# Methods and Semantics for
# Telecommunications Systems Engineering

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

**Stefan Leue**

von Deutschland

Leiter der Arbeit:
Prof. Dr. Dieter Hogrefe,
Universität Bern

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Der Dekan

Bern, den 19. Januar 1995

Prof. Dr. C. Brunold

Erschienen im Selbstverlag

Bern, Dezember 1994

*Für meine Eltern,*
*Christa und Rudolf*

*Mißverständnis zweier Surrealisten*

"es regnet"
sagte sie
"männer in schwarzen mänteln
gehen vorbei"
sagte sie

Magritte aber
hörte sie
nicht mehr genau
(sie sagte es nämlich erst Jahre
nach seinem Tod)

So hörte er nicht mehr
ihre letzten zwei Worte
und verstand nur
"es regnet männer in schwarzen mänteln"
Das malte er

*Erich Fried*

# Preface

This thesis addresses three aspects arising from the use of software engineering techniques, based on formal methods, in telecommunications systems development. Firstly, it will consider a formal semantics for Message Flow Graphs and Message Sequence Charts which are formal techniques of particular importance in telecommunications systems engineering. Certain aspects of the specification of quality of service (QoS) requirements of telecommunications systems are then addressed, with particular respect being paid to real-time requirements. Finally, a method for deriving optimized parallel implementations from formal protocol specifications is proposed.

Parts of the thesis are the result of joint work. The semantics of *Message Flow Graphs and Message Sequence Charts* has been developed jointly with Prof. Peter Ladkin, and the work on *parallel optimized protocol implementation* originates from a collaboration with Philippe Oechslin.

Some of the work described in this thesis has already been published or will be published in the nearer future. The work on the semantics for Message Flow Graphs and Message Sequence Charts will appear in the journal *Formal Aspects of Computing* [95]. Part of the work was also published in the proceedings of the *6th International Conference on Formal Description Techniques (FORTE'93)* [93], and a discussion of implications of the formal semantics appeared in the proceedings of the *7th International Conference on Formal Description Techniques (FORTE'94)* [94]. Work on the specification of Quality of Service requirements was presented at the *Montreal Workshop on Distributed Multimedia Applications and Quality of Service Verification* [104]; while the work on protocol implementation was presented at the *4th International IFIP Workshop on Protocols for High Speed Networks* [106], and at the *2nd IEEE International Conference on Network Protocols (ICNP-94)* [105]. (Precursors of this work were presented at the *4th IEEE Workshop on Future Trends of Distributed Computing Systems* [107]). Unless absolutely necessary, references to these publications within the text have been omitted.

## Acknowledgements

The work documented in this thesis has been carried out while I was a research assistant at the Department of Computer Science and Applied Mathematics of the University of Berne, Switzerland. The following organizations have supported my research financially: The Swiss Telecom, The Hasler Fund, The Swiss Federal Office for Education and Scientific Research, and The Swiss National Science Foundation. I wish to express my gratitude to these organizations for their generous support.

I would like to thank my thesis advisor Prof. Dieter Hogrefe for his guidance and advice, and for providing me with the excellent environment to allow me to carry out my research.

Prof. Reinhard Gotzhein, Prof. Peter Ladkin, and Prof. Claude Petitpierre were the external reviewers of my thesis. I wish to thank them for finding the time to do the reviews and for their many helpful suggestions for improvement, at early as well as at late stages of my work.

I am deeply indebted to Prof. Peter Ladkin for his constant encouragement, advice and friendship throughout the last five years since we first met in Berkeley in 1989. His constructive criticism and his collaboration have helped me greatly to appreciate the true nature of what it means to do research work in the field of computer science, and in developing the skills necessary to achieve my research goals.

My very special thanks are also due to Philippe Oechslin for his friendship and collaboration. His practitioner's perspective on problems in telecommunications systems engineering have greatly helped to relate my theoretical ideas to real-world problems.

In addition to the above mentioned individuals many more people have given me their valuable opinion on the research presented in this thesis. The comments I received from John Donaldson, Prof. Jean-Pierre Hubaux, Dr. Robert Kurshan and Dr. Ekkart Rudolph were particularly influential and helpful. From John Donaldson I also received extensive advice on linguistic questions, and I thank him for finding the time to review major parts of the text.

Finally, I would like to thank all of my colleagues, friends and relatives who have encouraged me in the past to pursue my research career – and I sincerely hope that they will continue to help me in very much the same way in facing future challenges.

Berne, December 1994 Stefan Leue

# Contents

# List of Figures

# List of Tables

# Part I

# Introduction

## Telecommunications Systems Engineering

The development of telecommunications software systems is a highly complex process. In order to manage this complexity various software engineering methods have been developed, ranging from requirements and design specification techniques to verification, validation, testing and implementation methods. In practise, we group all of these approaches under the broad term *telecommunications systems engineering*.

We will focus here on those methods in telecommunications systems engineering which have a formal foundation. The methods considered are expected to be based on formally defined specification languages with precisely defined syntaxes and formally defined semantics. Furthermore, these methods rely on formally well-defined transformations or at least they provide formal support for them. For example, the implementation of a specification is an important transformation for which a formal support is desirable.

The roots of a formal approach to telecommunications systems engineering can be traced back to *protocol engineering* based on formal methods in the 1970s and 1980s. Historically, the development of protocols was the main concern in the development of telecommunications systems. This was mainly due to the fact that protocols are distributed systems, and, as such, are subject to various difficult inherent design and verification problems[1]. A typical consideration in this field is that the design of protocols has had to be such that deadlock and undesirable lifelock situations were avoided. Other challenges in protocol engineering could include: (a) the detection and recovery from communication-media or communication-partner failure (e.g. by using timeout mechanisms), (b) the assurance of the completeness of a protocol machine with respect to a possible input/output alphabet, (c) the distributed testing of protocol implementations with respect to conformance to a given reference specification, and finally (d) verification that a protocol implements a specified service for a higher layer user instances.

Many of these approaches are still very important. However, with communication systems evolving towards high speed telecommunications infrastructures supporting heterogeneous traffic types, protocols are no longer the only subject of interest. Architectures have changed to be service oriented, with protocol mechanisms (for example in ATM) decreasing in overall significance with regard to the system's design. On the other hand, new requirements due to new classes of applications have evolved, such as the requirements relating to the quantitative aspects of the quality of the service provided by the telecommunications systems. It should also be pointed out that the classical layered protocol architecture model no longer has the same importance. Innovative communication architectures like *Open Distributed Processing* focus on object-oriented views, and network resource management protocols relying on object-oriented approaches have evolved. However, despite of their reduced importance the efficiency of protocol implementations have

[1]For overviews see [108] and [74]

become crucial, because in high speed communication environments the communication nodes have become the performance bottleneck. In order to encompass this variety of aspects we prefer to talk about *telecommunications systems engineering* instead of protocol engineering when referring to these problems and methods. The thesis addresses methods and semantics for use at various stages of a telecommunications systems engineering methodology. However, we will not refine in detail what this methodology should look like. We leave this point for further study, although it is intended that the methods and semantics provided here will be very helpful in a prospective telecommunications systems engineering methodology.

## Thesis Outline and Contributions

We now look at the motivation for this work, and introduce the various topics that are to be addressed in it. We also indicate the achievements arising from this work, and for which the reader will find the supporting arguments later in the text. The main body of the thesis is structured into three mainly independent parts. Part II presents a *formal semantics for Message Flow Graphs and Message Sequence Charts*, Part III suggests methods for *Quality of Service specification*, and Part IV finally presents an *efficient protocol implementation* methodology.

### The Semantics of Message Flow Graphs and Message Sequence Charts

Many specifications in telecommunications systems design focus on the specification of message exchanges between communicating systems, or components thereof. The systems considered can be either protocol or service specifications. *Message Sequence Charts* (MSCs) (also known as *Time Sequence Diagrams*, *Temporals Message Flow Diagrams* etc.) are a particularly appealing pictorial representation of message exchanges between systems. The common characteristic of these charts is that they graphically represent processes on different, most often vertical axes, and messages by directed arrows between points on the process axes. Recently, MSCs have also been incorporated into object-oriented specification and design methodologies, where they are used to describe communications between autonomous objects.

#### Outline of Contributions in Part II.

- We demonstrate that MSCs are a particular sort of Message Flow Graphs (MFGs), a notion originating from the analysis of code for parallel communicating systems. We also show how to map the graphical object 'MSC' into a mathematical object, the corresponding MFG, and we show how to translate a set of MSCs into an MFG by means of a syntactic interpretation of the composition of MSCs along conditions.

- We then argue for the necessity to define a formal semantics for MFGs and MSCs. To support this claim we illustrate the necessity for tool providers of MSCs to refer to an unambiguous semantics definition, and exemplify how in one case the definitions given there may lead into counterintuitive and logically contradicting specifications.

- We claim that the semantics we define for MSCs is applicable to a wide range of occurrences of MFGs and MSCs, namely telecommunications systems, object-oriented design methodologies, and the analysis of parallel code.

- One of the main underlying assumption for our work is that the semantics is a formal representation for the interleaved traces of communication events defined by an MSC specification.

- We argue that the semantics for MFGs and MSCs is inherently finite-state, and show that $\omega$-automata, of which the Büchi automaton is a well-known example, are a possible semantical model. We demonstrate that liveness properties are underspecified in MFG specifications, and we provide means to add liveness constraints by defining Büchi automata acceptance conditions for MFG specifications.

- By showing how an arbitrary Büchi-automaton can be simulated by an MSC specification, and from our semantic assumptions, we conclude that Büchi-automata and MSCs are expressibly equivalent.

- Next, we prove that temporal logic is a more flexible tool for the definition of the liveness criteria, and we show that our state-transition system based semantics avails itself easily to an interpretation as model for temporal logic specifications.

- We argue for the need to handle both synchronous and asynchronous communication in the semantics for MFGs (although the communication in standard MSCs is only asynchronous), and we provide a semantic interpretation for both communication mechanisms.

- We compare our definitions with informal descriptions of the semantics in the ITU-T standard document Z.120 for MSCs, and conclude that some of the suggestions there are infelicitous. This includes the textual representation of MSCs, which we prove *not* to be well-defined in Z.120.

- We also compare our approach with alternative approaches to a definition of the semantics for MFGs and MSCs, in particular with a recently standardized approach which has been added as Annex B to the ITU-T standard document Z.120. We point at different ambiguities and shortcomings of this approach, and we conclude that we interpret MSCs more completely.

- We show that seemingly innocuous syntactic choices, in particular the cross-over of messages, can have implications on hidden assumptions on the behaviour of the environment. We criticise this because in our view when dealing with a very simple and intuitive specification style like MSCs *what you see* should be *what you get*.
- As a consequence of the *what you see is what you get* requirement as well as of our arguments for a finite state semantics, we conclude that there are no queues involved in the communications between processes.
- Furthermore, we point out that the one-to-one communication relationship between sending and receiving of messages (later in the text called 'the property (*)') distinguishes communications in MSCs from many other concurrent specification techniques, like for example SDL.
- Finally, we show that the unimpeded use of conditions leads to so-called non-local choice situations, which can only be handled by using potentially unbounded history variables in the environment, or similar mechanisms. This contradicts both our finite-state assumption, as well as our *what you see is what you get* requirement.

### Quality of Service Specification

Telecommunications Systems are evolving towards highly complex systems providing heterogeneous services at very high communication speeds. A consequence of this development is that quantitative aspects of the quality of the service provided need to be specified, and mechanisms for assuring their satisfaction need to be implemented. Examples for these requirements are delay bounds, delay jitter bounds, throughput rates and loss rates which are essential to video transmissions in multimedia applications. These sorts of requirements are often referred to as *Quality of Service* (QoS) requirements, and they usually rely on real-time and probabilistic properties. The standard Formal Description Techniques (FDTs) like Estelle, LOTOS and SDL, however, do not provide for expressing these properties, therefore we investigate approaches for their specification in Part III.

#### Outline of Contributions in Part III.

- We analyze the real-time mechanism in SDL, and we conjecture that it is unsuitable to specify real-time progress or bounded response properties, due to a lack of urgence of events.
- We show that it is possible to interpret SDL specifications as models for temporal logic formulas, and we provide a sketch of such an interpretation.
- We define the concept of complementary specifications, which are joint SDL/MSC and temporal logic specifications.

- We then extend the interpretation to timed models and real-time temporal logics in order to specify hard real-time constraints for SDL specifications.
- Then we exemplify the application of these complementary specifications to the specification of some common real-time related quality of service requirements for telecommunications services, to real-time related aspects of protocols, and to QoS mechanisms.

### Efficient Protocol Implementation

A further consequence of the evolution of telecommunications systems and in particular of the underlying optical transmission technology is that, as opposed to conventional communications systems, the performance bottleneck is no longer the transmission link, but instead the protocol processing machine. This can be illustrated by a simple example: consider a standard workstation with a 32 bit architecture and a bus clock with a frequency of 25 MHz, then this yields a maximal data transfer rate inside the machine of 800 Mbit/sec, even if the processor runs at a multiple of the bus clock frequency [121]. This data transfer rate is easily exceeded by data transmission rates in broadband communication infrastructures like ATM. It is therefore imperative to have efficient protocol implementations available. In Part IV we therefore propose a method to transform the sequential structure of operations inside the processes of an SDL specifications into optimized relaxed dependence graphs which serve as a basis for for efficient parallel implementations of the specified protocol.

**Outline of Contributions in Part IV.**

- We show that it is inefficient to implement SDL specifications in a 'faithful' way by structuring the implementation according to the structure of the specification.
- It is argued that the lack of explicit parallelism inside SDL specifications, the structuring of SDL specifications into processes, and the asynchronous inter-layer communication mechanism object to the efficient direct implementation of SDL specifications in a 'faithful' way.
- We suggest the construction of a multi-layer dependence graph of statements in different layers of an SDL specification. We transform this graph into a relaxed dependence graph, mainly by discarding sequential control flow dependences and retaining data dependences.
- The relaxed dependence graph serves as a basis for the interpretation of different protocol implementation optimization methods, like combined execution of data manipulation operations, and for a parallel execution.

- Depending on the target hardware and the resource constraints of individual operations this leads to a scheduling problem, which may be solved at compile- or run-time.

**Acknowledgements.** As already mentioned, a major part of the work in Part IV arose from collaboration with Philippe Oechslin, and is based on his and the author's joint idea that control flow dependences need to be relaxed in order to allow for efficient implementations of the operations in a protocol stack. The ideas and concepts in Part IV due to contributions made by Philippe are: the determination and derivation of a *Common Path Graph*, the *Anticipation of the Comon Case*, the notion of *Auxiliary Dependences* which need to be added to data dependences to form the Relaxed Dependence Graph, and the ideas concerning a *Scheduling of Operations* in an implementation. The respective material will be published in [122].

# Part II

# The Semantics of Message Flow Graphs and Message Sequence Charts

# Chapter 1

# Introduction

> *"Formalized methods ... continue to rely on the intuitive understanding of the notations and concepts employed: they may replace a possibly wooly natural language description with, say, an apparently precise diagram – but the precision is illusory if there is no underlying semantics giving a strict meaning to the diagram."* [133]

The purpose of this part of the thesis is to give a precise formal semantics to a specification formalism often referred to as *Message Flow Graphs* (MFGs). Experience in both academic research and in industry has shown that MFGs lend themselves to easy pictorial representation of inter-process communications, and they are consequently found in telecommunications, distributed, and object-oriented system design, and are frequently used in textbooks. Informally, they make helpful pictures, which are easy for the reader to relate to, and this undoubtedly accounts for their popularity.

One type of MFG, is the Message Sequence Chart (MSC), defined in International Telecommunications Union (ITU-T)[1] Recommendation Z.120 [33]. MSCs provide a syntactically standardised description technique for telecommunications system design and validation. Throughout the remainder of this thesis, we shall refer to the ITU-T MSC standard simply as *Z.120*.

**What Are MFGs and MSCs Good For?** MFGs and MSCs describe process control structures and message exchanges of communicating processes. However they abstract from internal process computation. This distinguishes them from specification languages like SDL [32], Estelle [77] or LOTOS [78]. These languages specify the internal behaviour of communicating processes and the communication behaviour can only be inferred from the process code. Concludingly, one can say that MFGs and MSCs specify explicit communication behaviour while the process behaviour is implicit, whereas SDL, Estelle and

[1]The former ITU standardization body *CCITT* has been renamed *ITU-T* in 1993.

LOTOS specify the process behaviour explicitly while the communication behaviour is implicit. The system view represented by MFGs and MSCs can be helpful at all those stages of the telecommunications systems engineering process at which an easy and graphically appealing representation of a system's communication behaviour is particularly helpful, as for example at early design stages, or in conformance testing. For a discussion of some occurrences of MFGs and MSCs see Chapter 3.

**Why a Formal Semantics?** Work on formal semantics of MSCs has often been criticised by claiming that MSC specifications only show

(a) a partial view of the system behaviour, or

(b) an intuitive and possibly inexact description of behaviour traces or scenarios,

and that both points defeat the definition of an unambiguous, formal semantics. However, we are easily able to counter both of these points.

- Firstly, our work does not focus on methodological aspects. MSCs are used widely (sometimes intuitively, sometimes formally) at various stages of the software engineering cycle for telecommunications systems, and, used in such a manner, MSC specifications *do* describe system behaviours. Some opponents of a formal semantics argue that MSC descriptions only represent 'incomplete' traces of system behaviour. It remains unclear however, just what the completeness measure in this type of argument is, and we have come to the conclusion that it is irrelevant. Indeed, we provide a meaning to MSCs as they are given, independent of any particular context of application. However, we propose that the meaning we give is a canonical interpretation of MFGs and MSCs, and is thus applicable in any context.

- Secondly, we propose that for MFGs and MSCs to have any use at all, a precise meaning is indispensable. System specification methods used in industry can be very different from those investigated by researchers. One might say that while common industrial methods are good at book-keeping, well-engineered and relatively easy to teach, they can be fuzzy in stating system properties. In contrast, mathematical methods such as those based on logic or automata are more precise and expressive, but require greater depth of mathematical or logical understanding to use. We believe there is value in bringing the precision of logic-based specification methods to existing industrial methods.

  Rigorous specification methods such as Z, VDM, LOTOS, and the B Toolkit are already finding favor in industry. These methods seem to be following a path from use in academia to industrial research applications. In contrast, MFGs and MSCs are used in industry already, often informally. A precise semantics helps to illuminate system features and clarify issues during system development, and is highly desirable and almost certainly essential when wanting to use MSCs or MFGs in the context of system verification, validation and testing. In particular, it enables MFGs and MSCs to be used in *high reliability* or *safety-critical* contexts, in which precision is of the essence.

**Motivation.** Our motivation for this work came from two different directions. We believe that it is a touchstone of a worthwhile abstraction that it applies in different contexts.

- Firstly, it was demonstrated in [96] and [98] (summaries in [99], [97], with the complete material in [100]) that MFGs are very useful in deadlock and reachability analyses of parallel code. The MFGs were rather simple, involving loops but no branching. To extend the analysis, it became clear that some mechanism to keep track of branching was required.

- Secondly, in apparently unrelated work, we wanted to provide a rigorous semantics for MSCs and Time Sequence Diagrams (TSDs) [81] in an telecommunications systems engineering context, and we found it convenient to base their semantic interpretation on MFGs[2].

Given that MFGs have proved useful in different contexts, a natural next step is to define an unambiguous formal interpretation of each MFG, hence the present work.

[2] In earlier publications we sometimes referred to *ne/sig* graphs, a special form of MFG.

# Chapter 2

# What is a Message Flow Graph?

MFGs are a graphical, intuitive method for describing partial message-passing interactions between processes in communicating systems. They are frequently found in documents on design, validation and verification, as well as in textbooks. They are frequently used in describing aspects of telecommunications systems, and recently also gained importance in the description of communications in Object Models for object-oriented software development. One particularly important class of MFGs is that of Message Sequence Charts (MSCs), standardised by ITU-T Recommendation Z.120.

Telecommunications protocol and service specifications as well as the specification of communications in Object Models are distinguished amongst general system specifications by an emphasis on communication between processes rather than computation within a process, and by the relatively simple nature of the messages exchanged. *Message Flow Graphs* (MFGs) have been invented as a suitably abstract description method for this class of systems. They describe a system merely by the control structure of its processes, and by the structure of the inter-process message exchanges.

**Where are MFGs Found?** MFGs have been defined in the context of static analysis of parallel code. The currently most prominent area of application of MFGs is the design and development of telecommunications systems, where they can mainly be found as MSCs and TSDs. Recently, with the development of object-oriented design methods MFGs have entered a new field of application. For more information on the occurrences of MFGs see Chapter 3.

**Systems Employing MFGs.** MFGs have found their place in various software engineering methodologies and hence there are quite a number of commercial or non-commercial tools supporting MFGs that have been developed in academia and industry. Important groups of tools are those evolving from telecommunications systems engineering, and those related to object models. We shall mention some tools and discuss requirements on one

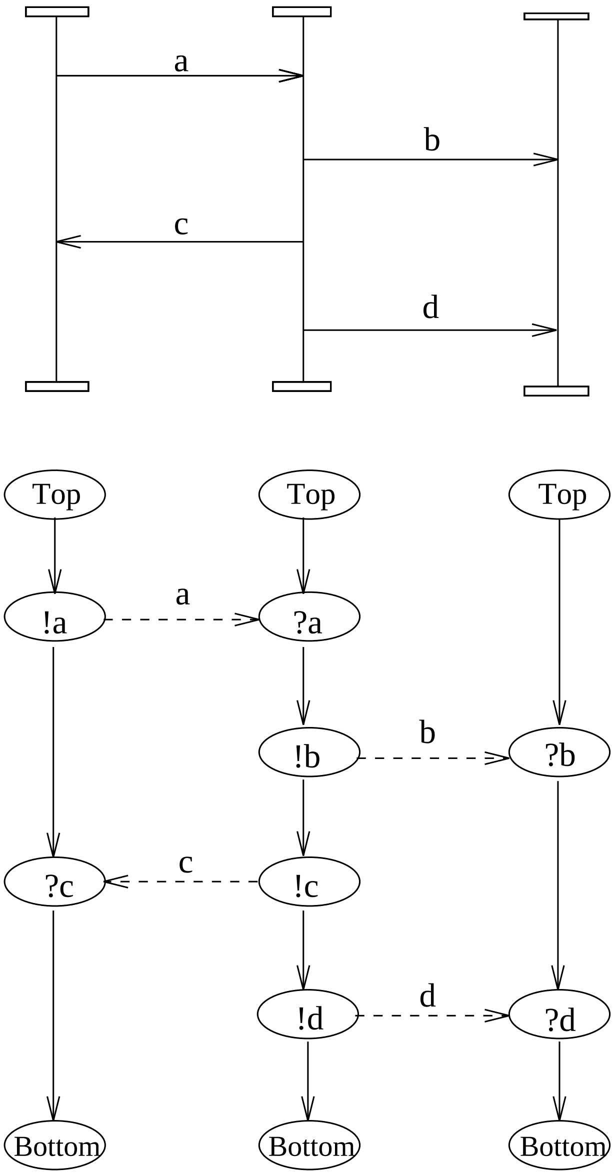


Figure 2.1: A simple Message Sequence Chart (top) and the corresponding simple Message Flow Graph (bottom).

particular tool in Chapter 6.

## 2.1 Simple Message Flow Graphs

MFGs are an algebraic representation of process control and message flow for communicating processes. MFGs may represent different descriptions of communicating processes, e.g. concurrent programming language code, abstract specifications of communication services or protocols, or high level message flow diagrams like MSCs or TSDs. In Figure 2.1

the MFG on the bottom represents the intuitive picture on the top which is similar to an MSC or TSD. The MFG in this example does not contain *conditions* (a notion introduced further down), we therefore call it a *simple* MFG.

In the picture on the top of Figure 2.1 processes are represented by vertical lines, and the signals sent between processes are represented by horizontal or sloping arrows. Communication is asynchronous. The junction between a vertical process line and a horizontal signal line represents an event at which a signal of the type specified is sent or received by the process. In each process axis, the events are temporally ordered from top to bottom, hence the ordering of events along a process axis is total. However, due to the concurrent nature of the different processes the picture describes a partial order of the communication events related to the sending and receiving of messages $a, b, c$ and $d$. The message `send`[1] and `receive` events are represented by the intersection of the message arrows with the process lines. In the example, the first process sends a signal of type $a$ to the second process, which upon reception sends a signal of type $b$ to the third process, a signal of type $c$ to the first process, and finally a signal of type $d$ to the third process. The system terminates when all processes have terminated.

The MFG corresponding to this picture is on the bottom in the same Figure. The basic idea of the MFG is that it is represented by a graph structure which has an underlying ontology of message `send` and `receive` events represented as nodes. MFGs have two kinds of edges, *next event* (*ne*) and *signal* (*sig*) edges, representing explicit relations on the nodes. The nodes are connected by solid arrows representing the *next-event* (*ne*) relation, indicating the next node in the same process (the process control), and dashed arrows corresponding to the *signal* (*sig*) relation, indicating from which node and to which node a message is passed. All nodes in an MFG, with the exception of the `start` and `finish` nodes, must be connected to precisely one other node.

The nodes (representing the events) are labeled with the event type. We use a variant of a common notation. The event node at the tail of a *sig* edge must be labeled with *!a* (`send` a message of type $a$), for some symbol '$a$' denoting the message type, and the event node at the head with *?a* (`receive` a message of type $a$), for the same '$a$'. (In some uses, it might be preferred to label the *sig* edge with $a$ and omit the node labels.) An MFG has `start` nodes (in the domain but not the range of the *ne* relation) labeled *Top*, and maybe `end` nodes (in the range but not the domain of ne) labeled *Bottom*[2]. We will present a formalisation of this informal definition of MFGs in Section 7.2.

[1] We sometimes abuse notation mildly by using the phrase 'message `A`' when we really mean 'instance of a message of type `A`', which is an awkward, although more accurate, phrase.

[2] In later MFG examples we sometimes also write a lower-case letter within a node to allow us to refer to that node in the text. These additional identifying letters do not occur in the MFG itself.

## 2.2 From MSCs to MFGs

MSCs are graphical devices, whereas MFGs are mathematical objects (graphs). As the semantics is based on the mathematical object we need to describe how to translate the former into the latter (although some readers may consider this a pedantry).

The translation is quite obvious. The MSC standard Z.120 defines notions like *instances* (the processes' control flows), *message output* and *message input* symbols. Z.120 defines these symbols for both a *graphical* and a *textual* representation of MSCs.

- In the graphical representation, an instance is represented by a vertical line, called instance axis. The instance axis may have horizontal or sloping arrows pointing away from it or to it. We call the intersection of the departing arrow with an instance axis an `output` symbol, and the intersection of an incoming arrow and an instance axis an `input` symbol.
- In the textual representation an instance is denoted by the keyword `instance` and comprises the subsequent code until an `endinstance` symbol is reached. The core code of an instance contains message input statements, denoted by keyword `in`, and message output statements, denoted by keyword `out`.

To cover both representations (although we mainly use the graphical form in our examples) we will from here on only use the terms `message input` symbol and `message output` symbol, with the obvious meaning in the context of any chosen representation. We will now simply *identify* these symbols with the nodes in the MFG (representing the corresponding events) and ensure that the MFG structure is consistent with the structure of the MSC. This means for example that the `message input` and `message output` symbols related to the message of type `a` in the MSC example in Figure 2.1 are mapped to a pair of nodes in the MFG so that this pair of send and receive nodes is in the *sig* relation of the MFG, and that it is labeled with signal type `a`. Furthermore, in order to represent the control flow of the left process in the corresponding MFG the node corresponding to the sending of message `a` must be connected with a node which corresponds to the `message input` symbol related to a message of type `c` in the MSC, etc. We formalise this mapping in Section 7.2.2.

## 2.3 Message Flow Graphs with Conditions

To this end, we define conditions in MFGs. Conditions are global labels on process axes[3]. We also define a composition operation which allows MFGs to be 'joined' at these conditions. This composition is a purely syntactic operation on MFGs. By allowing more

[3]The term *condition* has been introduced in Z.120. A condition is a syntactic label and should not be misunderstood as a condition in a logical sense.

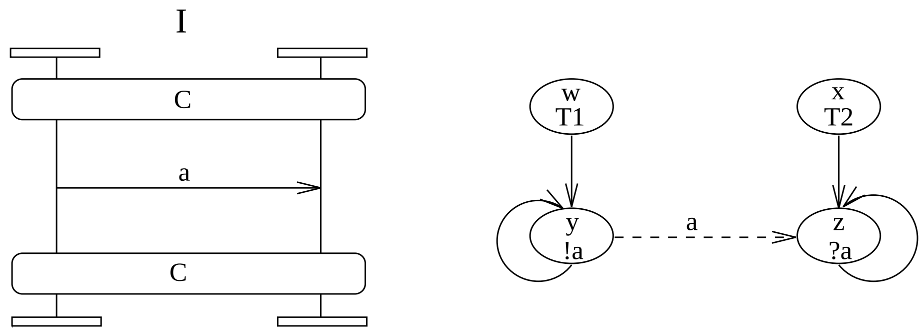


Figure 2.2: MSC I and corresponding MFG I

than one possible joining, one obtains the effect of non-deterministic choice in MFGs (but conditionals defined on the values of state predicates are still not possible). By writing the same condition at the beginning as well as the end of an MFG, one obtains non-terminating-loop-like behavior. This allows us to define the (partial) use of conditions as conceived in Z.120, but as we will note in Section 8.2, unrestricted use of conditions seems to entail that the environment has powerful implicit properties which are not explicit in an MFG specification.

The simple MSC and MFG example in Figure 2.1 does not show looping behavior of MFGs, as seen in Figures 2.2, 2.3, 2.4, and 2.5. A loop in an MSC specification is caused by conditions, as described below. A loop in an MFG is simply a cycle in the next-event (solid-arrow) relation. In such a case, the nodes in an MFG may no longer represent events, since in a trace of the MFG they are or may be traversed multiple times. Thus nodes in an MFG should be properly thought of in the manner of statements of a programming language, which may be executed multiple times, each execution of which is a message-passing event.

Whereas the MFGs in Figures 2.2 and 2.3 are simple non-branching non-nested loop structures, Figure 2.7 shows MFGs with conditions, represented by the diamond-shaped nodes, corresponding to the MSC specification in Figure 2.6. The idea is that control that arrives at a condition node may continue in another MFG from a condition node with an identical label, as if the MFGs were 'joined' at these condition nodes. Thus different MFGs starting with identical condition nodes provide different ways to continue control[4].

We now elaborate two reasons for incorporating looping and branching structures into MFGs: the need to represent *iterations* and *non-determinism* in MFG specifications.

### 2.3.1 Iterations in MFGs

[4]We are not totally convinced that this can be done without difficulty, as noted in Section 8.2. Nevertheless, we provide in Section 7.4 the apparatus to effect it.

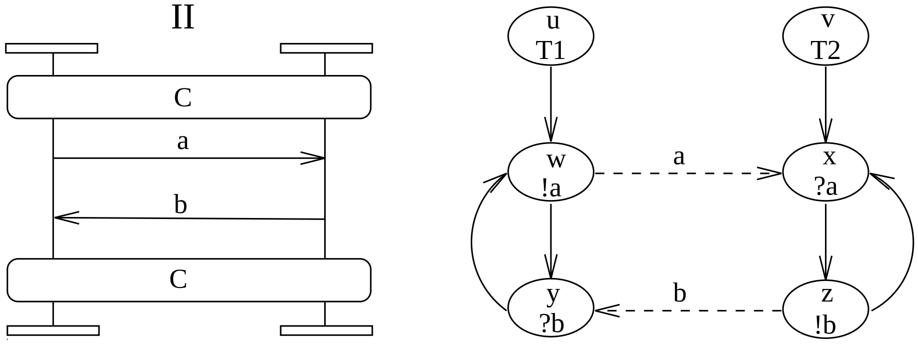


Figure 2.3: MSC II and corresponding MFG II

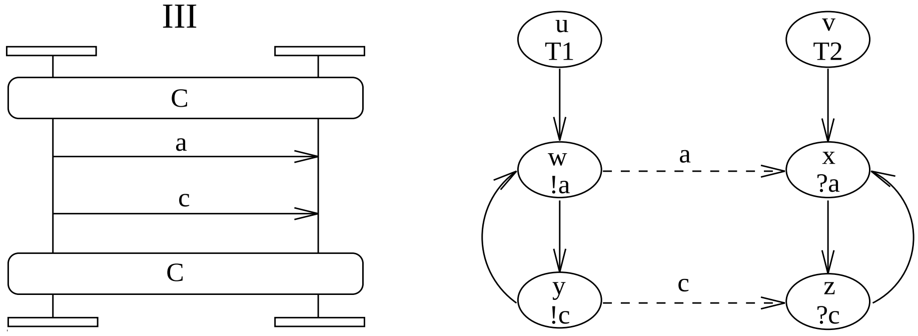


Figure 2.4: MSC III and corresponding MFG III

Representing entire MSC specifications[5] (which include multiple MSCs) as a single MFG may require branching or looping, which is disallowed in MSCs. For example, Figures

[5]We use here an intuitive 'definition' of *MSC specification* as a set of MSCs with conditions (see Figure 2.6). This is loosely correct but not strictly accurate. We do not refine this definition here because our purpose is to interpret MSCs, rather than to define a specification methodology. For example, it is outside the scope of this work to provide criteria for distinguishing 'allowable' MSC specifications from other non-allowable collections of MSCs (for example, those with 'bits missing'). Taking the phrase *MSC specification* to refer to some set of MSCs with conditions should cause the reader few problems if it is borne in mind that some such sets may have a meaningless content!

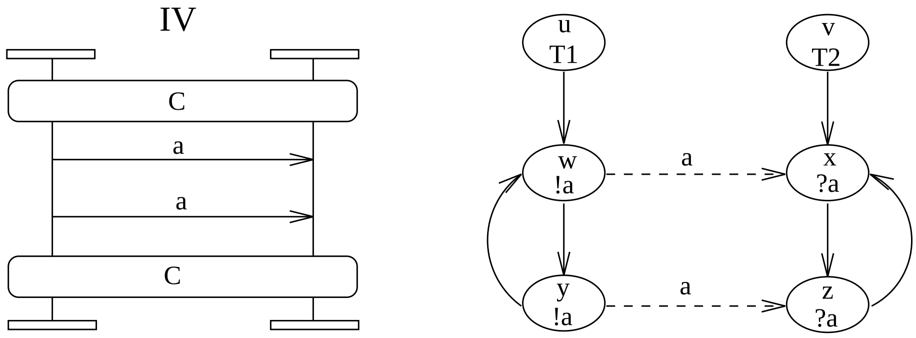


Figure 2.5: MSC IV and corresponding MFG IV

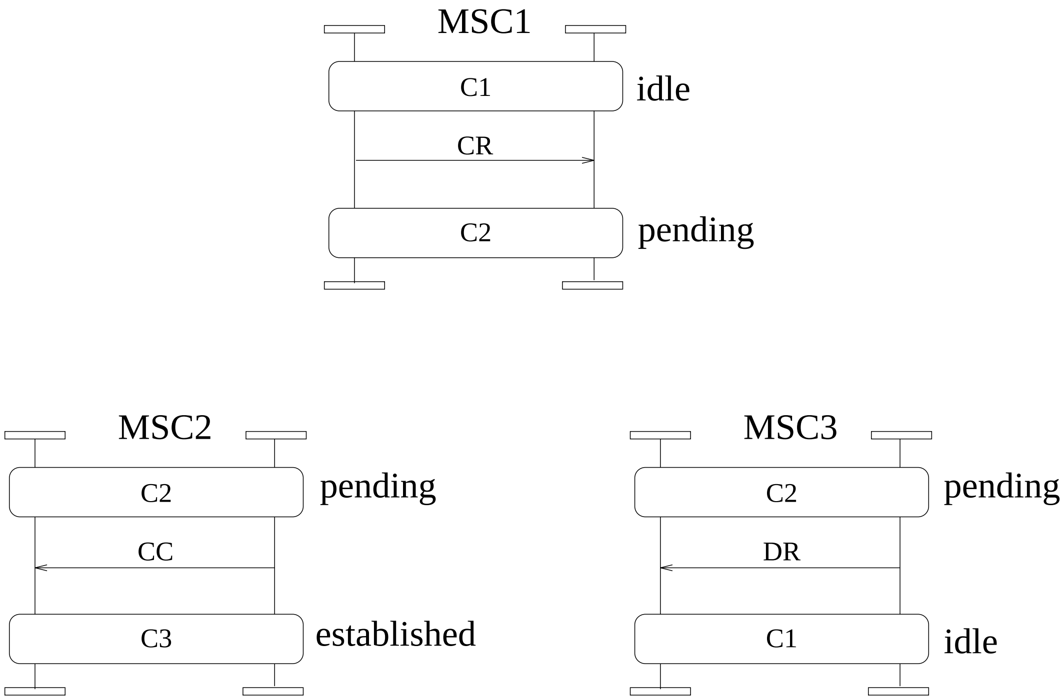


Figure 2.6: MSC specification with conditions

2.2, 2.3, 2.4 and 2.5 contain MSCs with conditions (later called MSC I, II, III, and IV, respectively), represented by the elongated symbols labeled `C` spanning the process axes. A condition is like a 'joint' for MSCs. The system is supposed to behave as though another MSC with an identically-labeled condition is joined on at the condition label. In MSC I, there is a single condition label `C` at top and bottom. Thus the MSC may be joined to *itself* at these conditions, creating a non-terminating loop in which the first process continuously sends signals of type $a$ to the second. MSC II is similar, in which $a$ signals alternate with $b$ signals in the other direction. Both MSCs are represented by MFGs in which the loops are explicit, as shown.

### 2.3.2 Non-determinism in MFGs

Conditions may also be used to specify non-determined behavior, as in Figure 2.6. We may understand this example as a protocol specification, namely as the connection establishment phase of some very simple connection oriented protocol. When the system is in state `idle`, which means that both processes are in that state, the first process may request the connection establishment by issuing a `CR` request and transit into a local `pending` state. Upon the reception of the `CR` signal the second process transits into its local `pending` state. A *global* state `pending` is reached, if both processes are in the respective local state. As mentioned above we may mark collections of local process states with common labels, in the case of the global system state `idle` with the label `C1` and in case of `pending` with the label `C2`. According to the syntactic definitions in [33], conditions may not cut through message arrows, thus they only represent global system states in which no message is in transit. Conditions only represent *possible* global system states - it is not required that these global system states are ever actually reached during execution of a system. At the condition `C2`, the second process may send a `CC` signal to the first, which indicates a confirmation to the connect request and a transition to the global state `connected`, or alternatively a `DR` signal to signal rejection of the connect request, before looping back to the beginning (condition `C1`). This gives rise to the branching and looping MFG in Figure 2.8.

**Non-Local Choice.** Using conditions in MFGs leads to potential branching of control in processes, in which processes may need to use information about other processes to choose a control branch. In Sections 8.2 we point out that an intuitive interpretation of non-local control choice necessitates using history variables to record the history of control-branch choices, as this information may be needed to determine the next state at any point in a computation. We regard any device that necessitates the use of history variables or equivalents in the semantics to be precluded by the arguments for an inherent finite-stateness of MFG specifications in Chapter 5. Thus the generality of non-local control choice arising from conditions precludes the unimpeded use even of global initial and final conditions.

**From MSCs with conditions to MFGs with conditions.** The translation is handled first by representing the MSCs with conditions as MFGs with condition nodes (Figure 2.7), which are an extra kind of node on each process axis, then joining the MFGs at these nodes and eliminating those condition nodes which are not required to synchronise non-local branching of process control. The formalisation of this *unfolding* is straightforward, but requires care in the details (see Section 7.2.4).

## 2.4 The Property (*).

Intuitively, an MFG is a graph representing concurrent processes exchanging messages. The nodes represent `send` and `receive` actions. There are two kinds of edges: *next-event* edges connect nodes to their successors within a process, and *signal* edges connect nodes to nodes in other processes with which they communicate. The essential property of an MFG is that

> (*) *each node is connected by precisely one signal edge to a unique node in another process.*

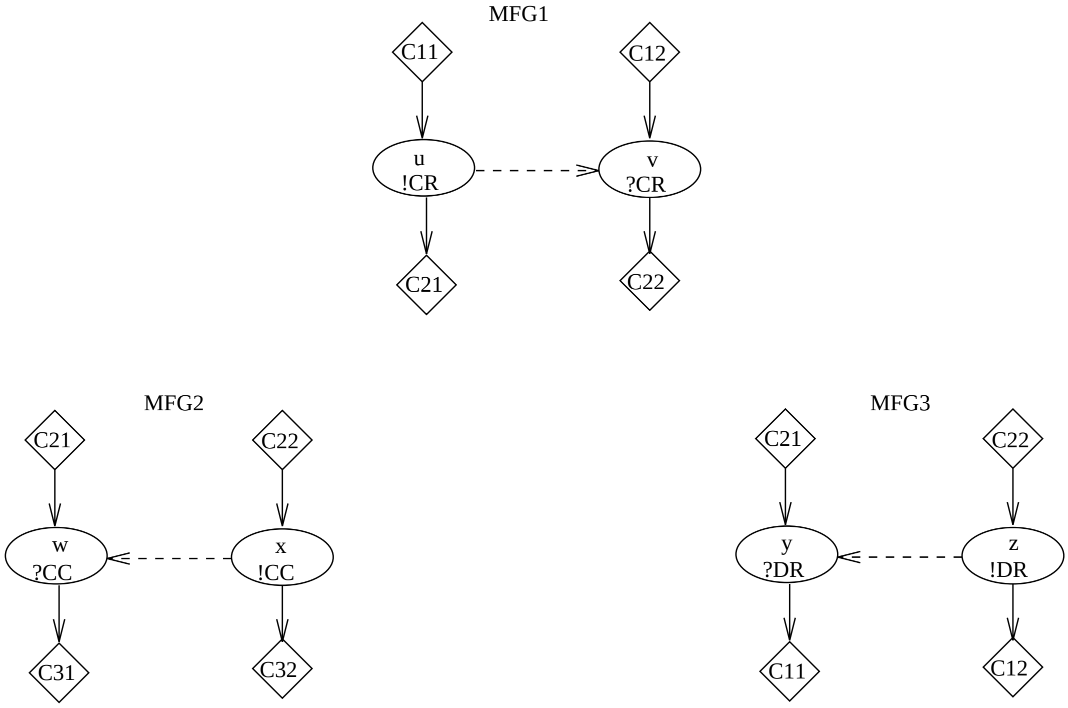


Figure 2.7: MFGs with conditions

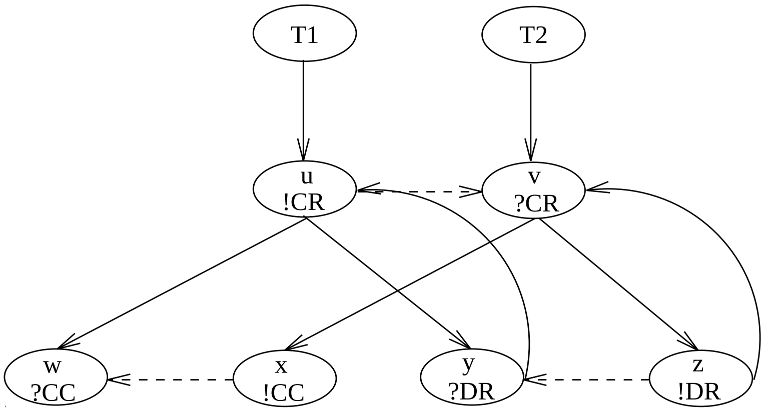


Figure 2.8: 'Unfolding' a set of cMFGs into a single pbMFG

The nodes may represent *atomic events*, as in Message Sequence Charts, or *communication statements* of a concurrent program as in [100]. Pure computation statements inside processes are ignored in the MFG, which focuses merely on the communication behavior of the processes, e.g. in order that deadlock and reachability analyses may be performed, or that the progress of the system may be checked for conformance to its specification.

The essential property (*) is guaranteed for such representations of message-passing as in MFGs or TSDs, but not for many other concurrent programming languages and specification techniques[6]. However, for other structures such as processes with a single non-terminating loop, proving the existence of an MFG describing the progress of the system can be non-trivial. An algorithm for constructing a minimal MFG from a collection of concurrent so-called loop processes was given in [100].

## 2.5 Message Flow Graphs: an Abstract Syntax

Different types of communication can have contrasting features, for example synchronous or asynchronous, channelled or broadcast, finitely or infinitely buffered, reliable or unreliable. Similarly, processes may be deterministic or non-deterministic, including parallel constructions or not, terminating or non-terminating. Because an MFG may be used to describe a mathematical syntax for the message-passing features of different specification methods, the graph is neutral with regard to the meaning of signal edges, the types of communication, or the control structure of processes. However, it does determine statically the sender and recipients of each communication (c.f. property (*)). Thus it is an abstract syntax: a *syntax* because it encodes only minimal semantic assumptions, and *abstract* because it abstracts from the details of the syntax of any specific application. The semantics arises from how the MFG is interpreted, as a particular kind of global state transition system. One major purpose of the MFG is to be able to derive a single MFG, where possible, from a collection of simpler descriptions of communicating processes. As seen above, for MSCs such a collection cannot be represented by a single MSC if the collection describes looping behavior of any sort. Representing a collection of MFGs with conditions as a single MFG without conditions (by means of the *unfolding* construction) enables us eventually to define the global state transitions and thus the automaton corresponding to a specification.

## 2.6 Overview of the MFG Semantics

In Chapter 3 we present some occurrences of MFGs and MSCs, namely their occurrence in the description of telecommunications systems (Section 3.1), in the analysis of parallel code (Section 3.2), and in object-oriented system analysis and design methodologies (Section 3.3).

We then specify our requirements for the semantics (see Chapter 4). These are most importantly that we use interleaved traces of message events, that the semantics is finite-state, that liveness is underspecified in MFGs and thus deserves special treatment, and that we include the handling of synchronous communication. In Chapter 5 we elaborate our arguments for a finite-state semantics. We further motivate our requirements on

[6]Note that in SDL *one* `OUTPUT(X)` statement can be matched by *many* `INPUT(X)` statements.

the semantic with a discussion of industrial tools employing MSCs, in particular on the GEODE toolset, in Chapter 6.

Chapter 7 contains the main part of the semantics definition. In Sections 7.3 and 7.4 we obtain a *global state transition graph* (GSTG) from an MFG. A GSTG is like a finite-state automaton but lacks definition of end-states. We consider end-state definitions in Section 7.5. Each possible end-state definition gives a Büchi automaton. The MFG specification under-defines the resulting automaton, in that end-state definitions are related to different safety and liveness properties that one might wish to require as additions to the specification. We show in Section 7.6 how this may be done via a connection with temporal logic, and we specify liveness properties for MFG specifications using temporal logic formulas there. We discuss some properties expressed in temporal logic which all MFGs satisfy, and some potentially desirable ones which some MFGs might be required to satisfy in some uses, in Section 7.8. We then show in Section 7.9 how synchronous communication can be accommodated along with asynchronous communication in MFGs, and give some reasons why this may be desired. We also note that the occurrence of synchronous communication in an MFG can simplify the liveness analysis. In Section 7.10 we show how to simulate an arbitrary Büchi automaton with an MFG specification, and conclude that MFGs are expressively equivalent to Büchi automata.

In Chapter 8 we scrutinize the semantics of MFGs. In particular, we show that the unimpeded use of conditions leads to the need of history variables of unbounded size, thus defeating our finite-state assumption. Furthermore, we show that crossing message arrows can cause undesirable implicit assumptions on the behaviour of the environment, that MFGs can 'count', and that liveness properties are better expressed by using Temporal Logic than by the use of acceptance conditions for Büchi automata.

In Chapter 9 we discuss the relation of our MFG semantics work to the MSC Z.120 standard document, including a description of the extent to which our work covers the MSC language as defined in Z.120. In Chapter 10 we discuss alternative approaches to a semantics for MSCs. Recently, a process algebra based semantics for MSCs has been standardized and added as Annex B to the Z.120 standard document. We offer a comparison of our approach with the standard in Section 10.1.

We suggest that readers primarily interested in the technical construction of our semantics directly proceed to Chapter 7.

# Chapter 3

# Occurrences of Message Flow Graphs

We discuss and give examples for the most important occurrences of MFGs: in telecommunications systems design, in the analysis of parallel code, and in object models.

## 3.1 Telecommunications Systems Description

The most prominent use of MFGs is in telecommunication systems design, in particular when specifying communication protocols and services. Most frequently, MFGs occur as MSCs or TDGs, both standardized graphical description methods (see [33] and [81]).

Many examples of MFGs can be found in textbooks on database systems and computer networks [140], where the use is mainly informal. In [147], where they are called *primitive sequences*, MFGs are used in the design of telecommunications systems. In industrial application contexts, MFGs are central to the work in [40] where they are called *temporal message flow diagrams*, and in [138] where they are called *MSCs*. For further use in industry see [41]. MFGs are also used in telecommunications standards, see for example [30] and [31].

As MFGs are widely accepted and reported in connection with telecommunications systems engineering, no particular examples are introduced here. However, the MFG examples used in this thesis (e.g. Figures 2.1 and 2.6) contain typical uses of MFGs in telecommunications.

## 3.2 Analysis of Parallel Code

MFGs were introduced in [96] as a means of analysing parallel code, in order to perform deadlock analysis and other optimisation at compile time. Analysis based on MFGs may be found in [97], [99], and [100].

The analysis is based on so-called *loop processes*, a pseudo-code notation for communication statements between concurrent processes. Internal computation is ignored in loop processes, loop processes reduced in this way to only contain communication statements are called *commstat-reduced*. A loop process has the form `A loop B endloop` where $A$ and $B$ denote linear sequences of communication statements without branchings and iterations. It is assumed that loops inside these blocks for which a finite loop count is known at compile time have been unwound into finite linear sequences. The expressiveness of loop processes is somewhat limited, they contain no conditionals, no indeterminate finite loops, and no execution indeterminism [100].

```
PROCESS S

send(R, CR)
receive(R, CC)

IF CC.flag = OK THEN
  LOOP
    send(R, DR)
    receive(R, DC)
    IF DC.flag = not OK THEN goto L
  ENDLOOP


L: TERMINATE
```

```
PROCESS R

receive(S, CR)
IF P THEN CC.flag := OK
send(S, CC)

LOOP
  receive(S, DR)
  IF Q THEN DC.flag := OK
         send(S, DC)
       ELSE DC.flag := not OK
         send(S, DC); goto L
ENDLOOP
L: TERMINATE
```

Figure 3.1: Concurrent pseudo code for abridged connection establishment and data exchange protocol

Loop processes contain communication statements of the form `W(AB)` where `A` is the name of the sending process, and `B` is the name of the receiving. Figure 3.1 gives an example of a concurrent programming pseudo code for a connection establishment and data exchange protocol. All variables are local to the processes `S` and `R`, the only communication between `S` and `R` is by message passing. Messages are of types $CR, CC, DR$ and $DC$. `P` and `Q` are local predicates to `R`. The message types denote data types with multiple components, one of which is called `flag`. It is designated to carry some control information about the state of the communication. The components of messages can be accessed like variables, so by `CC.flag` we refer to the value of the flag component of the last received message of type `CC`.

Processes `S` (sender) and `R` (receiver) first establish a connection through the exchange of `CR` (connect-request) and `CC` (connect-confirm) messages. If the connection establishment was successful, they perform a confirmed data exchange by use of `DR` (data-request) and `DC` (data-confirm) messages. Figure 3.2 gives the corresponding commstat-reduced loop process code.

```
process S             process R

W(SR)                 W(SR)
W(RS)                 W(RS)
  loop                  loop
    W(SR)                 W(SR)
    W(RS)                 W(RS)
  endloop               endloop
```

Figure 3.2: Commstat-reduced loop process code for example in Figure 3.1.

Message flow graphs in [100] are graphs over a set $N$ of communication statement nodes. There are two relations on these nodes, a relation $D$ (corresponding to our $ne$ relation) and a relation $U$ (corresponding to our signal edges in the $sig$ relation). There are some differences between the MFGs in [100] and ours. To make notation a bit clearer we shall denote the MFGs in [100] by *mfg*, and the MFGs introduced here by *MFG*.

There are some differences between MFGs and *mfg*s. In MFGs the *sig* edges are directed, whereas the $U$ edges in *mfg*s are undirected. However, the $U$ edges in *mfg*s become directed when an extension to asynchronous communication is introduced. Message arrows in *mfg*s do not have message types like in MFGs. It is however shown in [100] that the proposed analysis algorithms can be extended to handle message types when message types are determinable statically from the source code. *mfg*s are less expressive than MFGs with respect to conditionals, which can be represented in MFGs by *conditions* (see Section 2.3), but for which there is no device in *mfg*s. Both *mfg*s and MFGs satisfy the one-to-one condition on the sends and receives of messages which we called property (*) in Section 2.4.

More precisely, [100] distinguishes between *input graphs* and *message flow graphs*. An input graph is directly derived from the loop process code and contains a $D$ edge between nodes corresponding to successive communication statements, but no $U$ edges. These are added to the graph during the *mfg* construction. The construction involves matching communication statements in different processes. The matching process relies on a duplication of the loopbodies. When a match of communication statements in two processes is made, i.e. one `W(SR)` statement in process `S` and one in process `R` are matched, the corresponding $U$-edge will be added to the graph. The matching condition guarantees that a $U$ edge will be added to the graph if and only if no cycles are detected in $D$ and $U$ edges which are ordered previous to the current one. The generation algorithm guarantees that an *mfg* will only be completely generated if the loop processes are deadlock-free.

**Example.** Figure 3.2 contains the so called *commstat-reduced* loop process code for the example in Figure 3.1. The commstat-reduced code is an abstraction of the original process code. First, all statements in the code which do not relate to control- or message flow have been eliminated. Second, in process `R` there is a conditional `if Q then ...` inside the loop. Depending on the evaluation of `Q` messages with different types will be sent (one with `DC.flag = OK` and one with `DC.flag = not OK`). As *mfg*s do contain conditionals a direct translation of this code to *mfg*s is prima facie not possible. The algorithm in [100] however aims at deadlock detection, and for that purpose it is important to consider message flows. It is therefore appropriate to abstract away from the message type. As the direction of the message flow (from `R` to `S`) is not affected by the conditional we replace its branches by only one branch, which leads to the representation in Figure 3.2. Figure 3.3 presents the *mfg* for the example.

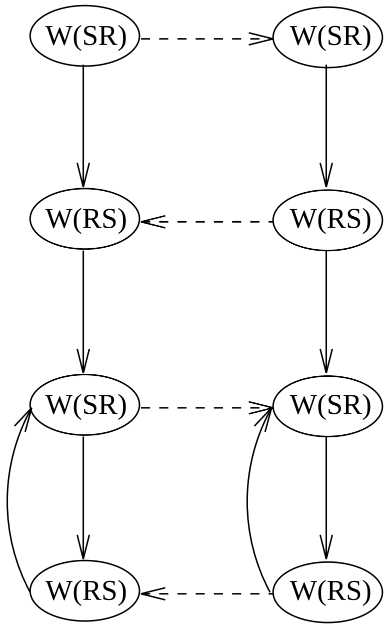


Figure 3.3: Message Flow Graph.

Although the purpose of the *mfg*s in [100] is different from the purpose of those in our work, our semantics can be directly applied without change to the *mfg*s in [100].

## 3.3 Object-Oriented Analysis and Design Techniques

There is a special interest in MFGs (and more generally in MSCs) in object-oriented analysis and design techniques. Systems are collections of independent modules, or objects.

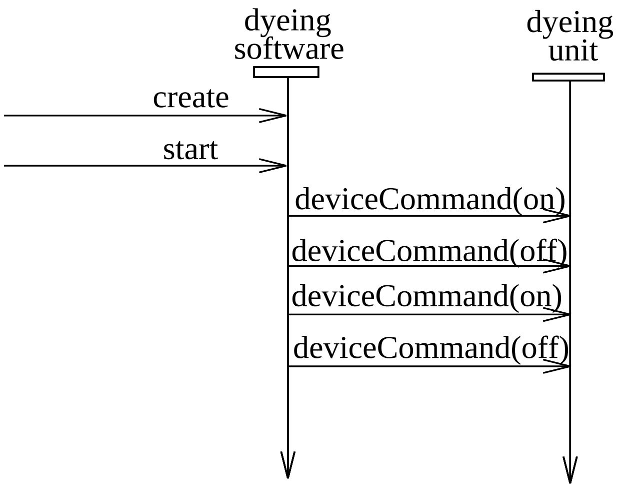


Figure 3.4: MSC describing Internal Message Sequence for the `DyeingSystem` class definition (taken from [137]).

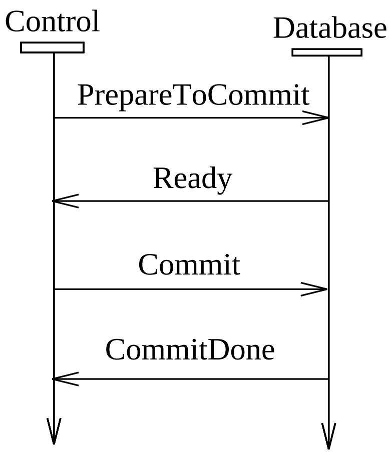


Figure 3.5: MSC describing a Two-Phase-Commit protocol (taken from [137]).

Any interaction between these objects (e.g. service calls) involves communication, and MFGs and similar creatures are used to describe the sequences of allowable messages exchanges between objects.

### 3.3.1 MSCs in Real-Time Object-Oriented Modeling

MSCs are used in Real-Time Object-Oriented Modeling (ROOM) [137] as general method to describe *"series of causally chained events typically spanning multiple tasks in a system that represent some meaningful high-level operation"*, in which they are called *scenarios*. They occur in two main contexts, as descriptors for *internal message sequence* specifications and for *protocol* specifications.

- First, the ROOM object model is structured into *actors* as main constituents. Actors are defined by actor classes. Actors can have concurrent *threads* of control, where each concurrent thread belongs to a component. MSCs are used to describe the internal message exchange between components of an actor. Figure 3.4 shows an internal message sequence for the `DyeingSystem` class definition in [137]. The `create` and `start` message arrows represent interactions of the `dyeingSoftware` component with the environment, the other arrows describe an internal message exchange between the `dyeingSoftware` and `dyeingUnit` components of this actor class.

- Second, MSCs are used in ROOM to describe so-called *protocols*. A protocol class is a high level construct of a specific structure describing the interactions between actors. Formally, a protocol class is described by sets of outgoing and incoming messages and a set of message types. For an example of the MSC based description of a protocol taken from [137] see Figure 3.5.

There are different reasons why our work is pertinent to the ROOM method. First, scenarios have a dynamic run-time function in ROOM. They are used to describe expected sequences of interactions of a model with its environment. When at run-time a divergence of the actual behaviour of an actor as described by an internal message sequence is detected, a warning message will be issued. Our semantics provides the means to determine (potentially already at compile time) what the allowable internal message sequences of an actor are and thus serves as a basis for generating the above described warnings.

Second, it is suggested in ROOM to use sets of scenarios for the derivation of a so-called *Complete State Definition* of an actor. It is apparent that because of the inherent global state complexity in concurrent systems automated ways of generating global systems states and the transition relation on these states are indispensable. The semantics given here provides a formal means for the definition of complete states, given that complete states are what we call global system states (see Chapter 7).

There are, however, some reasons why our semantics is adequate for MSCs as used in ROOM. Communication in the ROOM model is by synchronous or asynchronous message exchange, and our semantics provides for both types of communication, even in the same chart. Furthermore, we have a strong argument for requiring MFG specifications to be finite-state. This is of course an argument independent of the usage of MSCs, solely depending on which information is inherent in an MSC and which is not. However, as noted earlier, the usage of MSCs in ROOM is to describe communication between actors, or between their components. And these are finite state control objects in ROOM [136]. This means that the usage in MSCs inside ROOM, based on the semantics provided here, is consistent with the semantics of the control structure of ROOM actors.

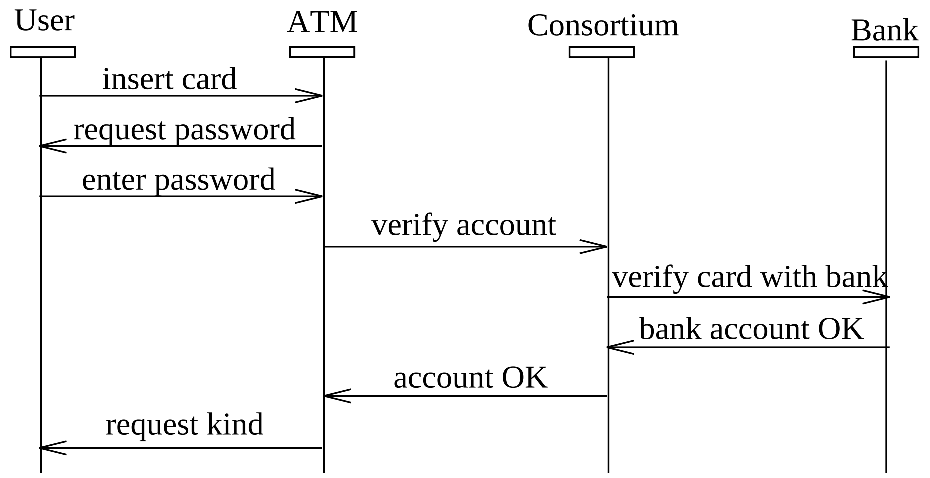


Figure 3.6: MSC describing an event trace for an ATM scenario (part of an example taken from [132]).

### 3.3.2 MSCs in Object-Oriented Modeling and Design

In the Object-Oriented Modeling and Design Methodology of [132] MSCs occur as so-called *event traces*. Figure 3.6 shows part of the message exchange in an object model describing an automated teller machine (ATM) example (taken from [132]). The event trace shows interactions between the objects `User`, `ATM`, `Consortium` and `Bank`.

Event traces are used at early stages of the design process. After objects and interfaces have been defined the design process continues with the development of scenarios, which are sequences of events (e.g. `<insert card, request password, ...>`) of particular executions of the system. Events transmit information from one module to another, and hence correspond to the *sig* edges and their types in our MFG model. When it has been determined which events transmit information from which object to which, event traces can be derived from the scenarios. Event traces are then used to derive event flow diagrams (by projection of events to pairs of objects), and then to derive state transition diagrams for implementations of these objects.

There are two main reasons why our semantics work is relevant to the use of MSCs within this design methodology. Our semantics generates a global state transition graph which may be traversed for many purposes. First, verification and model checking techniques (see for example in [74] [37, 102, 24]) rely on algorithms traversing state graphs and our semantics avails event trace specifications to these verification methods. Second, as a by-product a user may simulate the operating of the system by traversing the global state graph. This is of particular importance when the event traces are transformed into state diagrams for the individual objects. As proposed in [132], this translation is a manual heuristics, carried out by the designer. It targets at detecting repeating sequences of events in order to detect potential loops in the object's control flow, and thus to determine potential system states of individual objects. This step can be greatly enhanced by a simulation of the systems execution. Also, having a precise mathematical description of what the event trace means helps in automating this manual task.

# Chapter 4

# Requirements for the Semantics

We now discuss our requirements for the MFG semantics. These are mainly, that we consider our semantics to be a determination of the interleaved sequences of communication events specified by an MFG, that they be finite-state, that liveness properties are underspecified in MFGs and that therefore they deserve special treatment, that the complexity entailed by our semantics is not devastating, and that we handle both synchronous and asynchronous communications.

## 4.1 Traces of Message Events are Interleavings

We consider a semantics to be a precise determination of which execution *traces* a description allows. Since we focus only on communication events, we take *trace* to mean *trace of communication events only.* Internal process computation is ignored, although it can easily be added if desired. An interleaving model is one in which a trace is an interleaving of all observable atomic message-passing events of the system which is consistent with the linear ordering of events within each process, from the point of view of a global observer. Interleaving models are used for many important specification styles, including TLA [101], CSP [68], and LOTOS [78]. Taking traces of message passing events to be interleavings is consistent with industrial use of MFGs. MSCs are often used for talking about fragments of tests, as they are in GEODE [6]. Taking traces to be interleavings of events is consistent with the standard test definition language TTCN [79]. Thus we take traces to be interleavings of atomic communication events.

## 4.2 Finite-State Semantics

Even if it is assumed that there is only a finite number of possible control states with respect to messages, it nevertheless might be possible, if communication is asynchronous, for there to be an unbounded number of messages in the system (i.e. sent but not received).

Thus it is not immediately obvious that there are a finite number of global communication states if one includes the messages in the system as well as the process control predicates.

However, we shall argue in Section 5.6 that provided that the global state assumption is satisfied, and traces are interleavings, there are only finitely many global states with respect to the message-passing behavior of the system described by an MFG. Thus we may define a global finite-state automaton, whose accepted language is identical with the set of system traces described by the MFG.

Besides this argument which is particular to MFGs, justification for a finite-state requirement in general for telecommunications system specifications may be found in [74]. The primary advantage for finite-state interpretations is that reasonable verification and validation techniques may be applied. There are many practical techniques for analysis and verification of finite-state systems ([37, 102, 43] as well as [74]). Such methods can exhibit a high degree of automation. They have been used successfully to validate systems with up to $10^{14}$ states [42] by exhaustive state search, and up to $10^{20}$ states are being considered [29].

In contrast, non-finite-state methods usually employ theorem-proving techniques which are comparatively human-labor-intensive, are often research vehicles, and are currently limited in practical use. [91] argues for mixing the two sorts of techniques to obtain the advantages of both. The authors use COSPAN and TLA to verify a hardware multiplier design. As of the time of writing, the state-of-the-art in practical telecommunications system verification and validation is finite-statecraft.

One consequence of finite-stateliness is that MSCs, even those sets of MSCs with conditions (see below) that describe infinite traces, exhibit only finitely many global states. As we stated above, alternative proposals for the semantics of MSCs such as [134, 48, 64] do not define the semantics of composed MSCs, and therefore provide much less coverage than our semantics described in this document. Even if they did, our argument for finite-stateliness indicates that since the states of the system may be enumerated, these alternative approaches based on non-finite-state methods need to argue their case on technical grounds of greater efficiency, or maybe on aesthetic grounds, but no evidence has yet been given that either of these pertain. We will elaborate the finite-stateness requirement in our semantics in some more detail in Chapter 5.

## 4.3 Liveness Conditions

Given that a system is described by a finite state-transition graph, there is nevertheless a question as to whether all traces through this graph are acceptable traces of the system, or whether only a subset of them are. With asynchronous communication, a means is required for expressing liveness and safety assumptions in addition to the basic vocabulary of MFGs, argued in Section 7.5 and solved in Section 7.6. In order to facilitate the expression of a wider array of liveness properties, it is necessary to go beyond the Global State Transition Graph (GSTG) to consider which traces through the graph are allowed by the description (along with the liveness properties) and which are not. A standard way to express these conditions is to consider the GSTG as providing most of the definition of an $\omega$-automaton, lacking only an end-state definition, and to provide that end-state definition.

However, when communication is synchronous, the expression of safety and liveness are much simplified, as noted in Section 7.9, and any non-finite-state semantics for this case would be contrived. If one is defining a specification method, this would provide a good reason for preferring synchronous communication primitives, as argued in [68, 119]. However, MSCs as they are normally used utilise asynchronous message-passing.

## 4.4 Büchi- and Other $\omega$-Automata.

Since traces may be infinite, a finite-state semantics requires use of a finite-state automaton which accepts infinite strings. This is the class of so-called $\omega$-automata [143] of which the Büchi automaton is probably the most well-known. Büchi automata have been used in the determination of safety and liveness properties of distributed systems [8], [9]. Given a general MFG specification, involving a family of MFGs with conditions, we define in Section 7.3 the graph of global states with transitions, which is uniquely determined by the MFG specification. From this graph, various different end-state definitions will define various different $\omega$-automata, each of which identifies the set of system traces specified by the MFG with the set of accepted traces of the automaton. The Global-State Transition Graph itself defines a Büchi automaton, namely the one in which the end-states are the set of all states. These automata are similar to ordinary finite automata, except for the acceptance condition. For example, a (possibly infinite) string is accepted by a Büchi automaton just in case the automaton passes through a final state unboundedly often on the string (this definition also works for finite strings, normally turned into infinite strings for this purpose by simply repeating the last item for ever). Even though Büchi automata define a very rich class of trace-sets (in fact, they express a $\Pi^1_1$-complete set), in order to use them flexibly one must be at liberty freely to design the state set. We are constrained by having to use the global states defined in the GSTG, and we show in Section 8.5 that the Büchi acceptance condition does not suffice to define certain natural liveness conditions, given the GSTG states and transitions. Therefore other acceptance conditions may be preferable.

## 4.5 What About Complexity?

A single MFG without conditions obtained from a set of MFGs with conditions by eliminating the conditions (see *unfolding* operation in Section 7.2.4) may be exponential in the size of the specification. Also, the global system state graph generated from a message flow graph may be exponential in the size of the message flow graph. Are these kinds of complexities devastating to a proposed semantics?

We would argue that it depends what the semantics is used for. The point of our semantics for MFGs is to be precise about which traces they describe. To discriminate amongst traces, it is not always necessary to enumerate the global state set. Besides, in general, complexities of this nature should be expected if a description method is to be anything more useful than just a succession of cute pictures. There would be little point in using MFGs if it were just as easy to write down a global system automaton directly - one would simply write down the automaton and be done. Because this automaton is potentially exponentially more complex than a series of MFGs, it makes sense to write down the MFGs as shorthand for the larger object. It should still be analysable without explicit representation. The message flow graph is a structure of comparable size to the reduced state-graph structures used in [59], based on trace theory [115]. However, it is also a more picturesque, and therefore we would argue more intuitively appealing, structure.

## 4.6 Handling Synchronous Communication

In [100] the inter-process communication is both synchronous and asynchronous message passing. MSCs, however, are most frequently interpreted as asynchronous message passing, for example in Z.120. We give some reasons here why wanting to incorporate synchronous communication in our semantics, which, as we observe later, involves only minor modifications to the asynchronous case. Mixtures of synchronous and asynchronous message passing are also handled easily, as shown in Section 7.9.

**Definitions.** MFGs are most useful in an environment in which the sender and receiver of a process are statically determined, as in CSP or OCCAM, but unlike SDL, ADA or Remote Procedure Call. In the communication we consider, there is a single sender and a single receiver for every message type. Thus, all communication is *transparent two-process* in the terminology of [98].

- *Synchronous communication* in this work is an atomic action participated in by two processes; sender and receiver block if one of them is not ready. We translate this atomicity into one atomic global system state transition in our semantics.
- Sender action and receiver action in *asynchronous communication* are separate atomic actions and hence correspond to different global system state transitions. Sender never blocks, while receiver blocks if all sent communications have already been received. We translate the asynchronous communication mechanism into our semantics by treating sending and receiving events as separate atomic global state transitions.

**Reasons for Handling Synchronous Communication.** We provide here some reasons for wanting to include synchronous communication signals into MFGS, and for wanting to interpret this synchronous communication primitive semantically within the framework of our semantics.

1. Arguments given in [68], [119] suggest that effective formal methods are much easier to devise for communication relying on synchronous primitives. Synchronous primitives are useful as an abstraction.

2. Many specification and description languages rely on synchronous communication primitives, for example the process-algebra-based specification languages *CSP* [68], *CCS* [119] and *LOTOS* [78], the CSP implementation language *OCCAM* [75], and the family of so-called *synchronous languages* such as *ESTEREL* [23], *SIGNAL* [22, 21] and *LUSTRE* [20]. For such formal methods to avail themselves of MFG analysis techniques, it is necessary that synchronous primitives be handled in MFGs.

3. Within the context of telecommunications systems conforming to ISO OSI [76], synchronous events naturally occur at the interface between service layers, because the *same* event is looked at in two *different ways* by the two different layers. This point may be expressed another way, that the relation between different levels of abstraction in the design of a communications system can elegantly be expressed by means of synchronous communication primitives. It is closely related to the first point above.

4. There are examples where synchronicity and asynchronicity co-exist in one specification style, so for example in the dialect *ESTELLE** [44, 45] of the specification language *ESTELLE* [77]. In a suggested extension of SDL with synchronous communication primitives [72], it is argued that synchronous communication mechanisms are needed in open systems specifications. For example, it is argued there that synchronous communication is needed to specify a backpressure flow control mechanism, which cannot be described in standard SDL with only asynchronous communication.

5. Synchronous communication lends itself naturally to finite-state interpretation, and problems with liveness assumptions which we note later in Section 8.5 do not highlight inadequacies of the specification method in the same way. Similarly, an MSC with mixed synchronous and asynchronous messages has as few problems with liveness definition as an MSC with purely synchronous messages.

6. As we discussed earlier, MSCs are used in the object-oriented modeling method in [137] to specify inter-object communications. These communications can be either asynchronous or synchronous message passing, another reason to incorporate synchronous communication in our semantics.

In Chapter 7 we will first define a semantics for asynchronous communication in MFGs, which we extend to synchronous communication in Section 7.9.

**Remark concerning Z.120.** Synchronous communication is not yet a feature of MSCs in the ITU standard Z.120. It is beyond the scope of this work to evaluate in detail the arguments for incorporating synchronous communication into Z.120. However, given the arguments above for inclusion, and the relative simplicity of accommodating synchrony, we believe that allowing synchronous signal arrows would give the user of MSCs added flexibility at no extra cost.

## 4.7 Communication Mechanism

**What You See - What You Get.** We think that in a good specification method *'what you see'* should be *'what you get'*, i.e. there should not be any hidden semantic elements (e.g. properties of the environment, states of buffers) determining the functioning of the system. It is a standard requirement for a semantics to be *compositional* on the syntax, i.e. to follow the syntax of a term as it is built from subterms, and to say what the meaning of each subterm is as it is built. A semantics which pays attention to hidden elements that aren't represented syntactically cannot do that.

**Implication on the Communication Mechanism Employed.** Asynchronous communication in a telecommunications system is often thought of as based on (potentially unbounded) buffers or queues in between processes. In MSCs and MFGs there is no such thing like a buffer or queue visible in the specification, and we therefore do not include any concept similar to a buffer or a queue into our semantics. The asynchronous communication we shall define for MFGs only states that a message of some type is *on the way* (sent but not yet received), or *not*.

There is a further argument against the use of queues in our semantics. We will show in Chapter 5 that MFG and MSC specifications are inherently *finite-state*. Now, any device buffering messages for a looping MFG would have to remember a potentially unbounded number of messages, which entails that the system would have an unbounded number of states. This contradicts the inherent finite-stateness property of MFGs. As a consequence, there is no device in our semantics remembering for example *how many* messages of one type have been sent, but not yet received.

# Chapter 5

# Why a Finite-State Semantics?

One reason for wanting a finite-state semantics for MFGs is that finite-state property-checking methods are largely automatic. The major problem is controlling state-explosion when exhaustively checking properties through all possible states. However, this is an argument from convenience. There is a strong argument from the intuitive meaning of MFGs that MFGs are inherently finite-stately, which we provide here. It is based on inquiring what system information is explicit in MFG descriptions, and which information is hidden. We argue that the explicit information available from an MFG allows only finitely many global system control states.

## 5.1 What is the Event 'Connection'?

The connection between events in different processes is exhibited in the MFG in Figure 2.1 by means of dotted arrows, in the MSC by means of horizontal (or inclined) arrows, between different processes. We already introduced the property (*) earlier which describes the relation between `send` and `receive` events syntactically as a unique one-to one relation. What can this symbolic one-to-one connection correspond to in reality?

The connection is between `send` and `receive` events of the same type. Thus, maybe the connection is that *the identical message instance that is sent by the event statement at the arrowtail is received by the event statement at the arrowhead.* However, this is too strong an identity to be useful. Channels, even Ethernet channels, may be lossy. Protocols can try to ensure that if a message of a particular type is sent but not acknowledged, then the contents of the message, along with any message-ID, is regenerated and resent, until successful reception is acknowledged. If message-identity was taken to be message-instance identity, (the actual voltage values raised on the cable), MFGs would be unable to describe higher-level services based on an unreliable underlying protocol. So, rather than this strong identity condition, the connection could represent a successful reception of some uncorrupted message instance with a particular message-ID. This is a reasonable interpretation for the case in which an MSC is used to represent a higher-layer interaction, such as in a service description (e.g. for INRES in [19], or JVTOS in [51]), or in object models [132, 84]. In some sense, therefore, the message-arrow represents the 'same' message sent and received, where 'same message' is an individuation potentially finer than 'message of the same type', and potentially coarser than 'identical message-instance' (although it allows both extremes should they be appropriate to the description at hand). We shall call such a creature a *message occurrence*. Thus, MSCs represent `send`s and `receive`s of individual message occurrences. The sending and reception properties of message occurrences are guaranteed by underlying protocols such as Ethernet protocols. Further, and most importantly for our argument, message occurrences are the finest possible individuation of message objects. This means that it is impossible in principle to tell at the chosen level of description if the sending and reception of a message occurrence is also a sending and reception of one message instance, or of multiple instances (unless message occurrences are in fact message instances in the particular case at hand).

## 5.2 Finiteness of the Number of Message Occurrences

In order to demonstrate the proposition that the number of message occurrences in a given MFG specification is finite, we consider how message occurrences may be individuated. The usual means is by an identifier which is added at message-generation time, such as a timestamp. We shall generically refer to all such individuation codes as timestamps. We know of no timestamping mechanism used in real protocols that allows infinite timestamps. These methods frequently assign timestamps generated by a mechanism formally equivalent to picking numbers in some increasing order from the integers modulo $N$, for $N$ some large integer. Thus, the assignment of timestamps occurs cyclically, and there are at most $N$ different timestamps that may be used. We conclude that MFG descriptions can in principle only individuate finitely many message occurrences.

This conclusion also holds for MSCs, since they are MFGs, showing that MSCs are very different in principle from SDL specifications. In SDL, there are explicit data variables which may take values and be subject to operations that generate an unbounded set of different data values. In principle, one may use a data variable with an infinite range as a timestamp in messages in an SDL specification. There is thus the ability in SDL to distinguish unboundedly many message occurrences. Furthermore, in SDL the communication between processes is explicitly by means of unbounded FIFO queues. It is therefore explicit in the SDL definition that an SDL specification may have unboundedly many states. However, there is nothing in Z.120 or in the thinking about MSCs that would require MSCs to have unboundedly many states in the same way. Giving MSCs the ability to use unboundedly many timestamps would require an addition to the MSC definition - an addition which would be unmotivated by any practical criterion, and irrelevant to the purpose for which MSCs are used.

## 5.3 Timestamps May Be Eliminated

In Figure 2.1, the system generates and processes four message occurrences, and each of these message occurrences has a different type. Therefore, the types may be used to individuate messages. However, in Figure 2.2, an unbounded number of message occurrences is specified, indicated by the condition in the MSC, and the corresponding loop in the MFG obtained by 'joining' the two condition occurrences. Timestamps may be used to individuate the occurrences of messages. Suppose that timestamps modulo $N$ are used. Then, using timestamps, $N$ iterations through the loop of MSC I in Figure 2.2 may be individuated. Suppose we duplicate the loop body $N$ times. (For a formal definition of *loopbody duplication*, see [100].) This corresponds to a (maybe much!) larger MFG, in which there are $N$ message arrows. Thus message-occurrence individuation corresponds to different message-occurrence arrows. For MFGs with conditions but without explicit loops, such as MSCs, loopbody duplication corresponds to a syntactic operation of *MFG composition*, defined formally in Section 7.2.3.3. Although we have invited the reader to consider only a very simple example from Figure 2.2, we assert that this operation can be carried out for all MFGs with cycles.[1] We call this loopbody duplication/composition operation *timestamp-reduction*. Suppose timestamp-reduction has been performed. Different message instances generated by the same arrows in the timestamp-reduced graph cannot be individuated. Thus, the timestamps no longer individuate message occurrences finer than the different edges in the timestamp-reduced MFG. They individuate precisely the different message arrows in the timestamp-reduced MFG and we may thus remove them. The timestamp-reduced MFG may be much larger than the original MFG. However, it does not have timestamps. For purposes of semantics it is not necessary to pay attention to the increase in size that actually performing timestamp-reduction entails, just as it is unnecessary when giving a finite-state semantics for any system actually to write out the entire state graph. It is only important that *in principle* this reduction may be carried out.

We have shown how timestamps may be eliminated. The reduction yields an MFG in which message occurrences are individuated by the actual arrows that indicate them. In the reduction, two message instances generated by the same arrow cannot be individuated. Thus, we can identify the arrows in such a reduction with the message occurrences in the MFG. We shall assume from now on that all MFGs have been timestamp-reduced in this way.

[1] Writing out the details would be tedious and mathematically unilluminating, since the operation is quite simple.

## 5.4 There are Global States.

We assume that a system described by an MFG has a well-defined set of global partial states with respect to message sending and reception. Note that this does not require that global (total) states of the system are well-defined, but only that at any point of time control in each process is located between or at specific message-passing statements or events. Since each process $P_i, 0 \leq i \leq n$ is defined by a finite amount of code (i.e. a finite *ne*-component – c.f. Figure 2.1), there are a finite number of statements defining message-passing events $e_{i_1}, \ldots, e_{i_{n_i}}$ in $P$. These statements in $P$ correspond with nodes in an MFG. Control in $P_i$ always lies between or at one of these $e_m$. Thus we require that at any point, for any $P_i$, the Boolean value of the state predicate

$$Last(e)_{P_i} \triangleq \textit{the last message event that occurred in } P_i \textit{ was one corresponding to node } e.$$

is well-defined, where $e$ is one of the $e_{i_k}$. This entails there is a well-defined vector $\langle e_{p_1}, \ldots, e_{p_n} \rangle$ of the *next message event* for all processes. To handle the startup case, we also include a predicate

$$Last(\mathtt{start})_{P_i} \triangleq \textit{no message event has yet occurred in } P_i.$$

When interpreting MFGs, the only thing we care about concerning the control state of each process $P_i$ is the values of the state predicates $Last(e)_{P_i}$ (more exactly, the precise one of them which is true at that point).

## 5.5 The Different States Engendered by a Message Occurrence

We assume that MFG descriptions have been timestamp-reduced. Given the ontology of events, there are three state predicates that a system may satisfy with respect to a given message occurrence $m$: no `send` or `receive` of a message occurrence $m$ has occurred; a message occurrence $m$ has been sent but not received; a message occurrence $m$ has been sent and received.

The timestamp-reduction yields an MFG in which one instance of $m$ may not be discriminated from another. It follows that these three state predicates are mutually exclusive, and thus the system may be in one of precisely three states with respect to every message occurrence - precisely one of the three predicates above is true in any given system state, for each message occurrence (message arrow) $m$.

## 5.6 Finiteness and Uniqueness of the Global State Transition Graph

The state predicates of a given state of the MFG are therefore the predicates $Last(e)_{P_i}$ (precisely one of which has the value *true* at any time) indicating the position of the program counter of each process, and, for each message occurrence $m$, the truth values of the three state predicates above (precisely one of which has the value *true* at any time).[2] The potential global states of the system therefore consist of consistent assignments of truth values to these state predicates. There are precisely as many of them as there are nodes in the MFG plus the number of message arrows times three. This number is finite. We have thus shown that there are only finitely many global states of the MFG.

Since there is a finite collection of global states, it remains to determine the state transition function in order to obtain the *global state transition graph* (GSTG) that represents the traces consistent with the MFG description. The nodes of the GSTG are the states. State transitions may be represented by edges between pairs of states. State transitions are caused by events (nodes of the MFG), thus an edge of the GSTG may be labeled with the event triggering the transition. Every event causes a change in true value of precisely two predicates of the form $Last(e)_{P_i}$, and a change in truth value of precisely two message-occurrence predicates.

## 5.7 A General Argument for Finite-Stateness in Telecommunications

Although it is strictly unnecessary for the argument here, which concerns the finite-state behavior of MFGs, there is a general argument for requiring semantics of message-passing in any real telecommunications protocol or service to be finite-state, even those specified by SDL which in principle may utilise unboundedly many timestamps. We present that argument here.

In a protocol or service definition, each individual process control is usually a finite-state device with respect to `send`s and `receive`s. The unboundedly many states are usually attributed to the unboundedly many states a true asynchronous channel may have. But consider now system recovery from faults. Irrespective of its size, a finite state device can only remember a bounded computation history. Suppose, as the system runs, communication channels are compromised (someone cuts a cable). Also assume the processes themselves are not compromised. A consistent state must be reconstructed. Each process must be asked its state, namely where it thinks its control is, and what it

[2] These predicates are not independent Boolean variables, because which message occurrences have been sent and received of course depends on the position of the program counters.

remembers from what it has done. No other information may be assumed to be available. The system itself can have been operating for a very long time, much longer than the bounded memory of any single process. What can be reconstructed from the memories of the processes is bounded, no matter how long the system has been running previous to the fault. Hence, two such failures which result in the same local states to the processes are equivalent from the point of view of the potentially knowable state of the system. So each such equivalence class can be identified with a *global state* of the system. Since there are finitely many finite-state processes, the global states are some equivalence (probably the identity) relation on a subset of the cartesian product of the state spaces of the individual processes, and thus there are only finitely many global states. A conservative upper bound to the number of these states is the size of this cartesian product.

It is often suggested that asynchronous communication is equivalent to the presence of lossless queues of unbounded capacity on each channel, e.g. in SDL. It is well known that in theory queues may be configured to contain the entire system history information, which is finite at any one point, but unbounded through the history of the system. By the argument above, since the number of practically distinguishable global states is bounded, the contents of these theoretical queues cannot be part of the system, in general, and properties of 'queue' contents may only be inferred from the processes that generated and received those contents – and that information is bounded, as we have noted.

# Chapter 6

# Requirements for MSC Supporting Tools

## 6.1 Overview

Amongst companies developing or supplying MSC tools are Siemens AG (ZFE Division in München, Germany), Verilog (Toulouse, France), Telelogic (Malmö, Sweden), and ObjecTime (Ottawa, Canada), and AT&T Bell Labs (Naperville, Illinois, USA). Verilog's GEODE SDL tool, Telelogic's SDT tool and the ObjecTime tool are sold commercially. The GEODE and SDT tools both support the editing of MSCs and the interactive execution of MSC specifications. Both tools are typical telecommunications systems engineering tools centered around SDL specifications. As discussed earlier, MFGs have particular importance in Object Models. The ObjecTime tool supports the use of MSCs as run-time monitors for the behaviour of actors.

The non-commercial test case generation tool SAMSTAG, developed at the University of Berne, uses MSCs for the description of test cases. SDL specification of the system under test are simulated by message exchanges described as MSCs, and based on this simulation the tool generates TTCN test cases (see [63]). The MSC simulator part of this tool is based on the semantics described here.

We developed our requirements for the MFG semantics in Chapter 4. Three of these requirements, that there are global states, that traces are interleavings, and that the semantics is finite-state, are also requirements of the Verilog GEODE toolset. We therefore describe in more detail how MSCs are used in GEODE, and what the semantics assumptions inside GEODE are, in the next Section.

## 6.2 Requirements on the GEODE Toolset.

To further motivate our requirements, we describe briefly the role of MSCs in the GEODE toolset of Verilog. GEODE contains MSC tools, which are being enhanced within the AVALON project [6]. MSCs are used as a special kind of observer dealing with signal sequences, and they are used to specify parts of traces which may or may not be available under a given SDL specification. One may compare traces defined by MSCs with SDL specifications in GEODE in a variety of ways. The reader may find it useful to refer to the MSC in Figure 2.1 during the discussion.

GEODE employs two styles of interpretation, the *local ordering*, which only considers ordering of events relative to a given process, and consists in reading the event ordering off each vertical process axis independently, and the *global ordering*, in which an event occurs before another if and only if that event occurs graphically higher up in the entire MSC diagram. In other words, it's as if there is a global clock and the vertical process axes are all calibrated according to that global clock. Thus a given MSC defines a unique trace according to the global ordering.

Our MSC semantics produces a finite-state automaton from what [6] call the *causal ordering*. The causal ordering is favored by the Z.120 standard [33]. GEODE does not implement the causal ordering, because under this ordering MSCs "cannot be formalized easily as automata. This makes it difficult to use causal ordering [along with other GEODE methods]" [6]. Our work here formulates the causal ordering using finite state automata, incidentally providing a theoretical solution to this problem of integration within GEODE.

Other operators on MSCs are available in GEODE, and are being extended in AVALON, such as *sequencing* (our composition), *exclusion*, *exception* and *loops* (which appear in our MFGs). These operators on MSCs are transformed into operators on the FSMs that interpret the global or local semantics, so that a global MSC may be obtained. In general, this global MSC is non-deterministic, and is transformed into a deterministic machine for the validation tools [5].

There still remain open questions about the semantics of MSCs as used in GEODE. What is the interpretation of an MSC if both the `send` and the `receive` events are located at the same height in the MSC diagram? And what is the interpretation if the `send` event occurs further down than the `receive` event to which it is related by the message arrow? These open questions may have found a pragmatic answer inside the GEODE tool set, however we point out the necessity for a more general and unambiguous resolution, which our semantics provides[1].

[1] In particular our representation of the graphical *MSC* as an algebraic object, the MFG, avoids similar ambiguities. We define the *ne* and *sig* relations to imply an ordering of `send` and `receive` events which is independent of their location in the MSC diagram.

# Chapter 7

# The Semantics of Message Flow Graphs

We now develop the semantics for MFGs formally. As discussed before (see Chapters 4 and 5) our main requirements are firstly, to determine unambiguously exactly what execution traces are specified by an MFG, and secondly, to use a finite-state interpretation. As argued for above, our methods will function for both asynchronous and synchronous communications.

From a set of MFGs, we define a transition system of global states, and from that a Büchi automaton by considering safety and liveness properties of the system. In order easily to describe liveness properties, we interpret the traces of the transition system as a model of Manna-Pnueli temporal logic. Finally, we describe the expressive power of MFGs by mimicking an arbitrary Büchi automaton by means of a set of MFGs.

## 7.1 Overview

We introduced MFGs informally in Chapter 2. Section 7.2 formalises the notion of MFGs. In Section 7.3 we obtain a *global state transition graph* (GSTG) from an MFG. A GSTG is like a finite-state automaton but lacks definition of end-states. Section 7.4 formalises the GSTG notions. We consider end-state definitions in Section 7.5. Each possible end-state definition gives a Büchi automaton. The MFGs under-define the resulting automaton, in that end-state definitions are related to different liveness properties not explicit in the MFGs. We show in Section 7.6 how these may be made explicit via a connection with temporal logic, in which these properties may be formulated. In Section 7.7 we describe this connection formally. We discuss some properties expressed in temporal logic which all MFGs satisfy, and some potentially desirable ones which some MFGs might be required to satisfy in some uses, in Section 7.8. We then show in Section 7.9 how synchronous communication can be accommodated along with asynchronous communication in MFGs.

We also note that the occurrence of synchronous communication in an MFG can simplify the liveness analysis. Finally, in Section 7.10 we show how to simulate an arbitrary Büchi automaton with MFGs. For a definition of the notation see Appendix A.

## 7.2 Formal Definition of MFGs

In Chapter 2 we informally described MFGs to be graphs where the nodes represent communication events and edges represent either process control (*ne* relation) or message flow (*sig* relation). This Section provides the formal definition of MFGs.

### 7.2.1 Message Flow Graphs Formally

Let $S, C$ and $X$ denote arbitrary pairwise disjoint sets, the elements of which we call *sending* events, *receiving* events and *extra* nodes. Furthermore, let $ST$ and $ET$ denote arbitrary disjoint sets (also disjoint from $S, C$ and $X$), whose elements we call *signal* and *event* types. We define a *Message Flow Graph* as a tuple

$$\mathcal{G} = (S, C, X, ne, sig, ST, stype, ET, etype, Top, Bottom)$$

where $(S \cup C \cup X, ne, etype, ET)$ is a digraph with node[1] labels and $(S \cup C, sig, stype, ST)$ is a digraph with edge labels satisfying the following conditions:

1. $sig \subseteq S \times C$ is a (necessarily bipartite) bijective relation, where $S = domain(sig)$ and $C = range(sig)$ ($\mathcal{G}$ satisfies the property (*), see Section 2.4);

2. The set $ET = (\{!, ?\} \times ST) \cup \{Top, Bottom\}$ contains the *event types* (we write $!t$ for $(!, t)$ and $?t$ for $(?, t)$).

3. If the type of a signal is $t$, then the corresponding send and receive events are of type $!t$ and $?t$ respectively:

$$(a, b) \in sig \rightarrow (\exists t \in ST)(stype((a, b)) = t \ \wedge \ etype(a) = !t \wedge etype(b) = ?t);$$

4. Every component of the $ne$ relation graph contains at most one start event:

$$(e, e' \notin range(ne) \wedge (e, e') \in ne^*) \rightarrow (e = e').$$

[1] We have remarked previously that we may define a Message Flow Graph with either *sig* labels or node labels, and that either may be useful. For convenience, we define an MFG formally with both, along with a coherence condition saying that the *sig* label had better say what the corresponding node labels say, and *vice versa*.

A *Basic MFG* (bMFG) is an MFG which satisfies the following additional condition: Start nodes (defined to be nodes in the set $\{e \in X \mid (ne \rhd \{e\}) = \emptyset\}$) are of type *Top* and finish nodes (nodes in the set $\{e \in X \mid (\{e\} \lhd ne) = \emptyset\}$) of type *Bottom*:

$$e \notin range(ne) \leftrightarrow etype(e) = Top \textit{ and } e \notin domain(ne) \leftrightarrow etype(e) = Bottom.$$

A Basic MFG *with predicate labels* (pbMFG) has in addition a functional relation $predlab \subset ne \times PS$, where $PS$ is a set disjoint from all the others, interpreted as a set of predicate symbols. We will note in Section 8.2 how pbMFGs are used to interpret communicating systems with control branching.

#### 7.2.1.1 Process Type

A process is defined as a *connected component* of the *ne* relation. Since every component contains only one start node, we could define the set $PT$ of all process types to consist of all start nodes, i.e.

$$ptype(a) = e \textit{ iff } a \in range(\{e\} \lhd ne^+) \ \wedge \ e \notin range(ne).$$

However, we shall later wish to identify processes across different cMFGs when we define cMFG composition, so we specify only that $ptype \subseteq S \cup C \cup X \times PT$ is a functional relation relating every node of the MFG to its process type, and the set of process types $PT$ is disjoint from every other set in sight.

#### 7.2.1.2 Simple MFGs

Let $E \triangleq S \cup C \cup X$ the set of *all* events. An MFG is *simple* (an sMFG) if the following conditions are satisfied:

- $(\forall a \in E)(\mid domain(\{a\} \lhd ne) \mid = 1)$ (there is no branching in the *ne* relation)
- $ne^+ \cap id_E = \emptyset$ (there are no cycles in the *ne* relation),
- $(\forall (a, b) \in sig)(ptype(a) \neq ptype(b))$ (there is no self-sending),
- $(\forall e \in T)(range(\{e\} \lhd ne^+) = range(\{e\} \lhd ne^*))$ (all elements in some component are *reachable* from the start node), and
- $(\forall x \in ST)(\mid range(field(domain(stype \rhd \{x\})) \lhd ptype) \mid \leq 2)$ (for any signal type, there is a *unique sender* and a *unique receiver* process).

Note that sMFGs may not be basic, since they may include nodes, such as condition nodes, that start or finish the MFG, but are not *Top* or *Bottom* nodes. The three MFGs with conditions in Figure 2.7 are not bMFGs in that they do not start with start nodes or finish with finish nodes, however they are simple, and may easily be obtained from Message Sequence Charts describing the scenario which motivated the example.

Basic MFGs with or without predicates are for us the major descriptive objects. However, we have also noted the need, when interpreting Message Sequence Charts and also when analysing parallel code, for MFGs with condition nodes, which may be composed to form pbMFGs. Figure 2.7 showed three MFGs with conditions, in which the control branching was shown as two separate MFGs, which may be composed to form the pbMFG in Figure 2.8. We define MFGs with conditions below. Simplicity arises from purely practical considerations. In most examples we have seen in which it was necessary to compose MFGs, the MFGs are simple. Certainly, Message Sequence Charts and Time Sequence Diagrams yield simple MFGs. So we have defined simplicity here, and the reader may like to consider our further constructions under the assumption of simple MFG arguments, although the constructions also take non-simple arguments.

### 7.2.2 Formal Mapping of Basic MSCs to Basic MFGs

Earlier (see Section 2.2) we defined informally what it means to map a basic MSC on a corresponding simple MFG structure. We will now formalize this mapping. Given an simple MSC in graphical form (see for example Figure 2.1) we define a set of *sending* events $S$ of which each element corresponds to a `message output` symbol and a set of *consuming* events $C$ of which every element corresponds to a `message input` symbol. We call the arrow connecting a `message input` and a `message output` symbol a `message` symbol. For simple MSCs, the corresponding simple MFG is so close to the MSC that it may be regarded as just syntactic sugar. So we shall identify an sMSC with its MFG by identifying elements of $S$ and $C$ with their graphical MSC representation if they correspond in the above sense. Let $ne \subseteq (S \cup C) \times (S \cup C)$ denote a *next-event* relation and let $sig \subseteq S \times C$ denote a *signal* relation such that $(x, y) \in ne$ iff $y$ is a direct successor of $x$ on some instance axis, and $(v, w) \in sig$ iff $v$ and $w$ are connected by a `message` symbol.

### 7.2.3 MFGs with Conditions

We define MFGs *with conditions* (cMFGs). Generally, it is only necessary to consider cMFGs that are also simple, but we do not assume simplicity in the definition. We introduce a set $I$ of *condition nodes*, which are elements of $X$. Condition nodes intuitively correspond to particular segments of instance axes in between directly connected `message` symbols (namely, to those segments intersected by a condition node in the graphical representation).

**Definitions.** An *MFG with conditions* (cMFG) is a labeled digraph

$$M = (S, C, X, ne, sig, ST, stype, ET, etype, Top, Bottom, CL, cond)$$

where

- $M' = (S, C, X, ne, sig, ST, stype, ET, etype, Top, Bottom)$ is an MFG;
- $X = T \cup B \cup I$ with $T, B$ and $I$ pairwise disjoint (we call elements of $T$ *top nodes*, elements of $B$ *bottom nodes*, and elements of $I$ are *condition nodes*);
- $(\forall x \in T)(etype(x) = Top)$, $(\forall x \in B)(etype(x) = Bottom)$, and $(\forall x \in I)(etype(x) = \emptyset)$; (the event type of top nodes is $Top$ and that of bottom nodes is $Bottom$; condition nodes have no event type);
- $ne \subseteq ((S \cup C \cup I \cup T) \times (S \cup C)) \cup ((S \cup C) \times (S \cup C \cup I \cup B))$ (start nodes may only be condition nodes or $Top$ nodes, finish nodes may only be condition nodes or $Bottom$ nodes, and condition nodes may only be start or finish nodes);
- $CL$ is pairwise disjoint from any other set defined, and $cond \subseteq I \times CL$ is a functional relation: elements of $CL$ are called *condition labels* and *cond* the *condition labeling*;
- $(\forall l \in CL)(\mid domain(cond \rhd \{l\}) \mid = \mid range(domain(cond \rhd \{l\})) \lhd ptype \mid)$ (every condition node belonging to a given condition belongs to a different process).

We define a *condition* to be a set $C$ such that for some $q \in CL$, $C = \{c \in I \mid cond(c) = q\})$. The *set of all conditions* of a cMFG $M$ is *conditions(M)*.

#### 7.2.3.1 Types of Conditions

- A condition $C$ of some cMFG $M_s$ is *global* with respect to some set of MFGs $\mathcal{M}$ iff the set of all process types of $\mathcal{M}$ is equal to the set of process types of the condition nodes of $C$:
$$\bigcup_{i=1\ldots n} PT_i = domain(C \lhd ptype)$$
- A condition $C$ of some MFG $M$ is *initial* iff all its predecessor nodes in the $ne$ relation are top nodes:
$$range((domain(ne \rhd C)) \lhd etype) \ = \ \{Top\}$$
- A condition $C$ of some MFG $M$ is *final* iff all its successor nodes in the $ne$ relation are bottom nodes:
$$range((range(C \lhd ne)) \lhd etype) \ = \ \{Bottom\}$$

A cMFG may have only initial and final conditions, by definition, but conditions may or may not be global.

#### 7.2.3.2 Continuations

Given an MFG $M$, define $init_M \triangleq \{(a,b) \mid (a,b) \in ne_M \ \wedge \ ne_M \rhd \{a\} = \emptyset\}$, and $final_M \triangleq \{(a,b) \mid (a,b) \in ne_M \ \wedge \ \{b\} \lhd ne_M = \emptyset\}$. Let $\mathcal{M}$ be a set of cMFGs, $M_1, M_2 \in \mathcal{M}$, $C_1$ a condition in $M_1$ and $C_2$ a condition in $M_2$, with $cond(x) = c_i$ for every $x \in C_i$. $C_2$ is a *continuation* of $C_1$ ($cont(C_1, C_2)$) iff

- $c_1 = c_2$ (the labels are identical);
- $global(C_1) \wedge global(C_2)$ (both conditions are global);
- $(\forall x \in C_1)(x \in range(final_{M_1})) \ \wedge \ (\forall x \in C_2)(x \in range(init_{M_2}))$ ($C_1$ is a final condition and $C_2$ is an initial condition).

We shall restrict ourselves to composition of cMFGs via global initial or final conditions.

#### 7.2.3.3 Composition

The *composition* of cMFGs is the 'gluing together' of cMFGs at common conditions, i.e. where one is a continuation of the other. During this process, some condition nodes are removed. We also define the *composition graph* of a set of cMFGs.

Let $\mathcal{M}$ be a set of cMFGs, $M_1, M_2 \in \mathcal{M}$, and suppose the event sets of both cMFGs are disjoint, i.e. $S_1 \cap S_2 = \emptyset$, $C_1 \cap C_2 = \emptyset$. The *composition* of $M_1$ and $M_2$ is the cMFG $M' = (S', C', X', ne', sig', ST', stype', ET', etype', Top, Bottom, CL', cond')$, $M' \triangleq M_1 \oplus M_2$, iff

- $(\exists C \in conditions(M_1)\ \exists D \in conditions(M_2))\ (cont(C, D))$ (there is a condition in $M_2$ continuing a condition in $M_1$),
- $S' = S_1 \cup S_2$, $C' = C_1 \cup C_2$ (the event sets are unioned),
- $X' = I_1 \cup T_1 \cup B_2 \cup I_2 - C - D$ (start nodes of $M_2$, which form condition $D$, and finish nodes of $M_1$, which form condition $C$, are eliminated);
- $ne' =$

$$(ne_1 - (ne_1 \rhd domain(C \lhd cond_1) - (ne_1 \rhd B_1))$$
$$\cup\ (ne_2 - ((domain(D \lhd cond_2)) \lhd ne_2) - (T_2 \lhd ne_2)))$$
$$\cup\ \{(a,b) \mid ptype(a) = ptype(b) \wedge a \in domain(ne_1 \rhd (domain(C \lhd cond_1)))$$
$$\wedge\ b \in range((domain(D \lhd cond_2)) \rhd ne_2)\},$$

  (the new $ne$ relation is obtained as the union of the old $ne$ relations minus those pairs which have the connecting condition nodes in their range or domain and minus those pairs which connect these condition nodes with top and bottom nodes; we then add new $ne$ edges to connect $M_1$ and $M_2$)

- $sig' = sig_1 \cup sig_2$, $ST' = ST_1 \cup ST_2$, $stype' = stype_1 \cup stype_2$,
- $ET' = (\{!, ?\} \times (ST_1 \cup ST_2)) \cup \{Top, Bottom\} - ((B_1 \lhd etype_1) \cup (T_2 \lhd etype_2))$, $etype' = etype_1 \cup etype_2$,
- $CL' = (CL_1 - range(C \lhd cond_1)) \cup (CL_2 - range(D \lhd cond_2))$, and $cond' = (cond_1 - (C \lhd cond_1)) \cup (cond_2 - (D \lhd cond_2))$.

Let $\mathcal{M}$ be a set of cMFGs. We define the *composition* relation $comp \subseteq \mathcal{M} \times \mathcal{M}$ such that $comp \triangleq \{(M_i, M_j) \mid M_i, M_j \in \mathcal{M} \ \wedge \ M_i \oplus M_j \textit{is defined}\}$. From this we derive the *composition graph* $\mathcal{C} = (\mathcal{M}, comp)$ ($\mathcal{C}$ is a digraph whose nodes are individual cMFGs, and whose edges lead from a cMFG to its continuations).

### 7.2.4 Unfolding of MFG Specifications

Composition is defined between two cMFGs only. We obtain a pbMFG such as in Figure 2.8 from the cMFGs in Figure 2.7 by making all compositions possible from the cMFGs. The composition of cMFGs according to a composition graph yields a single graph, paths through which correspond to system traces. However, infinite traces could only be obtained in this manner from cMFGs (which specify a finite number of signals each) by infinite composition. We need a *finite* representation which contains the same information about traces as the 'infinite composition'. 'Infinite composition' may only happen from a set of MFGs for which there is a loop in the composition graph. If we 'fill in' the composition graph by 'plugging in' the cMFGs in the appropriate places, we obtain the desired finite structure. So, we define the *unfolding* operation on a set of MFGs which composes a cMSC with all possible successors, intuitively by taking the composition graph and 'plugging in' each actual cMFG (without its initial and terminal condition nodes) in the appropriate place. The result of this operation is a pbMFG with branching and cycles, and provides us with a single finite structure, a pbMFG, corresponding to the original set of cMFGs.

Let $\mathcal{M}$ be a set of cMFGs and let $\mathcal{C}$ denote the corresponding composition graph. We define the MFG $N_{\mathcal{M}} = (S, C, X, ne, sig, ST, stype, Top, Bottom)$ as the *unfolding* of $\mathcal{M}$ iff

- $S = \bigcup_{i=1,\ldots,n} S_i$, $C = \bigcup_{i=1,\ldots,n} C_i$,
  $X = \bigcup_{i=1,\ldots,n} X_i - \{C \in conditions(\mathcal{M}) \mid (\exists M_i, M_j \in \mathcal{M})\,(cont(M_i, M_j) \ \wedge \ C \in (final_{M_i} \cap init_{M_j}))\}$,
- $ne =$

$$\left(\bigcup \{ne_i \ \cup \ ne_j \mid (M_i, M_j) \in comp\}\right)$$
$$-\left(\bigcup_{i=1,\ldots,n} ne_i \rhd (domain(cond_i \rhd CL_i))\right)$$

$$- \left( \bigcup_{i=1,\ldots,n} (domain(cond_i \triangleright CL_i)) \triangleleft ne_i \right)$$
$$- \left( \bigcup_{i=1,\ldots,n} \{T_i \mid M_i \in range(comp)\} \triangleleft ne_i \right)$$
$$- \left( \bigcup_{i=1,\ldots,n} ne_i \triangleright \{B_i \mid M_i \in domain(comp)\} \right)$$
$$\cup \{(a,b) \mid ptype(a) = ptype(b)$$
$$\wedge (\exists C \in conditions(M_i), D \in conditions(M_j))\, cont(C,D)$$
$$\wedge (\exists c,d)(c \in (domain(C \triangleleft cond_i)) \wedge (a,c) \in ne_i$$
$$\wedge\, d \in (domain(D \triangleleft cond_j)) \wedge (d,b) \in ne_j)\},$$

(The $ne$ relation is obtained by a union of all the component $ne$ relations, minus all condition nodes, minus all $ne$ pairs which contain top and bottom nodes over which a composition is performed, plus all those event pairs which need to be connected as a result of the composition of two cMSCs.)

- $sig = \bigcup_{i=1,\ldots,n} sig_i$, $ST = \bigcup_{i=1,\ldots,n} ST_i$, $stype = \bigcup_{i=1,\ldots,n} stype_i$.

## 7.3 From MFGs to Global State Transition Graphs

Using unfolding, we may represent a set of cMFGs by a single pbMFG. The set of cMFGs we start with will have come from an attempt to describe the message-passing features of some system of communicating processes, and we wish to obtain a pbMFG as a description of this system. Accordingly, we call any set of MFGs whose unfolding yields a pbMFG, an *MFG specification.* Use of the word 'specification' should not be taken to suggest that we are advocating sets of cMFGs as a specification method. Sets of Message Sequence Charts and analyses of parallel code yield sets of cMFGs, so MFG specifications are the starting point for our semantic interpretation. We have shown already how to use unfolding to yield a single pbMFG. In order to obtain a finite-state automaton from such a pbMFG, we have to define the global states, the start state, and the state transition function, which we do in this Section. This triple defines the global state transition graph (GSTG), and is uniquely determined by the initial set of cMFGs. To make an automaton from the GSTG, we need further to define the set of final states, which will depend on a later discussion of liveness properties. We require that there must be a finite number of global states.

### 7.3.1 Obtaining the Global States, the Start State, and the Transition Relation

The *global state* of an MFG is determined by the local state of each of the processes, and by the "state" of each of the messages. We say that global states are certain sets of edges

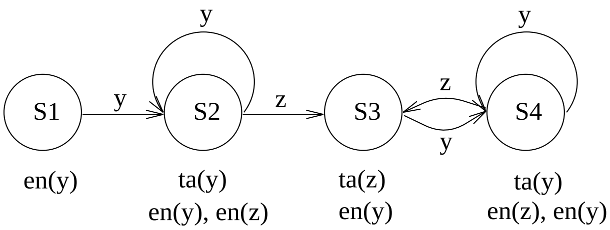


Figure 7.1: Global State Transition Graph for MFG I

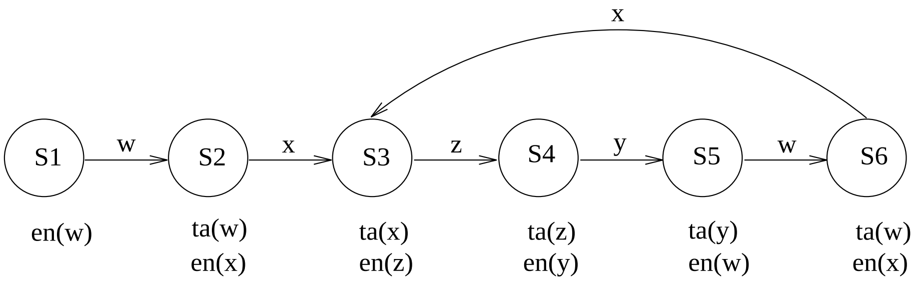


Figure 7.2: Global State Transition Graph for MFG II

of the MFG, and the transition relation between states is obtained by deleting particular edges from the state and adding others. The *ne* edges occurring in a state may be thought of as the set of positions where *control* lies in each process (the 'program counter'), and *sig* edges occurring in the state may be thought of as *signals sent but not yet received.* The *start state* $q_0$ is simply the set of edges leading from *Top* nodes in the graph. In MFG II (Figure 2.3), the set $G = \{(w, y), (z, x), < z, y >\}$ may[2] describe a potential global system state which indicates that the left process has last sent a message `a` and will next receive a message `c`, that the right process has last sent a message `b` and will next receive a message `a`, and that a message `b` is sent but not yet received.

**Note.** The set representation of our global system states can easily be transformed into a state predicate representation. Consider control state predicates *before(x)* and *after(x)* which denote whether the process control is pointing to statement $x$, or that $x$ has just been executed. Let *sent*$(< a, b >)$ denote a predicate which indicates whether message $< a, b >$ has been sent and not yet received. $(v, w) \in G$ can then be interpreted logically as *after(v)* $\wedge$ *before(w)*, and $< z, y > \in G$ as *sent*$(< z, y >)$.

**Example: Derivation of a GSTG.** We shall walk through the derivation of the GSTG for MFG II (Figure 2.3), given in Figure 7.2, to illustrate states and the *transition relation* between states. The labels we use in Figure 7.2 are those given to the nodes in Figure 2.3, so we can illustrate the *ta* and *en* predicates (one would normally just label transitions

[2] We shall later define which global systems states are actually *reachable*.

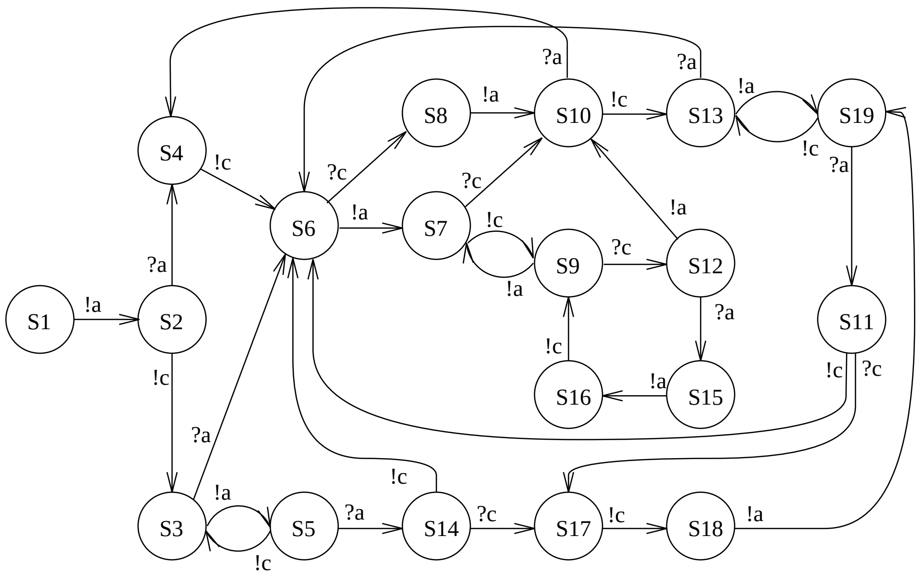


Figure 7.3: Global State Transition Graph for MFG III

with the message types, as in Figure 7.3). The start state is

$$q_0 = \{(u,w),(v,x)\}.$$

In state $q_0$ (labeled $S1$ in Figure 7.2) the event of type $!a$ at node $w$ is *enabled*, because node $w$ represents a `send` node (a `send` node $p$ is enabled in a state $S$ if there is an $ne$ edge with $p$ as second coordinate in $S$). Node $x$ is not enabled, because the `send` corresponding to it has not been taken in $S1$. Since $w$ is enabled, the event corresponding to it may be *taken*, i.e. executed, next to give a new state $S2$. The triple $\langle S1, w, S2\rangle$ will thus be a member of the *transition relation*. (Conversely, if $\langle S1, w, S2\rangle$ is in the transition relation, then $w$ is enabled in state $S1$.) The new state $S2$ is obtained by omitting the $ne$-edge $(u,w)$, and adding the $ne$-edge $(w,y)$ to the state (to represent the change in location of the 'program counter' of the first process), and adding the $sig$ edge $\langle w,x\rangle$ to represent the $a$ signal sent but not received. Thus

$$S2 = \{(v,x),(w,y),\langle w,x\rangle\}.$$

In $S2$, node $x$ is enabled, since it is a `receive` node and requires not only that its 'program counter' be at the right position (i.e. an $ne$ edge with $x$ as second coordinate is in the state), but that the signal has been sent (i.e. a $sig$ edge with $x$ as second coordinate is also in the state). When the action corresponding to node $x$ is *taken*, the edges $\langle w,x\rangle$ and $(v,x)$ are removed from the state $S2$, and $(x,z)$ is added to represent advance of the program counter. The resulting state is

$$S3 = \{(w,y),(x,z)\}.$$

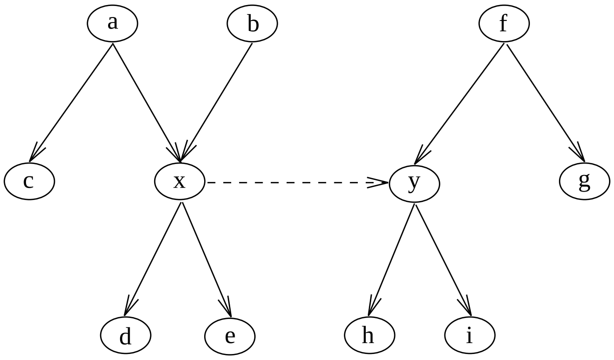


Figure 7.4: Part of an MFG with asynchronous communication

$\langle S2, x, S3\rangle$ is in the transition relation. Node $z$ is enabled in $S3$, and so on. The GSTG in Figure 7.2 is annotated with the list of actions enabled (*en()*) and taken(*ta()*) in each state.

**Consequences of Finite-Stateness.** Figure 7.1 shows the GSTG for MFG I (Figure 2.2). It should be noted that as a result of our finite-state requirement, which inhibits the use of signal queues, no history information on how many messages of one type have been sent is carried along the computation. Consequently, a single `receive` may disable repeated `send`s of one type, as it can be seen in the GSTG for MSC I where node $z$ is not enabled in $S3$. Furthermore, as we will argue in Section 8.4, MSCs I and IV (see Figure 2.5) are semantically distinct.

### 7.3.2 Enabling and State Transitions for Branching MFGs

In the previous example we showed how the system transits from one global system state to a successor state in case of a non-branching MFG. To illustrate the concept of *enabling* for branching MFGs, we walk through the partial MFG in Figure 7.4.

Assume that the graph is a part of a larger pbMFG, and that $c$ and $g$ are `send` events. All arrows in the chart belong to the $ne$ relation except for the pair $< x, y >$ which belongs to the $sig$ relation. As before, a global system state (gss) is a set of edges of the MFG. Consider a global system state

$$G = \{(a, c), (a, x), (f, y), (f, g)\}.$$

`Send` events $c, x$ and $g$ are enabled as the necessary (and sufficient) condition for enabling of send events, namely that at least one of their incoming ne-edges is in the current state, is satisfied. We will focus on the occurrence of event $x$. As it executes event $x$ the system will advance the 'program counter' by omitting the edge $(a, x)$ from $G$ and adding all outgoing $ne$ edges of node $x$, in this case $(x, d)$ and $(x, e)$, to the successor state $G'$. Thereby both potential successor events $d$ and $e$ are potentially enabled, depending only on whether they are `send` or `receive` events. However, one has to do a bit more, removing also the possibility of choosing the enabled actions represented by $c$ and $g$, which represent choice alternatives to the occurrence of the $x$ event. Hence the edges $(a, c)$ and $(f, g)$ are removed from $G$ as we transit via $x$ to $G'$. Finally, one has to represent that the sending event $x$ leaves a message in transit, and thus the *sig* edge $< x, y >$ is added to $G$ to form the successor state $G'$. Hence

$$G' = \{(x, d), (x, e), < x, y >, (f, y), (f, g)\}.$$

We define the transition relation formally in Section 7.4.

### 7.3.3 GSTGs can be Complicated.

It should be no surprise that GSTGs can rapidly become very complicated, for example the GSTG for MFG III in Figure 2.3 has nineteen states (Figure 7.3). This is partly due to the asynchronous communication, and partly to interleavings of non-related events. MFG II and MFG III are similar, differing only in that the second message goes in opposite directions. In MFG II this forces a unique execution sequence, and the GSTG is correspondingly simple (Figure 7.2). However, in MFG III, the two `send`s might occur before either `receive`, or alternatively `send`s and `receive`s might be interleaved. Thus the GSTG is more complex. However, it is not our intention to recommend explicit construction of the GSTG for every MFG, for the usual state-explosion reasons advanced in [74]. We use it later formally to relate liveness and safety properties as expressed in temporal logic or by Büchi automata to MFGs.

## 7.4 Formal Definition of GSTGs

This Section provides the technical definition of Global State Transition Graphs, and may be skipped on a first reading. We define the notions of a *global system state*, of *enabling* a set of events in a global system state, and finally the *global state transition graph*.

### 7.4.1 Enabling

A *potential system state* (pss) $G \subseteq ne \cup sig$ is any subset of the union of the *ne* and *sig* edges of the pbMFG. It is useful to define state transitions for pss's. An (actual) global system state (gss) will later be defined as a pss reached by taking the transitive closure of the transition relation from the *start* state (the set of all start nodes of the processes). Definition of a gss therefore waits upon definition of the transition relation.

Let $V \subseteq S \cup C$ denote a set of events and let $G$ denote a potential system state. We call $V$ *enabled in* $G$ iff for every event in $V$ one incoming *ne* edge is in $G$, and for every receive event in $V$ the corresponding *sig* edge is in $G$.

$$enabled(V, G) \triangleq range((ne \triangleright V) \cap G) = V \ \wedge \ (sig \triangleright V) \subseteq G$$

$$enableset(G) \triangleq \{V \mid enabled(V, G)\}$$

Let $G_1 = \{(c, e), (c, f)\}$ and $G_2 = \{(a, e), (c, e), (c, f)\}$ denote potential system states. Then $enabled(\{f\}, G_1)$ and $enabled(\{e, f\}, G_2)$. Note that in state $G_2$ two events are enabled simultaneously, which indicates an indeterministic behavior alternative.

### 7.4.2 Construction of a Successor State

We now define how a system transits between different global system states in relation to a set of enabled events. Assume that a system is in an actual state $G$. The following operations need to be performed in order to obtain the successor state $G'$.

- Select the event $a$ which is to be executed next from $enableset(G)$ (i.e. $\{a\} \in enableset(G)$),
- remove all *sig* edges pointing to $a$ from $G$,
- remove all *ne* edges pointing to $a$ from $G$,
- if $a$ has a directly preceding event $b$ which has multiple outgoing *ne* edges, remove all edges from $G$ which have source $b$,
- add all *ne* and *sig* edges which have source $a$ to $G$.

Let

$$prune(G, a) \triangleq ((G \ - \ (sig \triangleright \{a\})) \ - domain(ne \triangleright \{a\}) \triangleleft ne) \ \cup \ (\{a\} \triangleleft (ne \cup sig)))$$

Formally, we define the transition from $G$ to $G'$ on $a$, where $G, G'$ are pss's:

$$trans(G, a, G') \triangleq (a \in S \cup C) \ \wedge \ (G' = prune(G, a))$$

We define $G'$ to be a successor of $G$: $succ(G, G') \triangleq (\exists a)\, trans(G, a, G')$.

### 7.4.3 The Transition Relation

We define the global transition relation on a pss $G$, an event $a$ such that $\{a\} \in enableset(G)$ ($a$ is enabled in $G$), and a successor state $G'$ such that $succ(G, G')$. Let $N_{\mathcal{M}}$ denote an unfolding. The *global state transition relation* is $\mathcal{T}_{\mathcal{M}} \subseteq (ne \cup sig) \times (S \cup C \cup X) \times (ne \cup sig)$ such that

$$\mathcal{T}_{\mathcal{M}} \triangleq \{(G, a, G') \mid enabled(\{a\}, G) \ \wedge \ trans(G, a, G')\}$$

### 7.4.4 Global States and the Transition Graph.

We now distinguish system states that actually can occur in a run of the system. A system starts in its start state, and transits according to the transition relation, so every actual system state (called *global system state* below) is in the transitive closure of the transition relation starting from the start state. Finally, we restrict the transition relation to global system states to obtain the transition graph of the system.

Formally, let $\mathcal{M}$ be a set of MFGs, $N_{\mathcal{M}}$ the corresponding unfolding. Let $q_0$ be the set of start states, i.e. $q_0 = \{(a, b) \in ne \mid (\{a\} \lhd ne) = \emptyset\}$. We define $G$ to be a *global system state* (gss) iff $G \in Q$, where $Q = \{q_0\} \lhd \mathcal{T}^{+}_{\mathcal{M}}$ is the set of all gss's. Let $T_{\mathcal{M}} = Q \lhd \mathcal{T}_{\mathcal{M}}$ (the transition relation restricted to gss's). The *global state transition graph* corresponding to $N_{\mathcal{M}}$ is $\mathcal{G}STG_M \triangleq (Q, q_0, T_{\mathcal{M}})$.

## 7.5 From GSTGs to Automata via Liveness Properties

The global state transition graph, which we defined in the previous Section, is almost an automaton, lacking only a definition of end-states. We now turn to the definition of end-states.

### 7.5.1 Definition of Global State Automaton

Let $\mathcal{M}$ denote an MFG specification and $\mathcal{G}STG_M$ the corresponding global state transition graph. We can define a Büchi automaton which transits between global system states, by adding to $\mathcal{G}STG_M$ a definition of a set of *final states* $F$. The definition of a Büchi automaton is very similar to that of the usual finite-state automaton, except for the criterion for *acceptance* of a string. Büchi automata may accept infinite strings. A *global state automaton for* $\mathcal{G}STG_M = (Q, q_0, T_{\mathcal{M}})$ is $A_{\mathcal{M}} \triangleq (Q, q_0, T_{\mathcal{M}}, F)$, where $F \subseteq Q$ is a set of *final states*. Acceptance is Büchi-acceptance [143], namely an infinite word is accepted iff the automaton cycles through some state in $F$ infinitely often on the word (the alphabet is the set of events, e.g. $?a, !b$, and a word is thus a possibly infinite sequence of events, i.e. a possible trace).

Assume that the global state transition graph with 3 global states in Figure 7.5 is derived from some MFG specification, and $q_0 = S1$. The set of infinite paths through the graph is represented by the $\omega$-regular expression

$$(!a(!b?b)^{\omega}) + (!a(!b?b)^{*}?a)^{\omega} + (!a(!b?b)^{*}?a)^{*}.(!a(!b?b)^{\omega}).$$

Selecting $F = \{S2, S3\}$ as end-states means that traces of the form $!a(!b?b)^{\omega}$ would be accepted. Traces in this class do not satisfy the liveness requirement that a sent message will eventually be received (the counter example here is $!a$ in the first and third terms in the sum). However, selecting $F = \{S1\}$ ensures that only the *fair* traces of the form

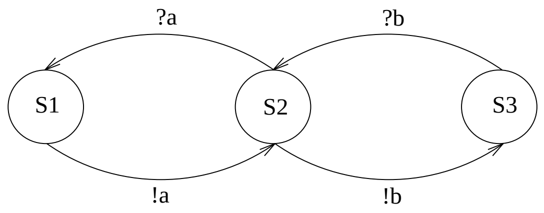


Figure 7.5: Global state transition graph

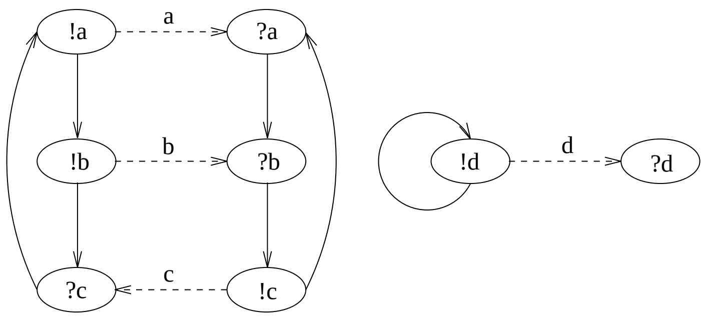


Figure 7.6: Strong and weaker liveness examples

$(!a(!b?b)^*?a)^\omega$ are accepted. Thus selection of a set of end-states depends fundamentally on the liveness and safety characteristics we wish to assume for a particular MFG specification. Applications of MFGs such as Message Sequence Charts and Time Sequence Diagrams omit explicit discussion of liveness properties. We show in this and succeeding Sections how this leads to ambiguity in which set of traces is specified by these methods. The explicit definition of liveness properties is thus required for these applications.

### 7.5.2 A Discussion of Two Liveness Properties

**A Strong Liveness Property for Loop Processes.** Consider a system whose pbMFG contains precisely one cycle per process, and assume no branching. Then the cycles are terminal, i.e. there are no outgoing edges from the cycles (which would violate branching). Then the processes are *loop processes* as defined in [100]. Let $P_i, i \leq n$ be the processes. Let $a_i \in P_i$ be some node in $P_i$'s cycle, for $i \leq n$, chosen such that $G = \{ne \triangleright \{a_i\} \mid i \leq n\} \in Q$ (i.e. $G$ is a *global system state* or *gss*). We omit the easy proof that there is some such $G$. Let $F = \{G\}$. A trace is accepted by the automaton with final-state set $F$ if and only if all processes iterate through their cycles infinitely often. This ensures strong liveness for the processes, as in the left-hand example in Figure 7.6, i.e. events preceding the cycle, and all events in the cycle, occur.

The example in Figure 7.7 shows that this condition does not ensure the strong liveness condition that all events eventually happen for examples in which the cycle is non-terminal (this can happen only if there is branching in the MFG). In this example, the language of

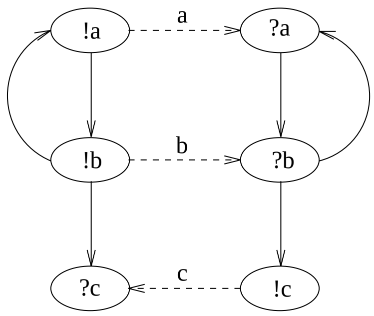


Figure 7.7: Strong liveness violated by branching

receive-events can be described by the expression $(?a?b)^\omega \cup (?a?b)^* ?a?b?c$ which denotes a set of finite and infinite regular sequences. In the infinite trace, the eventual reception of $b$ is actually ensured by the strong liveness requirement that $a$ is received infinitely often. But message $c$ will then never be either sent or received. However, this example does satisfy a weaker 'strong' liveness property that all signals sent will eventually be received.

**A Weaker Liveness Condition.** A weaker liveness property is to require: (*weak liveness*) for all processes which ever send, there is a state in the set of send-states which also is an end-state. Whereas the previous 'strong' liveness property expresses a general claim about the transmission medium, equivalent to requiring for loop processes that infinite sending leads to infinite reception, the 'weak' liveness property only addresses the local behavior of loop processes. For example, the system of loop processes described by the right hand part of Figure 7.6 satisfies the weak liveness condition, but not the strong liveness condition that all signals sent are received. Infinitely many signals are sent, but only one of the signals is ever received.

A final-states definition for loop processes which encodes this weaker liveness property is: If $P_i$ has a cycle, let $b_i = ne \triangleright \{a_i\}$, where $a_i \in P_i$ is any node in $P_i$'s cycle, for $i \leq n$. If $P_i$ has no cycle, then $b_i = \emptyset$. $G = \bigcup\{b_i \mid i \leq n\} \in Q$ (we again omit the easy proof that $G$ is a gss), and let $F = \{G\}$.

## 7.6 MFGs and their Connection to Temporal Logic

In the last Section we noted that liveness properties have bearing on the definition of the end-state set of the automaton. A discussion of the use of Büchi automata to specify such properties of distributed systems can be found in [8] and [9]. A complementary approach for expressing safety and liveness properties may be found in the use of temporal logic. Temporal Logic has also been advocated in the specification of open systems in [62], [61], [60], and in the specification of communication protocols in [148]. Temporal logic formulae are interpreted over infinite sequences of states, each state being defined by the truth values of state predicates. We relate these formulae to the automata obtained from the semantics definition. We remain informal here, referring the reader to the formal definitions in Section 7.7 for more precision. We base our temporal logic interpretation on the Manna-Pnueli approach [113].

**Basic Transition Systems.** Following [113] we interpret global state transition graphs as so-called *basic transition systems* (BTS). A BTS consists of a *finite set of states* $\Sigma$, a *transition function* $\tau$ mapping a state to a set of possible successor states, and an *initial condition.* We denote the set of all transitions $\tau$ by $T$. For an MFG $M$, $\Sigma$ will be the set of states $Q$ of $\mathcal{G}STG_M$, the transitions $\tau$ will be the communication events that lead from one global state to another, and the initial state of the BTS will be the initial state of the GSTG.

**Computations and State Predicates.** Manna and Pnueli define the following notions [113]. An infinite state sequence $\sigma = s_0, s_1, \ldots$ is a *computation* iff $s_0$ is the initial state of the BTS, and for all consecutive pairs $s_i, s_{i+1} \in \sigma$ there exists $\tau \in T$ such that $s_{i+1} \in \tau(s_i)$. The indices $i$ of $\sigma$ are *positions.* Transition $\tau$ is *enabled at position* $i$ of some computation $\sigma$, written as $en(\tau)$, iff $\tau(s_i) \neq \emptyset$. Transition $\tau$ is (has been) *taken* at position $i+1$, written as $ta(\tau)$, iff $s_{i+1} \in \tau(s_i)$.

To correlate these definitions with the global state transition graph, we need to define the *enabled* and *taken* predicates. Roughly speaking, a transition is *enabled* if it is enabled in the sense used earlier in Section 7.3. Similarly, a transition is *taken* in a state if that transition has led to the state from an immediately preceding state (notice the 'past tense' sense of the predicate *taken*).

**Temporal Logic.** Given these interpretations of a GSTG as a model for temporal logic, we may define a temporal logic in the usual way, e.g. [113]. The language has state predicates *en($\tau$)* and *ta($\tau$)* as only basic propositions, includes the Boolean connectives (we use just $\neg$ and $\vee$ for simplicity), and the temporal operators $\Diamond$ (eventually), $\Box$ (henceforth), $\diamondminus$ (sometime in the past), $\ominus$ (previous) and $S$ (since). The semantics is defined as usual. A temporal logic formula $p$ is interpreted over state sequences $\sigma$, and we define the usual model-theoretic notion $(\sigma, i) \models p$, that formula $p$ is *satisfied* in position $i$ of sequence $\sigma$.

## 7.7 Formal Definition of the Connection to Temporal Logic

In this Section, we formally define the Manna-Pnueli-style temporal logic interpretation of GSTGs. The Section may be skipped on a first reading.

Let $N_{\mathcal{M}} = (S, C, X, ne, sig, ST, stype, Top, Bottom)$ denote the pbMFG unfolded from some MFG specification $\mathcal{M}$ and let $\mathcal{G}STG_M = (Q, q_0, T_{\mathcal{M}})$ denote the corresponding

GSTG. First, we relate the GSTG to a *basic transition system* as defined in [113]. Then we consider *state predicates*, *computations* and finally the syntax and semantics of *temporal logic*.

**Basic Transition System.** $\mathrm{BTS}_{\mathcal{M}} \triangleq (\Pi, \Sigma, \mathcal{T}, \Theta)$ is a *basic transition system* corresponding to $\mathcal{G}ST G_M$, where

- $\Pi$ denotes a set of *state variables*, which is empty in the case of a set of cMFGs since they do not contain data[3],
- $\Sigma$ denotes the finite set of states, so $\Sigma = Q$.
- $\mathcal{T}$ denotes a set of *transitions*, $\tau : \Sigma \rightarrow 2^{\Sigma}$, for $\tau \in \mathcal{T}$. For $s, s' \in \Sigma$ let $\tau(s) \triangleq \{s' \mid (s, \tau, s') \in T_{\mathcal{M}}\}$ (the symbol $\tau$ now has, harmlessly, both a GSTG syntax and a $\mathcal{BTS}$ syntax). For MFGs, transitions are events of type $!a$ or $?a$, where $a$ is a signal type.
- $\Theta$ denotes an *initial* condition, in our case simply that the initial state *is* the initial state $q_0$ in $\mathcal{G}ST G_M$.

The states of $\mathcal{BTS}_{\mathcal{M}}$ correspond to global system states of $\mathcal{G}ST G_M$ (they are sets of edges of the *ne* and *sig* relations), and the transitions $\tau$ of $\mathcal{BTS}_{\mathcal{M}}$ correspond to communication events of the pbMFG obtained from the MFG specification.

**State Predicates.** Manna and Pnueli introduce an assertion which they call the *transition relation*[4] of the form $\rho_\tau : C_\tau(\Pi) \wedge (\bar{y}' = \bar{e})$ describing the change of the values of state variables in state $s$ to their values in state $s'$ into which the system transits from state $s$ by taking transition $\tau$. Since $\Pi = \emptyset$ for MFGs, $C_\tau$ is a constant, denoting the *enabling condition* which describes the condition under which the state $s$ may have a successor state by taking the $\tau$ transition. $(\bar{y}' = \bar{e})$ stands for a conjunct which expresses the values of a sequence of state variables after the transition has been performed. Since there are no state variables in MFGs, this conjunct is vacuous, and so the transition relation $\rho_\tau$ is equivalent to the enabling condition $C_\tau$ (which is just that there *is* some $\tau' \in \tau(s)$) thus $\rho_\tau$ holds in a state $s$ for some transition $\tau$ iff there exists $s' \in \Sigma$ such that $s' \in \tau(s)$.

Thus a transition $\tau$ is *enabled* in some state $s$ iff $\tau(s) \neq \emptyset$. Conversely, $\tau$ is *disabled* in $s$ iff $\tau(s) = \emptyset$. Mapping this to our MFG definitions we have that $\tau$ is enabled in state $s$ iff there is some state $s'$ such that $\langle s, \tau, s' \rangle \in T_{\mathcal{M}}$. Consequently, the Manna-Pnueli enabling condition for an action $\tau$ is true in precisely those GSTG states in which $\tau$ is *enabled* in the sense in which we defined this predicate for GSTGs earlier.

---

[3] The only data in MFGs is signal type information, which is encoded in state information.

[4] We will show that this notion is simplified for MFGs, so there will be no confusion with our notion of transition relation for GSTGs.

**Computations and State Predicates *en* and *ta*.** An infinite state sequence $\sigma = s_0, s_1, \ldots$ is a *computation* [113], iff

- $s_0 \models \Theta$, which means just $s_0 = q_0$, and
- for all consecutive pairs $s_i, s_{i+1} \in \sigma$ there exists $\tau \in \mathcal{T}$ such that $s_{i+1} \in \tau(s_i)$.

The indices $i$ of $\sigma$ are *positions.* Transition $\tau$ is *enabled (disabled) at position* $i$ of some computation $\sigma$ iff it is enabled (disabled) in $s_i$. We say that transition $\tau$ is *taken at position* $i+1$ iff $s_{i+1} \in \tau(s_i)$. We define the predicate *en(τ)* to hold in state $s_i \in \sigma$ iff $\tau$ is enabled at position $i$ and we define the predicate *ta(τ)* to hold in state $s_i \in \sigma$ iff $\tau$ is taken at position $i$[5]. As noted, these definitions cohere with our former GSTG definitions. In Figures 7.2 and 7.1, we annotate each state with the instances of *en* and *ta* that are true in that state.

**Temporal Logic.** We define a temporal logic in the usual way, following [112]. The language has state predicates *en(τ)* and *ta(τ)* as basic propositions, includes the Boolean connectives (we use just $\neg$ and $\vee$ for simplicity), and the temporal operators $\Diamond$ (eventually), $\Box$ (henceforth), $\diamondminus$ (sometime in the past), $\ominus$ (previous) and $\mathcal{S}$ (since).

The semantics are defined as usual. A temporal logic formula $p$ is interpreted over state sequences $\sigma$, and we define $(\sigma, i) \models p$, i.e. that formula $p$ is *satisfied* in position $i$ of sequence $\sigma$.

- If $p$ is a basic assertion, then $(\sigma, i) \models p$ *iff* $p$ is true in $s_i$ as defined above.
- $(\sigma, i) \models \neg p$ *iff not* $(\sigma, i) \models p$
- $(\sigma, i) \models p \vee q$ *iff* $(\sigma, i) \models p$ *or* $(\sigma, i) \models q$
- $(\sigma, i) \models \Diamond p$ *iff* for some $j \geq i$ $(\sigma, j) \models p$
- $(\sigma, i) \models p\mathcal{S}q$ *iff* for some $k, 0 \leq k \leq i, (\sigma, k) \models q$,
  and for every $j$ such that $k < j \leq i$, $(\sigma, j) \models p$
- $(\sigma, i) \models \ominus\, p$ *iff* $i > 0$ *and* $(\sigma, i-1) \models p$

As syntactic abbreviations we introduce the following notation.

- $\Box p \triangleq \neg \Diamond \neg p$
- $\diamondminus p \triangleq \mathit{true}\ \mathcal{S}\ p$

We say that a formula $p$ *holds* on sequence $\sigma$ iff $(\sigma, 0) \models p$, that it is *satisfiable* iff it holds for *some* computation, and *valid* iff it holds for *all* computations.

[5]In order to avoid notational confusion with our definitions for MFGs we distinguish our notation slightly from the notation used in [113].

## 7.8 Logical Properties of MFGs.

We can now give examples of properties expressed in temporal logic which can characterise pbMFGs obtained by unfolding sets of simple MFGs. In this Section, the MFG specifications considered all contain simple cMFGs only. The classification of properties as *safety, recurrence*, etc, refers to the classification in [112].

### 7.8.1 Properties Satisfied by all MFG Specifications

The following properties are satisfied by all computations derived from MFG specifications, as may be seen by inspection.

1. **Enabling of a `send` event (a *safety* property):** If a `send` event is taken, it must have been enabled previously. However, the enabling does not have to persist until the event is disabled, because a `send` event may also be disabled by a nondeterministic behavior alternative (some branch is taken rather than another) in the process's control flow.

   $$\Box(ta(x) \supset \diamondminus en(x))$$

   where $x \in S$.

2. **Persistent enabling of a `receive` event: (a *safety* property)** A `receive` event may only be taken if it has been previously enabled by a `send` event of the same type. Additionally, the enabling of a `receive` event can only be disabled by a `receive` event, therefore an enabling of a `receive` event persists up until the state when it is taken.

   $$\Box(ta(y) \supset \circleddash\, (en(y)\; S\; ta(x)))$$

   where $\langle x, y\rangle \in sig$.

### 7.8.2 Some Potential Requirements on MFG Specifications.

Some liveness properties are not automatically fulfilled by an MFG specification $M$. It was noted earlier that some of these properties were definable by making different selections of the set of final states of a Büchi automaton defined on $GSTG_M$. If it is required for an application that these properties should hold, they must explicitly be stated as an annotation to a pbMFG or an MFG specification. Well-known examples of such properties are

1. **Weak fairness (a *recurrence* property):** it is not the case that any transition $\tau$ is enabled continuously without ever being taken.

   $$\Box\Diamond(\neg en(\tau) \vee ta(\tau))$$

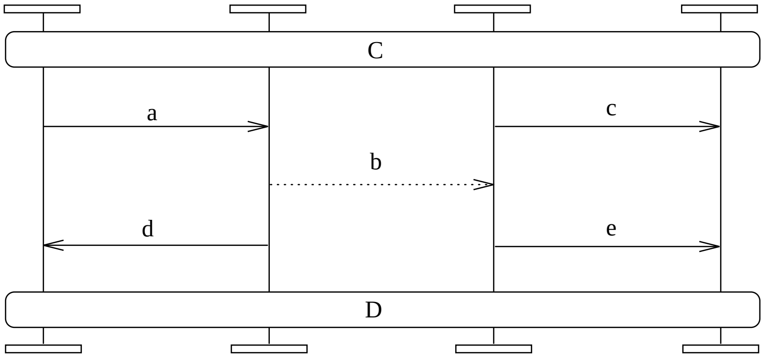


Figure 7.8: MSC with synchronous communication

2. **Strong fairness (a *reactivity* property):** if an arbitrary transition $\tau$ is enabled infinitely many times, then it is taken infinitely many times.

$$\Box\Diamond en(\tau) \supset \Box\Diamond ta(\tau)$$

It is known (and should be clear) that strong fairness implies weak fairness. We note that since `receive` events are persistently enabled, strong fairness and weak fairness just for `receive` events are equivalent statements. However, since a `send` event may be disabled without being taken, strong fairness and weak fairness are not equivalent for `send` events.

## 7.9 Representing Synchronous Communication in MFGs

So far, we have considered *sig* edges to represent asynchronous communication between processes. It is relatively straightforward to include synchronous communication *sig* edges in our definitions, and to provide a definition of GSTG transitions for synchronous communication events. We show the modifications needed to handle synchronous *sig* edges in the following sections.

### 7.9.1 Example

The example of Figure 7.8 shows an MSC specification which includes synchronous as well as asynchronous communication. We will denote the synchronous communication between two processes by dotted-line arrows and we will call the corresponding message symbol a `synchronous message` symbol. Solid line arrows will from now on correspond to `asynchronous message` symbols. Arrow tails and heads denote the respective `synchronous` or `asynchronous message input` and `message output` symbols. We translate MSCs into MFGs (see Figure 7.9 for the corresponding MFG) in a similar manner as before. However, different definitions of the transition relation must be given for synchronous and asynchronous messages, since synchronous symbols represent a single atomic

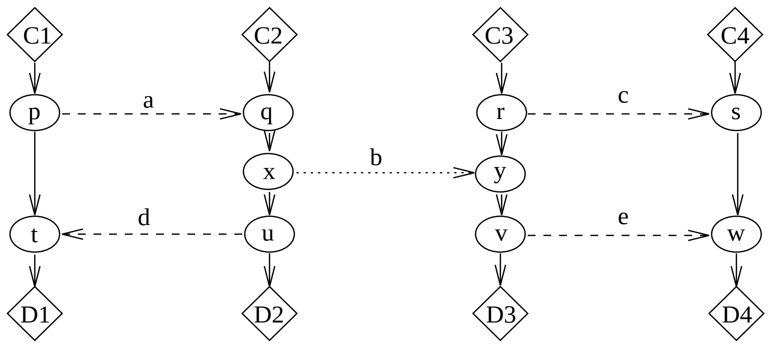


Figure 7.9: MFG with synchronous communication

action in which both processes participate (corresponding to an atomic transition in the GSTG), rather than the two separate, maybe temporally distinct, actions of asynchronous message-passing.

We introduce two disjoint *sig* relations: an *asynchronous sig* relation, e.g. the set of pairs $\{< p, q >, < r, s >, < u, t >, < v, w >\}$ in the example above; and a *synchronous sig* relation, in the example just the single pair $\{[x, y]\}$. Members of the asynchronous *sig* relation are *asyncsig*s, and members of the synchronous relation *syncsig*s. We write *asyncsig*s between angle brackets $< \ldots >$, and *syncsig*s between square brackets $[\ldots]$.

Sending and reception of a synchronous message are syntactically distinct, even though sending/reception is a single *atomic action* i.e. an action which is indivisible. In particular, this means that if $[x, y]$ is a *syncsig* with label $a$, there is no temporal ordering on the $!a$ represented by $x$ and the $?a$ represented by $y$. Hence, in a trace, these occur as one event. There are two ways this can be technically denoted in traces. One is by writing the $!a, ?a$ adjacent in every trace (in this order, since a *syncsig* retains a direction from sending to reception). This correctly represents the atomicity of this event, since the sending and reception are not interrupted by any other event in the system in any trace. The second is that, since sending and reception are merely syntactically distinct parts of a single event, the event should be represented in the trace by a single notation, say $[!a, ?a]$. Implementations of the specified processes have distinct code locations corresponding to sending and receiving primitives, which argues for the the first choice. However, it is plausible that a message of type $a$ may be sent over an *asyncsig* at one point, and another also of type $a$ over a *syncsig* at another. Under the first choice one would not be able to tell from the trace notation alone whether a particular occurrence of a $!a, ?a$ is a sync send/receive, or an async send followed immediately (for whatever reason) by an async receive. For this reason, we prefer the notation $[!a, ?a]$.

To show how the atomicity of synchronous communication restricts the sets of possible interleavings, consider the example of Figure 7.9. If message $b$ was asynchronous, a sequence of events $etype(x), s, etype(y)$ could be an admissible part of a system trace, which

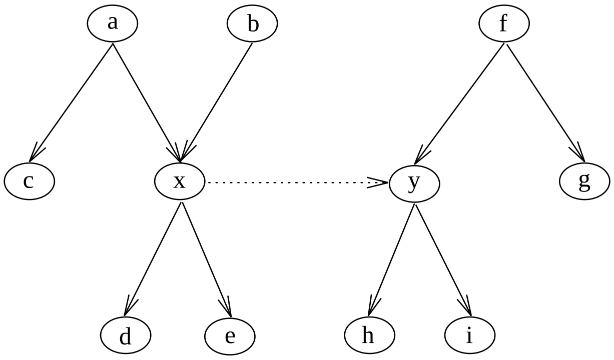


Figure 7.10: Part of an MFG with synchronous communication

is not the case if message $b$ is synchronous.

Since synchronous sending and reception are syntactically distinct but represent an atomic action, a synchronous communication event of sending and reception corresponds to exactly one transition in the global state transition graph (rather than the two distinct transitions required for *asyncsig* edges).

**Enabling.** To illustrate the concept of *enabling* for *syncsig*, we walk through the partial MFG in Figure 7.10. Assume that the graph is a part of a larger MFG, and that $c$ and $g$ are asynchronous `send` events. All arrows in the chart belong to the ne-relation except for the pair $[x, y]$ which belongs to the *syncsig* relation. As before, a global system state (gss) is a set of edges of the MFG. Consider a global system state

$$G = \{(a, c), (a, x), (f, y), (f, g)\}.$$

Events $c, x, y$ and $g$ are enabled. We focus on the synchronous events $x$ and $y$. As in the asynchronous case, a necessary condition for $x$ to be enabled is that at least one of the in-edges of node $x$ is in $G$. This condition is satisfied in $G$, but in contrast to asynchronous sends this is *not* a sufficient condition in the synchronous case for $x$ to be enabled. A synchronous send event may not occur independently of its corresponding receive event, so for either $x$ or its corresponding receive event $y$ to be enabled, we require also that for $y$ there is at least one incoming edge which is in the gss $G$. Hence for every *syncsig* $[x, y]$, the send node $x$ (respectively the receive node $y$) is *enabled* in $G$ if at least one ne edge with $x$ as target (second coordinate) is in G and at least one *ne* edge with $y$ as target is in $G$.

**State Transitions.** We now informally explain the state transition associated with a *syncsig* edge. Occurrence of the event associated with $[x, y]$ corresponds to the transition from global system state $G$ to a successor state $G'$. The transition corresponds to completion of the atomic synchronous communication action with a signal both generated and received. This signal can never be 'in transit' in any distinct system state, so it never appears as part of a system state. Both nodes $x$ and $y$ are enabled in $G$, and the transition occurs as both processes advance their program counters to the next event, represented by omitting ne edges of the form $(*, x)$ and $(*, y)$ and adding edges of the form $(x, *), (y, *)$ to the $ne$ relation.

As in the asynchronous case, however, one has to do a bit more, removing also the possibility of choosing the enabled actions represented by $c$ and $g$, which represent choice alternatives to the occurrence of the $[x, y]$ event. Hence the edges $(a, c)$ and $(f, g)$ are removed from $G$ as we transit via $[x, y]$ to $G'$. Hence

$$G' = \{(x, d), (x, e), (y, h), (y, i)\}.$$

We define the transition relation formally below.

### 7.9.2 Formalisation of Extended Message Flow Graphs

Our formal definitions are based on previous definitions. All parts of the definitions which are not explicitly redefined here remain valid. We refer to MFGs with both synchronous and asynchronous communication as *extended MFG*s or *XMFG*s.

**Extended MFGs.** To handle both *asyncsig*s and *syncsig*s, we simply split each set of send nodes (receive nodes) into two disjoint sets of syncsend (syncreceive) and asyncsend (asyncreceive) nodes, and the *sig* relation into two disjoint *syncsig* and *asyncsig* relations.

Let $S_a, S_s, C_a, C_s$ and $X$ denote arbitrary pairwise disjoint sets, the elements of which we call *asyncsend* nodes, *syncsend* nodes, *asyncrec* nodes, *syncrec* nodes, and *extra* nodes. Let $ST$ and $ET$ denote arbitrary disjoint sets whose elements we call *signal* and *event* types. For compatibility with the MFG definition we define $S \triangleq S_a \cup S_s$, and $C \triangleq C_a \cup C_s$. An *extended* MFG is a tuple

$$\mathcal{G}^X = (S_a, S_s, C_a, C_s, X, ne, sig_a, sig_s, ST, stype, ET, etype, Top, Bottom)$$

where $(S \cup C \cup X, ne, etype, ET)$ is a digraph with node labels, and $(S_a \cup C_a, sig_a, stype, ST)$ and $(S_s \cup C_s, sig_s, stype, ST)$ are digraphs with edge labels satisfying the following conditions:

1. $sig_a \subseteq S_a \times C_a$
2. $sig_s \subseteq S_s \times C_s$
3. $sig \triangleq sig_a \cup sig_s$,
4. all other conditions as for MFGs are satisfied.

This splitting of $S$, $C$ and $sig$ are the only changes required in the MFG (the 'abstract syntax'). Note that $sig_a$ and $sig_c$ are both bipartite relations.

### 7.9.3 Semantics of Extended MFGs

We now make the required additions to the formal definitions of *potential system state*, of *enabling* a set of events in a global system state, and finally of the *global state transition graph*, to accommodate XMFGs.

**Enabling.** Given an extended MFG

$$\mathcal{G}^X = (S_a, S_s, C_a, C_s, X, ne, sig_a, sig_s, ST, stype, ET, etype, Top, Bottom)$$

we define a *potential system state* (pss) $G \subseteq ne \cup sig$ to be any subset of $ne \cup sig_a$ (synchronous communication never leaves a message in transit, therefore there will never be a system state which includes an edge from the $sig_s$ relation).

Let $V \subseteq S \cup C$, and let $G$ denote a potential system state. $V$ is *enabled in* $G$ iff

- if $x \in V, [x, y] \in sig_s$ then $y \in V$, and if $y \in V, [x, y] \in sig_s$ then $x \in V$,[6]
- for every event in $V$ one incoming $ne$ edge is in $G$,
- for every receive event in $V$ the corresponding $sig_a$ or $sig_s$ edge is in $G$.

The condition that, for every synchronous send event in V there is at least one incoming $ne$ edge in G of the corresponding synchronous receive event and *vice versa*, is automatically ensured by the conjunction of the first two conditions above. Let $SS_V = domain(V \lhd sig_s)$ be the set of synchronous send events in $V$; similarly $SR_V = range(sig_s \rhd V)$ the set of synchronous receives in $V$; $AS_V = domain(V \lhd sig_s)$ the set of asynchronous sends in $V$; and $AR_V = range(sig_s \rhd V)$ the set of asynchronous receives in $V$. We formally define the extended *enabling* predicate as

$$enabled^X(V, G) \triangleq$$

$$SR_V = range(SS_V \lhd sig_s) \wedge SS_V = domain(sig_s \rhd SR_V)$$

$$\wedge \; range((ne \rhd V) \cap G) = V$$

$$\wedge \; ((sig_s \cup sig_a) \rhd V) \subseteq G$$

and the extended set of enabled events as

$$enableset^X(G) \triangleq \{V \mid enabled^X(V, G)\}.$$

[6] As we explained informally above, a synchronous node can only be enabled if both elements of the *syncsig* are enabled, hence the condition on $V$.

**Construction of a Successor State.** Assume that a system is in an actual state $G$. To obtain a successor state $G'$, all conditions as before in the definition of *succ* in Section 7.4 must hold; furthermore, if the triggered event is synchronous, its partner event must also go through the transition process. We recall the definition of $prune(G, a)$ from Section 7.4. Recall also that we can think of a transition on a synchronous event formally as two adjacent transitions consisting of the send followed immediately by the receive. We may use *prune* to generate the two adjacent formal transitions on a synchronous event and its partner. The transition on asynchronous events is as before. Thus, the following operations need to be performed:

- Select the event $a \in enableset^X(G)$ which is to be executed next;
- prune $G$ by $a$, i.e. perform $prune(G, a)$;
- if $a$ is a synchronous send event and $b$ is the corresponding synchronous receive event, $prune(G, a)$ and if $G_1$ is the result, follow with $prune(G_1, b)$;
- if $a$ is a synchronous receive event and $b$ is the corresponding synchronous send event, $prune(G, b)$ and if $G_2$ is the result, follow with $prune(G_2, a)$.

Formally, we define the transition relation from $G$ to $G'$ via $a$:

$$trans^X(G, a, G') \triangleq$$

$$(a \in (AS_V \cup AR_V) \wedge trans(G, a, G'))$$

$$\vee\ (a \in SS_V \wedge G' = prune(prune(G, a), range(\{a\} \lhd sig_s)))$$

$$\vee\ (a \in SR_V \wedge G' = prune(prune(G, domain(sig_s \rhd \{a\})), a)).$$

We may define the extended successor relation $succ^X(G, G') \triangleq (\exists a \in S \cup C)\, trans^X(G, a, G')$.

**The Transition Relation.** We define the global transition relation on a pss $G$, an event $a$ such that $\{a\} \in enableset^X(G)$ ($a$ is enabled in $G$), and a successor state $G'$ such that $succ^X(G, G')$. Let $N^X_{\mathcal{M}}$ denote an extended unfolding. The *global state transition relation* is $\mathcal{T}^X_{\mathcal{M}} \subseteq (ne \cup sig_a) \times (S \cup C_a \cup X) \times (ne \cup sig_a)$ such that

$$\mathcal{T}^X_{\mathcal{M}} \triangleq \{(G, a, G') \mid enabled^X(G, \{a\}, G') \wedge trans^X(G, a, G')\}.$$

The notions of global system state and transition graph are as defined for MFGs.

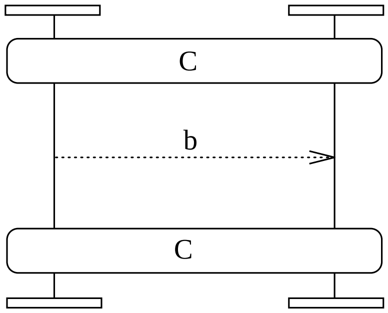


Figure 7.11: MFG with synchronous communication

### 7.9.4 Postscript

We have treated a synchronous communication event in an MFG by formalising the transition as a pair of adjacent send-receive events, without an intervening state. Since synchronous communication is an atomic action represented by two distinct nodes in an ne-sig graph, why did we not represent such an action by a single node in the MFG? The answer is that pursuing this alternative would have destroyed some nice properties of the MFG definition, namely (a) processes can be identified as connected components of the *ne* relation; and (b) the information about the direction of the communication (from which process to which other) is lost. In particular, the direction of a synchronous communication represents the fact that sending and receiving of messages refer to distinct code-locations in the processes involved, information which may be essential in conformance testing [80] and debugging.

The facility with which the extension to synchronous communication could be accomplished is evidence for us that the distinction we have emphasised between abstract syntax as represented by MFGs and the subsequent semantic interpretation by GSTGs and automata allows a very flexible treatment of the MSC semantics. It should be emphasised that the modifications involved only a minor change to the definitions of $S, C$ and $sig$, and a minor addition to the notions of pss, of enabling, and of the transition relation *succ* for enabled nodes in the $sig_s$ relation. All syntactic operations on MFGs like composition and unfolding were untouched by the extension.

### 7.9.5 Liveness Properties

We discussed earlier some liveness properties that MFGs with asynchronous communication might satisfy. Guaranteeing liveness properties is considerably simplified with the introduction of synchronous communication edges. A *syncsig* edge enforces synchronisation at its head and tail. Thus, all events preceding any event on a *syncsig* edge must already have occurred before the synchronous event.

For example, consider the MSC with synchronous communication in Figure 7.11 which

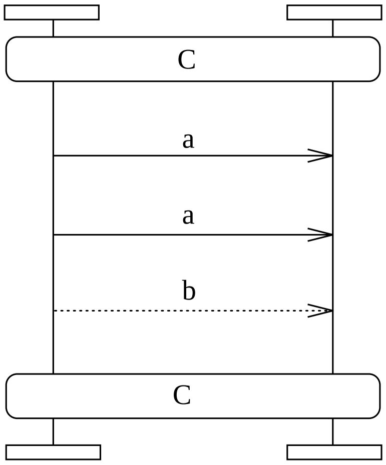


Figure 7.12: MSC with asynchronous and synchronous communication

is a modification of the asynchronous example MSC I (Figure 2.2). The enforced synchronisation of the two processes avoids the phenomenon of repeated sending without reception, noted before with respect to asynchronous communication. The unique trace of this MFG is simply $[!a, ?a], [!a, ?a], [!a, ?a], [!a, ?a], \ldots$. Similarly, in Figure 7.12, the allowable traces are precisely the language $(A + B)^{\infty}$ where $A = !a, !a, ?a, ?a, [!b, ?b]$ and $B = !a, ?a, !a, ?a, [!b, ?b]$.

Finally, supposing that the $c$ communication in MFG III (Figure 2.3) is synchronous. It becomes clear that synchronous communication not only makes the liveness issues easier, but also reduces the state space complexity. The unique trace in that case would be $!a, ?a, [!c, ?c], !a, ?a, [!c, ?c], \ldots$, and the corresponding automaton has five states compared to nineteen states for the original automaton in the asynchronous case of Figure 7.3.

It is easy to see that if all processes in an ne-sig graph with loops must pass through at least one synchronous communication, that there is only one possible definition of the Büchi automaton corresponding to the GSTG, i.e. the GSTG determines the automaton uniquely.

## 7.10 Abstraction of Automata

In this Section, we show how to simulate an arbitrary Büchi automaton by an MFG specification. The simulation relies on the notion of *abstraction* of a Büchi automaton.

Informally, an *abstraction* of a Büchi automaton $\mathcal{A}$ is an automaton $\mathcal{A}'$ whose states are a subset of the states of $\mathcal{A}$, and whose transitions occur not on letters from the alphabet of $\mathcal{A}$, but on sequences of such letters (i.e. words). In other words, the abstraction retains information only on some states, and how one gets from these particular states to others by treating a sequence of transitions of $\mathcal{A}$ as a single transition.

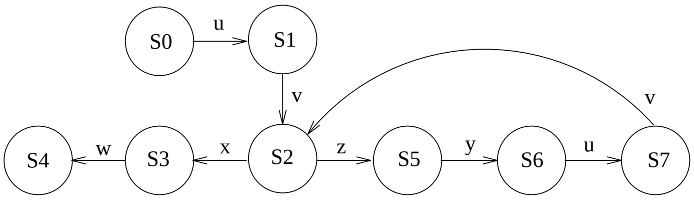


Figure 7.13: Global State Transition Graph

**An Example.** The GSTG for the MFG specification in Figure 2.7 is shown in Figure 7.13. We can form an abstraction of the GSTG, by retaining information only about the global states denoted by conditions C1, C2, and C3, which correspond to the state sets $\{S0, S6\}$, $\{S2\}$, and $\{S4\}$. The abstraction is shown in Figure 7.14 (in this case, it is in fact the composition graph of the MFGs).

The point of abstractions is that they are in an intuitive sense a summary, a less complex version, of the automaton that they abstract. We show below that an arbitrary Büchi automaton is an abstraction of the global-state automaton derived from some MFG specification, and therefore that MFG specifications are in this sense equally as complex as Büchi automata (and therefore much more expressive than temporal logic [149]) under our semantics.

**Abstractions Formally.** $\langle \mathcal{A}', I, h \rangle$ is an *abstraction* of $\mathcal{A}$ iff

- $I$ is a mapping of the state set of $\mathcal{A}'$ into the set of state sets of $\mathcal{A}$, i.e. $I$ picks out a subset of the states of $\mathcal{A}$ which correspond with a particular state of $\mathcal{A}'$;
- $h$ maps the alphabet of $\mathcal{A}'$ into words in the alphabet of $\mathcal{A}$, i.e. transitions among states in $\mathcal{A}'$ are translated into sequences of transitions of $\mathcal{A}$;
- there is a transition from $s$ to $s'$ on $p$ in $\mathcal{A}'$ iff $h(p)$ is a path from some state in $I(s)$ to some state in $I(s')$ in $\mathcal{A}$,

It is well-known that such an $h$ can be extended into a homomorphism of the words of $\mathcal{A}'$ into the words of $\mathcal{A}$. We shall abuse terminology and refer to $\mathcal{A}'$ alone as an abstraction of $\mathcal{A}$. A particularly important kind of abstraction is the *one-to-one abstraction*, in which every $I(s)$ is required to be a singleton, i.e. there is only one state of $\mathcal{A}$ corresponding to each state in $\mathcal{A}'$. The abstraction in Figure 7.14 is not a one-to-one abstraction. We categorise the expressiveness of MFGs by the following theorem.

**Theorem 7.10.1** *Every Büchi automaton is a one-to-one abstraction of an automaton derived from an MFG specification involving just two processes.*

*Proof* [Sketch]: Consider the MFG expressed in MFG V (Figure 7.15). It is easy to see that there is only *one possible sequence of events*, as shown in the GSTG. The following

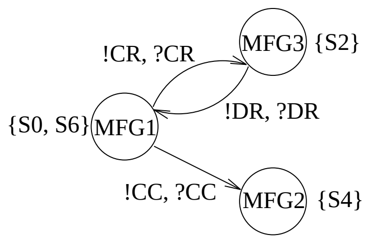


Figure 7.14: An Abstraction Graph

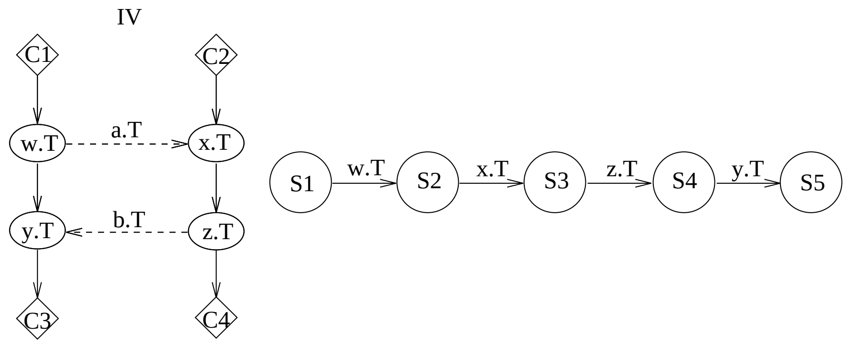


Figure 7.15: MFG V and its GSTG

lemma may easily be proved by recursion, and by inspection of the transition between states S1 and S5 in the GSTG.

**Lemma 7.10.2** *Let $\mathcal{B}$ be a Büchi automaton based on the GSTG of MFG V, which additionally satisfies weak liveness. If $\mathcal{B}$ attains the state represented by the beginning condition $C.s$, it will later attain the state represented by the terminal condition $C.s'$.*

To each pair of states $s$ and $s'$ of an arbitrary Büchi automaton $\mathcal{A}'$, and transition $T$ between them, we associate a copy of MFG V, with conditions $C.s$ and $C.s'$, and signals $a.T$ and $b.T$. This defines an MFG specification $\mathcal{M}$, with GSTG $\mathcal{G}STG_M$. We define $I(s) = C.s$ and $h(T) = !a.T?a.T!b.T?b.T$, and finally the global states of the automaton $\mathcal{A}$ derived from $\mathcal{G}STG_M$ are $\{C.s \mid s \textit{ is a final state of } \mathcal{A}'\}$. Using the lemma, it follows that $\mathcal{A}'$ is an abstraction of $\mathcal{A}$.

A crucial step in this proof is the definition of the end-states of the automaton $\mathcal{A}$ derived from the MFG specification. We are able to do this as we please, because the end-state set is not determined by a pure MFG specification. The reader should note that some additional liveness requirement for MFG specifications could restrict the choice of end-state set for the automaton $\mathcal{A}$, and preclude us from carrying out the simulation of an arbitrary Büchi automaton.

## 7.11 Concluding Remarks

We provided a finite-state semantics for MFGs. We defined cMFGs, and showed how to obtain pbMFGs from sets of cMFGs by unfolding. We defined a collection of global system states, and transitions between them, from the pbMFG. We discussed the completion of this GSTG to a Büchi automaton, noting the reliance on liveness properties not explicit in the MFG, leading to a connection to temporal logic via the standard semantics. We also showed how to incorporate synchronous communication with asynchronous communication in MFGs, and noted that it greatly simplifies the liveness analysis of MFGs to include synchronous communication signals. We showed how to simulate an arbitrary Büchi automaton by an MFG. Since we have also shown that an arbitrary MFG may be interpreted by a Büchi automaton, we conclude that in this sense MFGs and Büchi automata have equivalent expressive power.

# Chapter 8

# Discussion of Some Issues in the Semantics

We now discuss a few more issues arising from our semantics. We required that a system described by an MFG has global states with respect to its message-passing behavior, with instantaneous transitions between these states effected by atomic message-passing actions. Under this assumption, we argue now that

- the unrestricted use of 'conditions' requires processes to keep control history variables of unbounded size,
- that allowing 'crossing' messages of the same type implies certain properties of the environment that are neither explicit nor desirable,
- that MFG specifications can 'count' receptions of messages, and
- that liveness properties of MFGs are more easily expressed by temporal logic formulas over the control states than by Büchi acceptance conditions over the same set of states.

## 8.1 Introduction

The purpose of this Chapter is to discuss issues arising from the task of giving a precise mathematical semantics for MFGs and MSCs. Seemingly innocuous syntactic choices, made for example in the syntactic definition of MSCs in Z.120, may have profound and somewhat contrary semantic consequences. We point out some of those consequences here, not to resolve them (for resolution depends on standardisation group consensus), but to emphasise both the need for resolution, and the danger of introducing syntactic features without thinking through their semantic consequences.

It is generally accepted that the underlying ontology of MFGs (e.g. in their MSC form) is that of events (as also in the various proposed MSC semantics of [134, 48, 64]). We can summarise the representation of event features in an MFG as follows:

- (a) represented events are *message* `sends` and `receives` *only*;
- (b) the *occurrence* of an event is represented;
- (c) the *order of occurrence* of events within an individual process is represented;
- (d) the *types* of the events are represented.

These features are represented in the message-passing fragments of other imperative languages such as the original version of CSP [70], Esterel [23], Estelle [49] or SDL [32]. Representing this information in a graph such as an MFG is a mathematically more precise reformulation of the pictorial information about events and their ordering. However, this is not all the information that an MFG appears to contain. It is an additional feature of MFGs, not present in either CSP, Esterel, Estelle or SDL, that each of these events (in a simple MFG) or 'program statements' (in MFGs with conditions) is somehow 'connected' to precisely one other,[1] (c.f. the property (*) for MFGs in Section 2.4). A semantics of MFGs must explain how and what this connection is supposed to denote.

Next we briefly describe the issues concerning the semantics of MFGs discussed in the remainder of this Chapter.

**The problem of non-local choice.** Because of the requirements of the MSC standard, in order to treat MSCs as MFGs we defined a notion of *condition* for MFGs, and composition of MFGs by conditions (see Section 7.2.3.3). Conditions are global labels such that two MFGs may be 'joined' at this label. By allowing more than one possible joining, one obtains the effect of non-deterministic choice in MFGs (but conditionals defined on the values of state predicates are still not possible). By writing the same condition at the beginning as well as the end of an MFG, one obtains *non-terminating-loop*-like behavior. We argue in Section 8.2 that the unimpeded use of conditions requires the use of unbounded history variables ranging over decision predicates in order to resolve non-local choices. In other words, a complete history of control branching must be accessible to a process, and must either be held by the environment, and communicated implicitly to a process on demand, or must be communicated implicitly by the environment to a process dynamically, and held in a history variable of potentially unbounded size in the process itself.

[1] This may not quite be accurate. MSCs used informally sometimes have `sends` going nowhere, to indicate a lost message. But this is not in the MSC standard, and we do not think it's particularly significant for the argument made here.

Neither of these situations is particularly appealing. They lead to an additional source of non-finite-state behavior that we believe is even less warranted for MSCs than the usual arguments (which we also believe inappropriate) for non-finite-stateness due to potentially unbounded message buffering. We believe that a good specification style should only make explicit assumptions on environment behaviour.

**Crossing message arrows create anomalies.** In Section 8.3 we note that allowing crossing message arrows as in Figure 8.3 (which one may be tempted to interpret as 'message overtaking') leads to further implicit assumptions on implicit properties of the environment behaviour.

**MFGs can 'count' receptions.** One might make a further requirement that the state of each individual process should be not only determinable but also explicit in the specification style. For example, if you require a process to remember the last twenty messages (even if they are of identical type), then this requirement would entail that all twenty signals must appear in the MSC. In Section 8.4 we shall show that MSCs can 'count' in just this way.

**Büchi acceptance conditions are insufficient to express general liveness properties.** To define the set of execution traces specified by an MFG, we described how to compute a Global State Transition Graph (GSTG) from a given MFG in Chapter 7. However, this construction alone does not specify liveness properties for the MFG in a satisfactory way. We suggested using either Büchi acceptance conditions or Temporal Logic formulae instead. In Section 8.5 we prove that given the GSTG which is unambiguously defined by our semantics for a given MFG it may not be possible to specify every useful liveness property by using Büchi acceptance conditions, and suggest instead that temporal logic be used to specify the desired liveness properties for MFG specifications.

## 8.2 Conditions and Non-Local Choice

MSC specifications which include even simple use of conditions, such as in Figure 2.6, may induce multiple possible traces. From condition `C2` in `MSC1`, two different possible execution paths are possible, represented by `MSC2` and `MSC3`. This results in branching control, represented by the two next-event out-edges from each of nodes `u` and `v` in the MFG (Figure 2.8). These control branches have to be synchronised, in that if one process takes one branch and issues a `send`, the other process is constrained to take its corresponding branch to execute the corresponding `receive`. Since an MFG (and the MSCs) are supposed to represent message exchange, it is consistent with this view that the synchronisation should either be accomplished locally by each process, or explicitly in the

MFG by message exchange. However, there are cases in which synchronisation cannot be achieved by methods purely local to each individual process (see MSC specification in Figure 8.1, and the corresponding MFG in Figure 8.2)[2]. We show that this non-local decision-making leads to the necessity for each process to have access to its complete choice history, a record which is of unbounded size in non-terminating processes.

### 8.2.1 Non-Local Choice, and Choice History

We show that history variables or their equivalent are needed to handle control branching that cannot be achieved by local means. We note that the MSC standard Z.120 which suggests introducing conditions contains no recognition of the need for history variables in these circumstances. The spirit of MSCs would require that control choice synchronisation between processes which cannot be accomplished by each process acting independently should be accomplished by explicit exchange of messages indicating that a particular control branch is followed.

We make the technical argument leading to the necessity for history variables in two parts. The two parts therefore act as a *reductio ad absurdum* for the unimpeded use of global initial and final conditions. By defining MFGs which provide, in a natural way, the non-local synchronisation required for choosing branch of control without using explicit messages, we demonstrate that this leads to the necessity to remember potentially unbounded execution histories in unbounded history variables, of which there is no mention in Z.120.

We believe that any attempt to synchronise non-local control branching must employ some device similar to ours. However, whether Z.120 should embrace history variables, or instead severely restrict the use of conditions, is an issue on which we make no recommendation here. We are content merely to demonstrate the need for a resolution.

### 8.2.2 An Example

The example is a modification of Figure 2.6. In the MFG in Figure 2.8, the control-branch choice may be resolved locally. The first process, equipped with message-type-identification, awaits a signal from the second, and determines whether it is a `CC` or a `DR` message. Indeed, this would be the sensible way to implement it. Thus this example

[2] Another way of looking at the problem is to inquire whether it is possible to locally implement the system's decision making, or whether a global synchronization instance is needed. In the case of the local choice example in Figure 2.6, the left process may be implemented such that it sends a `CR` message, and then simply awaits either a `CC` or `DR` message, and reacts to receiving either of these messages accordingly. The right process may be implemented such that upon the reception of a message `CR` it performs a nondeterministic choice between sending a `CC` or sending a `DR`. The decisions are therefore made entirely locally within the right process. Such a local choice implementation is not possible in the case of the non-local choice example in Figure 8.1.

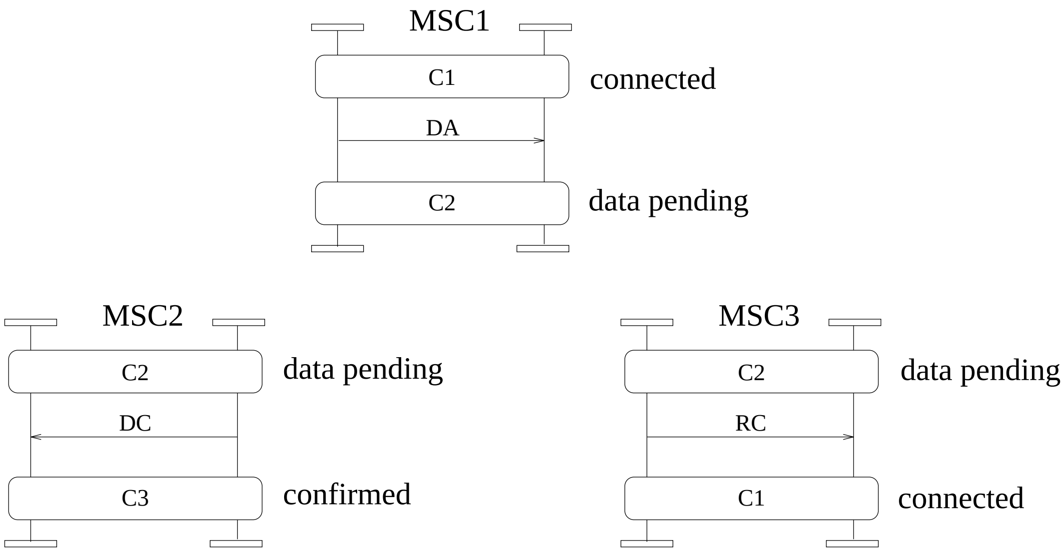


Figure 8.1: An MSC specification generating non-local control choice

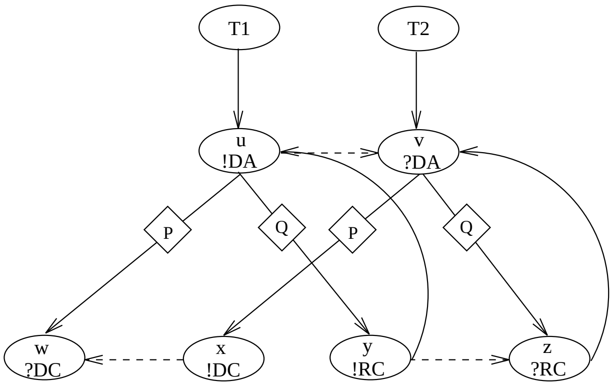


Figure 8.2: An MFG with non-local-choice nodes

involves no non-local control choice synchronisation. An example in which the control choice must be somehow communicated non-locally is given by the MSC specification in Figure 8.1, which generates the MFG in Figure 8.2. The example in Figure 8.1 could arise from part of a confirmed data exchange between two communication partners. Both partners are in a global `connected` state, represented by condition `C1` . Data may then be transferred by a *data request PDU* (`DA`). Upon receipt of the data the second process may acknowledge the receipt by a *data confirm PDU* (`DC`). Alternatively, if no `DC` is received, the data sending process may request acknowledgement from the receiving process by sending a *request for confirmation* PDU (`RC`) and then returning to the connected state, where for example the previous `DA` PDU may be retransmitted. In this example, the first and second processes must decide somehow 'together' whether they are going along the `DC` route or along the `RC` route. The MSC specifically disallows that one process could follow its left branch and the other its right branch.

We denote this required control synchronisation by labeling next-event edges at branches with predicates. We assume that there is an unbounded collection of predicate symbols $P_1, P_2, \ldots$, different from all other symbols used. Given a condition symbol for which there exist at least two MSCs starting with that symbol (e.g. the symbol `C2` in both Figures 2.6 and 8.1), we label corresponding next-event edges with predicate symbols as we construct the unfolding. For example, in Figure 8.2 we label edges corresponding to a transition through `C2` as we unfold. The labels are shown in the figure using diamonds on the edges (these diamonds here represent labels, not conditions). The labels used on the next-event edges are predicates from the list $P_1, P_2, \ldots$ that so far have been unused in the unfolding construction, say the first such ones. Labels are the same if each branch of each process with that label arises from the same MSC. In our example, labels `P` and `Q` used in Figure 8.2 would be respectively $P_1$ and $P_2$ according to this scheme, `P` being used to label the branches from MSC2 and `Q` those arising from MSC3. The modifications to the definition of MSC to allow next-event edge-labels and the formal definition of unfolding to accomplish this method of handling control branching in MSC specifications will be given in Section 8.2.3.

### 8.2.3 Definition of Transition Relation With Non-Local Conditions

The definition of the transition relation must be modified in the following way to deal with labeled conditional branches in a pbMFG (see Section 7.4). These predicates are needed to handle non-local choice conditions. Consider a transition from state $S$ to state $S'$, in which the transition occurs *through* node $u$ in process (*ne*-component) `A`. The intuitive meaning is that the transition happens because an event denoted by node $u$ occurs. We can formally define a transition to occur *through node* $u$ just in case the only out-edges added in the transition from $S$ to $S'$ are out-edges to $u$. Note that *ne* in-edges to $u$ must also be removed in a transition through $u$, and maybe also other in-edges from a predecessor of $u$.

Suppose $u$ has an incoming *ne* edge labeled `P` that is removed from $S$. The transition through $u$ indicates that the branch corresponding to predicate `P` has been taken, and represents a potentially non-local control choice. Suppose there are other *ne* edges labeled `P` in $S$. These edges are ne-edges to other processes. The branch on `P` has been taken on transition through $u$, and this information must be conveyed to other processes which have a control choice to make which includes `P`.

Consider such an edge $\langle x, y\rangle$ labeled `P`, belonging to state $S$ but in a different ne-component (thus in a different process). One way the information about the transition through $u$ might be conveyed is by retaining edge $\langle x, y\rangle$ in $S'$, but removing other branching possibilities for this process, i.e. all other *ne* edges of the form $\langle x, *\rangle$ with labels different from `P` are removed in the transition from $S$ to $S'$. This technical device retains the finite-state character, however, it is inappropriate for the following reason. Consider the case in which the node $u$ is part of a loop in the MFG. Transitions in this loop can potentially occur unboundedly often, and although this time the transition through $u$, and thus on predicate `P`, has occurred, other processes may not be on the same iteration through the loop. In particular the process containing $x$ may be on some earlier iteration, on which a different choice of control branch had been made. Thus it would be inappropriate to remove the non-`P` *ne* out-edges from $x$, on a transition through $u$ as suggested above, without knowing which iteration each process was on. Some processes may reach a particular control branch choice later than others, because of the asynchronous communication, thus some history of control branch choices must be retained.

Retaining the history of control branch choices necessitates the use of control history variables or their equivalent. History variables could be used as follows to maintain the history of control branch choices. A global list of control predicates is created by consecutive choices of branching control that are made. Define the *control vocabulary* of each process to be those control predicates which it uses. Associated with each process `B` is a sublist of the global list, consisting of some tail of the global list restricted to the control vocabulary of `B`, defined below. Call this list the `B`-*list*. The `B`-list is maintained in the following way. As a predicate `P` is added to the global list, it is added to a particular `B`-list if and only if `P` is in the control vocabulary of `B`. When control transitions through a node $u$ of `B` which involves branching control, it must transition along that *ne* edge whose predicate matches that at the head of the `B`-list, and at the same time the head of the `B`-list is removed from the `B`-list.

As we will show in more detail in Section 8.2.4 maintenance of the `B`-lists leads directly to a non-finite-state (in fact non-context-free) semantics. Thus such use of control predicates contravenes our desire to maintain a finite-state semantics.

### 8.2.4 Non-Local Choice May Imply Non-Finite-State Control

If it is argued that asynchronous message-passing is non-finite-state, the reason given is usually that message buffers are potentially unbounded, and that the state of the system must include the state of the message buffers. It is often granted that computation within a process may be finite-state. However, if conditions are allowed in MFGs, then some systems described by MFGs require non-finite-state computation within individual processes. Specifically, either unbounded history variables are required to keep track of choices made on conditions, or certain very simple MFGs with conditions that require non-local choice, such as those in Figure 8.1, must be regarded as ill-formed.

The set of MFGs in Figure 8.1 may represent a higher-level requirement on system behavior. A system implementing this behavior must conform to the requirement. Let us suppose that the implementation may perform some rudimentary amount of error recovery on, say, temporary loss of transmission, such that the system continues to satisfy the MSC requirement.

We refer to the process whose 'line' is on the left [resp. right] in Figures 8.1 and 8.2 as the 'left' [resp. 'right'] process. Suppose we label the left branches of both processes in the MFG in Figure 8.2 with $P$, to represent a choice of continuation with MSC2, and the right branches of both processes with $Q$ to represent a continuation with MSC3. Suppose now that the system executes a trace in which the successive choices between continuation with MSC2 and MSC3 are made as follows: 1 $P$ choice, followed by 2 successive $Q$ choices, then 3 successive $P$, then 4 successive $Q$, then 5 successive $P$, ...... The semantics of asynchronous communication allows the the left process to fall arbitrarily behind the right process. Somehow, the history of control choices made by the right process must be known by the left process, in order that the left process 'knows' which type of message to receive next. Suppose that the history of the last $n$ control choices in each process is retained, in a 'history variable', which we can assume is an array of length $n$. After at most $\Sigma_{k \leq n} k = n.(n+1)/2$ branch choices, the history array contains either (a) all $P$'s; (b) all $Q$'s; or (c) at most one change from $P$'s to $Q$'s or *vice versa*. It is easy to see there are $2.(n-1)$ such possibilities for (c), which along with (a) and (b) yields $2.n$ possible configurations of the history variable after $n.(n-1)/2$ branch choices.

Suppose a recoverable fault occurs, and the system is restarted with (i) all message buffers intact as when the fault occurred; (ii) all data (including history variables) intact; (iii) program counters in the same position. Suppose the fault has occurred after somewhat more than $n.(n-1)/2$ branch choices. There are only $2.n$ possibilities for the configuration of the history variables in each process. Hence there are only $(2.n)^2$ total pairs of values of both history variables. Suppose there are $k$ outstanding messages. It is easy to show that there are infinitely many possibilities of the choice history of each process compatible with each given configuration of the two history variables plus the number of outstanding messages. However, because of the construction of the example, unless the two processes start in corresponding places in the choice history, they will not fulfil the requirement expressed by the MSCs in Figure 8.1.

Importantly, not only the control branching history *of that process* but also the history of other processes that are 'further ahead', in order that the current process may make the same control branches! To ensure that this condition is fulfilled, under even these mild recovery conditions, either (a) the process must have an implicit means of communicating with the environment in order to access this history, which is held somehow by the environment in a history variable; or if not then (b) the environment must communicate this information implicitly to a process as it happens, and the process must itself retain this entire history in a variable, which must be of potentially unbounded size.

These conclusions reveal highly unsatisfactory semantic situations, since they require

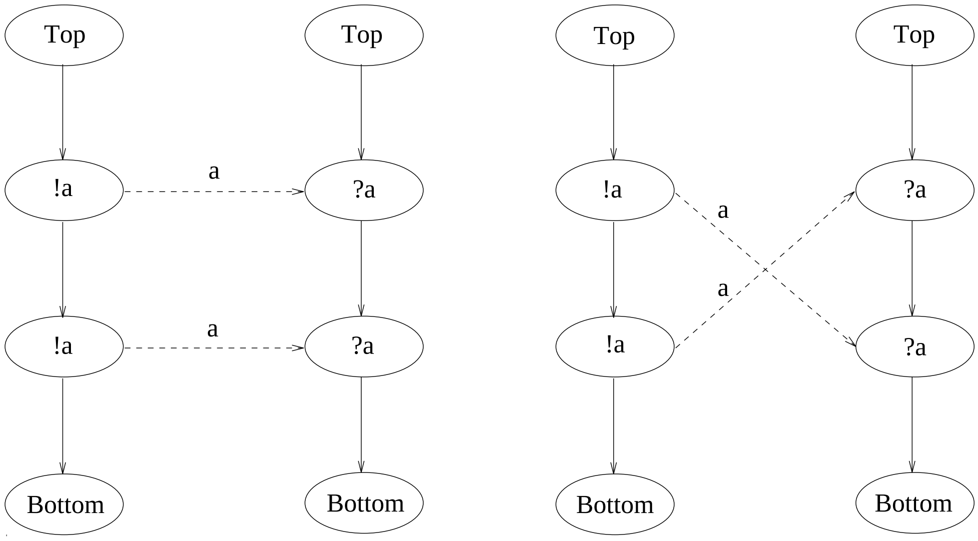


Figure 8.3: MFGs without (left) and with (right) cross-over of messages

either greater or lesser roles for the environment as an implicit information-passer, and in one case requires each process to have potentially unbounded memory to retain the choice history.

A third alternative is to simply consider the requirement as expressed in Figure 8.1 to be ill-formed. If conditions are allowed, presumably similar MFGs will not be ill-formed, if they do not require unbounded history variables. But is there any method to tell this in advance? Well-formedness should be a matter for syntax, not for semantical analysis. Hence this alternative is not really an appropriate alternative. We are left with the conclusion that allowing non-local choice leads to the requirement that processes must retain unbounded history, and thus the processes themselves, not just the message buffers, are non-finite-state. Hence allowing the syntactic ability to write choice notation which entails semantically non-local choice leads to strongly undesirable requirements on processes and implicit communication with the environment, which itself must satisfy strong history assumptions.

## 8.3 A Crossing Anomaly

In this Section, we point out an anomaly arising from allowing messages to 'cross' in MSCs. The anomaly leads us to conclude that there are properties of the environment implicitly but not explicitly required in certain types of MSC descriptions. We regard the fact that there appear to be implicit properties required of the environment as infelicitous.

The MSC standard Z.120 [33] allows crossing of signals to occur. The two MFGs of Figure 8.3 representing two simple MSCs (see Figure 9.2) describe different system behaviors. In both cases an identical type of signal is transmitted twice. The second case differs from the first in that a 'cross-over' of the messages is specified. The observable behavior of each individual process is identical in the two examples (one sends two '$a$' signals, the other receives two '$a$' signals), hence code implementing each process will be identical in both examples. However, the two examples have different sets of valid traces. The set of traces (interleaved observable events) of the first MSC is $\{<!a, !a, ?a, ?a>,$ $<!a, ?a, !a, ?a>\}$, and that of the second is is $\{<!a, !a, ?a, ?a>\}$. A system exhibiting behavior $<!a, ?a, !a, ?a>$ satisfies the first specification but not the second. However, a system exhibiting behavior $\{<!a, !a, ?a, ?a>\}$ and no other may satisfy either specification.

Since there is no difference in process code for the two examples, the different trace sets must be accounted for by a difference in the behavior of the environment. Thus, even though the environment is not explicitly represented in this specification, its properties must be invoked implicitly. One may try to resolve this problem by representing the environment explicitly, as a single vertical line like a process axis, provided it engages in message interaction with the processes (cf. [28]). However, this is no solution. Even if the environment is explicitly represented as such a third axis, firstly one can still obtain analogous process behavior by using cross-overs, and secondly this behavior may only be obtained by using a cross-over.

One criterion for a good specification method (to distinguish it, say, from the average programming language) is that all asserted properties be represented explicitly, including constraints from the environment. Our example shows that MSCs with cross-over do not pass this test. Further, even if the environment is explicit, cross-over is at best an unintuitive method of enforcing certain orderings on behavior, for which there is no other representation mechanism. Such 'programming tricks' have no place in a good description method.

## 8.4 MSC Specifications can 'Count' Receptions.

We now show that MSCs can 'count' receptions, which was pointed out in Chapter 5, where we used this ability to argue for the possibility of individuation of finitely many messages in an MFG, to be a feature which is a non-anomalous, expected consequence of a semantics, even though intuitively it may appear strange.

The example arises from our requirement that processes engage in finite-state control behavior with regard to messages (Chapter 5). Compare MSC I, Figure 2.2, with MSC IV, Figure 2.5. Both examples express a non-terminating `send` of a signal of type $a$ from the first process to the second, so at first glance one might imagine that they define the same set of traces. However, in MSC I, taking a $?a$ action removes the edge $\langle y, z\rangle$ from the state, and thus disables a further $?a$ action until after a further $!a$ action has put the edge back in the state. Thus there may be no two consecutive `receive`s, although there may be many consecutive `sends`. In contrast, in MSC IV, both $\langle w, x\rangle$ and $\langle y, z\rangle$ may occur in a state (for example, by first performing two `send` actions corresponding to nodes $w$ and $y$), and execution of a `receive` action corresponding to node $x$ removes $\langle w, x\rangle$ from the state, but not $\langle y, z\rangle$. Thus, a `receive` corresponding to $x$ may be directly succeeded by a `receive` corresponding to $z$, but then a further `send` must follow before either `receive` is enabled again. Thus, MSC IV allows two consecutive receives, but no more, and any number of consecutive sends. It should now be clear how to write down an MSC which enables $n$ consecutive `receives` of $a$, but no more, for any fixed $n$.

The interpretation of MSC I may be counter-intuitive. Firstly, why do I and IV not define the same set of traces? Secondly, why reject a trace in which there are, say 123 sends before the first receive? Formally interpreted, this suggestion says that the intuitive interpretations are those traces in which at any point in the trace, the number of `sends` is greater than or equal to the number of `receives`. This could easily be accomplished by simply counting the number of times the edge $\langle y, z\rangle$ is inserted into the edge-set defining the state. However, it is easy to see there would now be a state, for each positive integer $n$, in which the count of this edge is $n$. Thus we are no longer finite-state, indeed it is well-known that one cannot devise a finite automaton to accept precisely the suggested set of traces.

There is thus a conflict between naive intuition concerning the particular specifications here, and intuition concerning the finite-state nature of the individual processes. With all finite-state interpretations satisfying the arguments of Chapter 5, some similar such feature must arise. In our view, then, the non-intuitive interpretation of these particular MSC examples is both expected and appropriate.

## 8.5 Liveness Properties and Acceptance Criteria

Given that a system follows a finite state-transition graph, there is nevertheless a question as to whether all traces through this graph are acceptable traces of the system, or whether only a subset of them are. We showed in Section 7.5 that general MFGs define a very limited set of liveness properties. In order to facilitate the expression of a wider array of liveness properties, it is necessary to go beyond the Global State Transition Graph to consider which traces through the graph are allowed by the description (along with the liveness properties) and which are not. A standard way to express these conditions is to consider the GSTG as providing most of the definition of an $\omega$-automaton, lacking only an end-state definition, and to provide that end-state definition.

**Büchi- and Other $\omega$-Automata.** Since traces may be infinite, a finite-state semantics requires use of a finite-state automaton which accepts infinite strings. The Büchi automaton is probably the most well-known of these, and has been used in the determination of

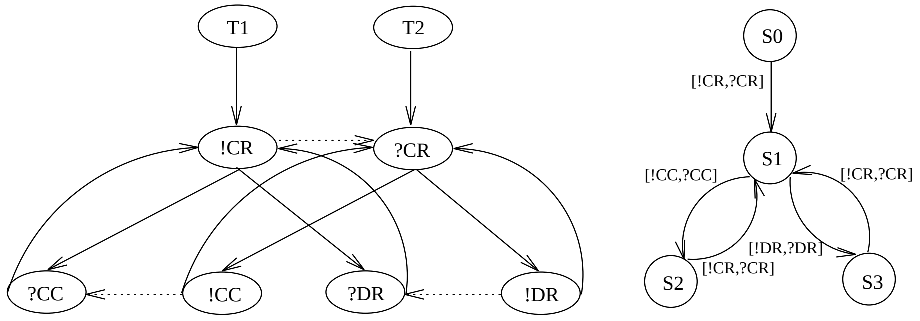


Figure 8.4: A MFG and the corresponding GSTG whose liveness may not be specified by Büchi acceptance

safety and liveness properties of distributed systems [8], [9]. As mentioned earlier, these automata are similar to ordinary finite automata, except for the acceptance condition. Büchi automata include in their definition a set of states called the *end-state set*. A (possibly infinite) string is accepted by a Büchi automaton just in case the automaton passes through an end state unboundedly often on the string (for finite strings, the final state must be an end state).

Given a general MFG specification, involving a family of MFGs with conditions, the GSTG is uniquely determined. From this graph, various different end-state definitions will define various different $\omega$-automata, each of which identifies the set of system traces specified by the MFG with the set of accepted traces of the automaton. The Global-State Transition Graph itself defines a Büchi automaton, namely the one in which the end-states are the set of all states. Even though Büchi automata define a very rich class of trace-sets (in fact, they express a $\Pi_1^1$-complete set), in order to use them flexibly one must be at liberty freely to design the state set. We are constrained by having to use the global states defined in the GSTG, and we show now that the Büchi acceptance condition does not suffice to define certain natural liveness conditions, given the GSTG states and transitions. Therefore other acceptance conditions may be preferable.

Büchi automata are sufficient to describe liveness properties of systems, provided that the specifier is free to choose the states of the automaton during the course of designing an automaton to accept precisely the desired traces. However, the global states of an MFG are specified, uniquely, by the MFG. We are not free to choose an alternative state set[3]. If one needs to specify liveness properties of an MFG description, the definition of Büchi acceptance may not suffice. Consider the GSTG on the right hand side of in Figure 8.4,

[3] It may be possible to devise a general transformation of a GSTG into another GSTG $G'$ satisfying identical safety properties such that a different set of of liveness properties may be defined for $G'$ based on Büchi acceptance. However, we do not know of one, or whether such a transformation might have other disadvantages.

derived from the MFG on the left hand side in the same Figure[4]. It represents a system which when in state $S1$ makes a non-deterministic choice between transiting into two states $S2$ and $S3$, and then returns to state $S1$. A important liveness property may be to require that the system performs a *fair* choice between the subbehaviours $S2$ and $S3$. This is expressed in temporal logic as $(\Box\Diamond at\_S2 \wedge \Box\Diamond at\_S3)$[5]. It is easy to see that there is no set of end-states under which this liveness condition is expressed by Büchi acceptance (just look at the 15 possible non-trivial end-state sets).

We conclude that the proper definition of liveness properties for the GSTG and therefore for the system described by the MFGs may be accomplished better by temporal logic formulae than by Büchi acceptance. This is because the design of the Büchi automaton is constrained by the necessary selection of a particular transition graph, the GSTG. It remains to be seen whether other acceptance criteria suffice to define automata suitable for all potential liveness criteria. In the meantime, temporal logic appears to be able to define the liveness criteria which users of MFGs may want. We have defined the precise relationship between temporal logic assertions and MFG 'executions' in Section 7.6.

[4]Note that the signal arrows in this MFG indicate *synchronous* communication, since we can make our point with this GSTG, which is much simpler than that derived from the same MFG with asynchronous communication. This may be another indication that life with synchronous communication is easier.

[5]As described in [113], $at\_S2$ and $at\_S3$ are state predicates asserting that the system is in state $S2$ or $S3$, respectively.

# Chapter 9

# Semantic Features of MSCs in Z.120

As mentioned before, standard MSCs are defined as an activity of the ITU-T standardization body, formulated in standard document Z.120. We note here some differences between an interpretation consistent with our requirements and some comments in Z.120, the current standard on MSCs. We do not regard this as a disadvantage of our semantics, but rather as a demonstration of the need for further formal analysis of the meaning of MSCs. There is no formal description of a semantics in Z.120, but some informal explanations are given. Our comments refer to what we understand from these informal explanations[1].

## 9.1 Commentary on Z.120

### 9.1.1 MSCs and SDL

Although MFGs are sometimes used in their MSC form in combination with SDL [32], we consider them an independent technique. Examples of use of MFGs independently of SDL are [140, 147, 132, 84, 137, 96, 100] (see also Chapter 3). We think it unwise to restrict ourselves to uses in the context of SDL. It is thus inflexible to define an MSC as representing a set of traces of an SDL-specification, as in Z.120. In our semantics, an MSC defines a set of traces of `send` and `receive` events for the message types specified in the MSC definition. This allows MSCs to be given meaning independent of an SDL specification, while remaining consistent with an interpretation in an SDL context, and accruing no disadvantages that we can see.

[1] A semantics for MSCs has recently been standardised and added as Annex B to the Z.120 document. For a discussion of this semantics see Section 10.1.

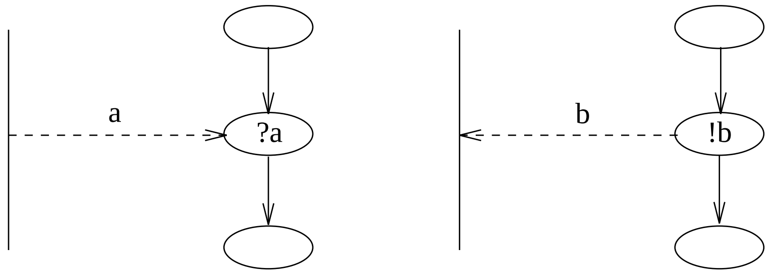


Figure 9.1: Partial MFGs with `environment receive` (left) and `environment send` (right) events

### 9.1.2 Environment

In Z.120, the behavior of the environment of a system is constrained in a particular way, namely events in the environment are considered to have an arbitrary event ordering. So far in our semantics we do not consider a distinct treatment of the communication of a system with its environment. As noted in Chapter 8, the environment may thus be forced to have implicit, and sometimes counterintuitive, properties by some kinds of MSCs, and we believe it is false to claim that the environment can always be represented explicitly by an additional process-like axis, as suggested in Z.120. However, a modification of our semantics to represent a distinguished set of environment communication events by a distinguished set of nodes in the MFG can easily be accomplished if desired[2]. Figure 9.1 shows partial MFGs communicating with the environment which is represented by a vertical solid line. Informally, the semantics of interactions of a process with the environment is explained in the following way in Z.120:

> *It is assumed that the environment of an MSC is capable of receiving and sending messages from and to the Message Sequence Chart; no ordering of message events within the environment is assumed. Although the behaviour of the environment is non-deterministic, it is assumed to obey the constraints given by the Message Sequence Chart.*

We interpret this in the following way. We assume that the processes specified by an MSC can rely on the fact that the environment will feed them with input messages whenever they wish to receive a message from the environment. Processes can also rely on the environment being ready to accept a packet at any time so that a process may send a message to the environment whenever it wishes to do so. The justification for this assumption is that we want to specify as few assumptions concerning the environment as possible, and that enforcing any constraints on its communications behaviour should be

[2]Note that a notion of *environment* in MSC specifications similar to the one in Z.120 is also known inside the ROOM methodology [137], see also Section 3.3.1.

avoided. As a consequence we do not see the need to represent a 'state of the environment' in our global system state, and no need to represent the communication of a process with the environment in the global state. Instead we introduce distinct `environment send` and `environment receive` events for MFGs. They distinguish themselves from normal send and receive events in the conditions on their enabling and the semantics of the state transitions. Also, we will not include incoming and outgoing *sig* edges related to environment communication events in the MFG. We will explain the necessary extensions to the formal definitions in 7.4 only informally here.

The partial MFGs in Figure 9.1 represent the receiving of a message of type `a` and the sending of a message of type `b` to the environment. The vertical lines representing the environment and the dashed line arrows are *not* part of the MFG. Assume the system to be in a global system state $G$. An `environment receive` event (e.g. `?a`) is *enabled* in $G$ if at least one of its incoming *ne* edges is in $G$. Note that there is no condition on an incoming *sig* edge. An `environment send` event is enabled like any other send event, i.e. if one incoming *ne* edge is in $G$. The state transition relation from $G$ to successor state $G'$ is as for normal communication events, except that an `environment send` event leaves no *sig* edge in $G'$, and an `environment receive` event removes no *sig* edge from $G$.

### 9.1.3 Conditions

Our semantics only covers global initial and final conditions. We note some reservations about their meaning, and discover technical difficulties even with these limited conditions (see in particular our discussion on non-local choice in Section 8.2). We conclude that the semantics of conditions in MSCs needs to be thoroughly investigated before they are freely admitted into the standard. However, Z.120 appears to allow conditions anywhere. We therefore refrain from treating further syntactic variants proposed in Z.120, like *non-local / non-global* conditions.

### 9.1.4 Message Types in Textual and Graphical Representation

Z.120 introduces a textual and a graphical representation for MSCs. Both are intended to be equivalent. We have some reservations concerning the way Z.120 suggests that the textual and the graphical notation are related.

In the graphical notation messages in Z.120 are labeled with names, e.g. *'off hook'* and *'dial tone on'* in the example `msc connection` in Z.120, Section 6, which describes a connection setup in a switching system. In Z.120 these are called *message names*, and this notion corresponds to the *message types* in our notation. The textual representation describes MSCs in an *instance*-wise fashion, where the instance description is embraced by an "`instance` *instance-name* ... `endinstance`" keyword construct (Note that an *instance* in Z.120 is what we call a *process* in our semantics). Message arrows are translated into

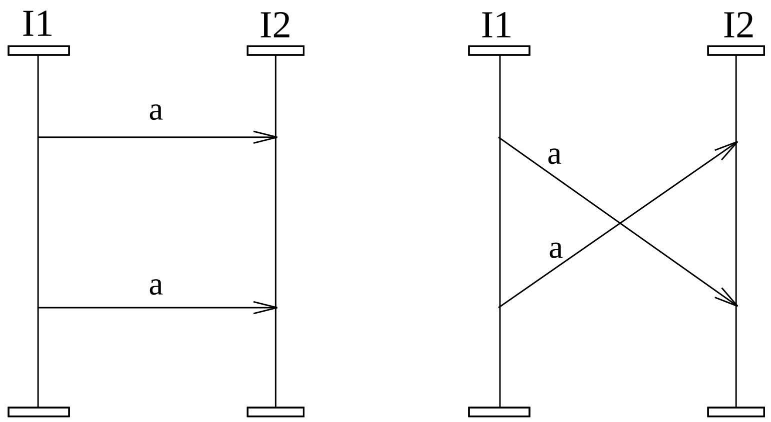


Figure 9.2: MSCs without (left) and with (right) crossing message arrows

syntactic objects of the form `out` *message-type* `to` *instance-name*.

Z.120 defines a syntax for both the graphical and the textual representation. However, there is no rule given concerning how to map the graphical representation onto the textual, and vice versa. In most of the examples in Z.120 we observe that the message types in the graphical representation are mapped by identity to the message type in the textual representation. In the example in Section 6 of Z.120 the two arrows we mentioned earlier are represented by statements `in` *off hook* `from` *env* and `out` *dial tone* `to` *env*.

However, the assignment of unique message type identifiers is crucial in some MSCs. We refer to the example *Message overtaking* in Section 6.2 of Z.120. This example is in principle identical to the MSC on the right hand side of Figure 9.2[3]. The peculiarity of both MSCs is that in both cases the left process sends two subjectively indistinguishable messages (of type `a`), and the right process receives two indistinguishable messages. If the processes cannot distinguish the messages, then the messages must be individuated by the environment. Consequently, we pointed out in Section 8.3 that the set of traces allowed by each of the two MSCs is different, which entails that the specification implies hidden assumptions on the behaviour of the environment.

Here we are concerned with the question how the graphical representation relates to the graphical representation in the context of the message crossing example. Simply mapping the message types in the MSC onto the message types in the statements of the textual representation would lead to the following MSC code:

[3]The MFGs corresponding to the MSCs in Figure 9.2 can be found in Figure 8.3.

```
msc example; inst I1, I2;
   instance I1;
      out a to I2;
      out a to I2;
   endinstance;
   instance I2;
      in a from I1;
      in a from I1;
   endinstance;
endmsc;
```

Apparently, this notation does not permit a distinction of the two MSCs in Figure 9.2, since both MSCs there correspond to the above textual representation. To overcome this deficiency Z.120 suggests informally the use of *message instance names* to disambiguate the textual representation and to ensure a unique correspondence between message outputs and message inputs. Z.120 suggests the following textual representation for the MSC with crossing arrows in Figure 9.2:

```
msc example; inst I1, I2;
   instance I1;
      out a,1 to I2;
      out a,2 to I2;
   endinstance;
   instance I2;
      in a,2 from I1;
      in a,1 from I1;
   endinstance;
endmsc;
```

The individuation of the message types has been accomplished by using an additional typing mechanism which, intuitively speaking, works like an enumeration of the message instances of identical type. The messages have been distinguished by representing the message types as `a,1` and `a,2`.

**Critique.** Z.120 acknowledges the need for an individuation of message instance names, and states the following:

> *The correspondence between message outputs and message inputs has to be defined uniquely. In the textual representation normally the mapping between inputs and outputs follows from message name identification and address specification. In case where the message name and the address is not sufficient for a unique mapping the message instance name has to be employed.*

We see a set of problems with this and related statements in Z.120:

1. The notion of a *message instance name* is ill defined. First, it is not clear whether it means a 'unique message identifier'. Second, as the word *instance* in Z.120 has a meaning which corresponds to our use of the word *process*, does this mean for example that the message instance name relates to the name of the (Z.120-)instance from which a message is coming or to which it is going, or is it used in the sense of a name associated with the instantiation of a message type?

2. Z.120 does not indicate how a process instance name should be generated, i.e. how it has to be constructed syntactically, and whether the construction mechanism has to ensure a unique assignment of names per instance type (e.g. as we indicated in the above example by suggesting it should be an *enumeration*). Furthermore, Z.120 does not specify whether message instance names should be different from other names in the MSC, e.g. message names (types), instance (process) names, or other identifiers used in an MSC.

3. Since Z.120 does not specify how to assign message instance names, it means that the same MSC can have an infinite variety of equivalent textual representations. By making names sufficiently complex (e.g. by picking them from an algebra with an undecidable word problem, so that it is undecidable whether two message instance name expressions are equal or not), one can make the problem whether a given textual representation represents a given MSC undecidable.

4. Z.120 does not say whether individuation by instance names should be done on some or on all of the messages. In particular, it is not clear whether the individuation should only be done in those cases where the textual representation is ambiguous, as in the above message crossing example, or whether it should also be done for the unambiguous cases. Deciding whether a situation is ambiguous is not obvious, and is a question of the semantics, as we have shown in Section 8.3.

5. Furthermore, we see a problem with the scope in which the uniqueness of the correspondence between ouput and input events is required to hold, according to Z.120. Assume an MSC similar to MSC I in Figure 2.2, with one message of type `a` sent from the left process to the right process, but with no initial condition and only a final condition labeled `C`. Now, assume a second MSC similar to this one with the only difference that it has an initial condition labeled `C`, and no final condition. Now, both MSCs are different MSCs, and both MSCs do not require a disambiguation of the message arrow types since they both only contain one arrow. The textual representation of both MSCs (called `MSCIA` and `MSCIB`) is therefore, according to Z.120, as follows:

```
msc MSCIA; inst I1, I2;
   instance I1;
      out a to I2;
      condition C shared all;
   endinstance;
   instance I2;
      in a from I1;
      condition C shared all;
   endinstance;
endmsc;

msc MSCIB; inst I1, I2;
   instance I1;
      condition C shared all;
      out a to I2;
   endinstance;
   instance I2;
      condition C shared all;
      in a from I1;
   endinstance;
endmsc;
```

According to Z.120 the condition `C` defines a potential composition of both MSCs. Composing the two MSCs according to the definitions in Z.120 (as far as we understand them) by considering `MSCIB` as a syntactic continuation of `MSCIA` would lead to a composed MSC with a non-unique correspondence of output and input events. Hence, a disambiguation would be required, however there is no recognition in Z.120 for the necessity of a disambiguation when composing MSCs via conditions.

We conclude that the syntax in Z.120 is not well defined with respect to the aspect of the individuation of message instance names. The issues we rose should be clarified in the standard document. In our semantics, we individuate message instances by individuating the send and receive events related to messages of the same type, and avoid the above described shortcomings. Our semantics is precise, and meets the criteria Z.120 is aiming for, as far as we can determine them.

### 9.1.5 Miscellaneous Concepts

**Sub Message Sequence Chart.** We do not handle any structuring concepts for MSC specifications. We think that whether a given MSC is a main- or a sub-MSC has no bearing on the semantic interpretation of the chart. However, state transition models avail themselves of state and event refinement and abstraction operations, c.f. the abstraction operation in Section 7.10.

**Abstraction.** We do not consider refinement and abstraction corresponding to the *process* and *sub-MSC* concepts of Z.120.

**Process model.** The process model discussed employed in our semantics is *static*, without dynamic generation of processes. However, it may easily be modified to include dynamic process generation, in the following way. A `create` instruction to generate a process is introduced as a node in the MFG. Its meaning would be similar to the semantics of a synchronous communication. The predecessor node of a node corresponding to a `create` receive event may only be a `Top` node. The termination of an instance, in Z.120 called *process stop*, can be treated similarly.

**Coregion.** Events along instance axes are totally ordered in our approach. We have no concept comparable to the *coregion* of Z.120. A treatment of the coregion concept could be accomplished similarly to the treatment of the event ordering for the environment, allowing for an arbitrary interleaving of events in the coregion with any other event in the system.

**Timer.** We do not address the timer concept in Z.120 here. However, [117] presents an algorithm for the verification of real-time constraint annotated MSCs, and in Part III we will discuss the use of real-time temporal logic for the specification of real-time constraints for MSCs.

**Parameter Lists.** Z.120 allows messages to be parametrized. However, there is no data concept in MSCs, and thus no data type mechanisms are available. We conclude that there can be only finitely many different parameter lists in an MSC specification, and we define the parameter lists to be part of the message types.

**Instance Names.** Z.120 uses in both the graphical and the textual representation process names (there called *instance names*). We do not include instance names into our semantics. However, we noted that processes form connected components of the *ne* relation in MFGs (see Section 7.2), and it is straightforward to define a labeling of each connected component of the MFG into a name alphabet[4]. Figure 9.2 shows an example of two MSCs where the instances have been named `I1` and `I2`, respectively.

**Action.** Z.120 introduces so-called *actions* which represent non-communication events. We doubt that this is a useful concept as MSC and MFG specifications seem to focus on message exchanges between processes and abstract from internal computation. However,

[4] C.f. the process name alphabet $PT$ and the labeling function *ptype* in our semantics.

actions can be interpreted similarly to the environment communication events we defined above.

## 9.2 Global System States in Z.120

According to Z.120

> *a global system state is determined by the values of the variables and the state of execution of each process and the contents of the message queues.*

This seems to contradict the informal semantics definition given in Z.120, which contains no concept of data and thus no concept of variables. Furthermore, message queues are not represented explicitly in MSCs, and we have presented an argument why this should remain so in Chapter 5 above.

There is a further human-factors argument against incorporating potentially unbounded message queues, as seemingly suggested in Z.120. Users of MSCs do not always want to think of the states of queues in order to see what their MSCs define. In contrast to many other specification methods, the appeal of MSCs lies in their relatively simple graphics for talking about sequences of messages. If determining the global system state required information about the contents of queues at any point in a trace, then 'what you see' in an MSC would not be 'what you get', i.e. there would be non-explicit semantic information concerning the history of a computation that had to be taken into account when determining the next state of the system. This contradicts the argument put forward in Section 4.7. In our MSC semantics, we only employ features which are explicitly represented in MSCs, thus we do not consider data, and we do not explicitly represent queues. We thus consider the above quoted global states argument of Z.120 to be misguided.

# Chapter 10

# Alternative Approaches to a Semantics for MSCs

The syntax of MSCs has been standardized in Z.120. Various attempts have been made to also standardise the semantics of MSCs, and ITU-T has recently added one of these proposals as Annex B to the Z.120 standard document [83]. We have some reservations concerning this semantics, and we offer a discussion and comparison with our work in Section 10.1. An approach towards the semantics based on Petri-Nets will be discussed in Section 10.2, and we will finally mention some more approaches in Section 10.3.

## 10.1 Comparison with an ITU-T Standardized Semantics

The ITU document [83] standardizes a formal semantics for MSCs based on Process Algebra. A condensed version of the semantics appears in [114]. In the remainder we will refer to the standards document as *Z.120 Annex B*, or simply as *Annex B*. The semantics is based on an interpretation of the textual representation of MSCs as defined in Z.120. Annex B defines a notion of *Basic Message Sequence Charts*, identical to our sMFGs, which are then translated into expressions of a process algebra called $PA_{MSC}$ which is based on $ACP$ [16].

### 10.1.1 Textual Representation

MSCs are graphical objects arranged according to a graphical syntax. In our semantics we interpret these graphical objects as mathematical objects, namely as labeled graphs called MFGs, and we base the semantics directly on these graphs. In contrast to that Annex B prefers not to interpret the graphical notation directly, but instead to use the textual representation according to Z.120 as the starting point for the semantic interpretation. We noted in Chapter 9.1 problems with the relation of graphical to textual representation in

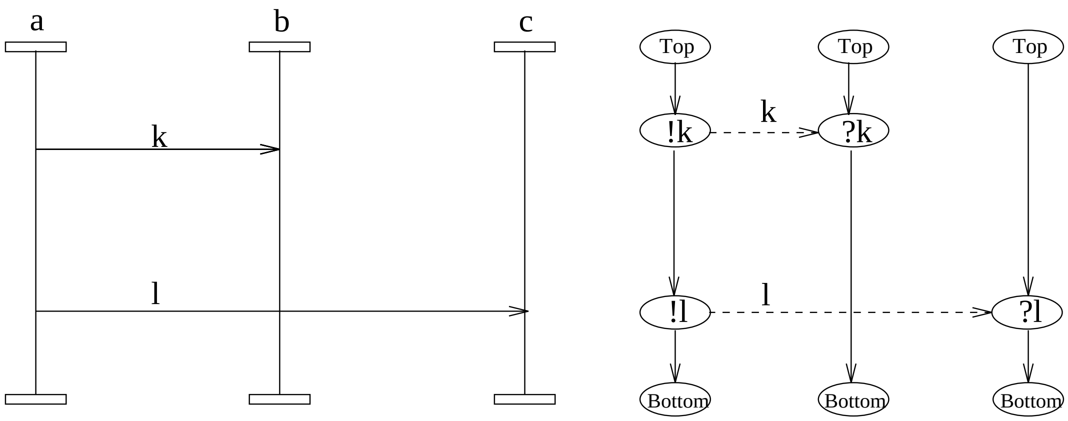


Figure 10.1: MSC / MFG *example3* from [114]

Z.120, in particular with respect to message individuation. We noticed that the attempted syntactic 'definitions' concerning *message instance names* are ambiguous and insufficient. We therefore conclude that the semantics in Annex B is not based on a syntactically well defined language. Our semantics avoids these shortcomings. It does not rely on the textual representation but interprets the graph directly.

### 10.1.2 Computation of Allowable Orderings

The semantics in Z.120 Annex B translates MSCs, as mentioned in their textual representation form, first into behaviour expressions of the $PA_{MSC}$ process algebra. To derive $PA_{MSC}$ from ACP requires substantial mathematical effort. First, the algebra $PA_{\epsilon}$ is derived from MSC, and the addition of a state operator $\lambda_M$ transforms this into a process algebra for so-called *Basic MSCs* called $PA_{BMSC}$. Then, further axioms are added to reflect particular MSC language constructs, yielding the process algebra $PA_{MSC}$. [114] gives a structural operational semantics using Plotkin rules for $PA_{BMSC}$, however, there is no definition of an operational semantics included in Annex B. Also, there is no operational semantics given for the complete algebra $PA_{MSC}$. However, it is claimed that the semantics computes the allowable orderings of sends and receives of messages:

> *[...] the semantics of Basis Message Sequence Charts is the free merge of the semantics of its constituents instances. By this construction we enable all interleavings of the message outputs and message inputs.* [114, Section 5.2]

Hence, Annex B considers a straightforward ordering semantics. But calculating the interleavings of a Basic Message Sequence Chart is easy, and does not need such apparatus, as we will show now. Consider the MSC/MFG *example3* taken from [114] and Annex B in Figure 10.1. According to Annex B the derivation of a $PA_{BMSC}$ expression for this MSC is as follows:

- A function $S_{inst}$ is applied to the textual representation of the MSC which gives the following three $PA_{BMSC}$ expressions[1]:

$$S_{inst}[\![a]\!] = out(a,b,k) \cdot out(a,c,l)$$

$$S_{inst}[\![a]\!] = in(a,b,k)$$

$$S_{inst}[\![a]\!] = in(a,c,l)$$

- The next step is the the application of the parallel composition operator $||$ to the three expressions, which is claimed to represent behaviour of the MSC.

$$S_{inst}[\![a]\!]||S_{inst}[\![b]\!]||S_{inst}[\![a]\!]$$

- The next step involves some calculations, which are neither included in Annex B nor in [114], yielding an expression in the process algebra involving 34 terms of the form $out(a,b,k)$ etc. Speculating what the nature of these computations is we assume that it is the application of the axioms of $PA_{BMSC}$ and of the state operator $\lambda_M$ to the above mentioned expression.

- The above described terms still allow undesired executions, which can be removed by applying the state operator $\lambda_\emptyset$, which gives the following expression of 9 terms:

$$out(a,b,k) \cdot (in(a,b,k) \cdot out(a,c,l) \cdot in(a,c,l)$$

$$+out(a,c,l) \cdot (in(a,b,k) \cdot in(a,c,l) + in(a,c,l) \cdot in(a,b,k)))$$

  This $PA_{BMSC}$ expression is supposed to define the semantics of the MSC.

- However, it is questionable whether one should consider a process algebra expression as giving a semantics to another formalism. Naturally, the interpretation of a process algebra expression depends on the process algebra's formal semantics. The interpretation of the above $PA_{BMSC}$ expression as a Labeled Transition System relies on the definition of an operational semantics for $PA_{BMSC}$. This definition is given as a highly complex operational semantics as Plotkin rules in [114], but not included in Annex B of the standards document.

We contrast this with the derivation of the Global State Transition Graph for the example3 MSC/MFG. The derivation is given in Table 10.1, and the GSTG can be found in Figure 10.2. Note that Table 10.1 contains the *complete* set of calculations necessary to derive the semantic model from the MFG in Figure 10.1.

[1] Because of space limitations we will not be able to introduce the whole highly complex definition of the $PA_{BMSC}$ notation here, but these are informally the most important features. The sendings of messages are described by expressions of the form *out(a,b,k)* (send a message of type $k$ from process $a$ to process $b$) and receivings by expressions of the form *in(a,b,k)* (receive a message of type $k$ from process $a$ at process $b$). $\cdot$ denotes sequential composition, and $+$ nondeterministic choice.

| | State | enabled | Successor State |
|---|---|---|---|
| $S_0$ | {*(Top, !k), (Top, ?k), (Top, ?l)*} | *!k* | $S_1$ |
| $S_1$ | {*(!k,!l), <!k, ?k>, (Top, ?k), (Top, ?l)*} | *!l* | $S_2$ |
| | | *?k* | $S_3$ |
| $S_2$ | {*(!l, Bottom), <!l, ?l>, <!k, ?k>, (Top, ?k), (Top, ?l)*} | *?k* | $S_4$ |
| | | *?l* | $S_5$ |
| $S_3$ | {*(!k,!l), (?k, Bottom), (Top, ?l)*} | *!l* | $S_6$ |
| $S_4$ | {*(!l, Bottom), <!l, ?l>, (Top, ?k), (Top, ?l)*} | *?k* | $S_7$ |
| $S_5$ | {*(!l, Bottom), <!k, ?k>, (Top, ?k), (?l, Bottom)*} | *?k* | $S_7$ |
| $S_6$ | {*(!l, Bottom), <!l, ?l>, (?k, Bottom), (Top, ?l)*} | *?l* | $S_7$ |
| $S_7$ | {*(!l, Bottom), (?k, Bottom), (?l, Bottom)*} | | |

Table 10.1: GSTG derivation for *example3*

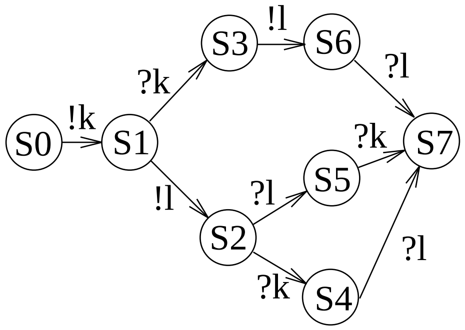


Figure 10.2: GSTG for MSC example3

Concludingly, the Annex B semantics defines process algebra and state-operator axioms, and uses these axioms to calculate a term which denotes essentially the same thing. The size of the resulting term appears to be of similar complexity as the entire state-transition calculation itself. But the state-transition calculation does not require the use of 24 axioms and a Plotkin semantics, as exemplified in Table 10.1. There seems to us to be little point in using process algebra for interpreting MSCs, when a finite-state automaton suffices.

### 10.1.3 Coverage of the Z.120 Language

The coverage of the semantics definition in Z.120 Annex B is described in the following way (c.f. [83, p. ii]):

> *The document presents a formal semantics of Message Sequence Charts [...]. The semantic constructions are introduced incrementally. This means that first the semantics of Basic Message Sequence Charts is given, and that subsequently*

> *additional features are added until the complete language is covered.*

In other words, Annex B claims to give a semantics definition of the complete MSC language as defined in Z.120. We show that this is false. As we noted earlier, the MSC standard includes *conditions*, which are to be used to 'join' MSCs, in such a way that different MSCs may describe consecutive parts of the same system execution. Basic MSCs do not include conditions. However, as indicated above Annex B pursues an incremental extension of the semantics, and in Section B.4.8 a treatment of conditions is consequently described. The function mapping MSC textual language constructs onto $PA_{MSC}$ expressions maps a `condition` symbol to the empty process $\epsilon$. Consequently, the explanatory text reads:

> *Note that the semantics of a chart containing conditions is simply the semantics of the chart with the conditions deleted from it.* [83, p. 27]

This can clearly not be considered a treatment of the condition construct, and we conclude that the claim to cover *"the complete language"* is false. This has various implications, namely that the class of MSCs described by the resulting process algebra expression has a trivial semantics, and that it only describes finite, non-iterating and non-branching behaviour. We have in our work addressed the semantics of conditions by defining a composition operation and by interpreting the branching and iterating MFGs which potentially result from the use of conditions.

### 10.1.4 Finite-Stateness

We have argued at length in Chapter 5 that MSCs are inherently finite-state systems. The semantics in Annex B is finite-state, for a trivial reason: as we argued above, it only describes finite behaviours. However, extending the semantics in Annex B to infinite behaviours, e.g. by giving a meaning to iterating MSC specifications due to the use of conditions, would require means of expressiveness for infinite behaviours. In process algebras this is usually accomplished by the use of recursion, and it is mathematically a non-trivial task to restrict recursive process definitions to only express finite-state equivalent systems. It is equally difficult to show that a recursive process definition corresponds to a finite-state system. Furthermore, there is evidence that the semantics as described in Annex B is inherently *non-finite state*, when extended to cover infinite behaviours. According to [114, Section 4.2]

> *this operator* [the state operator $\lambda_M$, see above] *remembers all message outputs that have been executed in a set $M$ and only allows a message input if its corresponding message output is in that set.*

We see a few problems in this definition.

- First, we doubt that this operator can allow the retention of complete execution histories even for finite MSCs. The definition of the operator as given in Table 4 of [114] specifies that when applying it to an *out(a,b,l)* expression, then the string representing this expression will be *added* to the set $M$ of sent but not yet received messages. In analogy to that the application to an *in(a,b,l)* expression will remove the *out(a,b,l)* expression from $M$. Now, let's look at the left MSC in Figure 9.2. Applying the definition of $\lambda_M$ to this example would mean that there is an allowable execution in which two times *out(a,b,l)* will be added to $M$, followed by just one removal of *out(a,b,l)*, which means that two message sendings will be followed by one reception. It is obvious that this is not the intended semantics of the MSC in Figure 9.2. Although the functioning of the $\lambda_M$ operator is similar to the adding of a *sig* edge to the current global system state in our semantics the distinction between communication events and signal types in our semantics avoids the described problem[2].

- Second, if the semantics in Annex B was extended to capture infinite executions like our semantics, and the $\lambda_M$ operator was actually defined in such a way that it could remember a potentially unbounded number of sends of messages, then this would lead to a non-finite state system[3]. Furthermore, as we point out in Chapter 5 and Section 8.2, there is nothing in the state of any telecommunications system implementing a given MSC corresponding to keeping track of the system's complete execution history.

We conclude that the semantics in Annex B does not extend nicely to infinite systems.

### 10.1.5 Pragmatics

When discussing the application of formalisations of MSCs the Z.120 Annex B document states:

> *Tool builders can use the semantics for derivation of prototypes directly from the definitions provided or they can base their computer applications on these definitions.* [83, p. ii]

Exactly how tool builders are supposed to derive prototypes from MSCs is not specified. For example, BMSCs include no branching behavior, and we've noted that the branching behavior of general MSCs is incompletely and confusedly specified by Z.120. However,

[2] In our interpretation we would map the the signals in Figure 9.2 to two pairs of communication events (*sig* edges), say $< u, v >$ and $< x, y >$ where the signal type of both edges is $a$. We can thus add two distinct *sig* edges to the current system state, $< u, v >$ for the first message $a$, and $< x, y >$ for the second.

[3] For a similar argument see our discussion of non-local choice and the resulting need for potentially unbounded history variables in Section 8.2.

most software includes at least one branch, with usually a clear meaning. Z.120 gives no clue how to specify such a branch. We conclude that the statement that *software developers may derive prototypes from ...* is gratuitous.

More particularly, we doubt that it will be possible to derive system prototypes from MSC specifications because their behaviour will be of a much higher degree of complexity than any behaviour which could be expressed by an MSC. However, we note that MSCs may be useful for simulations of the message passing of a system, or for deriving object control state machines as in some object-oriented design methodologies (c.f. [132]). We feel that the models for MSC specifications resulting from our semantics (the Global State Transition Graph) is much closer to any use in a software tool than those resulting from the semantics in Annex B, and therefore more practical. This for two reasons:

- The main application of a formal semantics lies in the use of the generated semantic model for verification purposes. Nowadays, most of the practical verification is done by so-called *model-checking* which entails traversals of the state graphs of systems (see for example in [74]).

- As an important by-product software engineering tools may use a formalization to execute a specification by traversing the state transition graph. This may be helpful when wanting to debug a specification, or when animating the system's behaviour.

We conclude that the ability to traverse a state transition graph is an important requirement for a semantics to be practical. It has been argued that a process algebra model can be translated into a state-transition model. However, this is a non-trivial operation[4], and we do not see a point in doing so when a direct translation into a state-transition model can easily be done, as we have demonstrated by our approach. This means that the semantics in Annex B is much less practical than our approach, and therefore much less helpful in serving as a basis for computer applications, as required in Annex B.

### 10.1.6 Communication Mechanism

The Semantics in Annex B describes only asynchronous communication, which is the only communication mechanism known to Z.120. However, we believe that our semantics, which covers both synchronous and asynchronous communication, is more flexible and therefore has a broader scope of applications, as for example in object-oriented methodologies (see Section 3.3) which make use of both mechanisms.

[4] C.f. the work on translating LOTOS specifications into state-machines described in [86].

## 10.2 A Petri-Net based Approach

An approach to MSC semantics based on Petri Net interpretations was suggested in [64]. The example given there yields an interpretation which is easily and intuitively translated into a finite-state automaton. The authors do not address many of the questions that concern us in our approach to a semantics, such as varying interpretations due to different liveness properties, problems of synchronising non-determinism under composition by conditions, and assessing the expressive strength of MSC specifications, and they do not address the question of how to handle composition under conditions, as shown in Section 7.2.3. One of the authors has suggested that a Petri Net-based semantics will not in general be finite-state [131]. We see this as a problem, since we argue in Chapter 4 that MSCs are inherently finite-state. Furthermore, using a semantics not based on finite-state methods may cause difficulty with proper use of MSCs inside some tools which, like the GEODE tool, assume MSCs to be finite-state.

## 10.3 Miscellaneous Approaches

Some further approaches to a semantics for MSCs focus on formalisations, data structures, and operations on MSCs [39] [40] [144]. However, they do not provide an operational semantics for MSCs in the way we do in our work. A formal interpretation of SDL appears in [28], where MSCs are introduced as ways of specifying 'traces' with which a given SDL specification may be compared. MSCs in [28] seem to be regarded as more of a test-derivation language, and are not themselves given a formal semantics therein.

[48] describes a different process-algebra based approach to the semantics of MSCs. It also considers only condition-free MSCs, and therefore only simple finite behaviors. A message corresponds to two separate `send` and `receive` events. This is a simple translation of a finite MSC into a finite collection of partially ordered events.

# Part III

# Quality of Service Specification

# Chapter 11

# Introduction

Many telecommunications systems engineers find it convenient to specify functional properties of their systems using state-transition based formal description techniques like SDL or Message Sequence Charts (MSCs). However, the expressiveness of these techniques does not capture Quality of Service (QoS) requirements because many of these rely on real-time constraints which SDL and MSC specification do not allow to express. However, suitably extended temporal logics allow for a description of these requirements. We introduce a method for the integration of functional system specifications given in SDL or MSC with temporal logic based specifications of QoS requirements, and we call the resulting specifications *complementary specifications*. We show how SDL and MSC specifications fit together with temporal logic and real-time extended temporal logic specifications. Then we give examples of real-time related delay bound, delay jitter, and isochronicity constraint QoS specifications. We discuss how our method helps in the specification of: system performance to QoS mapping problems, of QoS negotiation mechanisms, and of QoS monitoring. Finally we hint at methods for the formal verification of QoS specifications.

## Properties of Telecommunications Systems

In telecommunications systems design, a lot of work has been devoted to using formal descriptions for the specification of functional properties of the designed systems. SDL [32] and Message Sequence Charts (MSCs) [33] are frequently used for this purpose, and they will therefore play a central role here[1].

Properties are sets of observable sequences of events in a system. Many authors have introduced notions of *safety* and *liveness* properties, see for example [7, 8]. Safety properties ensure that *nothing bad happens*, whereas liveness properties ensure that *eventually something good will happen*. Informally, the requirement that *a connection has to be*

[1]For an overview of the use of formal description techniques in telecommunications systems engineering see for example [108, 19, 145].

*established before the data exchange may begin* is describing such a functional property, according to the classification in [7, 8] a *safety* property. Another property can be described by the requirement that *whenever a message has been sent, then it will eventually be received* which is according to [7, 8] classified as a *liveness* property. We consider safety and liveness properties as well as all combinations of these to be *functional* properties of a system[2].

In addition to the functional aspects real-time mechanisms have been used in the design of protocols in order to ensure progress of the system, and also in order to detect errors like message losses or unavailability of resources. For the use of the SDL timer mechanism in order to achieve the second goal see for example [19]. However, as we will show later the asynchronous SDL timer mechanism is unsuited to ensure progress of a system. This has also bearing on QoS aspects as far as they are related to *hard* real-time bounds [55], e.g. delay and delay jitter bounds for multimedia services.

## Overview

In Chapter 12 we investigate the SDL timer mechanism and we observe limitations in its expressiveness; in particular with regard to the specification of real-time bounded response properties. We recommend the use of complementary real-time temporal logic specifications to remedy this pitfall. This requires providing a model for SDL specifications based on which temporal logic formulas can also be interpreted.

In Chapter 13 we introduce a state transition model and show how SDL specifications can be interpreted based on this model. This interpretation is similar to the interpretation of SDL specifications as communicating extended finite state machines, for which some suggestions exist in the literature (see Chapter 13). However, the existing proposals are either informal, incomplete, or they do not adequately capture the SDL semantics, in particular as far as the potentially iterative structure of SDL process transitions and the particular semantics of the SDL `INPUT` statement are concerned.

In Chapter 14 we show how temporal logics can be used in combination with SDL specifications when these are interpreted based on the state transition model as defined in Chapter 13. The underlying idea is that both the SDL specification as well as the temporal logic specification constrain the allowable behaviour of the system, and we require both specifications to be satisfied by a system. We say that the temporal logic specification is a *complementary* specification of the basic SDL specification. Temporal logic, however, allows for the specification of state-based properties whereas we are often interested in specifying properties of sequences of events, such as `INPUT` and `OUTPUT` events. We show in Chapter 14 how these events can be defined in terms of state predicates.

[2]However, we should note that like MFG and MSC specifications SDL specifications do not express liveness properties.

In Chapter 15 we specify a range of different real-time constraint based QoS requirements complementing SDL and MSC specification examples. These include service response and message transmission delay bounds, delay jitter bounds, isochronicity related requirements, and requirements on transmission rates.

In Chapter 16 we exemplify how some QoS related mechanisms can be specified using our approach, like QoS negotiation and reaction to QoS guarantee violation.

We conclude with a discussion in Chapter 17. Section 17.1 exemplifies the use of our specification method in the context of system performance to QoS mapping problems. In Section 17.2 we discuss some issues concerning the formal verification of QoS requirements.

**Related work.** A wide range of literature is available on many of the topics discussed in this part of the thesis. We will mention the appropriate references in places where they are relevant. However, some general literature on QoS should be mentioned here. The proposed standardisation of QoS concepts in the ISO/OSI and ODP context is documented in [82]. [90] gives a complementary overview over QoS topics.

# Chapter 12

# A Critique of the SDL Real-Time Mechanism

SDL has a built-in real-time mechanism, relying on an asynchronous timer mechanism. We will argue here that this mechanism is inexpressive with respect to the most important class of real-time requirements, namely hard real-time or bounded response constraints (see for example [55] [14]). We will briefly explain why this sort of constraints is an important requirement for real-time systems, and we will then address the unsuitability of the SDL mechanism.

## 12.1 Real-Time Requirements

We introduced liveness properties as properties of a system which state that "something good will *eventually* happen". This class of theoretically interesting properties has proved to be of limited practical use. By asserting that one can rely on the fact that when one has requested a service, the request is eventually going to be served does not exclude the possibility that one may need to wait a finite but *apparently* limitless period of time for the servicing of the request[1]. It is theoretically possible to specify situations which are perfectly "legal" from a liveness point of view but which could result in the user having to wait for an impractically long period of time before the servicing of his request (e.g. exceeding human life expectancy !). Using liveness assertions in such an empirical manner are clearly of little value and alternative approaches have to be considered if apposite use is to be made of their benefits. It should also be pointed out that a closely related problem associated with pure liveness requirements is the fact that liveness properties are not testable [92], and which provides still further proof of the limited practical use of this class of properties.

[1]For a similar argument in the context of a lift system specification see [116].

To overcome this problem, real-time models enforcing progress have been introduced. These models introduce the idea of the *urgence* of events which means that events are required to happen after a specified period of time. This implies that we need to introduce a notion of *time* into the so far purely untimed state and event sequence model. A suitable timed execution model for our purposes is the model of timed traces [14], where steps in system traces are labeled with monotonically increasing timestamps. The requirement that a service request be serviced within $t$ time units of the current moment in time is expressed in the timed trace execution model as:

> the request will be serviced in a state $S_i$, $i \geq j$, so that the timestamp $ts(S_i)$ differs from the current time stamp $ts(S_j)$ by not more than $t$ time units.

We call such a requirement a *bounded response* requirement [67]. Bounded response requirements are crucial in many control system specifications, e.g. in communication protocols and safety-critical systems [55][2].

## 12.2 The SDL Real-Time Mechanism

Real-time is introduced into SDL by an *asynchronous timer* mechanism[3]. An SDL specification can access the value of a global clock by reference to a variable called `NOW`. This variable always refers to the current moment in time.

*Timers* are similar to variables: they can be set, reset, and they can expire. A timer can be "set" in the course of a state transition by the SDL command `set`, usually being set to a value greater than the current value of the system clock. For example, the SDL command `set(now+t, T)` would be used to set the value of a timer called `T` to a point of time $t$ time units greater than the current moment of time. This command is implemented by synchronously reading the *global time* value in variable `NOW` and adding the *time distance* value $t$, which yields the value to which the timer is set. We call a process which sets a timer the *timed* process.

The set timer is administered by a *timer process*. Each time a timed process sets a timer an instance of the timer process will be generated. The timer process runs independently and asynchronously from the timed process. The timer process continuously compares the value to which the timer is set with the current global time value. When the value to which the timer is set is reached or exceeded, the timer process communicates the expiry to the

[2] It is interesting to note that from a theoretical point of view the liveness property that the event will eventually be serviced becomes a safety property when it is transformed in such a bounded response requirement [67]. This is due to the fact that in every state of the system it is possible to evaluate whether the requirement is satisfied by the system execution up to the current state, or not. This is not possible for liveness properties.

[3] For an instructive explanation of the SDL timer mechanism on which we base our discussion here see [19].

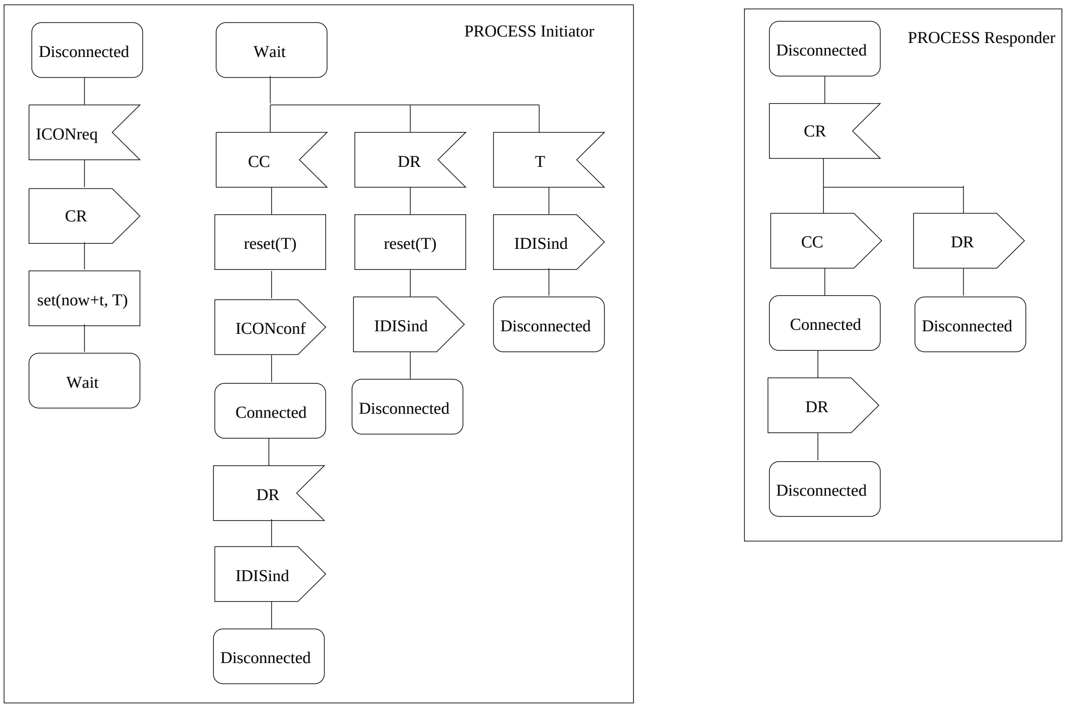


Figure 12.1: SDL specification of the INRES connection establishment

timed process by placing a *timer signal* at the end of the input queue of the timed process. The timer signal is then treated like any other input signal, and the timed process may consume the timer signal from its input queue whenever it is at the head of the queue, and react accordingly. Timers may also be *reset* by the timed process in which case the timer process *deactivates* the respective timer. If the timer had already expired by the time the reset gets executed, the corresponding signal will be removed from the timed processes' input queue, which means that the pure FIFO queue access strategy (stating that a queue can only be accessed through its head) is violated in this case. To summarize, a "reset" may be caused explicitly by a `reset` command, or implicitly by consumption of the timer signal.

**Example.** In Figure 12.1 we present an SDL specification of the INRES connection establishment protocol, using the SDL timer mechanism. We consider an *initiator* and a *responder* process, the initiator plays the central role. The behaviour of the responder is obvious. The initiator transits from the `disconnected` state to the `wait` state upon reception of a `ICONreq` service primitive signal from the service user. In the course of this transition a `DR` PDU is sent to the responder and the timer `T` is set to the value of the global time `NOW` plus the time distance value `t`. It should be noted that the state transition from `disconnected` to `wait` is considered to be an atomic action of the SDL execution model. When the system is in the `wait` state the initiator may:

- receive a `CC` PDU from the responder, which indicates that the responder accepts the connection establishment, and it may then transit to state `connected`,
- or it may receive a `DR` PDU which indicates a rejection of the connection request by the responder,
- or it may receive a timer signal `T`.

After the reception of `DR` or `T` signals the initiator indicates closing of the connection or unsuccessful connection establishment respectively by issuing a `IDISind` service primitive to the Initiator-user.

This specification example represents a very common usage of the SDL timer mechanism in protocol specifications. It is generally assumed that the above use of the SDL timer ensures progress of the system, and thereby ensures its liveness. The underlying assumption is that the expiry of the timer forces the timed process `Initiator` to react, within a bounded time frame. We will now show that this assumption is false.

## 12.3 Critique

We claim that the above described timer mechanism is unable to express real-time properties relying on the urgence of events, as seen, for example in the bounded response requirement introduced in Section 12.1. We consider and comment upon the following aspects concerning the timer mechanism in SDL.

- Processes receive the timer signals through their input queue. This implies that the reaction on the expiry of the timer is decoupled from the expiry itself. We may therefore only infer that the system reacts *some time after* the timer expires.
- Furthermore, no estimation can be made for the time it takes to consume all events in the queue which may (potentially) have arrived earlier than the timer signal. This means that the time span between adding the timer signal to the input queue and when it is at the head of the queue is finite but unbounded.
- The interaction between the input queue and the process is asynchronous. Even if a timer signal has arrived when the input queue of the timed process was empty it cannot be guaranteed how long it will take for the timed process to actually consume the timer signal and react accordingly. However, if one assumes a basic trivial liveness assumption (namely that every enabled transition will eventually be taken) then it is at least guaranteed that the reaction will *eventually* happen.

More formally, the argument is stated as follows. Assume that at time $T_{now}$ a process sets a timer $\tau$ to a time value $T_{now}+T_v$. When the timer process expires (the system clock reaches the value $T_{now} + T_v$), then a *timer expiry signal* (hereafter called a *timer signal*) is generated. The timer signal is placed in the timed processes' input queue some $T_1 \geq 0$ time units later. It will be consumed (i.e. removed from the head of the receiving process' input queue) some $T_2 \geq 0$ time units later by the timed process. Note that a finite but unbounded number of messages can be in the input queue of the timed processes ahead of the timer signal. Note also, that depending on the structure of the specification, the timed process may run into some state from which it will never neither reach a reset instruction nor a timer signal input statement. This will imply that at some point the timer signal will reach the head of the timed processes' input queue, but it will be discarded and the system will then not react on the timer expiry at all. Finally, the earliest reaction to the timer expiry will happen $T_3 \geq 0$ time units after the consumption of the time expiry signal. This means that the delay $\delta$ between the point of time when the timer expires, and the moment at which the SDL specification reacts to the expiry, can be estimated as $0 \leq \delta \leq T_1 + T_2 + T_3$, and there is no upper bound for the value of $T_1 + T_2 + T_3$. This leads to the following points of criticism:

- As there is no upper bound for the value of $T_1 + T_2 + T_3$, it is not possible to specify a bounded response requirement using the timer mechanism.

- The only property, however, which can be expressed is that *before* the global time has reached the value $T_{now} + \delta$ the timer signal `T` will not be consumed. In the Inres example this means that when the system remains in state `wait`, before the time $T_{now} + \delta$ has been reached a `IDISind` signal will not be issued, unless the system has received a `CC` or `DR` signal. It is questionable, however, whether this is a useful property.

We conclude that it is not possible to enforce system progress or liveness by using the SDL timer mechanism, and therefore that SDL does not provide an expressiveness for specifying time-critical systems. This implies that SDL is not capable of specifying many important real-time QoS requirements which rely on real-time bounds.

## 12.4 Remedies

A large number of proposals for real-time formalisms and semantics has been made in the literature, and space limitations do not allow for a complete overview here. SDL is a specification language based on a state transition model, and the transitions are described using a programming language-like formalism. It therefore seems obvious here to discuss approaches to providing SDL with real-time expressiveness based on automata models and temporal logics. We see in particular the following possibilities for a remedy of the above described shortcoming of the SDL real-time mechanism.

- It has been suggested that to describe real-time constraints on state transition based systems may be achieved by so-called *timed automata* [118] [146, 111] [12, 13]. A common feature of these approaches is that transitions of automata are being attributed by time constraints. Time is introduced into these systems by clock variables, and transitions may only be taken if associated constraints on the clock values are satisfied. We do not directly pursue this approach because it requires the translation of an SDL specification into a state-transition system, and then to define the clock values and constraints on them based on the state-transition model.

- [18] suggest labeling SDL transitions by transition time and transition rate parameters. These parameters refer to time constraints on repeated executions of a transition. We need a much more flexible mechanism here, allowing us to specify timing constraints referring to different transitions (c.f. the discussion of the INRES example in Section 12.3), or even to events or transitions in different processes. A further deficit of the approach described in [18] is that no formal semantics is introduced for the suggested constructs.

- [15] suggests a method to *'take a programming language off the shelf and upgrade it into a real-time programming language'*. This upgrade is accomplished by the introduction of clock variables and a so-called *guarded wait* statement. Applying this suggestion to SDL is certainly a very appealing idea, in particular because the `NOW` variable in SDL already offers a time variable concept, but we refrain from changes to the SDL language for the time being.

- Finally, the use of real-time extended temporal logics has been put forward by many authors to specify and reason about real-time constraints for reactive, state-transition based systems [67, 14] [1] [87] [125]. The logic formulas in most of these approaches allow to refer to state predicates representing time values (a.g. the values of distinct clock variables), and in some of the approaches the modal operators are extended by time indices. This approach enjoys a high degree of flexibility in the specification of real-time constraints, and we will therefore pursue the idea in the following Chapters.

# Chapter 13

# A State-Transition Model for SDL Specifications

In Chapter 12.1 we suggested using a combination of SDL specifications and temporal logic formulas to express real-time requirements for which SDL does not have the necessary expressiveness. In this Chapter we will connect the semantic models underlying SDL specifications and temporal logics by interpreting SDL specifications as state transition systems. We describe states of these state transition systems in terms of logical predicates.

## 13.1 Introduction

In this Chapter we define a rudimentary computational model for SDL specifications, a so-called *Global State Transition System* (GSTS). This model has certain similarities with the state transition model GSTG introduced in the semantics for MFGs in Chapter 7. As in the GSTG model we define the global state in the GSTS model to be determined by the local state of the processes plus the state of the communications in between processes. However, we note that as opposed to the GSTG model for MFGs the GSTS model for SDL specifications is not finite. This is due to the fact that SDL specifications contain data variables over infinite domains, and that message queues between processes may contain a potentially unbounded number of messages.

The unwinding of the GSTS model will describe all admissible sequences of states of an SDL specification. In describing sequences of states, the model also describes sequences of state transitions, which are in turn triggered by events in the system. The most interesting of these events are inputs and outputs of signals. The GSTS defines a computational model for SDL specifications, the computations can be derived from the GSTS by unwinding all valid state sequences. The computations described by the GSTS will later on serve as models for what we call *complementary* temporal logic specifications. These specifications are complementary because they rely on the GSTS model and express all those properties which SDL specifications cannot express, like liveness properties, real-time constraints, etc. Only those specifications which satisfy both the properties expressed by the SDL specification as well as the properties expressed by the temporal logic specifications satisfy the composed specification. The temporal logic specifications may thus be considered a filter on the computations defined by the GSTS model.

It should be emphasized that the goal here is not to define a, or as some would prefer to say, *yet another* formal semantics for SDL. A formal semantics has been standardised in [32], and alternative formal treatment of the SDL semantics have been widely discussed, e.g the approach based on stream functions in [28]. The goal of the work presented here is to make SDL specifications available to a complementary treatment by temporal logics, and none of the above cited formal semantics provides for that directly. Therefore, we understand this work as putting SDL into a suitable model theoretic context, based on an intuitive understanding of the elsewhere specified formal semantics. For the remainder of this Part we assume that the reader has some familiarity with the SDL syntax and semantics. The main components of the GSTS model are:

- *Process control and data manipulation.* This component describes the local behaviour of an SDL process. An SDL process executes transitions between *symbolic* states. In the course of these transitions, variables will be manipulated and process control progresses.
- *Communication.* SDL processes communicate via potentially unbounded queues, and each SDL process has exactly one input queue handling all incoming communication from any other process[1]. We describe the local state of an SDL process as the combination of current values for the data variables, the point of local process control, and the state of the input queue.
- *Global System States and State Transitions.* The global system state (GSS) is the product of all local states of all processes of an SDL specification. SDL processes run concurrently. The approach we take to model this concurrency aspect is an interleaving semantics, which is consistent with the SDL semantics as defined in the standard document [32]. In a given GSS a number of transitions of the individual SDL processes may be enabled. A nondeterministic algorithm decides which one of the enabled transitions is selected for execution. The execution of a selected transition changes the local state of the respective process. Furthermore, in case the selected statement is an `OUTPUT(X)` statement, the local state of the receiving process is modified by adding the signal `X` to the tail of its input queue. The result is a new global system state.

[1]For reasons of conciseness we do not address inter-process communication mechanisms like *viewing* or *remote procedure call*. However, a treatment of these communication mechanisms within our framework is straightforward.

**Overview.** We proceed in three steps. First, in Section 13.2 we define the notion of a process state transition system (pSTS). A pSTS has components similar to an extended finite state machine, plus a process unique input queue which we model as a local data structure. We also define a transition relation and the notion of an admissible state sequence for pSTS here. The interpretation of pSTS as SDL processes is presented in Section 13.3. We explain how to map SDL process states to symbolic states in the pSTS, and how to formally treat `INPUT` statements, variable assignments, decision statements, and iterating transitions in the pSTS model. In Section 13.4 we demonstrate how to augment `INPUT` and `OUTPUT` statements to state-propositions, which helps us later in using references to these events in temporal logic specifications. In Section 13.5 finally we define global state transition systems (GSTS). SDL specifications consist of collections of concurrent processes. In the definition of the pSTS we attributed the message queues to the receiving process, thus a global state can be considered to be the cartesian product of all local pSTS states. We show in this Section also how to formally handle `OUTPUT` statements, and we define global system state sequences which yields the computational model over which we will later interpret temporal logic formulas.

**Related Work.** Our definitions here are close to the definitions of state transition systems in [113] where they are called *Basic Transition Systems*. They can be seen as a generalisation of *Finite State Machines* (FSM) and *Extended Finite State Machines* (EFSM). A partly formal introduction to FSM and EFSM can be found in [108]. EFSM are usually distinguished from FSM in that they allow the explicit representation of data variables and symbolic control states. Typically, due to the infinite range of data variables EFSM represent infinite state spaces. The modeling of SDL processes as EFSM has been suggested in [19]. However, as we will see later the mapping of SDL process transitions as informally described in [19] is too coarse in order to represent the structure and the complexity of the computation occurring in the course of an SDL transition. A similar criticism applies to the formalization given in [135]. [110] contains a formalization of SDL based on FSM, hence without treating data variables over infinite domains. Formalizations of EFSM can be found in [74] (where the state space is finite by limitation of the range of data variables and variables representing the state of communication channels to finite domains), and in [34] and [88] (from where we take part of our formalization). [25] describes and formalizes the use of queues to model the collective behaviour of concurrent FSM which communicate asynchronously via queues (there called *protocols*). We use a similar approach when constructing a global state transition system representing an SDL specification. [27] present a temporal logic based semantics for the specification language Estelle.

## 13.2 Process State Transition Systems

The process state transition systems (pSTS) we define here represent an SDL process by a set of symbolic states, a set of program variables (consisting of control and data variables), and by its interactions with the environment (input and output of signals). The 'logic' of an SDL process is encoded in its state transition relation. A state transition relates a current state of the system (involving enabling conditions on the current values of the program variables) and an input signal to a successor state (including an update of the program variables) and an output signal. For a definition of the mathematical notation we use here we refer the reader to Appendix A.

### 13.2.1 Definition Process State Transition System (pSTS)

A Process State Transition System P is defined as a tuple $(S, D, V, O, I, Q, T, C)$ where

$S$ is a finite set of *symbolic states*,

$D$ is an n-dimensional linear space where each $D_n$ is an *interpretation domain*,

$V$ is a finite set of *program variables*, $V = \{\pi, v_1, \ldots, v_n\}$ where $\pi$ is a *control* variable ranging over elements of $S$ and $v_1, \ldots, v_n$ are *data* variables so that $v = (v_1, \ldots, v_n) \in D$,

$O$ is a finite set of *output signal types*,

$I$ is a finite set of *input signal types*,

$Q$ is a linear sequence $q_1, \ldots, q_m$ (in the standard mathematical sense) of elements from $I \times D$ which we call *input queue*,

$T$ is a *transition relation*, with $T : S \times 2^D \times Q \to S \times 2^D \times Q$, and

$C$ is an *initial condition* on $S \times 2^D \times Q$.

A *state* $s$ is a function $s : V \times Q \to 2^S \times 2^D$ assigning a value to every variable in $V$ and to Q. By $s[x]$ we denote the value of variable $x$ in state $s$. We denote the set of all variables by $V$. Apparently, $V$ can be infinite.

### 13.2.2 Transition Relation, Admissible Sequences, and Reachable States.

We associate a set $\mathcal{T}_T = \{\tau_1, \ldots, \tau_m\}$ of *transitions* with the transition relation $T$ of an pSTS. With each transition $\tau_j$ we associate a pair of state propositions $P_j$ and $Q_j$ and we call $P_j$ a *precondition* and $Q_j$ a *postcondition* of transition $\tau_j$[2]. We assume the existence of a satisfaction relation $\models_P$ which relates assertions about the system state to system states for a given pSTS $P$[3]. In particular, we write $s \models p$ iff state $s$ satisfies state-proposition $p$[4]. Now, in order to relate states $s$ and $s'$ we say that $(s, s') \in T$ iff

$$(\exists \tau_j \in \mathcal{T})(s \models P_j \wedge s' \models Q_j).$$

Let $\sigma = s_0, \ldots, s_k$ denote a finite sequence of states. We call this sequence *admissible* iff $(\forall 0 \leq j < k)((s_j, s_{j+1}) \in T)$. This definition extends to infinite sequences in the obvious way. A state $s_k$ is a *reachable* state iff the sequence $\sigma = s_0, \ldots, s_k$ is admissible and $s_0 \models C$, i.e. $s_0$ is the initial state. In state formulas, when referring to states $s$ and $s'$ with $(s, s') \in T$ we sometimes denote $s[v]$ by $v$ and $s'[v]$ by $v'$. In order to express that a transitions $\tau_k$ is *enabled* in a state $s$ we write $s \models en(\tau_k)$ iff $s \models P_k$. For a pair of states $(s, s')$ we say the transition $\tau_l$ has been *taken* iff $s \models en(\tau_l)$ and $s' \models Q_l$. We denote this by $ta(s, s', \tau_l)$

### 13.2.3 Input Queue Formally.

Let the variables $X$ and $Y$ range over the queues of a pSTS, i.e. over sequences of signal types, and $A$ over signal types. The concatenation of a sequence and a singleton element is expressed by juxtaposition. For a signal queue $X$ and a signal type $A$ the term $XA$ describes a sequence where A is the *last* element. Conversely, $AY$ describes a sequence where $A$ is the *first* element.

## 13.3 Interpreting SDL-Processes as pSTS

After having defined the pSTS model we now explain the mapping of an SDL process specification to the components of an pSTS P. So-called *transitions* in an SDL specification describe the change of processes control from one symbolic state to a symbolic successor state. In SDL symbolic states are identified by the `STATE` and `NEXTSTATE` keywords. In the example in Table 13.1 the two symbolic states are `S1` and `S2`. We map the symbolic states onto elements of $S$, which means $S = \{S1, S2\}$ in this case.

[2]Note the similarity to Hoare-triples consisting of a pre-condition, a program(-statement), and a post-condition [69]. [27] argue that Hoare-triples are insufficient to describe the semantics of Estelle and suggest the use of Dijkstra's predicate transformers instead. However, their reason for requiring predicate transformers is that they are more useful for verification purposes. Our main interest is not in availing SDL specifications to verification, we therefore prefer the use of more intuitive Hoare triples.

[3]We omit the reference to $P$ when this is clear from the context.

[4]We will not define all details of the relation $\models$ formally. For a detailed description of how to define such a relation for a given state transition system we refer the reader to [113]. We will concentrate here on the formal definition which are particular for an SDL process based pSTS.

The body of a transition consists of the specification of different sorts of statements, like assignments, decisions, communication statements, etc. In order to describe the state of the system before and after the execution of a transition we assign pre- and postconditions to every transition. In a few cases, when the transition body has a trivial structure, the determination of pre- and post-conditions is straightforward. However, as we shall see in the remainder, this is not always the case, and we shall discuss the treatment of more complex transition structures later in this section.

### 13.3.1 Formal Treatment of `INPUT` Statements

For the time being we only consider local systems, we do not yet interpret effects of communication. Therefore, we only give meaning to `INPUT` statements, `OUTPUT` statements will be formally interpreted later when we construct a global state transition system out of a set of pSTS. `INPUT` statements in SDL have, surprisingly, a purely local semantics. When executing an `INPUT` statement a process simply removes the signal at the head of the input queue and assigns its value to a local variable.

Table 13.2 shows the mapping of an SDL transition to transitions $\tau_j$ of the corresponding pSTS. The above given explanation of the semantics of an `INPUT` statement is not yet quite accurate. We need to observe the following particularity of the semantics of `INPUT` statements:

- When executing a transition associated with an `INPUT(X)` statement the process reads the value of the head of the input queue and assigns its value, if any, to a local variable with the name `X`[5]. More precisely: it is first checked, whether the signal at the head of the queue is of type `X`, if this is true then the signal is consumed as described above.

- However, if the signal at the head of the queue does *not* have the expected type, i.e. it is not of type `X`, then the message is removed from the head of the queue, discarded, and the same `INPUT` statement is re-enabled.

Concludingly one can say that the `INPUT` statement in SDL is an abridged notation for a much more complex operation.

We therefore need to split the treatment of `INPUT` statements into two logical cases, the first being the one where the expected signal type is not at the head of the queue, and the second being the one where the expected signal is at the head. This also means that we treat transitions with `INPUT` statements as two transitions which are mutually exclusive. Therefore, even though the Example in Table 13.1 only contains one transition, we need two transition predicates $\tau_1$ and $\tau_2$ to describe this transition. The logical exclusion is

[5] For reasons of conciseness we do not treat the handling of `SAVE` statements here, for their modeling in the context of an FSM interpretation we refer the reader to [110].

```
STATE S1;
INPUT(A);
OUTPUT(B);
NEXTSTATE S2;
```

Table 13.1: SDL Transition I

| $\tau_j$ | $P_j$ | $Q_j$ |
|---|---|---|
| $\tau_1$ | $\pi = S1 \wedge Q = AX$ | $\pi' = S2 \wedge Q' = X$ |
| $\tau_2$ | $\pi = S1 \wedge Q = CX \wedge C \neq A$ | $\pi' = S1 \wedge Q' = X$ |

Table 13.2: pSTS predicates for Transition I

encoded by the test $Q = AX$ which is *true* in case the head of the input queue contains the message of expected type `A`, and the test $Q = CX \wedge C \neq A$ which evaluates to *true* iff this is not the case.

Attention has also to be paid to the control flow in a transition. If we consider a transition which brings a process from symbolic state `S1` into symbolic state `S2`, then this can be interpreted as though control lies in code location `S1` before execution of the transition, and in location `S2` afterwards. Now, we defined a particular variable $\pi$ to range over code locations, called symbolic states, and we use this variable to formulate pre- and postconditions on the control flow inside an SDL process. To describe the transition from a state `S1` into a state `S2` we use the precondition $\pi = S1$ and the postcondition $\pi = S2$, see also Table 13.2.[6]

### 13.3.2 Formal Treatment of Variable Assignments

Variable assignments are treated on a very standard way, as for example described in [101]. Let $x$ and $y$ denote variables in a state $s$, let $x'$ and $y'$ denote these variables in the successor state $s'$, and let the system transit from $s$ to $s'$ through the execution of a statement `y:= x + 1`. The we describe this transition by the state predicate $y' = x + 1$ which is required to hold in state $s'$. This is the postcondition for this transition, there is no obvious precondition characterising this statement. The SDL transition in Table 13.3 involves the update of the local variable $x$, and Table 13.4 shows the predicates we use to

[6] It is interesting to compare the notion of a transition and it's enabling here with those used in the context of our MFG semantics in Section 7.4. For MFGs, a transition is only enabled if the process control lies in the right place, and the expected signal has been sent but not yet received (in the case of a receive event). As opposed to that, an SDL transition is enabled to be executed as soon as the process control has reached the right point, in our above example symbolic state `S1`. However, it may happen that the signal at the head of the input queue is not the one which is expected, in which case the transition is executed, the signal discarded, and control returns to the initial point. This exemplifies again that the semantics of transitions in MFGs is much more generic than the much more high-level construction of SDL transtions.

```
STATE S1;
INPUT(A);
TASK x := y + 1;
NEXTSTATE S2;
```

Table 13.3: SDL Transition II, with variable assignment

| $\tau_j$ | $P_j$ | $Q_j$ |
|---|---|---|
| $\tau_1$ | $\pi = S1 \wedge Q = AX$ | $\pi' = S2 \wedge Q' = X \wedge x' = y + 1$ |
| $\tau_2$ | $\pi = S1 \wedge Q = CX \wedge C \neq A$ | $\pi' = S1 \wedge Q' = X$ |

Table 13.4: pSTS predicates for Transition II

describe this transition.

### 13.3.3 Formal Treatment of `DECISION` Statements

The logical treatment of `DECISION` statements is straightforward, as these statements already define a predicate which determines the subsequent flow of control. Consider a `DECISION P(x)` statement, we decompose this into two, again mutually exclusive transition alternatives. The first is that the decision predicate holds, namely $P(x)$ is *true*, the second is that $P(x)$ is *not* true. As an example see the treatment of the decision in Table 13.5 in Table 13.6.

### 13.3.4 Handling Iterative Transitions

So far we assumed that the symbolic states in the set $S$ are identical to the symbolic states used in the SDL specification. This works in all those cases in which an SDL transition represents a finite linear sequence of operations. However, SDL transitions may also be iterative structures, for an example see the loop in the control flow in Table 13.8. `DECISION` statements may introduce branching into the control flow of an SDL transition, iterations

```
STATE S1;
INPUT(A);
DECISION D(A);
 (true):
   NEXTSTATE S2;
 (false):
   NEXTSTATE S3;
ENDDECISION;
```

Table 13.5: SDL Transition III, with decision predicate

| $\tau_j$ | $P_j$ | $Q_j$ |
|---|---|---|
| $\tau_1$ | $\pi = S1 \wedge Q = AX \wedge D(A)$ | $\pi' = S2 \wedge Q' = X$ |
| $\tau_2$ | $\pi = S1 \wedge Q = AX \wedge \neg D(A)$ | $\pi' = S3 \wedge Q' = X$ |
| $\tau_3$ | $\pi = S1 \wedge Q = CX \wedge C \neq A$ | $\pi' = S1 \wedge Q' = X$ |

Table 13.6: pSTS predicates for transition III

```
STATE S1;
INPUT(A);
/* S1-1 */
l1:
DECISION D(A);
 (true):
   NEXTSTATE S2;
 (false):
   OUTPUT(B);
   TASK A:=A-1;
   JOIN l1;
ENDDECISION;
```

Table 13.7: SDL Transition IV, with decision predicate and looping transition branch.

are achieved by a goto and labeling mechanism, where labels are introduced in a fairly standard way (the goto statement is called `JOIN` in SDL).

This means that we need to abandon the idea that a transition in an SDL process leads from one symbolic state to a symbolic successor state. This structure is too coarse to capture iterative transition structures. We need to refine the idea of control flow in SDL transitions by allowing cyclic control flow structures. We suggest introducing further symbolic states, called *auxiliary* symbolic states, which correspond to the target locations in the control flow to which a process jumps back or forth when executing `JOIN statements`. These target locations are the locations of the labels in the control flow. In the example in Table 13.8 we introduced an additional symbolic state `S1-1`, corresponding to the point of control which is reached when jumping to label `l1` (we introduced a comment `/* S1-1 */` in the SDL code at the location corresponding to auxiliary state *S1-1*).

The rising complexity of this looping iteration results in the fact that the transition IV is represented by five pre- and postcondition pairs. The conditions $\tau_4$ and $\tau_5$ represent cases in which control lies in the auxiliary symbolic state `S1-1`.

| $\tau_j$ | $P_j$ | $Q_j$ |
|---|---|---|
| $\tau_1$ | $\pi = S1 \wedge Q = CX \wedge C \neq A$ | $\pi' = S1 \wedge Q' = X$ |
| $\tau_2$ | $\pi = S1 \wedge Q = AX \wedge D(A)$ | $\pi' = S2 \wedge Q' = X$ |
| $\tau_3$ | $\pi = S1 \wedge Q = AX \wedge \neg D(A)$ | $\pi' = S1-1 \wedge Q' = X \wedge A' = A-1$ |
| $\tau_4$ | $\pi = S1-1 \wedge D(A)$ | $\pi' = S2$ |
| $\tau_5$ | $\pi = S1-1 \wedge \neg D(A)$ | $\pi' = S1-1 \wedge A' = A-1$ |

Table 13.8: pSTS for Transition IV

## 13.4 Input/Output Labeling of Transitions

The definition of the state transition system allows us to specify requirements on state predicates, like for example on the current point of control (e.g. $\pi = S1$) or on the state of data variables (e.g. $Q = AX \wedge A = \texttt{DR}$). However, sometimes one would much rather specify properties of events to happen, in particular referring to communication events. The communication events are usually interactions of a system with its environment, i.e. input or output of signals that are about to take place or that have just been executed. These communication events occur in the course of state-transitions, therefore we need to encode these state transitions in our state proposition language, as state-predicates.

What we do to solve this problem is to introduce state predicates which indicate which transition has been taken as last step in a computation, and whether this transition involved any communication events. Technically, we introduce two relations *inlabel* and *outlabel* which label the transitions of the pSTS with the `INPUT` or `OUTPUT` statements which are executed in the course of a transition. We omit the straightforward technical construction of this labeling here, and just explain the construction by example. If we consider the example in Tables 13.7 and 13.8 we see that for example $inlabel(\tau_3) = \{\texttt{INPUT(A)}\}$ and $outlabel(\tau_3) = \{\texttt{OUTPUT(B)}\}$.

We augment these labels to state propositions in the following way. Let $s = s_1, s_2, \ldots$ an admissible state sequence for a given pSTS, and let $\mathcal{T}_T$ denote the set of transitions for this pSTS. We say that

$$s_i \models INPUT(A) \text{ iff } (\exists \tau \in \mathcal{T}_T)(ta(s_{i-1}, s, \tau) \wedge (\texttt{INPUT(A)} \in inlabel(\tau))), \text{ and}$$

$$s_i \models OUTPUT(A) \text{ iff } (\exists \tau \in \mathcal{T}_T)(ta(s_{i-1}, s, \tau) \wedge (\texttt{OUTPUT(A)} \in outlabel(\tau))).$$

## 13.5 Global State Transition Systems

### 13.5.1 SDL Specifications Formally

In the previous section we considered single SDL processes and formalized their state-transition behaviour. SDL specifications, however, consist of collections of concurrent SDL processes. We say that the Global State Transition System (GSTS) $G_P$ corresponding to an SDL specification $P$ is a tuple $G_P = (P^0, \ldots, P^n)$ where each $P^i$ for $i = 1, \ldots, n$ is a pSTS. $P^0$ represents the environment. $P^0$ (which represents the environment behaviour) is not a full pSTS, it only consists of an input and an output alphabet, and an input queue. $P^0$ has no state and we rely on the facilitating assumption that $P^0$ will provide any of the other processes with input signals whenever these wish to consume any such signal, and that $P^0$ consumes instantly any signal which it receives from any process of the SDL system.

### 13.5.2 Formal Treatment of Communication in SDL Specifications

SDL processes communicate asynchronously via infinite queues. There is one input queue per SDL process. For an SDL specification we interpret the sending of a signal $A$ from a process $P^1$ to a process $P^2$, indicated by an `OUTPUT(A)` statement, such that a signal of type $A$ is appended to $P^2$'s input queue $Q^2$. We slightly simplify the SDL mechanism of mapping of an output signal to a receiving process by assuming that a signal `A` is sent from a process $P^i$ to a process $P^j$ iff $A \in I^j$[7]. Furthermore, we require $(\forall i = 1, \ldots, n)(\forall a \in O^i)(\exists j \neq i)(a \in I^j)$ and $(\forall i = 1, \ldots, n)(O^i \cap I^i = \emptyset)$. As we saw in Section 13.3, the execution of an `INPUT(A)` statement (which in the SDL terminology is often just referred to as *signal-consumption*) represents an action purely local to an SDL process, and we have given the semantic interpretation there.

**Transition Predicates for OUTPUT statements.** The execution of an `OUTPUT` statement involves a non-local action. It means that the execution of the statement is a local event to the sending process, whereas the reception (which in SDL is different from the consumption of the message and just means that the message will be appended to the tail of the receiving process' input queue) is a local event of another (the receiving) process. Therefore, one can not formalize these transitions by state propositions that solely refer to state variable of only one process. Table 13.10 presents a simple example of a two-process SDL specification $P = (P^0, P^1, P^2)$. Transition $\tau_1^1$ describes both the state change in $P^1$ and the appending of the signal $B$ to the input queue of $P^2$. Although strictly speaking this transition also changes the state of process $P^2$ for our formal treatment we consider transition $\tau_1^1$ to be a transition belonging to process $P^1$.

### 13.5.3 Global System States and Transitions

Let $G_P = (P^0, \ldots, P^n)$ denote the GSTS for an SDL specification $P$. We say that the vector $s = (s^1, \ldots, s^n)$ is a *global system state* (GSS) of the SDL specification $P$ iff $s^i$ is a

[7] In SDL this involves a mapping of signal names via signal lists to signal routes which point to the receiving process.

```
PROCESS P1;        PROCESS P2;
  STATE S1;          STATE S3;
  INPUT(A);          INPUT(B);
  OUTPUT(B);         NEXTSTATE S3;
  NEXTSTATE S2;
```

Table 13.9: Transitions involving inter-process communication

| $\tau_j^1$ | $P_j^1$ | $Q_j^1$ |
|---|---|---|
| $\tau_1^1$ | $\pi^1 = S1 \wedge Q^1 = AX \wedge Q^2 = Y$ | $\pi'^1 = S2 \wedge Q'^1 = X \wedge Q'^2 = YB$ |
| $\tau_2^1$ | $\pi^1 = S1 \wedge Q^1 = CX \wedge C \neq A$ | $\pi'^1 = S1 \wedge Q'^1 = X$ |

Table 13.10: Predicates describing inter-process communication

state of pSTS $P^i$ for all $i = 1, \ldots, n$.

**Global System State Sequences.** We now extend the notions of admissible state sequences to GSS. In the course of each change of the GSS exactly one pSTS changes its local system state. We assume an interleaving model of global system state sequences to model the concurrency of an SDL specification. This means that in a given GSS $s$ a demon decides nondeterministically which out of all enabled transitions of all pSTS is going to be executed next, giving the successor GSS $s'$. Let $\sigma = s_0, \ldots, s_k$ denote a finite sequence of GSS. We call this sequence *admissible* iff $(\forall 0 \leq j < k)(\exists \tau_l^i)((s_j^i, s_{j+1}^i) \in T^i))$. This definition extends to infinite sequences in the obvious way. Also, the interpretation of the state propositions *en*, *ta*, *INPUT* and *OUTPUT* extend in the obvious way from pSTS states to GSS.

**Satisfaction of an SDL Specification.** Based on the above definitions we may now define a satisfaction relation $\models_{\mathrm{SDL}}$ for SDL specifications. Let $P$ an SDL specification and let $\Sigma_P^\omega$ the set of all infinite sequences of GSS of $P$. For a $\sigma \in \Sigma_P^\omega$ we write $\sigma \models_{\mathrm{SDL}} P$ iff $\sigma$ is an admissible sequence with respect to $P$.

# Chapter 14

# Using Temporal Logic for SDL Specifications

Many authors have advocated the use of temporal logics for high level specification of abstract requirements on communication protocols and services (see for example [148] and [61, 62]). The characterisation of properties by the use of temporal logic is accomplished by interpreting the temporal logic specification as a filter on the set of all state sequences, resulting in a set of admissible state sequences.

Now, as we have seen in Chapter 13, SDL specifications also specify admissible sequences of states. But, as we have noted before, not all desirable properties are expressible in SDL, as for example liveness properties and hard real-time bounds. What we suggest now is to use a combination SDL and temporal logic specifications, where the temporal logic specification acts as a filter on the admissible sequences described by an SDL specification. We call these combined specifications *complementary* specifications.

When using this specification approach a crucial point is the selection of a suitable temporal logic language. For an overview over temporal logics we refer to [52]. In the remainder we will use a temporal logics similar to the logic described in [113], called Propositional Temporal Logic (PTL), and extensions based on PTL. However, it should be fairly easy to translate formulas into other temporal logic frameworks, for example into the *Temporal Logic of Actions* (TLA) [101], or into branching time temporal logics [52].

**A State Proposition Language.** When using complementary QoS specifications we need to identify a set of state propositions which we may use in the temporal logic formulas. When determining the set of state propositions one also determines which part of the state information is observable[1]. We will not treat this question in depth here. The determination of the visible state component is mainly dependent upon the particular

[1]This corresponds to the determination of the sets $\mu(\tau)$ in [14].

specification problem considered. We assume that the state propositions we use all refer to observable components of the system state, and we use in particular the following state propositions for an SDL specification $P$:

- *Actual State:* let $S = S_1^i, \ldots, S_n^i$ denote the symbolic states for a given process $P^i$ of $P$, then $at\_S_k^i$ denotes the state proposition that the i-th component of the global system state is in symbolic state $S_k^i$, i.e. $\pi^i = S_k^i$.

- *Input and output:* we use the state propositions *INPUT* and *OUTPUT* as defined above to denote that we are in a state where an input or an output of a signal has just occurred in the last GSS transition.

- Data: we allow the reference to visible data variables and allow standard comparison operators on the variables. However, we require that the resulting expressions remain state propositions.

We allow state formulas to be constructed by using boolean operators between state propositions.

**Example.** The state formula $n \leq 3 \wedge \mathit{INPUT}(A)$ holds in all GSS in which the value of variable $n$ is less than or equal to 3 and an input of a signal of type $A$ has just been executed. The state formula $at\_S1 \supset n \geq 3$ holds in all those GSS in which if the control is in symbolic state $S1$ then the value of variable $n$ is greater than or equal to 3.

## 14.1 Propositional Temporal Logic

The Propositional Temporal Logic (PTL) we use here is a linear time temporal logic taken from [113]. For a formalization of the syntax and semantics of PTL we refer the reader to [113]. Let $p$ denote a state predicate. This means that p is constructed from state propositions. We say that the formula $\Diamond p$ holds in a state $s$ iff $p$ holds in $s$ or in some future state. In addition to the standard operators of PTL as defined in [113] we define a *strong eventuality* operator $\diamondsuit$ so that $\diamondsuit p$ holds in some future state $s$[2]. The formula $\Box p$, which denotes a syntactic abbreviation for $\neg\Diamond\neg p$, holds in a state $s$ iff $p$ holds in $s$ and in every successor state of $s$. Temporal Logic includes the propositional calculus. The formal semantics of PTL define a satisfaction relation $\models_{\mathrm{PTL}}$. An execution sequence $\sigma = s_0, \ldots$ of states $s_i$ satisfies a formula $\phi$ iff $\phi$ holds in $s_0$, and we write $\sigma \models_{\mathrm{PTL}} \phi$. We say that a system satisfies a formula $\phi$ iff all its execution sequences satisfy $\phi$.

[2]The formal definition of the semantics of this operator is $s_i \models \diamondsuit p$ iff $(\exists j > i)(s_j \models p)$.

## 14.2 Metric Temporal Logic

Real-Time extended temporal logic has been suggested in various places as a suitable tool for the specification of real-time systems (see for example [67], [1], [87] and [124]). We apply a variant of these logics called *metrical temporal logic* (MTL) to the specification of QoS requirements[3]. The language of Propositional Temporal Logic (PTL) is a proper syntactic subset of MTL.

**Timed Observation Sequences.** The models over which we interpret PTL formulas are timed observation sequences $o = o_1, \ldots$ (see [14]). Each $o_i$ corresponds to a pair $s_i, I_i$ where $s_i$ is a state and $I_i$ is a numeric interval expression. Let $l_i$ and $r_i$ denote the left and right boundaries of the Interval $I_i$, then for example the timed observation $(s_i, [l_i, r_i[)$ means that state $s_i$ can be observed in the interval starting with $l_i$ and ending with, but not including, $r_i$. As we only consider instantaneous state changes the sequence of intervals $I_I$ can be replaced by single time stamps, for example for an interval $I_i$ always by the left interval boundary $l_i$. We assume sequences $l_i$ to be monotonic. We assume finite precision of our clocks, i.e. we assume that every state change coincides with a click of the clock from which we derive the timed observation. Therefore the set of natural numbers $N$ suffices as domain for the interval expressions [14]. We use MTL to specify properties of concurrent systems based on an interleaving interpretation of concurrent state transitions. Assume that $s_1, s_2$ and $s_3$ are GSS of an SDL specification. Furthermore, let both $s_1, s_2, s_3, \ldots$ and $s_1, s_3, s_2, \ldots$ be admissible sequences in the untimed model. If we now want to express that both $s_2$ and $s_3$ may occur at the same time (which means that they have the same time stamp) in any order we have to allow that both timed observation sequence $(s_1, l_1) \rightarrow (s_2, l_2) \rightarrow (s_3, l_3) \rightarrow \ldots$ and $(s_1, l_1) \rightarrow (s_3, l_2) \rightarrow (s_2, l_3) \rightarrow \ldots$ are admissible and that $l_2 = l_3$. Hence in this interleaving model we assume the sequence $l_i$ to be *weakly*-monotonic [14].

**MTL language and semantics.** MTL contains formulas of the form $\Diamond_I \phi$ which assert that *one* of the following states within the time-interval described by expression $I$ is a state which satisfies $\phi$. Formulas of the form $\Box_I \phi$ assert that *all* states in the time-interval described by $I$ satisfy $\phi$. The expression $I$ describes an either open or closed interval over the time domain and we sometimes use semi-algebraic expressions to refer to these intervals. As an example the formula

$$\Box_{\leq 5}(\neg OUTPUT(A))$$

[3] Our introduction will be rather informal and we will not present all possible operators, we restrict ourselves to a minimal subset of the language which we need to carry out our example. For a complete formal definition of the syntax and semantics of MTL we refer the reader to [14] and [67, chapter 3.4]

expresses the property that in all subsequent system states $i$ in which the time stamp $T_i$ is less than or equal to 5 the state proposition $OUTPUT(A)$ is false. In analogy to the satisfaction relation for PTL we write $o \models_{\mathrm{MTL}} p$ iff the sequence $o$ satisfies the MTL formula $p$.

## 14.3 Complementary Specifications

Assume we have an SDL specification $P$ and a set of formulas $M$ in MTL. Now, $P$ and $M$ are *complementary* specifications if we require from the specified system that for all its timed observation sequences $o = (s_0, t_0), \ldots$ the following condition holds:

$$s \models_{\mathrm{SDL}} P \wedge o \models_{\mathrm{MTL}} M.^4$$

**Example.** Let us consider the INRES connection establishment example in Figure 12.1 again. The SDL specification describes a sequence of possible executions. We will use complementary specifications to specify an interesting liveness property and a bounded response hard real-time constraint.

First we will look at liveness. It is important to require that the system is live, namely that when a request for a connection establishment has been issued by sending a `CR` message, then eventually the process `Initiator` will eventually receive either a `CC` or a `DR` signal, or it will eventually issue a `IDISind` signal to the service user to indicate that a connection establishment was not possible. Liveness properties are not expressed by SDL specifications, so this requirement can be expressed by the following complementary PTL formula:

$$\Box(OUTPUT(CR) \supset \Diamond(INPUT(CC) \vee INPUT(DR) \vee OUTPUT(IDISind))).$$

Now, as we argued in Chapter 12.1, it is important to know that any of these responses to the sending of the `CR` signal happens within a reasonable period of time, say within $t$ time units. In the SDL specification the timer `T` has been used to require this, but we have argued above why the usage of the timer in this context cannot guarantee this condition to hold. Therefore we transform the above liveness requirement into a real-time bounded response requirement which we specify using MTL in the following way:

$$\Box(OUTPUT(CR) \supset \Diamond_{\leq t}(INPUT(CC) \vee INPUT(DR) \vee OUTPUT(IDISind))).$$

## 14.4 Using PTL and MTL for MSC specifications

In Section 7.6 we explained how GSTGs derived from MSC specifications relate to Manna-Pnueli Basic Transition Systems and how PTL can be used to specify liveness properties

[4] Note that PTL is a syntactic subset of MTL and hence included in this definition.

for MSCs. The state propositions we use as basic propositions for PTL formulas are the predicates $en(\tau)$ and $ta(\tau)$ where $\tau$ is a transition in the GSTG. In a similar way like for SDL specifications we would like to make assertions not only on these state predicates, but also using propositions referring to communication events.

It is therefore necessary to use a similar labeling technique like for the transitions in an SDL specification. We label transitions with the names of the communication events which cause the transition to happen, so in the GSTG in Figure 7.1 the transition from $S1$ to $S2$ is labeled by $y$. It should be noted that for example the state proposition $ta(y)$ holds in several different states ($S2$ and $S4$), namely all those states that can be entered by executing a $y$ event. If the transition $y$ represents a send (receive) event of a message of type $a$, then we sometimes use the event types as state propositions, e.g. we write $!a$, instead of $ta(y)$, and $?a$ instead of $ta(z)$.

**Example.** If we were for example to require that in our example a send event will eventually be followed by a receive event, which is a liveness requirement that prevents the system from looping on state $S2$ forever, we would write in PTL

$$\Box(!a \supset \Diamond ?a).$$

The interpretation of MTL formulas over models generated by MSC specifications is a straightforward extension of the interpretation of the formulas over SDL specifications.

# Chapter 15

# Specifying QoS: Delays

Figure 15.1 presents the example of a simple *Sender/Receiver Service* (SRS) specified using MSCs. The example works as follows. A user `U1` of the service requests the transmission of some data by sending a `UDreq` signal to the sender process `S` which in turn requests the transmission of the data from a medium service `M` by sending a `MDreq`. The medium service is unreliable. However, in case the transmission is successful the medium service will deliver the data to the receiver process `R` by means of an `MDind` message, and the receiver delivers the data to the user process `U2`. Although the medium service is unreliable we nevertheless assume that it is capable of reliably indicating to the sender process by means of an `MDcon` signal whether the data has been delivered successfully to the receiver process, or by an `MDrej` that this is not the case. Successful delivery will be indicated to the service user `U1` by an `UDcon` signal, and unsuccessful delivery by an `UDrej` signal.

The example does not reflect any particular real-world telecommunications service, however as pointed out in [103] some *Asynchronous Transfer Mode* (ATM) adaptation layer service will have to implement a similar functionality as the medium service in the SRS example. The SDL specification presented in Figure 15.2 presents a similar service. However, in the SDL specification we focus on the specification of the behaviour of the sender `S` and receiver `R` processes and omit a specification of the behaviour of processes `U1`, `M` and `U2`.

**Related work.** For an overview on specifying real-time constraints using MTL we refer the reader to [14]. [51] describes aspects of the JVTOS service using MSCs, and describes QoS measurements in TLA [101]. It has been suggested to attribute MSCs with timing information, and [117] investigates how timing constraints attached to MSCs can be verified using the Dechter, Meiri and Pearl algorithm. This work relies on timing information requiring earliest and latest occurrence of one event on the occurrence of another event.

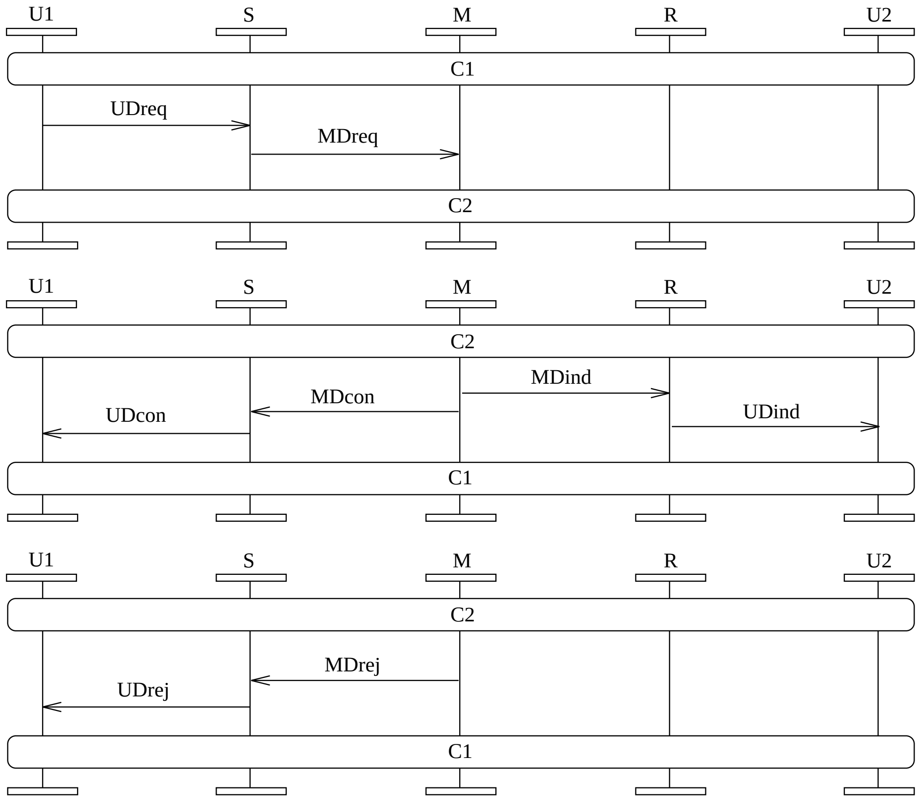


Figure 15.1: MSC Specification of SRS example.

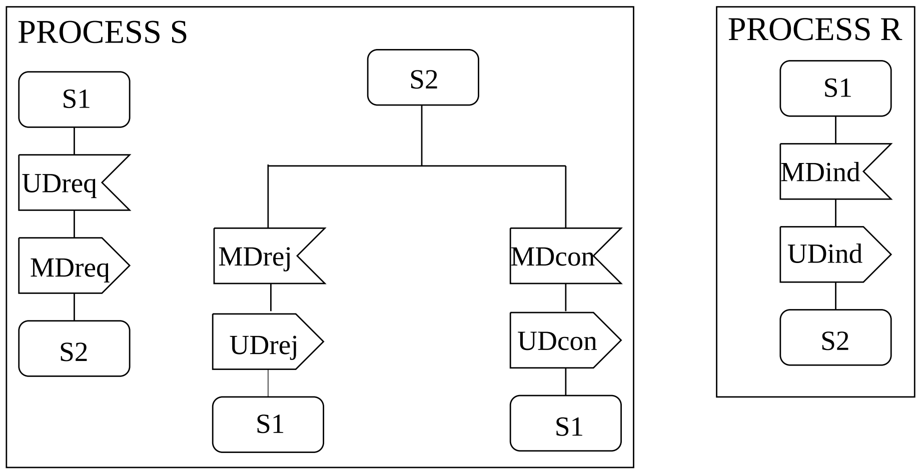


Figure 15.2: SDL Specification of SRS example.

## 15.1 Delay bounds on SRS

We now discuss a few potential delay bounds on the SRS service and specify them using complementary MSC and PTL specifications. The specifications given here are inteded to demonstrate the applicability of the approach in principle.

### 15.1.1 Service Response Delay Bound

Similar to what we discussed in the context of the INRES example an important liveness requirement for the SRS service is that if the service process has received a `UDreq` it will eventually indicate either an `UDcon` or a `UDrej` signal to the service user in order to indicate successful or unsuccessful delivery of data. We describe this requirement by

$$\Box(!\mathit{UDreq} \supset \Diamond(?\mathit{UDcon} \vee ?\mathit{UDrej})).$$

However, many real-time applications may require an indication to be given within a real-time bound of for example $t_1$ time units, which is expressed by the following bounded response requirement:

$$\Box(!\mathit{UDreq} \supset \Diamond_{\leq t_1}(?\mathit{UDcon} \vee ?\mathit{UDrej})).$$

### 15.1.2 Service Processing Delay Bound

The sender process `S` may need some time to process the user data before sending out a `MDreq` to request data transmission from the medium service. The service processing delay bound requirement reads

$$\Box(?\mathit{UDreq} \supset \Diamond_{\leq t_2}!\mathit{MDreq}).$$

### 15.1.3 Message Transmission Delay Bound at Service Interface

Depending on the communication mechanism used at the service interface between processes `S` and `M` the transmission of a message between both may consume time. In order to limit the transmission time to at most $t_3$ time units we write

$$\Box(!\mathit{MDreq} \supset \Diamond_{\leq t_3}?\mathit{MDreq}).$$

### 15.1.4 Medium Transmission Delay Bound

The process `M` is an abstraction for the whole communication subsystems which we use in order to transfer the data. One may wish to constrain the time between the events of the reception of the `MDreq` signal and the output of the `MDind` signal to $t_4$ time units, but only if the transmission is successful.

$$\Box((?\mathit{MDreq} \wedge \Diamond!\mathit{MDind}) \supset \Diamond_{\leq t_4}!\mathit{MDind})$$

This requirement also constrains the medium service *not* to deliver messages to process `R` after $t_4$ time units after sending.

### 15.1.5 Minimal Medium Service Response Time

In a verification context it may also be interesting to state that between two events there is a *minimal* time that will always pass. The following formula states that if after the request the data will eventually be successfully delivered by the medium service by issuing a `MDind` signal, then this will happen at least $t_5$ time units after the request has been issued.

$$\Box((?MDreq \wedge \Diamond !MDind) \supset \Box_{<t_5} \neg !MDind)$$

The SRS example is based on MSCs with asynchronous communication. However, with the obvious exception of the message transmission delay bound requirement all other requirements apply analogously to the same example with synchronous communications.

## 15.2 Delay variation: Jitter

### 15.2.1 Delay Jitter

Successive data units routed through a complex network may be subject to varying delays over time. The delay variations may be caused either by the network management changing the routes which successive data units are using through a multi-hop network, or by variations of the background load of the network. The ATM service is, as one example, prone to this sort of delay variation [103]. However, in particular multimedia applications which need to reconstruct continuous signals require data to be delivered within a time interval around the mean value of the transmission delay, depending on the coding scheme used. The delay variance is called *delay jitter* and formally defined as follows: let $d_{min}$ denote the minimal and let $d_{max}$ denote the maximal delay between sending and receiving of a sequence of transmitted data units, then $J = d_{max} - d_{min}$ denotes the delay jitter. We use the SRS example specified in SDL (see Figure 15.2) here to exemplify the specification of jitter constraints, similar formulas would apply to the SRS example specified by MSCs. We assume that $d_{min}$ and $d_{max}$ are known constant values. The requirement bounding the delay jitter for the user interface service can then be specified by the formula

$$\Box(INPUT(UDreq) \supset (\Box_{\leq d_{min}} \neg OUTPUT(UDind)) \wedge (\Diamond_{\leq d_{max}} OUTPUT(UDind))).$$

### 15.2.2 Isochronicity

Isochronicity is a characteristics of many multimedia applications. The isochronicity we refer to means that events, for example sending and receiving of data units, occur periodically at equally distanced points of time. Again, the example formulas given here refer to the SDL specification of the SRS example.

**Isochronous sending and receiving.** Isochronous sending is a characteristic of a traffic source. It is a characteristical requirement on simple coding schemes for audio or video data where samples of the analogous signal are taken and sent periodically. The characterization of isochronous sending reads

$$\Box(\mathit{INPUT}(\mathit{UDreq}) \supset (\neg \Diamond\!\!\!\cdot_{<t} \mathit{INPUT}(\mathit{UDreq}) \wedge \Diamond_{=t} \mathit{INPUT}(\mathit{UDreq}))).$$

On the receiving side the receiver may require to have successive data units available at isochronous moments in time. This may be expressed in a way very similar to the isochronous send characterization, namely as

$$\Box(\mathit{INPUT}(\mathit{UDind}) \supset (\neg \Diamond\!\!\!\cdot_{<t} \mathit{INPUT}(\mathit{UDind}) \wedge \Diamond_{=t} \mathit{INPUT}(\mathit{UDind}))).$$

### 15.2.3 Rates

Rates like throughput are usually measured by the number of data units processed per time period. The temporal logic we have proposed so far does not permit the counting of events. Event counting requires a non-trivial extension of the logic[1]. However, it may be useful to specify that the time interval in between two events of some particular type, e.g. the sending of events, is restricted to a time span $t$. This is a reciprocal consideration compared to the number of events per time unit based rate specification. For example, in order to specify a constraint on the output rate of the medium service used in the SRS example we may require that the time between two successive *MDind* events is limited to be less than or equal to $t_6$.

$$\Box(\mathit{OUTPUT}(\mathit{MDind}) \supset \Diamond\!\!\!\cdot_{\leq t_6} \mathit{OUTPUT}(\mathit{MDind})).$$

It is more appropriate to call this requirement the bounded *inter-send time* requirement instead of calling it a throughput requirement because throughput may be achieved by small inter-send times for a while, followed by a period of silence. This is not captured by the requirement presented here.

[1] For an example of how to incorporate event counting in Temporal Logics see [61, 62].

# Chapter 16

# Specifying QoS-mechanisms

In this Chapter we show a method for the specification of QoS mechanisms based on the complementary use of SDL and MSC specifications and temporal logic formulas. The QoS mechanisms we refer to are QoS negotiation, reaction on QoS guarantee violation, and delay jitter compensation.

## 16.1 QoS Negotiation

Figure 16.1 describes a QoS negotiation scenario. Assume that this specification is somehow related to the specification of the SRS example in Figure 15.1[1]. The functioning of the negotiation is quite obvious. The user `U1` requests an increase in bandwidth by sending a `UINCreq` signal which the service forwards to the medium (`MINCreq`) (it is assumed that there it is processed by the appropriate network management process). The medium either grants the increase (`MINCcon`) or it refuses the increase (`MINCrej`). Both reactions are indicated accordingly to the user. The following formula limits the constraining impact of the response time requirement on the medium service by only requiring the QoS guarantee to be satisfied if the QoS level has been granted (by the medium issuing the `MINCcon` signal).

$$\Box(!MINCcon \supset \Box((?MDreq \wedge \Diamond !MDind) \supset \Diamond_{\leq t_4} !MDind))$$

## 16.2 Reaction on QoS Violation.

It may be useful to specify a reaction on the violation of QoS requirement without requiring that the violation invalidates the behaviour against the system specification. Let us look at the SRS example again and let us assume that we monitor the response time behaviour

[1]The example has been inspired by an example of QoS negotiation in the context of ATM based virtual private networks given in [58].

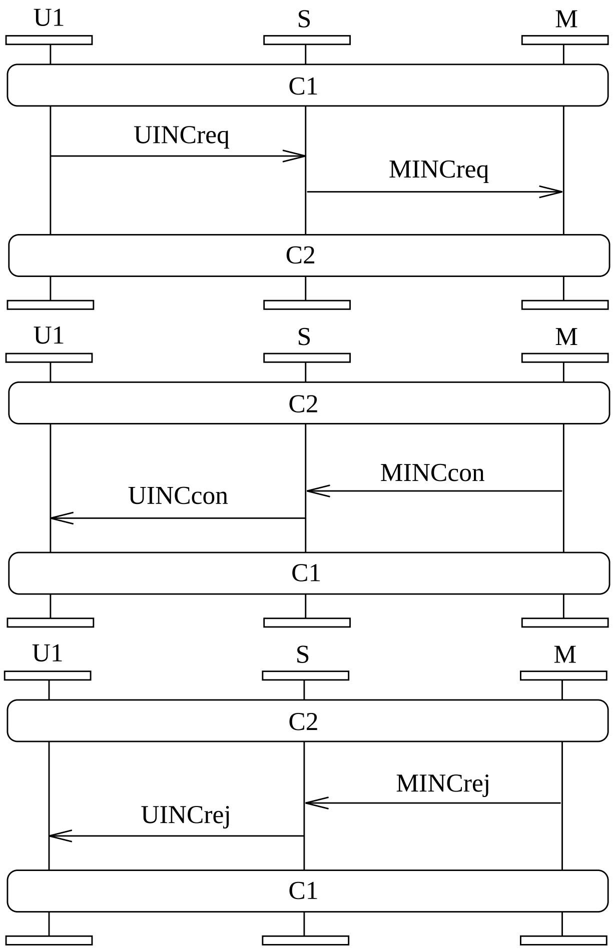


Figure 16.1: MSC Specification of QoS negotiation.

of the medium service. Assume that we want the monitor to rise an `ALARM` if it has been detected that the medium service does not respond by either `MDind` or `MDrej` within $t_7$ time units after the `MDreq` has been issued, and that we want the `ALARM` to be issued at most $t_8$ time units after this situation has been detected. We specify this as

$$\Box(\neg(OUTPUT(MDreq) \supset$$

$$\Diamond_{\leq t_7}(INPUT(MDind) \vee INPUT(MDrej))) \supset \Diamond_{\leq t_8} OUTPUT(ALARM)).$$

## 16.3 Delay Jitter Compensation

Guaranteeing a bound on the delay jitter does not yet guarantee isochronous delivery of messages to a user, even if the source is sending data isochronously. In order to compensate the residual delay jitter and to guarantee an isochronous delivery of data units to a user it is often suggested to use a jitter compensation buffer between the network service and the user. In the context of ATM this buffer is often called *playout* buffer (see [103]). At this point we leave the SRS example as underlying model to a certain degree. We assume that the process `R` also has the functionality of a playout buffer, as described for example in [103][2]. The playout buffer functionality of `R` is the following. `R` accepts the possibly non-isochronous but jitter-bounded data stream from the Medium service by `MDind` signals. Every signal will be delayed for a minimum time span of $d_1$ time units. This means that the first data units in a stream will fill the buffer up to a certain threshold number. Then, at latest $t_2 > t_1$ time units after the arrival at the buffer the data units will be delivered to the user by means of a `UDind` signal. The delivery of successive `MDind` signals then occurs isochronously with an inter-signal delivery time of $p$, which ideally should correspond to the inter-send event time at the sender in order to ensure an isochronous traffice with identical inter-send times on the sender as on the receiver side. The jitter compensation requirement for the process `R` reads

$$\Box(INPUT(MDind) \supset ((\Box_{\leq t_1} \neg OUTPUT(UDind) \wedge \Diamond_{\leq t_2} OUTPUT(UDind)))$$

$$\wedge \Box(OUTPUT(UDind) \supset \Diamond_{=p} OUTPUT(UDind)).$$

[2] It is a fairly easy exercise to model the *functional* aspects of a playout buffer in SDL.

# Chapter 17

# Discussion

We will discuss some problems arising from the use of complementary specifications in the context of quality of service specifications. First, we discuss the relation of system performance and quality of service. We address the specification of the characteristics of a lower layer communication system, a communication mechanism, and how this relates to the QoS characteristics offered to a higher layer user. This leads to a verification problem, and we discuss which verification methods can be applied.

## 17.1 System Performance to QoS Mapping

A communications system generally offers a service to a higher-layer user, and usually uses a lower layer communications infrastructure. We will now investigate the question how the QoS requirements which the higher layer user imposes on the system, the communication mechanisms inside the communications system, and finally the QoS requirements on the lower layer communications infrastructure are interrelated. To make the terminology less ambiguous we call the requirements of the upper layer application on the service *QoS* requirements, and the description of the properties of the underlying communication infrastructure *system performance* (a similar terminology is used in [82]).

Assume that we are given a specification $\mathcal{P}$ of the system performance, specification $\mathcal{S}$ of the service or the protocol which we investigate, and a specification $\mathcal{Q}$ of the user QoS requirements. Assume that all three specifications are given in terms of logic[1]. We now ask whether $\mathcal{P}$ together with $\mathcal{S}$ can ensure that $\mathcal{Q}$ can be satisfied, or more formally:

$$\mathcal{P} \wedge \mathcal{S} \supset \mathcal{Q}.$$

**Example.** Let us consider the SRS example again, and let us assume that SRS has been translated into logic and is given as a specification $\mathcal{S}$. Furthermore, assume the system

[1] We have shown earlier how to translate SDL specifications into logic descriptions.

performance to be described by the following minimal response time formula:

$$\mathcal{P} : \Box((?MDreq \wedge \Diamond !MDind) \supset \Box_{<t_5} \neg !MDind).$$

Let the QoS requirement be described by the following formula:

$$\mathcal{Q} : \Box(!UDreq \supset \Diamond_{\leq t_1}(?UDcon \vee ?UDrej)).$$

Based on an instantiation we would now require a verification method to verify, whether the assertion $\mathcal{P} \wedge \mathcal{S} \supset \mathcal{Q}$ holds. Intuitively, the answer depends on the choice of values for $t_1$ and $t_5$.

To formally establish this conjecture it is necessary to either employ theorem proving or model checking techniques.

## 17.2 Verification of QoS Requirements

The goal of a formal verification method for QoS requirements is to prove that the specification of a system $S$ satisfies a set of QoS requirements $Q$. In particular, $S$ may be the specification of a protocol or a service and may include the guarantees provided by the underlying system performance of the underlying network. $Q$ may be the specification of QoS requirements that the System specified in $S$ is expected to guarantee. The verification methods available are *formal verification* and *model checking*.

### 17.2.1 Formal Verification or Theorem Proving

We translate the specification $S$ into a set $T_S$ of temporal logic formulas, unless $S$ is already specified in temporal logic[2]. It then remains to prove that

$$T_S \supset Q.$$

Formal verification requires a proof system for the particular temporal logic calculus used. In case the underlying temporal logic is a decidable logic the proof can be fully automatized. If this is not the case manual or machine supported reasoning is necessary.

### 17.2.2 Model Checking

We take the state transition model $M_S$ of of the system $S$ and prove formally, that this model satisfies the QoS requirements Q, formally

$$M_S \models Q.$$

[2]The translation of an SDL specification into a set of temporal logic formulas is a straightforward extension of the logic based interpretation of SDL specifications in Chapter 13.

Model checking amounts essentially to an exploration of the state space of the system $M_S$. State exploration is a well known technique in the field of protocol validation (see for example [74]). In real-time systems, model checking additionally requires a verification whether a state-transition model satisfies certain real-time constraints, see for example [11].

## 17.3 Conclusions

We described a method for the specification of real-time constraint based QoS requirements. Starting point was an analysis of SDL specifications and the insight that the SDL timer mechanism is unsuitable to express an important class of real-time requirements. We mapped SDL specifications to global state transition systems and we showed, how systems states and state transitions can be described in terms of logic formulas over state propositions.

Next we connected temporal logic specifications to SDL specifications. We called the combinations of SDL/MSC and temporal logic specifications complementary specifications. The temporal logics we used were standard propositional and metric temporal logic.

This was one way to overcome the inexpressiveness of the SDL timer mechanism. Independent of the shortcomings of the SDL timer mechanism it should be noted that using real-time temporal logic based specifications instead of timer mechanisms means a much higher degree of flexibility and abstraction in the specification. Timers are specifications of the *how* to ensure real-time requirements, whereas temporal logics just express *which* requirement has to be satisfied by the system, and *how much* of it.

We then gave some general example specification for QoS requirements for SDL and MSC specifications. Examples included delay bounds, delay jitter bounds, and isochronicity requirements. We then showed how QoS mechanisms can be specified in the framework of our method, in particular QoS negotiation and QoS monitoring. Finally we briefly discussed methods for the formal verification of QoS specifications, and pointed to extensions towards a probabilistic expressiveness.

# Part IV

# Efficient Protocol Implementation

# Chapter 18

# Introduction

In this part of the thesis we present a method for the automatic derivation of efficient protocol implementations from a formal specification. Optimized efficient protocol implementation has become an important issue in telecommunications systems engineering as recently network throughput has increased much faster than computer processing power. Efficiency will be attained by two measures. First, the inherent parallelism in protocol specifications will be exploited. Second, the order of execution of the operations involved in the processing of the protocol data will be allowed to differ from the order prescribed in the specification, thus allowing operations to be executed jointly and more efficiently. The method will be defined formally which is useful when implementing it as a tool.

Our method starts with the SDL specification of a protocol stack. We first derive a data- and control flow dependence graph from each SDL process. Then, in order to perform cross-layer optimizations we combine the dependence graphs of different SDL processes. Next, we determine the common path through the multi-layer dependence graph. We then parallelise this graph wherever possible which yields a relaxed dependence graph. Based on this relaxed dependence graph we interpret different optimization concepts that have been suggested in the literature, in particular the combination of data manipulation operations. Together with these interpretations the relaxed dependence graph can be used as a foundation for a compile-time schedule on a sequential or parallel machine architecture.

## 18.1 Overview

In Figure 19.2 we present a (partial) view of the SDL specification of a two layer protocol stack. It will serve as a running example in this part and we will refer to it as TLS. The specification in Figure 19.2 is given in graphical SDL syntax (GR), Figure 19.3 contains the same specification in the textual SDL syntax (PR). We also give this textual representation to emphasize that the syntactic analysis steps we present here can be automated. A first attempt towards automation of the data- and control flow analysis for SDL specifications has been made in [151]. It is the purpose of our optimization and implementation method to transform specifications similar to TLS into parallelised and optimized implementations.

In Chapter 19 we discuss the sort of layered SDL specifications we consider in this part. Here, we also argue why a direct and faithful implementation of SDL specifications would lead to inefficient implementations. This is mainly due to the structuring of SDL specifications into per-layer processes and the resulting inter-layer asynchronous queue based communication mechanism. Then we turn to a description of our analysis and optimization method:

- First, we construct a dependence graph representing control-flow and data dependences among statements in an SDL specification. This leads us to so-called *Transition Dependence Graphs*. Their construction is explained in Section 20.1. For the dependence graphs for example TLS see Figure 20.1. The dependence graph construction is an application of methods known from the domain of compiler optimization and parallel compilation as they are for example described in [56] and [17]. Control flow dependences relate directly successive statements (e. g. `S2` and `S3` in Figure 20.1) whereas data dependences relate statements where the depending statement uses a variable that is defined in the other statement (e. g. `S3` and `D1` in Figure 20.1).

- We optimise and parallelise operations related to processing a packet. We consider the way the packet takes from the point where it enters the protocol stack to where it exits. Therefore we combine transition dependence graphs belonging to different SDL processes. We do so by eliminating the inter-layer communication statements, e. g. the statements `S4` and `S9` in the example TLS. The result is a Multi-Layer Dependence graph. We describe the constructions in Chapter 21, for an example see Figure 21.2.

- Third, we identify the path a packet takes through the protocol stack in the so-called common case, from the root node representing the point where a packet is accepted from the environment to the exit node, where the packet is conveyed to the environment. For example, we assume that in the example TLS decision `D1` has one common and one uncommon branch, whereas decision `D2` has two common branches. This resulting graph is called *common path graph*, for an example see Figure 22.2. We will apply our later optimizations only to the common case part of the specification.

- Fourth, we relax dependences on the common path graph in the following steps.

  - *Anticipation of the common case:* In this step we ignore that certain statements depend on a decision, namely for those decisions for which we assumed a common outcome. Henceforth we treat these decision nodes as if no other node depends on their execution. An example is decision `D1` (see Figure 22.2).

– *Parallelising:* We construct a relaxed dependence graph by taking the data flow dependence relation of the CPG and by adding additional dependences which ensure that a node is never executed before the last decision node on which it depends in the control flow dependence relation has been executed (see Section 23.2). For example node `S10` (Figure 23.1, right hand side) is not data flow dependent on decision node `D2`, but still both nodes may not be executed in any order, because the execution of `S10` depends on the evaluation of `D2`. However, `S10` and `S11` are not data dependent and may thus be executed in parallel (meaning in any order).

- Finally, in Chapter 24 we show how suggestions that have been made in the literature to optimize the implementation of communication protocols can be interpreted based on the relaxed dependence graph. We refer to the concepts of *Lazy Messages* (see [123]), and, in particular, *Grouping of Data Manipulation Operations* (see [35], [36] and [2]).

The optimized and parallelised graph now serves as a foundation for an implementation on either a sequential or a parallel machine architecture. We discuss some issues concerning an implementation of the optimized graph in Chapter 25. In Chapter 26 we discuss how to accommodate our dependence analysis method to alternative SDL inter-process communication mechanisms.

## 18.2 Related Work

Efficiency of implementation has become an imperative requirement in the context of high speed protocols. Aspects of hardware and software architecture that increase an implementation's efficiency are discussed in [35], [36], [123], [46] and [142]. Hardware implementations for high speed protocols have been proposed in [89]. In the literature on optimized protocol implementation special attention has been paid to parallelising protocol implementations, so for example in [26] and [150]. However, the parallelisation proposed in these papers depends entirely on the intuition of the designer and thus its efficiency may be non-optimal. Therefore automated support for the parallelisation is desirable. An approach based on the scheduling of parallel tasks generated by an Estelle compiler is presented in [57]. [120] describes the determination of data-flow dependence graphs for parallel implementations of stream processing programs on transputers.

Work presented in [128] analyses data flows in networks of Communicating finite state machines for the purpose of the detection of so-called non-progress properties. [130] suggest a method for the analysis of data flows in distributed communicating processes. The main objective of the work presented here is the detection of unreachable program statements, and the compile time determination of values of program expressions. Closely related work is included in [100] (see also Section 3.2) which analyzes the data- and message flow dependences between communicating processes for static analysis purposes (e.g. compile-time deadlock detection). The algorithms given are highly complex. Our later assumption that there is a one-to-one mapping of send and receive primitives in the code greatly facilitate the message flow analysis in our model and, in fact, makes it trivial.

**Precursors.** An earlier version of our method has been applied to an IP/TCP/FTP protocol stack SDL specification [107].

## 18.3 The Role of SDL

The formal specification technique we consider is the CCITT standardized *Specification and Description Language* SDL [32]. We consider this language because it enjoys wide acceptance in the protocol engineering community. For an overview of SDL see [19] and [145]. The choice of a formal description technique as starting point connects our method to existing techniques and methods in the domain of telecommunications systems and protocol engineering (see for example [108]). We may for example assume that as result of a previous verification step the specifications on which we base our optimization are dead- and live-lock free. Also, conformance tests developed based on the formal specification can be directly applied to the implementation.

Part of our method (dependence analysis and construction of multi-layer dependence graphs) are specific to features of SDL. However, we claim that for many other procedural specification methods an easy adaptation is possible. The later steps (starting with the CPG construction and down to the optimization steps we describe) are independent of the specification method on which the dependence graph is based.

# Chapter 19

# A Discussion of SDL Specifications

In this Chapter we discuss some features of layered SDL specifications of protocol stacks, like communication and concurrency issues. We then argue why 'faithful' implementations of these specifications are inefficient which gives rise to our 'non-faithful' implementation method.

## 19.1 SDL Specifications of Protocol Stacks

### 19.1.1 Communication and Concurrency

SDL is a Formal Description Technique frequently used in the specification of telecommunications systems, in particular for the layered specification of communications protocols. Figure 19.1 shows a schematic model of the representation of a protocol stack by an SDL specification. Each layer of a protocol stack consists of a number of interacting protocol entities. Protocol entities of adjacent layers form a protocol stack (see Figure 19.1). In SDL processes communicate with the environment as well as with other processes via asynchronous communication through process-unique input queues of unbounded capacity. In the example in Figure 19.1 the process `n-Entity`, which represents the layer n protocol machine, communicates with the adjacent layer process `n-1-Entity` via the exchange of `N-1-SDU` messages, and with the user located in the environment by exchange of `UDAT` messages.

The processing inside an SDL process is sequential. However, at run-time all processes belonging to an SDL specification run concurrently, so an SDL specification can be seen as a collection of sequential processes that run in parallel. Each process can be structured into a set of transitions, each transition leading from a symbolic state to another or the same symbolic state, triggered by an input signal (see for example Figure 19.2). The labels that identify the different states allow for loops and branchings in the control flow of the

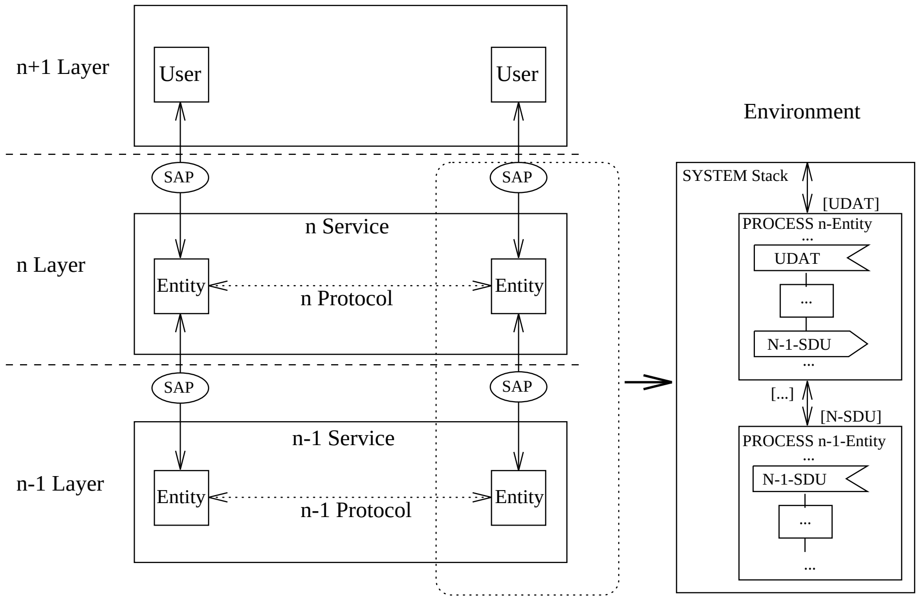


Figure 19.1: Layered protocol architecture and schematic SDL specification of two-layered protocol stack.

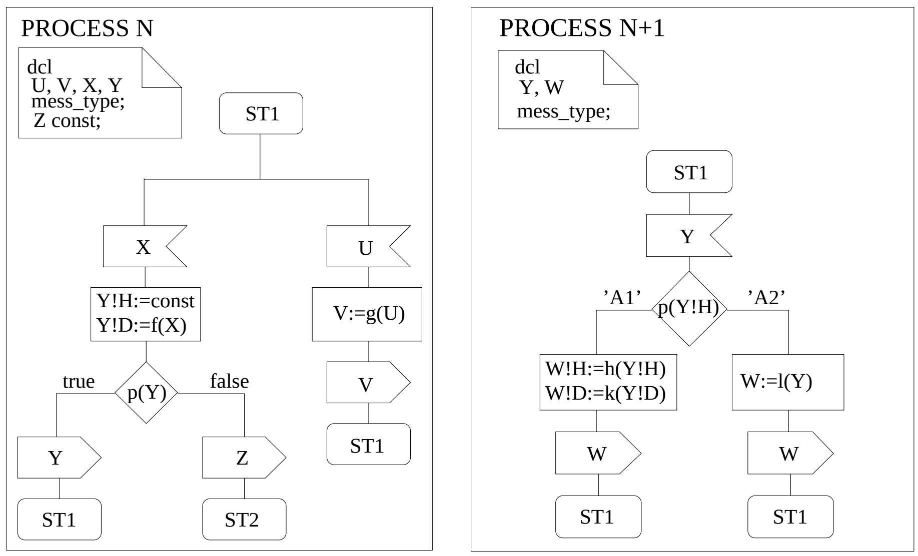


Figure 19.2: The Two Layer Protocol Stack (TLS) Example, SDL-GR representation

```
    PROCESS N;
    ...
     STATE ST1;
S1   INPUT(X);
S2   TASK Y!H:=const;
S3   TASK Y!D:=f(X);
D1   DECISION P(Y);
      (true):
S4      OUTPUT(Y);
        NEXTSTATE ST1;
      (false):
S5      OUTPUT(Z);
        NEXTSTATE ST2;
     ENDDECISION;
S6   INPUT(U);
S7   TASK V:=g(U);
S8   OUTPUT(V);
     NEXTSTATE ST1;
    ...
    ENDPROCESS N;
```

```
     PROCESS N+1;
     ...
      STATE ST1;
S9    INPUT(Y);
D2    DECISION p(Y!H);
       ('A1'):
S10      TASK W!H:=h(Y!H);
S11      TASK W!D:=k(Y!D);
S12      OUTPUT(W);
         NEXTSTATE ST1;
       ('A2'):
S13      TASK W:=l(Y);
S14      OUTPUT(W);
         NEXTSTATE ST1;
     ...
     ENDPROCESS N+1;
```

Figure 19.3: The Two Layer Protocol Stack (TLS) Example, SDL-PR representation

processes. A transition may lead to many successor states, the choices are either made by logical decision predicates, or by checking the different `INPUT` events by which a transition can be triggered. For many examples of protocol and service specifications based on SDL see [19] and [145].

Asynchronous message exchange using the SDL primitives `INPUT` and `OUTPUT` seems to be the mechanism most frequently used for inter-layer communication in protocol specifications. However, the SDL standard introduces further mechanisms. Communication between processes can also be through *remote procedure calls*, through a so-called *viewing* mechanism allowing processes to share variables, and finally an *import/export* mechanism which, however, only hides an asynchronous message exchange. Finally, an extension of SDL by a synchronous communication primitive has been suggested in [72]. In the next Sections we will assume that inter-layer communication is only through asynchronous message exchange. In Chapter 26 we will then sketch modifications necessary to accommodate our method to these alternative communication mechanisms.

### 19.1.2 The Two-Layer Protocol Stack Example

The *Two Layer Protocol Stack* (TLS) example of two protocol processes `N` and `N+1` which we assume to belong to adjacent layers of some protocol stack are presented in Figure

19.2[1]. Both processes are only partially specified:

| | | |
|---|---|---|
| Process `N` | **either** | accepts a message of type `X` from a non-specified lower layer service, which is then processed and sent out as a message of either type `Y` or type `Z`, |
| | **or** | it accepts a message of type `U` which after processing is being sent out as a message of type `V`. |
| Process `N+1` | | accepts a message `Y` which is processed and sent out as a message `W`. |

Hereafter, we shall sometimes abbreviate the terminology by saying *a message* `X` instead of *a message of type* `X`. In SDL, the mapping of the output signals of the sending process to the corresponding input signals of the receiving process is done using a relatively complicated mapping of signal names to signal routes, where the signal routes carry the sender and receiver identification information. For reasons of conciseness of the presentation we will use an abstraction of this mechanism and will simply identify sender and receiver of messages by the identity of the message type. Thus, in the TLS example the message `Y` sent out by process `N` is consumed by process `N+1`.

## 19.2 Inadequacy of 'Faithful' Implementations

By the term *faithful implementation* we refer to an implementation which follows in its structure and in the sequence of operations exactly the original SDL specification from which it is derived. This may for example mean (a) that the SDL specification is directly compiled so that every statement in the SDL specification is mapped to a (sequence of) statement(s) in the implementation, (b) that every SDL process corresponds to a process in the implementation, and (c) that the processes in the implementation communicate using the SDL asynchronous communication mechanism via infinite queues. However, as we argue in the following such a faithful implementation is not efficient.

- *No explicit parallelism:* Although SDL processes run concurrently the processing inside an SDL process is strictly sequential. This means that the structuring of the specification into processes, which in many cases is influenced by general design decisions, determines the degree of parallelism of a specification. It also means that without optimizations the sequential processing of operations inside a process may be inefficient compared to a parallel execution.

[1]Note that this will be the running example throughout the subsequent development of this part of the thesis.

- *Structuring of the specification into processes:* The structure of the specification often means that there is one process per protocol layer peer entity of the protocol (see for example the specifications presented in [19] and [145]). The design of communication protocols is often governed by the principle that *'a good specification is a highly modular and layered specification'*. Though from a structured-design point of view a layered design may be desirable, we stipulate that in order to derive efficient parallel protocol implementations such a layered design is obstructive. This is can mainly be attributed to the fact that the parallel scheduling and combined execution of operations belonging to different protocol layers, which can lead to a considerable gain in efficiency, are inhibited by the layer-wise structuring of the specification. Similar arguments can be found in [46].

- *Asynchronous inter-layer communication via infinite queues:* An efficient implementation of a protocol stack for one peer entity will usually be a non-distributed system. Apparently it is very inefficient to implement the exchange of data in a non-distributed system via asynchronous queues. Instead, the protocol data will be stored in a local memory and the communication between the processes will be by shared variables.

The objectives of our method are therefore to remove the boundaries between processes, to remove the asynchronous communication between processes, and to analyze dependences between statements so that parallel and combined execution of statements belonging to different processes is enabled.

# Chapter 20

# Dependence Analysis for SDL Processes

In this Chapter we explain how a data- and control-flow dependence graph can be obtained by syntactic analysis from an SDL specification. For a definition of the mathematical notation we use here and in later Chapters see Appendix A. First we will explain how transitions as basic building blocks of SDL process specifications can be formalized and then how entire protocol stacks can be represented as graphs, built up from the graphs representing single transitions.

## 20.1 Transitions in SDL Specifications

**Syntactic structure.** A *transition* in an SDL specification is a construct which describes the transition of an SDL process from one *symbolic state* into a successor symbolic state. The body of a transition consists of a collection of statements which we group in the set of statements $S$. We only consider a limited subset of SDL-statements, namely `INPUT`, `TASK`, `DECISION` and `OUTPUT` statements, and we identify one of these four statement types with every element of $S$. The statement `STATE` denotes the current symbolic state and precedes a transition. The statement `NEXTSTATE` denotes the next symbolic state into which the system transits after executing the steps in the transition body. The `STATE` and `NEXTSTATE` statements do not belong to the transition body. We assume that the transition body has the following syntactical structure. A transition starts with an `INPUT` statement. An `INPUT` statement may be followed by a `TASK`, `DECISION` or `OUTPUT` statement, or it may be directly followed by a `NEXTSTATE` statement. The `TASK` statement may be followed by a `DECISION`, `TASK`, `OUTPUT` or `NEXTSTATE` statement. A `DECISION` statement may be followed by a `TASK`, `DECISION`, `OUTPUT` or `NEXTSTATE` statement. An `OUTPUT` statement is the final statement of a transition and always followed by a `NEXTSTATE` statement.

**Justification.** The syntactic subset we have chosen is a reduced subset of the full SDL syntax. For the sake of conciseness we have limited our considerations to the language subset described above but we conjecture that an adequate treatment of other language constructs is a straightforward extension. Furthermore, we conjecture that the language subset chosen here allows for an analysis of most of the standard protocol specifications as presented in [19].

## 20.2 Control Flow and Data Flow Dependences

The syntactical analysis of the SDL specifications that we describe in this Section yields a graph structure over the set of statements $S$ of an SDL specification. This so-called *dependence graph* identifies the two types of dependences between members of $S$, namely *control flow* and *data flow* dependences.

**Dependences.** We now describe the different types of dependences between statements informally.

- Statements, which according to the syntactical and semantical rules of SDL are direct successors, are part of the *control flow dependence* relation *cfd* over the set $S$. A statement of type `DECISION` has two or more directly succeeding statements, all pairs of a `DECISION` statement and it successor statements are part of the *cfd* relation. The execution of a statement directly succeeding a `DECISION` statement depends on the run-time evaluation of the decision predicate. This is represented by a branching of the *cfd* graph. We will in later optimization steps, in particular when parallelising the dependence graph, have to ensure that statements will only be executed when the decision on which they depend has been taken.
- A statement usually describes operations on process variables in which these are usually referenced in two different ways.
  - We say that a statement $S_n$ *uses* a variable $x$ iff it references the variables current value without modifying it. Note that in one statement more than one variable may be used. A typical use of a variable would be to reference its value in the expression on the right hand side of an assignment statement.
  - We say that a statement $S_n$ *defines* a variable $x$ iff it assigns an initial or new value to the variable without referencing its previous value. A typical example is the definition of a variable on the left hand side of an assignment statement. It should be noted that for reasons of simplicity we only allow one variable to be defined in one statement, hence all assignment statements are *single assignment statements*.

- A pair of statements $(s_1, s_2)$ is in the *data flow dependence* relation *dfd* if $(s_1, s_2)$ is in the transitive closure of the *cfd*-relation[1] and $s_2$ *uses* a variable which is *defined* in $s_1$. For simplicity we assume that no re-definition of variable names inside transitions occurs[2]. Also, we assume that every variable name used in a transition is defined inside of the transition, therefore no data dependences from statements in other transitions exist. Function calls are assumed to have no side-effects and to return a single value. Assignments to structured variables are decomposed into component-wise assignments. An `INPUT(X)` statement is a *define* statement with respect to a variable named $X$, an `OUTPUT(Y)` statement is a *use* statement with respect to variable named $Y$[3].

## 20.3 Transition Dependence Graphs (TDG)

**Definition Transition Dependence Graph.** Let $S$, $STT$ and $X$ denote pairwise disjoint sets, the elements of which we call *statements*, *statement types* and *variables*. Formally, we define a Transition Dependence Graph (TDG) as a tuple

$$T = (S, STT, X, sttype, use, define, cfd, dfd)$$

where

- $cfd \subseteq S \times S$,
- $dfd \subseteq cfd^+$,
- $STT = \{input, decision, task, output\}$,
- $sttype \subseteq S \times STT$ is a functional relation (relating a statement to a statement type),
- $use \subseteq S \times \mathcal{P}(X)$ is a functional relation (relating a statement to the set of variable names which are being *used* in it), and
- $define \subseteq S \times X$ is a partial functional relation (relating a statement to the variable name which is being *defined* in it),

satisfying the following conditions:

1. $(S, cfd)$ is a tree.
2. $\forall s \in S$ the following conditions hold:

[1] Thus our definition of the data dependence implies that an 'earlier' statement in the control flow cannot be data dependent on a 'later' one.

[2] This avoids additional *output* dependences, see [126].

[3] The data dependences we consider are purely local to the processes, we do not consider data dependences between processes caused by message flows.

- $(sttype(s) = \{input\}) \leftrightarrow (\mid \{s\} \lhd cfd \mid = 1 \wedge root(S, cfd) = \{s\})$ (an `INPUT` statement has exactly one successor, and it is the root of the tree),
- $sttype(s) = \{decision\} \rightarrow \mid \{s\} \lhd cfd \mid \geq 2$ (every `DECISION` node has at least two successors),
- $sttype(s) = \{task\} \rightarrow \mid \{s\} \lhd cfd \mid \leq 1$ (every `TASK` node has at most one successor), and
- $sttype(s) = \{output\} \rightarrow s \in leaves(S, cfd)$ (an `OUTPUT` statement is a leaf of the tree).

3. $(\forall (v, w) \in dfd)(define(v) \in use(w))$.

## 20.4 Example SDL Processes and TDGs

In the following examples we give the SDL specification of a transition in graphical representation (SDL-GR) on the left hand side of the charts, and the dependence graph resulting from our syntactic analysis on the right hand side. We add labels `Sn` and `Dn` to help us to identify regular and decision statements, respectively. However, these labels are not part of the specification, they are only intended to facilitate the reference to statements in the text.

The example in Figure 20.1 shows a partial view of the specification of a process `N` of which we show only two transitions. The transition on the left hand side leads from state `ST1` via statements `S1`, `S2`, `S3`, `D1` and either via `S4` to a successor state `ST1` or via `S5` to successor state `ST2`, depending on the evaluation of the decision predicate `p(Y)`. This transition is triggered by the input of an `X` signal. In statements `S2` and `S3` the variable `Y` is defined. We assume that a variable of type `mess_type` is defined as a record, and that for example the expression `Y!H` refers to the first component of the record `Y` and `Y!D` to its second component[4]. The evaluation of the decision predicate `p(Y)` determines whether a message `Y` or a message `Z` will be issued, and hence whether the successor state will be `ST1` or `ST2`.

The dependences are as follows. The *control flow* dependence follows the linear sequence of the statements `S1`, `S2`, `S3` and `D1` and then branches to either `S4` or `S5`. The `DECISION` statement `D1` has possible successor statements `S4` and `S5`, the respective control flow dependence edges are labeled for illustrative purposes by *true* and *false*. The data flow dependences are so that `S3` depends on `S1` because of variable `X`, whereas `D1` and `S4` both depend on `S2` and `S3` because of the use of variable `Y`.

Figure 20.1 presents a graphical representation of this TDG which we call $T_1$. *Solid* line arrows represent control flow dependencies, thus elements of *cfd*, and *dashed* line

[4] Think of `Y!H` to stand for the header and `Y!D` for the data part of a protocol data unit or a packet.

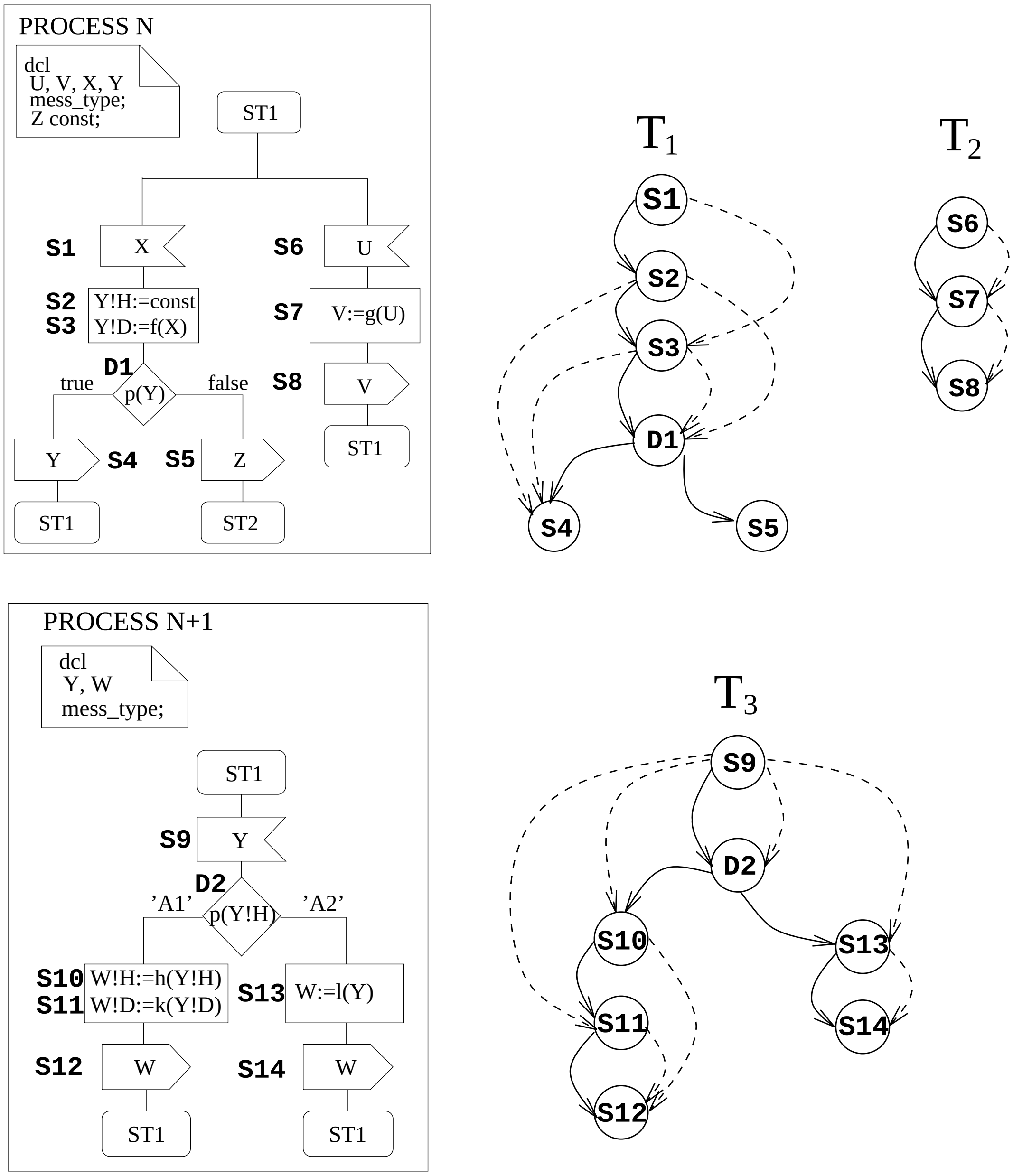


Figure 20.1: Data and control-flow dependence graphs for processes of the TLS Example

arrows represent elements of *dfd*. It should be noted that the labels in the nodes are only annotations which allow us to refer to single nodes more easily. $T_1 = (S_1, STT_1, X_1, sttype_1, cfd_1, dfd_1)$ consists of the following components:

- $S_1 = \{S1, S2, S3, D1, S4, S5\}$
- $X_1 = \{X, Y, Z\}$
- $sttype_1 =$

$\{(S1, input), (S2, task), (S3, task), (D1, decision), (S4, output), (S5, output)\}$

- $use_1 = \{(S1, \emptyset), (S2, \emptyset), (S3, \{X\}), (D1, \{Y\}), (S4, \{Y\}), (S5, \{Z\})\}$
- $define_1 = \{(S1, X), (S2, Y!H), (S3, Y!D)\}$
- $cfd_1 = \{(S1, S2), (S2, S3), (S3, D1), (D1, S4), (D1, S5)\}$
- $dfd_1 = \{(S1, S3), (S2, D1), (S2, S4), (S3, D1), (S3, S4)\}$

When in state `ST1` process `N` may execute two different transitions, depending on whether the signal available at the head of the input queue is of type `X` or of type `U`. Above we described the transition for the first case, for the second case the second transition leads from `ST1` via statements `S6`, `S7` and `S8` to state `ST1`. The syntactical analysis as described above leads to the transition dependence graph $T_2 = (S_2, STT_2, X_2, sttype_2, cfd_2, dfd_2)$ which consists of the following components:

- $S_2 = \{S6, S7, S8\}$
- $X_2 = \{U, V\}$
- $sttype_2 = \{(S6, input), (S7, task), (S8, output)\}$
- $use_2 = \{(S6, \emptyset), (S7, \{U\}), (S8, \{V\})\}$
- $define_2 = \{(S6, U), (S7, V)\}$
- $cfd_2 = \{(S6, S7), (S7, S8)\}$
- $dfd_2 = \{(S6, S7), (S7, S8)\}$

For our later argumentation, which aims at combining multiple processes to one process, we need a further example SDL process. We will use the process named `N+1` first presented in Figure 19.2. The syntactical analysis leads to the transition dependence graph $T_3$ in Figure 20.1. The structure of $T_3$ is quite similar to the structure of $T_1$. It should also be noted that the decision `D2` does not have a boolean evaluations, instead it evaluates to strings `A1` or `A2`. Algebraically, $T_3 = (S_3, STT_3, X_3, sttype_3, cfd_3, dfd_3)$ consists of the following components:

- $S_3 = \{S9, D2, S10, S11, S12, S13, S14\}$
- $X_3 = \{Y, W\}$
- $sttype_3 = \{(S9, input), (D2, decision), (S10, task), (S11, task), (S12, output),$ $(S13, task), (S14, output)\}$

- $use_3 = \{(S9, \emptyset), (D2, \{Y\}), (S10, \{Y\}), (S11, \{Y\}), (S12, \{W\}), (S13, \{Y\}), (S14, \{W\})\}$
- $define_3 = \{(S9, Y), (S10, W), (S11, W), (S13, W)\}$
- $cfd_3 = \{(S9, D2), (D2, S10), (S10, S11), (S11, S12), (D2, S13), (S13, S14)\}$
- $dfd_3 = \{(S9, S13), (S9, D2), (S9, S10), (S9, S11), (S13, S14), (S10, S12), (S11, S12)\}$

**Preview.** The purpose of the transformation of SDL specifications into dependence graphs is to obtain an algebraic representation as input for later optimizations steps (in particular for the parallelisation). The parallel execution of statements is allowed if they are neither directly nor indirectly data flow dependent on each other. In TDG $T_1$ in Figure 20.1 statement `S3` is control flow, but not data dependent on statement `S2`. This reveals an opportunity for parallelising that we shall exploit later on.

# Chapter 21

# Dependence Graphs for Protocol Stacks

As we saw in Chapter 19 protocol stacks are usually specified by a set of independent concurrent processes. Each of these processes consists of a number of transitions. In Chapter 20 we described how to syntactically analyze each of these processes in order to derive a set of transition dependence graphs for each process. As we argued in Section 19.2, it is advantageous to remove the boundaries between layers of SDL processes and to eliminate the inter-layer communication via infinite queues. In this Chapter we describe the necessary steps to combine the transition dependence graphs of different SDL processes and to remove the communication between them. Technically, we perform this in two steps:

- First, we label all TDGs of all processes by so-called *input/output labels*. These labels are the names of the signals exchanged by the `INPUT` and `OUTPUT` statements at the beginning and at the end of each transition.

- Second, we combine all TDGs with matching input/output labels, eliminate the `OUTPUT(X)/INPUT(X)` statement pairs, and perform a cross-layer data dependence analysis. We may do this because we assume that every `OUTPUT` statement can be mapped to a unique `INPUT` statement of another process. The result is a graph which we call *Multi-Layer Dependence Graph.*

## 21.1 Input/Output labeled Transition Dependence Graphs (IOTDGs)

We assume that all transitions we consider for the combination process start with an `INPUT` statement accepting a data packet from an adjacent layer process, and end with an `OUTPUT` statement which delivers the processed packet to the next adjacent layer process. Hence, we assume that all the processing for a packet in a layer process is carried out in the course of *one* transition, and that no looping and branching due to `JOIN` statements inside a transition occurs. Thus, our dependence graphs are always trees. Different transitions starting in different states in one process may exist, but they only represent the process to be in different states (e. g. state *waiting* and state *transmission*). Furthermore, we assume that the packet passing is unidirectional, either from the medium towards the user or vice versa.

**Formal Definition of Input/Output labeled TDGs.** Based on the above stated assumptions on the structure of the SDL Transitions we formalize the concept of labeling of root and leaf nodes of TDGs by the appropriate signal names as follows. Let $T = (S, STT, X, sttype, use, define, cfd, dfd)$ denote a TDG and let $SIG$ denote a set disjoint from any other set in sight, the elements of which we call *signal names*. Furthermore, let $insig \subseteq ((S \cap root(T)) \times SIG)$ and $outsig \subseteq ((S \cap leaves(T)) \times SIG)$ denote functional relations. We define an Input-Output labeled Transition Dependence Graph (IOTDG) as a tuple

$$I = (S, STT, X, SIG, sttype, cfd, dfd, insig, outsig)$$

for which the following conditions hold[1]:

- $sttype(root(I)) = input$, and
- $(\forall x \in leaves(I))(sttype(x) = output)$.

These two conditions imply that all transitions we consider start with an `INPUT` statement and end with an `OUTPUT` statement. In other words, we exclude all those transitions that do not end with an `OUTPUT` statement.

**Example IOTDG** In Figure 21.1 we show the three IOTDGs representing the TDGs for Example TLS.

## 21.2 Multi-layer Dependence Graph (MLDG)

What we have obtained so far is a set $\mathcal{T} = \{T_1, \ldots, T_n\}$ of IOTDGs. $\mathcal{T}$ represents the dependences of all transitions of the specification that we analyze. In this section we describe an algorithm that transforms $\mathcal{T}$ into a set $\mathcal{M}$ of Multi-Layer Dependence Graphs (MLDG). Each MLDG represents the dependences of the processing of one packet or protocol data unit in adjacent layers of the protocol stack. We are interested in following the processing of one packet from the code location where it enters into the protocol stack

[1] We omit the mentioning relations *use* and *define* in this and later definitions of modified dependence graphs.

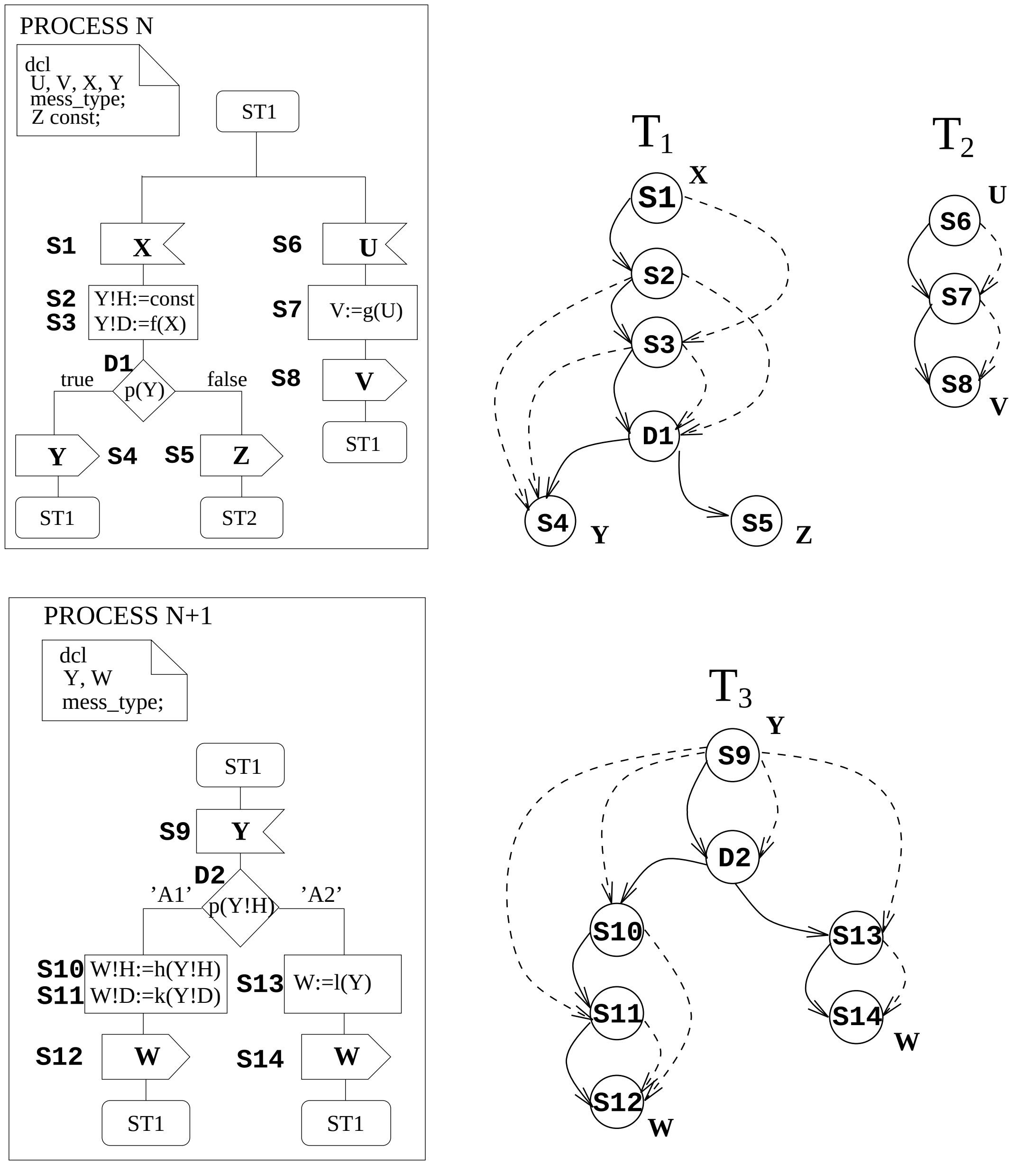


Figure 21.1: IOTDGs for Example TLS

to the location where it exits. In our example this means that we will derive a connected control flow dependence graph from statement `S1`, where the packet `X` enters the processing in process `N`, to the statements `S12` and `S14`, where it exits the stream of processing in process `N+1` as a message of type `W`. Thus we have to compose the individual IOTDGs in $\mathcal{T}$. The criterion for composing two IOTDGs will be that they exchange a message with identical names, e. g. one IOTDG ends with an `OUTPUT(Y)` statement and another IOTDG begins with an `INPUT(Y)` statement. We assume that the names of the types of the messages exchanged are unique at the interfaces between two processes, and that the direction of the message flow is uniquely determined by the message type names. Also, we assume that every `OUTPUT` statement can be mapped to a unique `INPUT` statement. Note that SDL transitions are deterministic on `INPUT` signals, i. e. in one state the future behavior is uniquely determined by the type of the message that is consumed next.

**MLDG Construction Algorithm.** The functioning of Algorithm 1 is as follows. First, a set $\mathcal{T}'$ of initial IOTDGs is selected (step I.). This set contains all those IOTDGs that do not input a message that is output-ed by another IOTDG. The algorithm then loops over all these IOTDGs (III.). The set $\mathcal{Z}$ (V.) contains all those IOTDGs that can be appended to a leaf node of an IOTDG from $\mathcal{T}$. The next loop (VI.) performs the merging of two IOTDGs (VII. to XVI.) for all elements of $\mathcal{Z}$. The merging of two IOTDGs comprises the elimination of the two nodes $x$ and $root(Z)$ by which the two graphs are merged (IX.), this corresponds to the elimination of the `OUTPUT/INPUT` statements. XIII. describes the construction of the new *cfd* relation. Every node which depended on $root(Z)$ is made dependent on every node from which $x$ depended. The construction of the new *dfd* relation (XIV.) is very similar, but we additionally check whether a node on which $x$ depended defines a variable which is used in a node that depended on $root(Z)$. XVII. constructs the result, a set $\mathcal{M}$ of MLDGs.

**Algorithm 1**

*I. SELECT* $\mathcal{T}' = \{T'_1, \ldots, T'_m\} \subseteq \mathcal{T}$ *SO THAT*
$(\forall T'_i)(\forall T_j)(\text{insig}(\text{root}(T'_i)) \cap \bigcup_{j \neq i} \text{outsig}(\text{leaves}(T_j)) = \emptyset)$;

*II.* $\mathcal{M} := \emptyset$;

*III. FOR ALL* $T'_i \in \mathcal{T}'$

*IV.* $M := T'_i$;
*V.* $\mathcal{Z} := \{T \in \mathcal{T} \mid \text{outsig}(\text{leaves}(M)) \cap (\text{insig}(\text{root}(T))) \neq \emptyset\}$;
*VI. WHILE* $\mathcal{Z} \neq \emptyset$
*VII. FOR ALL* $Z \in \mathcal{Z}$
*VIII. SELECT* $x \in \text{leaves}(M)$ *SO THAT*
$(\text{outsig}(x) \in \text{insig}(\text{root}(Z)))$;
*IX.* $S'_M := S_M \cup S_Z - \{x\} - \text{root}(Z)$;
*X.* $X'_M := X_M \cup X_Z$;
*XII.* $\text{sttype}'_M := S'_M \lhd (\text{sttype}_M \cup \text{sttype}_Z)$;
*XIII.* $\text{cfd}'_M := \text{cfd}_M \cup \text{cfd}_Z - (\text{cfd}_M \rhd \{x\}) - (\text{root}(Z) \lhd \text{cfd}_Z)$
$\cup \{\text{domain}(\text{cfd}_M \rhd \{x\}) \times \text{range}(\text{root}(Z) \lhd \text{cfd}_Z)\}$;
*XIV.* $\text{dfd}'_M := \text{dfd}_M \cup \text{dfd}_Z - (\text{dfd}_M \rhd \{x\}) - (\text{root}(Z) \lhd \text{dfd}_Z)$
$\cup \{(v, w) \in \{\text{domain}(\text{dfd}_M \rhd \{x\}) \times \text{range}(\text{root}(Z) \lhd \text{dfd}_Z)\} \mid \text{define}(v) \subseteq \text{use}(w)\}$;
*XV.* $M := (S'_M, STT_M, X'_M, \text{sttype}'_M, \text{cfd}'_M, \text{dfd}'_M)$
*XVI.* $\mathcal{Z} := \{T \in \mathcal{T} \mid \text{outsig}(\text{leaves}(M)) \cap (\text{insig}(\text{root}(T))) \neq \emptyset\}$;

*XVII.* $\mathcal{M} := \mathcal{M} \cup M$

As a result we obtain a set of MLDGs $\mathcal{M}$. Each $M \in \mathcal{M}$ is a multi-edged labeled tree $(S, STT, X, SIG, \textit{sttype}, \textit{cfd}, \textit{dfd})$. Note, however, that not all of the conditions we required for IOTDGs still hold. For example it is not true any more that a node of type *input* has no predecessor in the *cfd* relation.

**Entry, Exit and Branching Nodes.** For a MLDG $M$ we say that a node in *root*($M$) is an *entry* node, that a node in *branchnodes*($M$) is a *branching* node, and that a node in *leaves*($M$) is an *exit* node. An entry node represents a statement where a message (in most cases a packet or protocol data unit) is accepted from the environment, and an exit node refers to a statement in the code where a message is delivered to the environment.

**Example MLDG** Figure 21.2 shows the set $\mathcal{M}$ which we obtain by applying our algorithm to the IOTDGs of our example TLS. It contains two MLDGs, one with root `S1` and one with root `S6`. Note that the *cfd*-relation forms the skeleton of the MLDGs. The nodes `S4` and `S9` have been eliminated, reflecting the elimination of the `OUTPUT(Y) / INPUT(Y)` statement pair. The additional *cfd* pair $(D1, D2)$ has been added. Furthermore, data de-

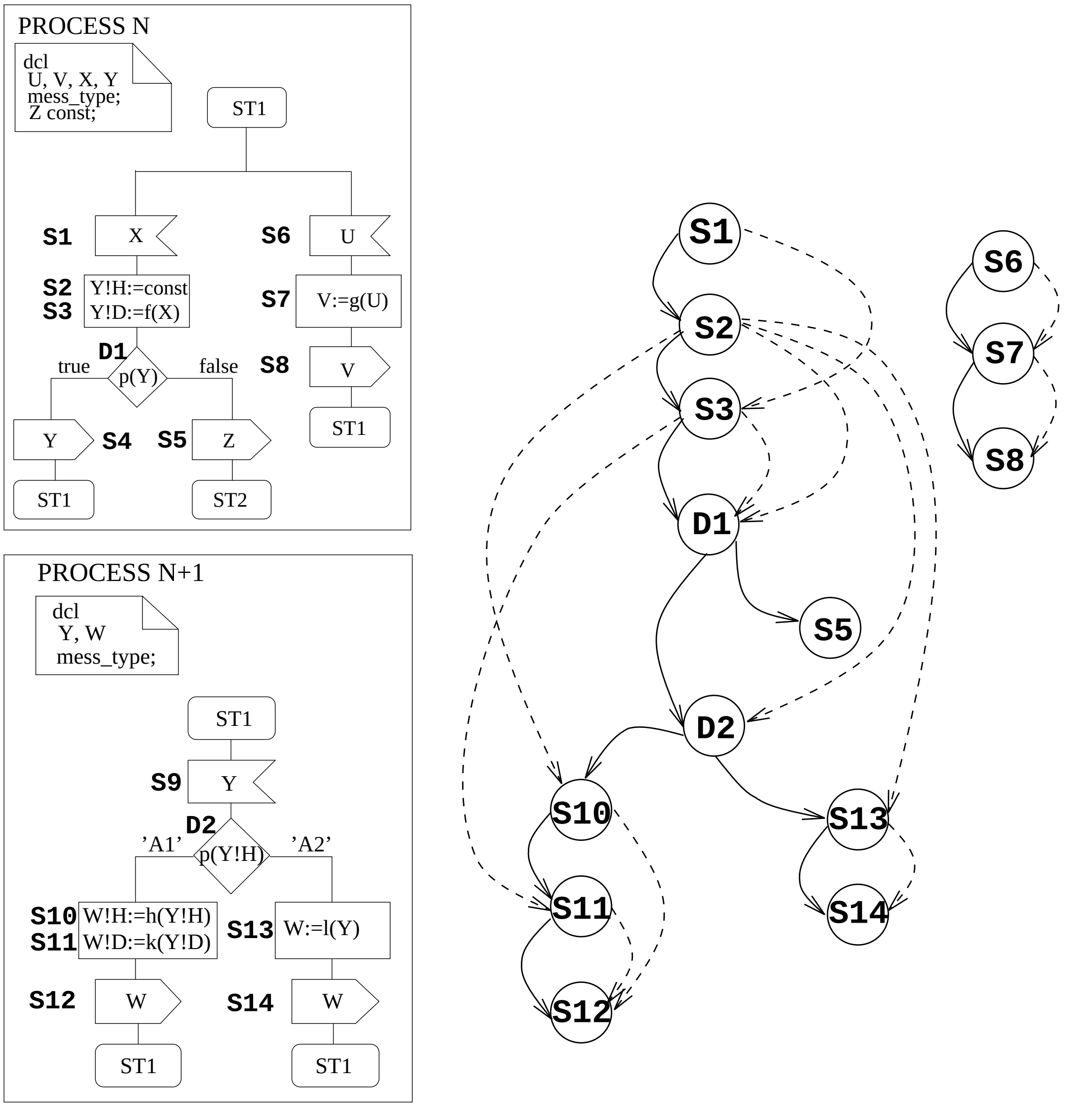


Figure 21.2: MLDGs for Example TLS

pendences between statements of the two merged graphs have been added, so for example $(S2, D2)$.

**Justification for the MLDG construction.** When building the MLDG we modified the original SDL specification in two ways. Firstly, we ignored the asynchronous queue communication mechanism, and secondly, we eliminated the corresponding `OUTPUT / INPUT` statement pair. The justified question arises whether these modifications preserve the correctness of the original specification. We argue that ignoring the queue can be justified because this is a refinement step which preserves two essential queue properties, namely 1. the *safety* property that it is always true that if something is received it must have been sent before, and 2. the *liveness* property that it is always true that if something is sent it will eventually be received.

The safety property is trivially satisfied because the order of the `OUTPUT(X)` and `INPUT(X)` statements is preserved. The liveness property is satisfied if we assume our implementation to be live, namely that every transition which is continuously enabled will eventually be taken.

Furthermore, the elimination of the `OUTPUT / INPUT` statement pair can be justified by the fact that we preserved all control flow and data flow dependences. Thus, the above argument concerning the safety properties now holds for all those statements which are direct predecessors or successors of the `OUTPUT / INPUT` statements that we eliminated.

Another way of looking at it is to consider the traces generated by each of the alternatives. Let $!X$ stand for an `OUTPUT(X)` and let $?X$ stand for an `INPUT(X)` event and let the system be in an infinite loop. Then the language of events that can be observed in the case of asynchronous queue communication can be described by the $\omega$-regular expression $(!X^+?X^+)^\omega$ whereas our implementation generates the expression $(!X?X)^\omega$. Hence, the traces generated by our implementation are a subset of the traces allowed by the original SDL specification semantics, and we argue that our implementation is a correct implementation with respect to a trace inclusion implementation relation.

Concluding we can say that out of the many interleavings of events which are possible according to the original specification we only implement one possible representative, namely the interleaving where a packet is accepted at one end of the protocol stack, entirely processed, and finally handed over at the other end before the next packet is accepted for processing.

As opposed to the ESTELLE related work reported in [71] we do not eliminate the asynchronous queue communication mechanism between adjacent layer processes by replacing these processes by one product automaton because this would induce an extreme blow-up in the complexity of the implementation.

# Chapter 22

# Determination of the Common Path Graph

The later steps of our optimization method rely on the assumption that we optimize the processing of a packet only for the 'common case' (we will come to a clearer understanding of this expression in this Chapter). Restricting the optimization to the common case has the advantage of reducing the complexity of the code that needs to be optimized and therefore leads to more compact optimized code modules. Furthermore, in Section 23 we introduce optimization steps that anticipate certain common decision results according to a common case assumption. These optimization steps, which rely on relaxing the dependences of statements before and after certain decisions, would be impossible without the common case assumption. We consider our common path determination a generalization of the *Common Path* optimization as advocated in [35].

Protocols usually have the task of hiding imperfect behavior of lower layer services from upper layer users. This means that a major part of their functionality aims at detection and treatment of many kinds of exceptions and errors. Exceptions and errors, however, are usually uncommon, in particular in typical high speed communication environments. On the other hand, optimizing the common case implies that we need to take care of uncommon cases using alternate non-optimized error-case implementations. But, as we argued above, because of the low probability of these error handling cases we can tolerate the non-optimized processing of these error cases without risking a considerable degradation of the performance of the protocol. However, not all branching in the control flow can be classified so that *one* branch is common and *all others* are uncommon. It may as well be the case that more than one alternative is a common choice, namely when the branching does not aim at handling exception cases.

Now, what does the term *common case* mean technically? We distinguish the decision edges (outgoing *cfd*-edges of a node with outdegree $> 1$) of the *cfd* relation of an MLDG $M$ disjointly into those which are taken with a probability above a certain value (the *common*

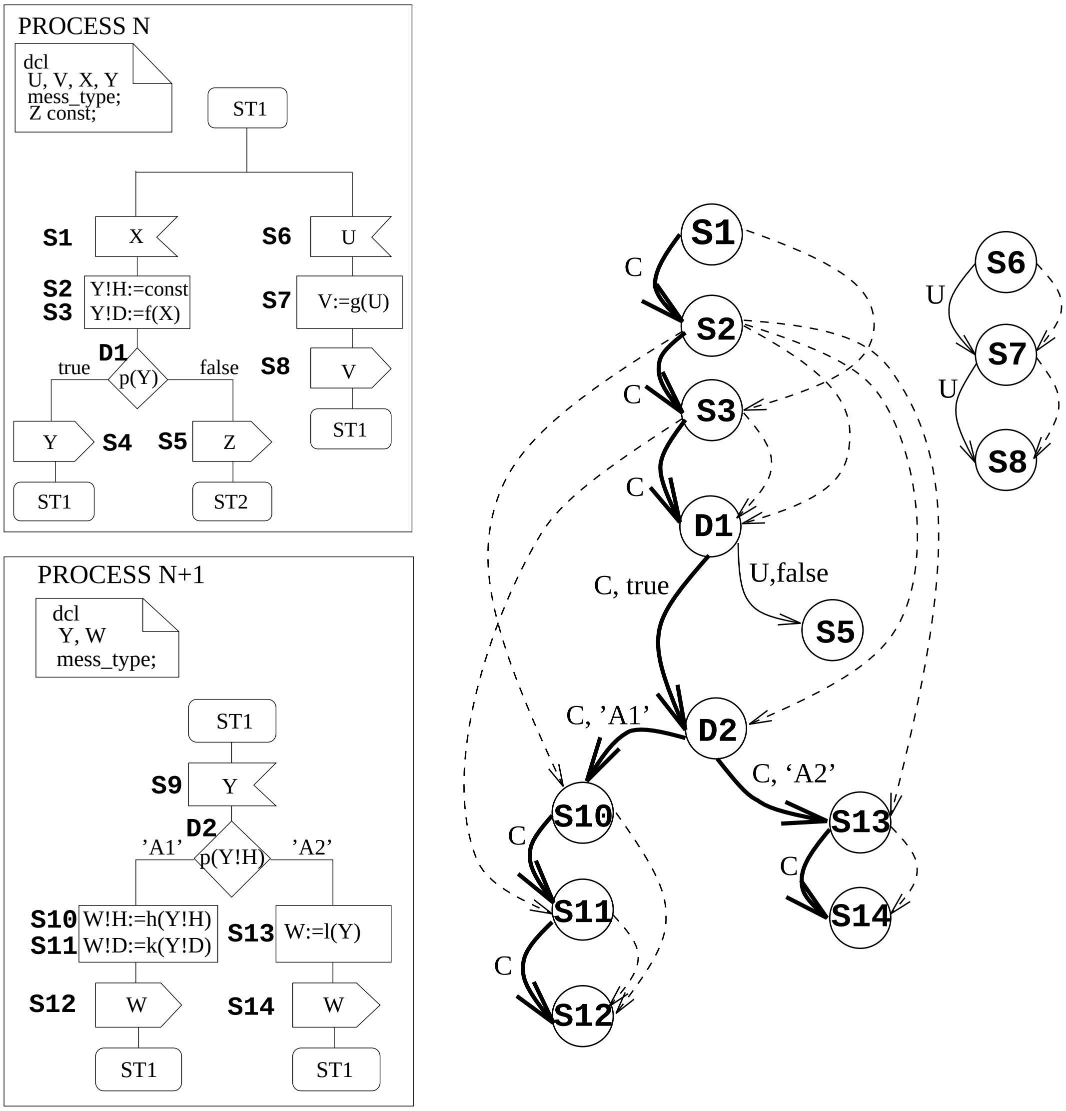


Figure 22.1: Common/uncommon labeled MLDGs for Example TLS

ones, labeled with 'C') and those for which the probability is below a certain value (the *uncommon* ones, labeled with 'U'). The labeling of the decision edges is described in Section 22.2. It defines a *common path graph* which is a subgraph of the *cfd* graph. Hence, our further optimization will only address the common way a packet takes through the protocol stack, along a common path, and not the uncommon cases. In order to obtain what we call the *Common Path Graph* (CPG) we drop those subgraphs of $M$ which start with an edge labeled as uncommon from every decision node (see Section 22.1).

Common path labeling is a step in which we define which path through the *cfd* relation of an MLDG represents the common case. In order to do this we analyze all decision edges of the MLDG and label them with values *common* (C) and *uncommon* (U).

**Common/Uncommon Labeling of MLDGs.** Let $M$ denote an MLDG and let $\mathcal{C} = \{C, U\}$ a set disjoint from any other set in sight. Furthermore let $cul \subseteq (branchedges(S, cfd) \times \mathcal{C})$ a functional relation. We say that $cul$ is a *common/uncommon labeling* of the MLDG $M$.

**Example Common/Uncommon Labeled MLDG.** Figure 22.1 shows a common / uncommon labeling for the example TLS. Note that the labeling of the branching edges yields a tree which represents the normal way the packet takes through the protocol stack from an entry to an exit point. This normal path is common to many packets, therefore the name. The tree is identified in the Figure by bold solid line arrows.

**Discussion.** Whether a decision edge is common or uncommon depends in part on the environment in which a protocol is running. The common / uncommon attributes can thus not be automatically derived from the protocol specification. The attribution has to be provided by the implementor as an input for our method. One way of finding out which decisions are uncommon is to analyze a working implementation using for example a code analysis tool as it has been proposed in [3][1]. In case such analyses are not available it may be necessary to use simulation techniques or estimations in order to determine whether a particular decision edge belongs to the common or the uncommon case.

## 22.1 Common Path Graph (CPG)

Given an MLDG $M$ we now describe an algorithm to remove those subgraphs that depend on an uncommon decision in $M$. Technically, this means that we drop those subgraphs of $M$ which start with an edge labeled as uncommon from every decision node.

**Algorithm for the Construction of the CPG.** Let $M$ an MLDG and let $cul_M$ the corresponding common / uncommon labeling. The algorithm for the construction of the common path graph $C_M$ is as follows:

**Algorithm 2**

*I.* $C_M := M$

*II. FOR ALL* $x \in \mathrm{domain}(\mathrm{domain}(\mathrm{cul}_M \rhd \{U\}))$

  *III.* $C_M := \mathrm{mlprune}(C_M, x)$

## 22.2 Labeling of MLDGs

[1]The authors describe a tool called *Chitra* which analyzes program execution sequences yielding a semi-Markov chain model representing the time behavior of a program.

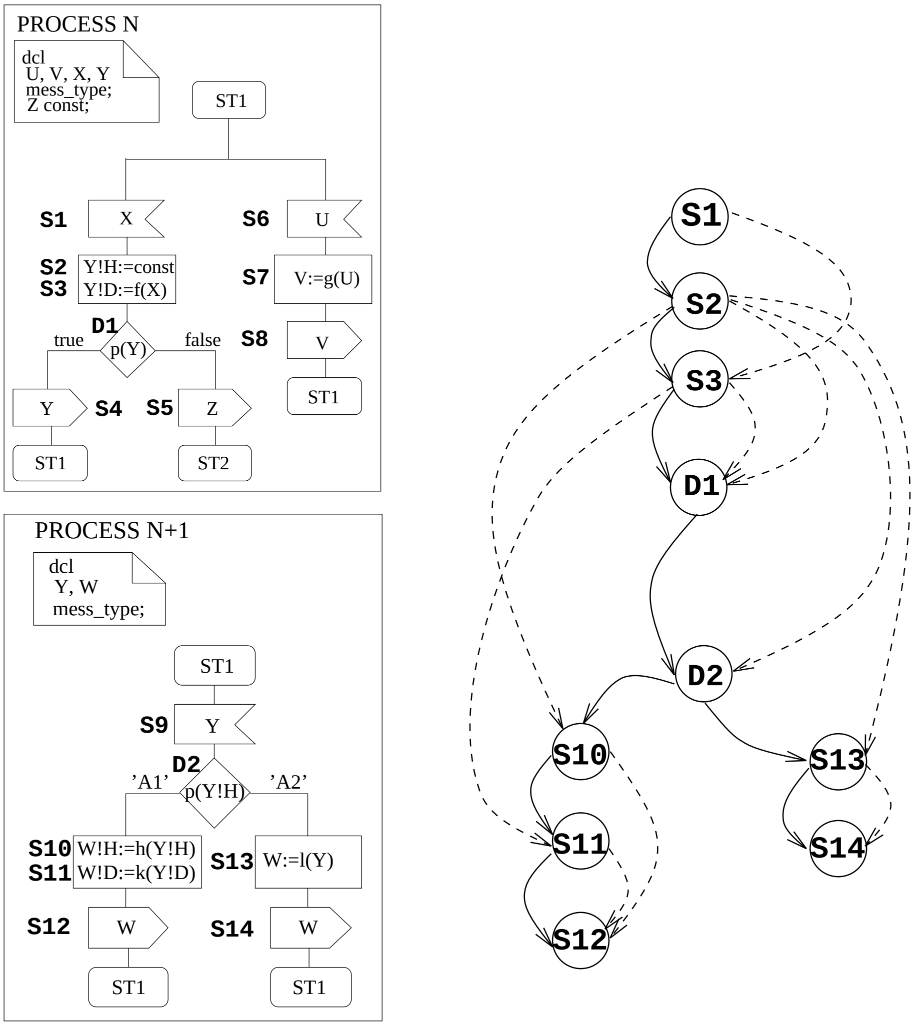


Figure 22.2: CPG for Example TLS

**Example CPG.** In Figure 22.2 we present the CPG derived from the common / uncommon labeled MLDG in Figure 22.1. The subgraph starting with the edge $(D1, S5)$ has been removed. The subgraphs starting in node $D2$ have both been retained as they both represent common branches of a decision. Also, the TDG starting in node $S6$ has been removed as it has no edge belonging to the common path.

**Preview.** In the rest of this document we will focus on the optimization and implementation of the common path of protocols. However, the result of the dependence analysis in Chapter 20 has been a set $\mathcal{M}$ of MLDGs, whereas this Chapter only addresses the determination of a CPG based on a single MLDG. We expect that the user decides which elements of $\mathcal{M}$ he wishes to be optimized by the later optimization steps, based on a similar common/uncommon decision as we discussed earlier.

# Chapter 23

# Construction of the Relaxed Dependence Graph

In the previous Chapters we have shown how a common path graph (CPG) can be derived from an SDL specification based on a control and data flow dependence analysis. In this Chapter we will construct a relaxed dependence graph (RDG) which will be the starting point for later optimization and implementation steps. The relaxation will be mainly a relaxation of the sequentiality constraints imposed by the sequential control flow dependence relations in the CPG. The implementations of the CPG will be, as we argue, correct implementations of the original specification, but they will execute more efficiently than 'faithful' implementations. The relaxation will consist in the following steps:

- *Anticipation of the common case:* Most nodes in the CPG depend on[1] a decision node, and normally nodes depending on a decision node can only be executed when the last decision node on which they depend has been executed. However, we have identified some nodes in the CPG which are of type *decision* but only have one outgoing *cfd* edge (cf. node `D1` in Figure 22.2), so they do not represent a decision along the common path. We therefore *anticipate* the outcome of this decision to be always the way which we predicted when determining the common path. We henceforth treat these decision nodes as nodes representing 'normal' statements, and as if no other node depended on their execution. Thereby we reduce the amount of linear sequential execution conditions. We call the resulting graph an *anticipated CPG.*

- *Parallelising:* We relax the anticipated CPG such that we first strip away the *cfd* relation and only retain the *dfd* relation. However, we need to add some additional dependences which ensure that a node is never executed before the last decision

[1]For a node $x$ to *depend on* a node $y$ here means that $x$ is in the transitive closure of the *cfd* relation restricted to $y$ on its first component.

node on which it depends in the *cfd* relation has been executed. The result is a *Relaxed Dependence Graph* (RDG). In the later implementation two statements can be executed in parallel iff they do not depend on each other in the RDG.

## 23.1 Anticipation of the Common Case

The CPG may contain nodes corresponding to decisions with only one outcome belonging to the common path. Decisions enforce an execution order because a node can only be executed after all decisions it depends on have been taken. Decisions thus limit potential parallelism. To enhance potential parallelism we anticipate the outcome of decisions that have only one outcome in the CPG, which means that we treat such decisions as if they represented nodes of type *task* instead of nodes of type *decision*.

What does this exactly imply? A successor of an anticipated decision can henceforth be executed before the outcome of the decision is known. If the outcome corresponds to the anticipation we have a potential gain in parallelism. However, in the very few cases where we process a packet for which our anticipation of the common outcome of a decision was wrong, e.g. an erroneous packet has been processed and the error was detected, then statements which have already been executed in anticipation of the common outcome of the decision may need to be undone. In Chapter 25 we discuss the handling of the uncommon case in an implementation and argue that there is always a way to handle these situations consistently.

Anticipation of the common case step is applied to the CPG using Algorithm 3. Given a CPG $C$, the algorithm selects all decision nodes from the set $S_C$ that have only one (I.) and changes the type of these nodes to *task* (III.).

**Algorithm 3**

*I. SELECT* $\mathcal{D} = \{D_1, \ldots, D_m\} \subseteq S_C$ *SO THAT*
$(\forall D_i)((sttype(D_i) = decision) \wedge (\mid \{D_i\} \lhd \mathrm{cfd} \mid = 1))$

*II. FOR ALL* $D_i \in \mathcal{D}$ *DO*

*III.* sttype$(D_i) := task$

Note that the result of the algorithm is a graph in which all nodes of type *decision* have more than one successor in *cfd*. All decision nodes are thus branching nodes as defined in 21.1.

**Example.** Anticipating the common case in our example results in changing the statement type of D1 from *decision* to *task*. When the sequential *cfd* dependences have been removed this will allow us to execute node `D1` in Figure 22.2 *after* node `S11`.

## 23.2 Relaxation of Dependences

In this transformation we remove the *cfd* dependences from the CPG in order to increase the potential for parallel execution[2]. More precisely, we remove all *cfd* edges, retain the *dfd* edges, and add some auxiliary dependences. We obtain a graph, called relaxed dependence graph (RDG), which has the same set of nodes as the anticipated CPG, but only one dependence relation on its nodes. We call this relation the *relaxed dependence* relation (*rxd*).

There are three types of precedence constraints which the relaxed dependence graph has to enforce:

- *Data flow dependences*: the data flow dependence relation as defined by the CPG has to be respected (a node using a variable may not be executed before a node which defines that variable).
- *Control flow dependences*: a node which is (directly or transitively) control flow dependent on a decision or root node may not be executed before the decision or root node has been executed[3].
- *Final execution of exit nodes:* Exit nodes must be the last nodes to be executed because they are the point where a protocol interacts with its environment and makes the result of the processing available to the environment. Thus all non-exit nodes must be forced to be executed prior to executing an exit node, and auxiliary dependences need to ensure this.

**The Algorithm.** Starting from an anticipated CPG $C$ we create the RDG and its *rxd* relation in three steps. First we include all elements of the original CPG's *dfd* relation in *rxd*. This will ensure that data dependences are respected in the RDG. Then we examine each node of the RDG to see if it already depends (directly or transitively) on its nearest preceding decision or root node in the original CPG's *cfd* relation. If not, we add a dependence between the examined node and that nearest decision or root node. This ensures that a node is not executed before the last decision it depends on is executed. Finally, we check whether all exit nodes reachable from a given node in the CPG are also dependent of that node in the RDG. If this is not the case, then we add relaxed dependences between the given node and the concerned exit nodes. This last step ensures that the exit nodes which check whether the anticipations of common outcomes of decisions

[2] By parallel execution of the graph we more precisely mean the parallel execution of the implementation of the statements represented by the nodes of the graph.

[3] Note that some former decision nodes were anticipated in the anticipation step and that these node are now considered regular non-decision nodes.

are justified for the respective packet or whether the execution has to be rolled back are actually executed as last steps.

Algorithm 4 is the RDG construction algorithm. Starting with an anticipated CPG C it uses the $cfd_C$ and $dfd_C$ relations to create the $rxd$ relation over $S_C \times S_C$ of the resulting RDG. The algorithm first selects a set D of all decision nodes of the graph plus the root of the graph (I.). It includes all elements of $dfd_C$ into $rxd$ (II.). Then, for every node $x$ of the graph, it finds the nearest node in D from which $x$ is transitively dependent in the $cfd_C$ relation (III.). If in the $rxd$ relation $x$ is not yet transitively dependent of that nearest node, a new auxiliary dependence is added. Next, all nodes except the exit nodes of the graph are examined. A dependence is added (V.) between an examined (VI.) node $y$ and each exit node which is transitively dependent on $y$ in the $cfd_C$ relation of the CPG but not in the $rxd$ relation of the RDG (VII. and VIII.).

**Algorithm 4**

*I. SELECT* $\mathcal{D} = \{D_1, \ldots, D_m\} \subseteq S_C$ *SO THAT*
$(\forall D_i)(sttype(D_i) = decision \vee D_i = root(C))$

*II.* $\mathrm{rxd} := \mathrm{dfd}_C$

*III. FOR ALL* $n \in S_C - \mathrm{root}(C)$

*IV. SELECT* $D_n$ *SO THAT*
$\{s \in \mathcal{D} \mid (s, n) \in \mathrm{cfd}_C^+ \wedge (D_n, s) \in \mathrm{cfd}_C^+\} = \emptyset$

*V. IF* $(D_n, n) \notin \mathrm{rxd}^+$ *THEN* $\mathrm{rxd} := \mathrm{rxd} \cup \{(D_n, n)\}$

*VI. FOR ALL* $m \in S - \mathrm{leaves}(C)$

*VII. FOR ALL* $x \in \mathrm{leaves}(C)$

*VIII. IF* $(m, x) \in \mathrm{cfd}_C^+ \wedge (m, x) \notin \mathrm{rxd}^+$ *THEN*
$\mathrm{rxd} := \mathrm{rxd} \cup \{(m, x)\}$

We call the resulting directed graph $R = (S_C, rxd)$ the relaxed dependence graph for CPG C. It should be noted that R is not a tree any more.

**Example.** Figure 23.1 shows the RDG for the anticipated CPG in Figure 22.2. The dependence graph in the middle is the one obtained after executing step II of the algorithm when only the *dfd* relation of the original graph is retained. The complete RDG is shown on the right hand side of the Figure. It is obtained by adding auxiliary dependences. In order to ensure the dependence of nodes from their closest decision or root node the edges $(S1, S2), (D2, S10), (D2, S11)$ and $(D2, S13)$ have been added. In order to ensure that the exit nodes $S12$ and $S14$ are actually executed last, and to avoid the anticipated decision node $D1$ to be executed last, the dependences $(D1, S12)$ and $(D1, S14)$ were added. We see that $S2$ and $S3$ depend both on $S1$, but they do not depend on each other. This means that once $S1$ has been executed $S2$ and $S3$ can be executed in parallel.

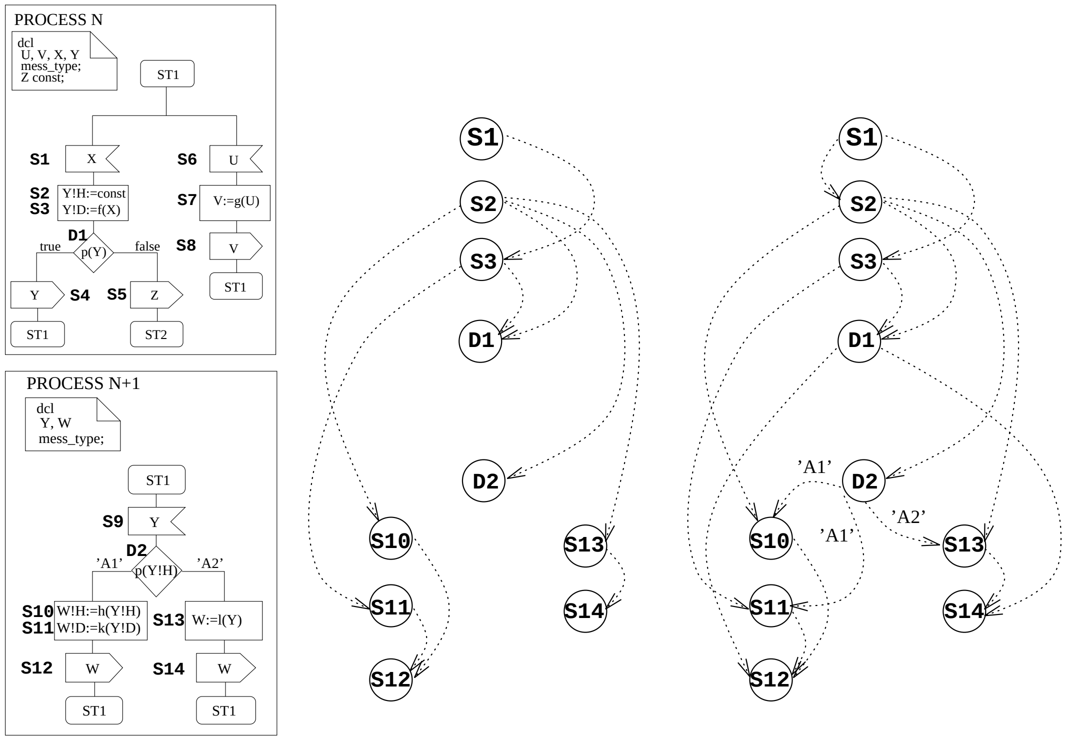


Figure 23.1: Control-flow dependence relaxed (middle) and complete RDG (right) for Example TLS

# Chapter 24

# Optimizations based on the RDG

The RDG will be the basis for an implementation of the common path portion of the protocol stack. The run-time or compile-time scheduling of the operations in the RDG on a given hardware architecture is an important task of an implementation. When scheduling the operations, the scheduler may take advantage of the relaxation of dependences in the RDG. In particular, relative to other operations the execution of an operation may be scheduled in an order different from the order prescribed in the original sequential SDL specification. Therefore, the scheduler may schedule the processing of certain operations whenever it seems optimal, for example when the required resources and data are available. This also ensures that the inherent potential for parallelism in the RDG can be exploited. A further gain in efficiency can be achieved by combining the execution of so-called *Data Manipulation Operations* (DMOs), which also depends on some freedom in the ordering of operations. In the next Section we will discuss how this concept can be interpreted based on an RDG.

## 24.1 Grouping of Data Manipulation Operations.

We call data manipulation operations (DMOs) operations that manipulate entire data parts of protocol data units. Examples are checksum calculation and encryption of data. Combining two such operations into one that does two manipulations at the same time saves an extra storing and fetching of all the data and therefore executes much faster than the non-combined execution of both operations. This has already been demonstrated in [35]. It is also central to the work reported in [36] and [2]. Particularly, it has been shown in [123] that in presence of decisions along the path of execution of a packet in the protocol stack it is advantageous to wait with the execution of DMOs until all decisions have been taken. At that point the set of DMOs to be executed is known and the DMOs can be combined. The technique of deferment of the execution of operations is referred to as *lazy messages* in the literature.

In this section we present an algorithm which manipulates the RDG such that the scheduler is enabled to schedule the execution of DMOs in a combined fashion. The algorithm is a generalization of the 'lazy messages' technique. The grouping of DMOs means that we tightly couple the processing of operations which in the RDG are not dependent, so that their joint processing requires less execution time. In order to enable joint scheduling of operations the RDG has to be modified. It has to be taken into account that when grouping the execution of two DMOs so that one operation depends on a decision higher up in the RDG than the other operation, the higher operation must be executed along every possible path through the RDG. It is thus necessary to distribute DMOs over the RDG.

**Example.** Let us assume that the operations $S3$ (`TASK Y!D:=f(X)`) and $S11$ (`TASK W!D:=k(Y!D)`) in the TLS example (see Figure 23.1) are DMOs. In a real-world example $S3$ might be a translation routine translating every byte of the message `X` and assigning it to the data part of message `Y`, whereas $S11$ might be another such operation on the same data resulting in the data part of message `W`. The identification of DMOs as such is a manual task here, however, it is certainly possible to partly automate the detection of DMOs in the SDL specification.

We include all identified DMOs in an RDG in a set called $\mathcal{DMO}$, hence in this example $\mathcal{DMO} = \{S3, S11\}$. However, in order to allow two DMOs to be combined the following condition needs to be satisfied. It is necessary to require that there may not exist a node which is *rxd* dependent on one DMO, so that the second DMO is *rxd* dependent on this node. Such a node would clearly have to be executed after the first DMO but before the second, thus defeating the combined execution of the DMOs. For two DMOs to be executed at the same time, all decisions on which their execution depends must have been taken before the combined execution can be permitted. In our TLS example, even if $S3$ does not depend on $D2$ we would nevertheless have to execute $S3$ after $D2$ because only then we know if $S11$ will need to be executed at all. To make sure that $S3$ is executed after $D2$ we modify the RDG so that $S3$ depends directly on $D2$ rather than on $S1$, the node it is originally depending on in the RDG.

We have, however, to take into account that $S3$ will have to be executed independently of what the evaluation of the decision node $D2$ is. Thus we need to "distribute" $S3$ over all possible evaluations of $D2$, or, more precisely, over all subgraphs with root note $D2$. Distributing a DMO over the possible evaluations of a decision predicate means that we make one copy of the node representing the DMO for each possible outcome of the decision. In our example there will be two copies, one corresponding to the `'A1'` evaluation of $D2$ (we'll call this new node $S3'_1$), and one to corresponding to the `'A2'` evaluation ($S3'_2$). If $D2$ evaluates to `'A2'` we can execute a combined DMO $S3'_2/S11$. If $D2$ evaluates to `'A1'`, then we execute $S3'_1$ alone.

## 24.2 An Algorithm for Grouping of DMOs

We propose a recursive algorithm that starts at the root of a given RDG (see Algorithm 5). The algorithm is first applied to the root node of a RDG $C$, and then recursively to the whole rest of the graph. The algorithm also takes as input the *cfd* relation of the CPG from which the RDG $C$ was originally derived. This will help in determining from which closest decision or root node a node is *cfd* dependent. Let $B$ be the name of the node the algorithm is currently applied to. The algorithm distributes the DMOs that depend on $B$ over each decision depending on $B$ (we refer to any one of these as $B'$) if and only if other DMOs exist which can only be executed after $B'$. The algorithm is then recursively applied to all decisions $B'$ which depend on $B$.

Starting from a node $B$, the algorithm functions as follows. For each DMO $D$ which depends on $B$ (I.) a second DMO $D_2$ depending on another decision node is searched in the subset of nodes that may be executed[1] if $D$ is executed (II.). If such a DMO exists, the decision node $B'$ depending on $B$ and leading to $D_2$ is found (III.). Then $D$ is removed from the graph (IV.-VI.) and several copies of $D$, called $D'_i$, are created, one for each possible evaluation of $B'$ (VII.-X.). Dependences are added from $D'_i$ to all exit nodes which can be reached from $B'$ with the corresponding evaluation of the predicate (XI.). Once all DMOs depending on $B$ have been treated, the algorithm is applied to all decision nodes depending on $B$ (XII. and XIII.). The algorithm stops when $B$ has no

[1] In the algorithm this subset is found using the transitive closure of the *cfd* relation of the original CPG.

more successors which are decision nodes.

**Algorithm 5**
*RecursiveCombine($B$)*

*I. FOR ALL* $\{D \in \mathcal{DMO} \mid (B, D) \in \mathrm{rxd}\}$

*II. IF* $\exists D_2 \in \mathcal{DMO} \mid (D, D_2) \in \mathrm{cfd}^+$ *AND*
$\{s \in S_C \mid (D, s) \in \mathrm{rxd}^+ \wedge (s, D_2) \in \mathrm{rxd}^+\} = \emptyset$

*III.* $B' = s \in \mathrm{branchnodes}(C) \mid (B, s) \in \mathrm{rxd} \wedge (s, D_2) \in \mathrm{rxd}^+$

*IV.* $S_C := S_C - \{D\}$

*V.* $\mathcal{DMO} := \mathcal{DMO} - \{D\}$

*VI.* $\mathrm{rxd} := \mathrm{rxd} - \{D \lhd \mathrm{rxd} \cup \mathrm{rxd} \rhd D\}$

*VII. FOR ALL* $N_i \in B' \lhd \mathrm{cfd}$

*VIII.* $S_C := S_C \cup \{D'_i\}$

*IX.* $\mathcal{DMO} := \mathcal{DMO} \cup \{D'_i\}$

*X.* $\mathrm{rxd} := \mathrm{rxd} \cup (B', D'_i)$

*XI.* $\mathrm{rxd} := \mathrm{rxd} \cup \{(D'_i, x), x \in \mathit{leaves}(C) \mid (N_i, x) \in \mathrm{cfd}^+\}$

*XII. FOR all nodes* $newB \in B \lhd \mathrm{rxd}$ ***and*** $\mathrm{sttype}(newB) = \mathit{decision}$

*XIII. call RecursiveCombine($newB$)*

The algorithm creates groups of DMOs that are selected by the same evaluation of a decision. Such groups of DMOs can be combined to one complex DMO which can be implemented in a much more efficient way. The actual rewriting of the DMOs into one complex DMO is done during implementation. One method for doing this has been suggested in [2].

**Example.** An application of the algorithm to our example is exemplified in Figure 24.1. We defined two nodes, namely $S3$ and $S11$, to be DMOs. $S3$ is replicated for each evaluation of $D2$, yielding $S3'1$ and $S3'2$. If $D2$ evaluates to `'A1'` then a combined DMO $S3'1/S11$ can be executed. If $D2$ evaluates to `'A2'`, then $S3'2$ is executed alone. In the final implementation the schedule of the operations has to be such that depending on the evaluation of the decision predicate $D2$ either $S3'1$ or $S3'2$ is executed before $D1$, but not both.

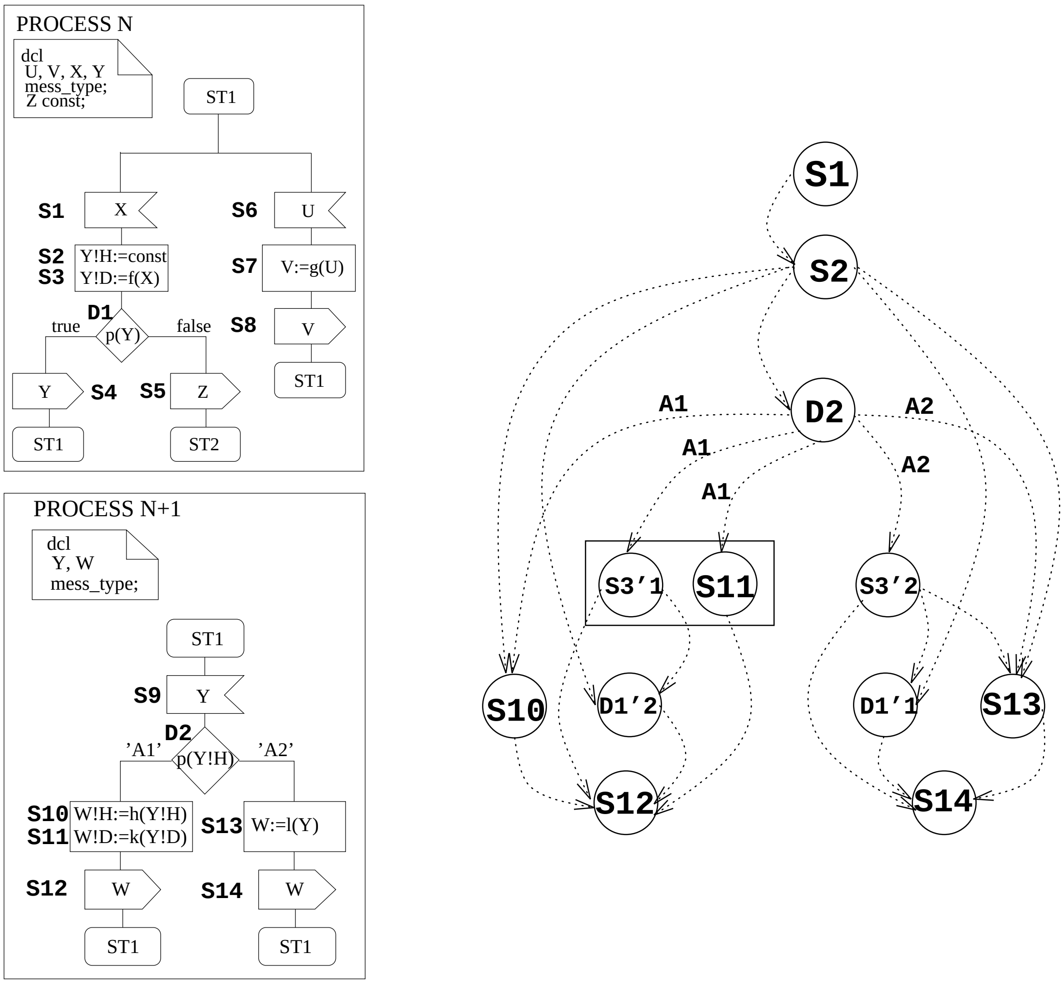


Figure 24.1: Dependence graph with grouped DMOs

# Chapter 25

# Implementing the Optimized Graph

In the previous chapters we have shown how a multi-layer dependence graph (MLDG) can be derived from an SDL specification, how a common path graph (CPG) can be extracted from the MLDG, and how this CPG can be transformed into a relaxed dependence graph (RDG). This chapter addresses final aspects of the method, namely the implementation of the considered protocol stack based on the derived RDG.

Implementing the RDG means that we map the statements corresponding to each node to a set of software- or hardware instructions. When performing this mapping we have to consider the following three aspects:

- First, we have to respect the *ordering constraints* on the operations as specified by the *rxd* relation of the RDG.

- Second, assuming the availability of parallel processing resources, the operations have to be *scheduled* on the hardware resources according to the ordering constraints, the qualitative resource requirements, and the expected time consumption of every operation on particular hardware components.

- Finally, we have to take care of the fact that the RDG only describes the common case of packet processing, i. e. we need to provide for an alternate processing when a packet belongs to an uncommon case. This includes assuring that the system is in a consistent state after a packet has been detected not to comply with the common case.

## 25.1 Preserving Ordering Constraints

The RDG imposes a set of ordering constraints on the operations to be executed. In general, this is a partial order. If we look back at the RDG in Figure 23.1 it is easy to see that for the subset $\{D2, S10, S11\}$ of operations the following partial order, expressed informally in terms of a process algebra like behavior expression, is `(D2; (S10 || S11))`. Any interleaving trace derived from this expression is the trace of a valid implementation, e. g. the traces $(D2, S10, S11)$ and $(D2, S11, S10)$. However, for the exact derivation of an optimal implementation these possible interleavings do not provide sufficient information, in particular for the following two reasons.

- The operations may be executed in a machine environment with limited parallel processing resources, so the theoretical maximal possible degree of parallelism may not always be attainable. Also, the processing resources may not be homogeneous and certain operations may have particular requirements of the particular characteristics of the resources on which they are to be executed.

- Furthermore, operations are not atomic, as the interleaving model suggests, but they have a duration. This also means that they may be executed partly simultaneously, and one operation may be executed simultaneously with a sequence of different other operations. All relations that are valid for two or more convex intervals are possible for the operations in the RDG. However, for two operations $A$ and $B$ where $B$ depends on $A$ we require that $A$ has to be finished before $B$ starts.

## 25.2 Scheduling

Concludingly, the RDG defines ordering constraints on the operations that need to be executed in an implementation. However, in order to come to an implementation the target hardware architecture also has to be taken into account. Assuming that the implementation is supposed to run on a parallel hardware architecture this leads to the problem of deriving an optimal schedule. The schedule does not only reflect the order of the execution of operations, but also answers questions about how long an operation will occupy a certain hardware component. Last, because we cannot assume that all parallel components of the hardware have equal qualitative characteristics, the schedule will also have to respect qualitative constraints, like which operation has to be executed on which hardware component.

## 25.3 Ensuring Consistency - Treatment of Uncommon Cases

The RDG we derived from the initial specification is based on the so-called common case assumption. This means that we assume that the packets processed inside the RDG all comply with the assumptions made to determine the common path through the protocol stack, e.g. that they are error-free, that they do not require exception handling etc. As a consequence we anticipated the results of some of the decisions along the common path. This means that we presumed a certain evaluation of some of the decisions and removed dependences of statements depending on these presumed decisions. In other words, some operations have been decoupled from the decision predicates by which they were 'guarded' in the original specification. This may lead to inconsistent sequences of operation. For example, a division by zero may be executed concurrently with the test for non-zeroness of the respective operand if we assumed that non-zeroness is the common case. In the original specification of our example TLS (see Figure 20.1) the execution of `D2` (through `S4`) depends on the evaluation of decision `D1` to `true`. However, in the RDG in Figure 23.1 `S4` does *not* depend on the evaluation of `D1`. This implies that `D2` may even be executed *before* `D1` is evaluated. A possible inconsistency can only be detected when the processing of a packet reaches an exit node.

Apparently, consistency ensuring mechanisms have to be applied. This leads to the following three requirements.

- First, as we argued before we need to have a faithful and complete backup implementation of the whole protocol stack available. The backup implementation covers all decisions, exception handling mechanisms etc. as foreseen in the original specification. It takes over control when the optimized implementation detects that a packet violates the common case assumption, namely if a test does not evaluate to the value which was anticipated during the common path determination.

- Second, because we saw that operations may be executed prior to the evaluation of a decision predicate by which they were originally guarded, all operations must be robust. This means that no matter when an operation is executed it is ensured that the system will not enter a failure state.

- Third, when the processing control is handed over to the classical implementation, the state of the system when the packet has entered the protocol stack through an entry node has to be reestablished. To ensure that this initial state can always be reestablished we suggest using the following mechanism.

  – We distinguish operations in reversible and irreversible operations. We claim that most operations are irreversible, in particular operations reading data or copying data from one storage location into a register, modifying the data, and writing it to a second storage location. Operations of this sort are reversible (because the unmodified data is still available in the old location), and they can easily be undone when control needs to be transferred to the backup implementation.

- All those operations which are irreversible, and we claim that that is only a minor part of all operations, need to be secured by a checkpointing mechanism. This means that the data which is affected by these operations will be checkpointed before the respective operation is executed. If not all decisions are evaluated in the way it was anticipated, i. e. the packet is not processed according to the common case, the checkpoint information can be used to undo all irreversible operations.

**Discussion.** It arises the justified question how advantageous our optimization is in light of these time consuming consistency ensuring mechanisms. We claim that the resetting to the initial state only occurs very infrequently, namely when an uncommon case has been reached. This holds in particular in high speed communication protocols where error rates are low, and flow control mechanisms are very often omitted. Also, we think that only very few operations in high speed protocols are reversible and require a checkpointing for the state of the protocol stack. However, when uncommon cases occur more and more often it is clear that there will be a break-even point between the efficiency gain due to the parallel and resequenced operation, and the resource consumption for consistency ensuring mechanisms.

## 25.4 Case Study: an IP/TCP/FTP Protocol Stack

In [107] we presented the application of an earlier version of our method to the SDL specification of an IP/TCP/FTP protocol stack. The SDL specification on which we based this example was developed in the context of our work. We first mapped operations or sequences of operations in the protocol stack to statements in the SDL specification. (The granularity of the resulting set of operations in the SDL specification greatly influences the complexity of the dependence graphs). We identified 21 statements (operations and decisions) in the specification. Some of the operations were procedure calls which hid more complex operations. We determined a common path, constructed a dependence graph, and determined a relaxed dependence graph.

Based on the relaxed dependence graph we combined two DMOs, namely the TCP checksum calculation, and the translation from internal into external ASCII representation inside the FTP layer. We scheduled the operations on a hardware architecture with limited parallelism which consisted of independent *medium* and *host* interface components, two *FIFO queues* feeding the interfaces, a special purpose *Data Manipulation Unit*, a general purpose *microprocessor*, and a *random access memory* unit. We assigned resource consumptions and qualitative resource constraints to each of the operations, and applied an enumerative scheduling algorithm to this problem.

In the optimal schedule the both DMOs were executed jointly, and in parallel with other operations (both DMOs were scheduled to be executed on the data manipulation unit, whereas the other operations were executed in parallel on the microprocessor). The optimal schedule would have been executed within 413 process cycle time units, whereas the strictly sequential execution of the packet processing along the common path in a 'faithful' fashion according to the SDL specification would have taken 1036 processor cycle time units.

# Chapter 26

# Alternative SDL Communication Mechanisms

We assumed that the inter-layer communication mechanism in the SDL protocol stack specification relies on the asynchronous exchange of messages by means of `INPUT` and `OUTPUT` statements. For all we know, this is the common way to specify data exchange at layer interfaces in SDL specifications, c.f. the many examples in [19] and [145]. However, SDL offers alternative inter-process communication mechanisms. We do not know whether these are used anywhere for specifying inter-layer communications, and we do not necessarily recommend using them for this purpose, however we feel that a short discussion of how our dependence analysis method extends to these communication mechanisms is necessary.

We shall discuss the following alternative communication mechanisms in the following Sections: [72] suggests extending SDL by a *synchronous communication* primitive. The SDL-92 standard [32] introduces a synchronous communication mechanism by means of a *remote procedure call* (rpc) mechanism (c.f. [53]). Furthermore, there are two mechanisms which rely on so-called *shared values*, namely the *viewing* mechanism and the *export and import* of variable values.

## 26.1 Synchronous Communication Primitive

[72] suggests extending SDL by a multi-way synchronization primitive called `SYNCHRONIZATION`. It refers to a signal type, and all processes having this signal type in their input alphabet (or, more precisely, having this signal type in their incoming signal list) will have to synchronize in order to jointly execute the synchronization statement. Note that this is only a proposed extension and not included in the current SDL standard.

Let us restrict the multiway synchronization to the two process case and let us assume that a process `A` wishes to synchronize with process `B` by sending a signal of type `X`.

Let us furthermore assume that X has one parameter, and that upon synchronization A wants to assign the value of the parameter to a variable u, whereas B wants to fill the parameter position with the value of its variable v. Finally, let both A and B have signal name X on their incoming signal list. Then, A would contain a `SYNCHRONIZATION X(?u)` and B would contain a `SYNCHRONIZATION X(!v)` statement. Both statements will only be executed in the course of one atomic action if in one global system state both processes are prepared to execute the synchronization. Note that like the synchronous communication mechanism we defined in Section 7.9 for MFGs and MSCs the synchronization here is directed, representing a direction of data flow.

In the context of the dependence analysis for inter-layer communications in the protocol stack the `SYNCHRONIZATION X(?u)` is a *define* statement with respect to variable u. The `SYNCHRONIZATION X(!v)` statement is a *use* statement with respect to variable v. Note that one `SYNCHRONIZATION` statement may contain data flows in different directions.

## 26.2 Remote Procedure Calls

Synchronous communication has been introduced into the SDL-92 standard by means of a *remote procedure call* (rpc) mechanism. Procedures can return values, and if they are labeled `exported`, then they can be declared as `imported` by another process. This other process may call such a remote procedure and receive the return value of the procedure. The calling process will be blocked until the remote procedure terminates.

As mentioned above we currently do not know of any example of the use of remote procedure calls for inter-layer communication, but in principle the rpcs could be used for this purpose in the following way. Assume that a process A wishes to receive a value from process B, and that process B has this value stored in a variable U. Furthermore, assume that B contains the declaration of a remote procedure P which does nothing else than taking the global variable U, assigning it to a reserved name called `result` (i.e. `result := U`), and then terminates[1]. The value in `result` will be the value returned to the calling process. Now, to achieve synchronization, process B has to be in a state in which it can execute P (for details concerning when this is the case see [53]), and process A has to call P by executing a `V := call P` statement. A will be blocked until the value of P has been returned and assigned to V. According to [53] the notification of a call to the procedure P as well as the notification that P has been executed and the return of the result value is implemented as an implicit asynchronous signal exchange. Hence, the rpc mechanism is merely an implicit protocol enforcing process synchronization, based on the standard SDL asynchronous signal exchange mechanism.

[1] Note that remote procedures are local scope units, but variables of the global scope are accessible and modifyable.

From the point of view of the data flow dependence analysis the above scenario would need to be treated in the following way. Inside process `A` the `V := call P` statement is a *define* with respect to the variable `V`. However, to determine the dependences relating to the procedure call requires some more apparatus, and for an exact definition of the dependence analysis we refer the reader to [56]. Informally, the `V := call P` statement is of course also an indirect *use* statement with respect to the variable `U` of process `P`.

## 26.3 Shared Values

**Viewing Mechanism.** The *viewing* mechanism allows to access variables defined in other processes within the same block. A process `A` making a variable `X` available to another process would declare this variable as *revealed*: `dcl revealed X ...;`. The process `B` would access the value of `X` by a *view* statement: `view (process A X);`. Apparently, the assignment of a value to `X` in process `A` would mean a *define* of the variable. In process `B`, the access of the variable `X` of process `A` would mean a *define* of the variable `A` inside the scope of process `B`. According to [145] the usage of the viewing mechanism is not recommended and is only included in the SDL language for backwards compatibility reasons with older versions. Also, it should be noticed that the viewing mechanism provides for no synchronization between the processes, i.e. the process `B` in the above example will not know when the value of variable `X` has been updated. To make this communication mechanism useful in the context of inter-layer communication would require the implementation of additional synchronization mechanisms.

**Import and Export.** A process `A` may export a variable `X`, which has been declared `exported`, by executing the statement `export(X)`. A process `B` may import the variable by use of an `import` statement, for example in the course of the following assignment: `task Y:= import(X, A)`. It should be noted that according to [19, 145] the export/import mechanism is implemented as an asynchronous signal exchange like for `OUTPUT` and `INPUT` statements. Hence, the import/export mechanism is not a true *shared memory* mechanism. The treatment with respect to the data flow analysis is hence similar to the treatment of `OUTPUT` and `INPUT` statements. The `export(X)` statement is a *use* of variable `X`, while `task Y:= import(X, A)` is both a *define* of `X` in the process scope of `B` and a *use* of $A$, as well as of course a *define* with respect to `Y`.

# Chapter 27

# Conclusions

In this Part of the thesis we presented formalizations and algorithms for the derivation of optimized protocol implementations from SDL specifications. We started with a syntactical dependence analysis for SDL processes. We then showed how multiple dependence graphs can be combined to multi-layer dependence graphs. Next we determined the common path graph, a subgraph of a multi-layer dependence graph which represents the common case of processing of a packet in the protocol stack. This graph was the basis for an optimization by anticipating the evaluation of some decision statements in the CPG, and then by relaxing the dependences. This essentially meant to omit control flow dependences and to only consider data flow dependences and dependences that express the dependence of a statement from the evaluation of a decision predicate. We called the result a relaxed dependence graph. When scheduling the operations on a given hardware architecture the scheduler may take advantage of the relaxation of dependences in the RDG in particular by scheduling certain operations at a different point of time compared to the sequential execution in the SDL specification. In particular we showed how the optimization concepts of lazy messages and grouping of Data Manipulation Operations can be interpreted based on the Relaxed Dependence Graph.

In general, implementing the RDG means that we map the statements corresponding to each node to a set of software instructions or hardware modules. When performing this mapping we have to consider three aspects. First, we have to preserve the ordering constraints imposed by the RDG. Second, assuming the availability of parallel processing resources the operations have to be scheduled according to the ordering constraints, the resource requirements and the expected time consumption of every operation. Finally we have to take care of the fact that our RDG only addresses the common case, i. e. we need to solve the problem of the alternate processing when a packet belongs to the uncommon case. A preliminary description of these implementation aspects is given in [107] (see also Chapter 25).

# Part V

# Conclusion

# Chapter 28

# Concluding Remarks

We arrived at the point where it is time to recapitulate the chief technical arguments made so far, and to point at directions for future research.

## 28.1 Recapitulation

We have been addressing two major problems arising from the use of formal methods in the telecommunications systems engineering process.

### Specification and Semantics

First, we were concerned with formal semantics for specification methods in telecommunications systems engineering.

**Semantics for Message Flow Graphs and Message Sequence Charts.** We were looking in particular at a specification formalism called *Message Sequence Chart*, which is a specialization of the *Message Flow Graph* concept. We noticed their occurrences in different domains of software engineering, namely in telecommunications engineering but also in the analysis of concurrent code and in object models, and we argued for a need to define a formal semantics for these charts.

Our requirements were that the semantics are infinite traces of interleaved communication events, and that these traces have to be representable by a finite state automaton because MFG and MSC specifications represent inherently finite-state systems. We furthermore required that the semantics handles both asynchronous and synchronous communications, even in the same chart. We briefly described the requirements on the GEODE toolset and observed that it shares the finite-state assumption with our semantics.

The basic model we chose as representation for the sets of traces specified by an MFG was the Büchi automaton, however we noticed that liveness properties are underspecified in MFG specifications, and that these need to be added to MFG specifications in order to determine the end-state selection criteria for the Büchi automaton. We noticed that temporal logic can be a more flexible tool for the specification of liveness properties compared to the definition of end-state sets for the Büchi automaton, and we showed how to interpret the global state transition graphs resulting from our interpretation of an MFG specification as models of propositional linear time temporal logic. Furthermore, we proved that MFG specification are expressibly equivalent to Büchi automata.

We located problems and ambiguities in some of the concepts underlying MFG specifications. First, we showed that the unimpeded use of conditions in MFG specifications leads to non-local choice conditions. The resolution of these hinges on the availability of hidden devices capable of remembering potentially unbounded execution histories, which contradicts our finite-stateness requirement. Furthermore, we proved that crossing message arrows may entail hidden and non-explicit assumptions on the environment behaviour, in our view an undesirable feature.

We also investigated the ITU-T standards document Z.120 for MSCs and its Appendix B which standardizes a Process Algebra Based semantics, and we related our approach to both documents. We discussed problems in the relation of the textual and the graphical representation of MSCs as described in Z.120. This entailed as we showed that the semantics in Annex B of Z.120, which relies on the textual representation of MSCs, is not based on a well-defined language. Furthermore, we criticised that from a pragmatic viewpoint the process algebra based semantics of Z.120 Annex B is the less useful one, because it avails itself to state based verification methods only indirectly by means of a further translation step.

**Quality of Service Specification.** We noted that hard real-time constraints are an important class of requirements in telecommunications systems engineering, and that the semantics of the specification language SDL is inexpressive with respect to hard real-time constraints. This is mainly due to the timer mechanism relying on asynchronous message exchange. We put forward the idea to use real-time extended temporal logics as a complement to SDL specifications in order to express hard real-time constraints.

This entailed the definition of a common model theoretic foundation for SDL and linear time temporal logic. In the course of doing so we exemplified how SDL can be interpreted in terms of logic, and we clarified the relation of symbolic states and system states in SDL specifications. By usage of complementary Metric Temporal Logic (MTL) formulas in combination with SDL specifications we showed how to specify a number of real-time related Quality of Service constraints for SDL specifications. The previously defined state-transition based semantics for MFGs allowed us to use MTL specifications also in combination with MFGs and MSCs.

### Implementation

Next we considered a method for the derivation of optimized, parallel implementations from SDL specifications of protocol stacks. We argued that the efficiency of the protocol processing is crucial because communication bandwidths have increased significantly in recent years and turned the protocol processing machines into a performance bottleneck.

First, we noted that it is inefficient to implement SDL specifications of protocol stacks 'faithfully'. We noted that in particular sequential control flow dependences as well as the buffered inter process communication mechanism and the distribution of protocol functions of different processes obstruct an efficient implementation.

To overcome this deficiency we suggested a data- and control flow analysis of the transitions in SDL processes. Then we assumed that packets are processed in a sequence of steps on their way through the protocol stack and we removed the boundaries between different protocol layers. Next we determined a so-called common path of a packet through the protocol stack, which allowed us in the next step to relax the dependences inside the graph by abstracting away from the sequential control flow dependences. Based on the relaxed dependence graph we were then able to perform a grouping of data manipulation operations, and the resulting graph finally acted as a basis for the implementation of the protocol stack, subject to the solution of a scheduling problem.

The fact that we have provided a rigorous formal description of our method clearly supports the implementation of our algorithms in a comprehensive toolset. It also connects our method well to other formally supported steps in telecommunications systems engineering, like testing, verification and validation.

## 28.2 Directions for Future Research

### The Semantics of MSCs and MFGs

The definition of the semantics has provided us both with an operational model for an MFG specification as well as with insight in the inherent intricacies of specifications based on MFGs. While this has been a worthwhile endeavour, and while the operational model is certainly useful when wanting to use verification algorithms for MFGs, we suggest that from the abstraction point of view the translation of MFGs into a logic based framework like TLA [101] could be a worthwhile extension of our work[1].

This for different reasons. First, a translation into TLA would make the underspecification of liveness properties in MFG specifications even more apparent. Second, logic based formalisms avail themselves very directly of formal verification, namely theorem proving. Finally, as noted above, MSCs and MFGs enjoy a high degree of acceptance amongst

[1]See [51] for the concerted use of MSCs and TLA in the specification of a telecommunications service. The relation between MSCs and TLA remains informal there, though.

telecommunications systems engineers. The translation into TLA may therefore convince many engineers to start incorporating the use of logic based methods into their design methodologies, albeit we concede that the software tool support for TLA is currently not satisfactory.

From a pragmatic point of view we hope that the constructions presented here will easily find their way into software tools relying on MSC and MFG specifications. As mentioned above, a by-product of the semantics is that the GSTG we define allows for a simulation of the behaviour specified by an MSC, and we see this as one of the most important practical uses of our semantics, next to availing MSCs of formal verification methods.

### QoS Specification

**Probability Aspects.** So far we described the specification of QoS requirements based on real-time constraints. However, many practical QoS requirements require probabilistic expressiveness, so for example in the informal requirements that every sending of a packet will result in a delivery of a packet *with probability* $\geq p$ within $t$ time units. We think that it is desirable to extend our method of complementary QoS specification to these aspects.

The essentials of extending real-time temporal logics to probabilities have already been presented in the literature. [66, 65] and [10] have extended real-time temporal logics by probability operators, albeit both approaches are based on branching time temporal logics whereas MTL relies on linear time models. The underlying state-transition model is extended by a probability labeling yielding a generalized markov-chain model in [66, 65] and [10]. It is fairly straightforward to extend the concept of complementary specifications to probability aspects and to combine SDL and probabilistic real-time temporal logic specifications. This means that the state transition model obtained from SDL specifications must be extended by an assignment of probability labels to the transitions. Both [65] and [10] provide model checking algorithms for the extended logics they present, where the model against which the specified requirements are checked is the underlying generalized markov chain model.

In addition to the probability labeling of choices in an SDL specification[2] probabilities can also be used to specify requirements on unreliable media, like for example cell loss rates, an important QoS parameter in ATM [103]. However, communication channels in SDL are reliable and do not lose messages. It will therefore be necessary to consider probabilistic reliability requirements as a refinement of SDL channel specifications.

[2]Note that the SDL-92 standard introduced a notion of *nondeterministic choice* into SDL.

### Optimized Parallel Protocol Implementation

**Improved Message Flow Graph Analysis.** We made significant facilitating assumptions concerning the assumed message flows between processes in our model. These were in particular the assumption that one sending of a signal corresponds to exactly one receiving of that signal by the partner process. The method will gain a lot in flexibility if more sophisticated message flows can be treated.

**Lateral Communication.** In our method we have so far assumed that the processing of a packet is a non-interrupted sequence of operations from the point where the packet enters the protocol stack, to where it exits. We have not treated effects of lateral communication, namely when processes exchange control data like flow control information in addition to the protocol data we considered. Each such lateral communication would entail in our model an exit point from the protocol stack, and many exit points reduce the possible efficiency gain of our method considerably.

**Tool Support.** First attempts have been made to implement parts of the described method as a toolset. The software described in [151] uses a Yacc/Lex based SDL parser in order to derive TDGs from SDL specifications. The construction of an MLDG is a straightforward application of the algorithm described here. The labeling of the common path graph has to be contributed manually, but then the generation of a CPG and a RDG is straightforward, and first steps towards an implementation of the respective algorithms have been taken at EPF Lausanne [122]. Finally, the derivation of an optimal schedule based on the RDG, the resource constraints, and the hardware architecture can be automated. The optimal schedule which we developed in [107] for the IP/TCP/FTP protocol stack example was generated automatically.

# Part VI

# Bibliography

# Bibliography


[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *[47]*, pages 1–27, 1992.

[2] M. Abbott and L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.

[3] M. Abrams, N. Doraswamy, and A. Mathur. Chitra: Visual ananlysis of parallel and distributed programs in the time, event, and frequency domains. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):672–685, 1992.

[4] S. Aggrawal and K. Sabnani, editors. *Protocol Specification, Testing and Verification, VIII.* Proceedings of the IFIP WG 6.1 Eighth International Symposium on Protocol Specification, Testing and Verification. North Holland, 1989.

[5] B. Algayres. Personal communication, 1993.

[6] B. Algayres, Y. Lejeune, F. Hugonnet, and F. Hantz. The AVALON Project: A VALidatioON Environment for SDL/MSC Descriptions. Unpublished Manuscript. Verilog, Toulouse, France, February 1993.

[7] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985. North Holland.

[8] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[9] B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages*, 11(1):147–167, 1989.

[10] R. Alur, C. Courcoubetis, and D. Dill. Model checking for probabilistic real-time systems. In M. Rodriguez Artalejo J. Leach Albert, B. Monien, editor, *International Colloquium on Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[11] R. Alur, C. Courcoubetis, and D. L. Dill. Model checking for real-time systems. In *Fifth Annual Symposium on Logic in Computer Science*, pages 414–425, 1990.

[12] R. Alur and D. Dill. Automata for modeling real-time systems. In M. S. Paterson, editor, *Automata, Languages and Programming, Lecture Notes in Computer Science 443*, volume 443 of *LNCS*, pages 323–335. Springer Verlag, 1990.

[13] R. Alur and D. Dill. The theory of timed automata. In *[47]*, pages 45–73, 1992.

[14] R. Alur and T. A. Henzinger. Logics and models of real-time: A survey. In *[47]*, pages 45–73, 1992.

[15] R. Alur and T. A. Henzinger. Real-time system = discrete system + clock variables. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development*, pages 1–30, 1994. To appear.

[16] J. C. M. Baeten and W. P. Wijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[17] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, feb 1993.

[18] F. Bause and P. Buchholz. Protocol analysis using a timed version of SDL. In *[129]*, pages 269–285, 1990.

[19] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification*. Prentice Hall International, 1991.

[20] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. Research Report 581, IRISA, Rennes, France, 1991.

[21] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions on Automatic Control*, 35(5):535–546, May 1990.

[22] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[23] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[24] G. v. Bochmann and D. K. Probst, editors. *CAV'92: Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[25] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, April 1983.

[26] T. Braun and M. Zitterbart. Parallel transport system design. In A. Danthine and O. Spaniol, editors, *Proceedings of the 4th IFIP conference on high performance networking*, 1992.

[27] J Bredereke, R. Gotzhein, and F. H. Vogt. Design of a formal Estelle semantics for verification. In *[50]*, pages 153–168, 1993.

[28] M. Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing*, 3:21–57, 1991.

[29] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Los Alamitos, CA, 1990. IEEE Computer Society Press.

[30] CCITT. Recommendation Q.65: Stage 2 of the method for the characterization of services supported by ISDN. CCITT, Geneva, 1988.

[31] CCITT. Recommendation Q.699: Interworking between the digital subscriber system layer 3 protocol and the signaling system no. 7, ISDN user part. CCITT, Geneva, 1988.

[32] CCITT. Recommendation Z.100: CCITT Specification and Description Language (SDL). CCITT, Geneva, 1992.

[33] CCITT. Recommendation Z.120: Message Sequence Chart (MSC). CCITT, Geneva, 1992.

[34] K.-T. Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th Design Automation Conference DAC-93*, pages 86–91, 1993.

[35] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[36] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM SIGCOMM '90 conference*, Computer Communication Review, pages 200–208, 1990.

[37] E. M. Clarke and R. P. Kurshan, editors. *Computer Aided Verification: Proceedings of CAV'90*, volume 531 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[38] W. R. Cleaveland, editor. *CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[39] A. A. R. Cockburn. A formalization of temporal message-flow diagrams. In *[85]*, 1991.

[40] A. A. R. Cockburn and W. Citrin. An executable specification language for history-sensitive systems. Technical Report IBM RZ 2162, IBM Rüschlikon Research Laboratory, Zürich, 1991.

[41] A. A. R. Cockburn, W. Citrin, R. F. Hauser, and J. Känel. An environment for interactive design of communication architectures. In *[109]*, 1990.

[42] D. Cohen and N. Dorn. An experiment in analysing switch recovery procedures. In *[50]*, pages 23–34, 1993.

[43] C. Courcoubetis, editor. *Computer Aided Verification: Proceedings of CAV'93*, volume 697 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[44] J.-P. Courtiat. ESTELLE*: a powerful dialect of ESTELLE for OSI protocol description. In *[4]*, 1988.

[45] J.-P. Courtiat. Estelle and Petri nets: introducing a rendezvous mechnism in Estelle: Estelle*. In *[49]*, pages 175–203. 1989.

[46] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica. Is layering harmful? *IEEE Network Magazine*, pages 20–24, january 1992.

[47] J. W. de Bakker, C. Huizing, W.P. de Roever, and G.Rozenberg, editors. *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[48] J. De Man. Towards a formal semantics of message sequence charts. In *[54]*, pages 157–166. 1993.

[49] M. Diaz, J.-P. Ansart, J.-P. Courtiat, P. Azéma, and V. Chari, editors. *The Formal Description Technique Estelle*. North-Holland, 1989.

[50] M. Diaz and R. Groz, editors. *Formal Description Techniques, V.* IFIP Transactions C, Proceedings of the Fifth International Conference on Formal Description Techniques. North-Holland, 1993.

[51] A.J.M. Donaldson. Specification of quality of service measurement points in JVTOS. Master's thesis, University of Stirling, Scotland, U.K., September 1993.

[52] E. A. Emerson. Temporal and modal logic. In J. v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 16. Elsevier Science Publishers B. V., 1990.

[53] O. Færgemand and A. Olsen. FORTE-92 tutorial on new features in SDL-92. In M. Diaz and R. Groz, editors, *Fifth International Conference on Formal Description Techniques, Participant's Proceedings: Tutorials*, 1992.

[54] O. Færgemand and A. Sarma, editors. *SDL '93: Using Objects*. North-Holland, 1993.

[55] S. R. Faulk and D. L. Parnas. On synchronisation in hard-real-time systems. *Communications of the ACM*, 31(3):274–287, March 1988.

[56] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, pages 319–349, July 1987.

[57] S. Fischer and B. Hofmann. An Estelle compiler for multiprocessor platforms. In *[141]*, pages 171–186, 1994.

[58] J.-P. Gaspoz, T. Saydam, and J.-P. Hubaux. Object-oriented specification of a bandwidth management system for ATM-based virtual private networks. Unpublished manuscript, submitted for publication, March 1994.

[59] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *[37]*, pages 332–341, 1992.

[60] R. Gotzhein. Formal definition and representation of interaction points. *Computer Networks and ISDN Systems*, 25(1):3–22, August 1992.

[61] R. Gotzhein. Temporal logic and applications - a tutorial. *Computer Networks and ISDN Systems*, 24(3):203–218, May 1992.

[62] R. Gotzhein. *Open distributed systems: on concepts, methods, and design from a logical point of view*. Vieweg advanced studies in computer science. Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig/Wiesbaden, Germany, 1993.

[63] J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by MSCs. In *[54]*, pages 253–265, 1993.

[64] P. Graubmann, E. Rudolph, and J. Grabowski. Towards a Petri Net based semantics definition for Message Sequence Charts. In *[54]*, pages 179–190. North-Holland, 1993.

[65] H. A. Hanson. *Time and Probability in Formal Design of Distributed Systems*. PhD thesis, Uppsala University, Sweden, 1991.

[66] H. Hansson and B. Jonsson. A framework for reasoning about time and reliability. In *Real Time Systems Symposium*, pages 102–111, 1989.

[67] T. A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems.* Phd thesis, Stanford University, Department of Computer Science, August 1991. Also published as Report No. STAN-CS-91-1380.

[68] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[69] C. A. R. Hoare. Programs are predicates. In C.A.R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 141–155. Prentice-Hall, 1985.

[70] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[71] B. Hofmann and W. Effelsberg. Efficient implementation of Estelle specifications. Technical report Reihe Informatik, Nr. 3/93, University of Mannheim, Mannheim, Germany, 1993.

[72] D. Hogrefe. SDL and OSI: On the use of CCITT-SDL in the context of OSI. Habilitation Thesis, University of Hamburg, 1989.

[73] D. Hogrefe and S. Leue, editors. *Formal Description Techniques, VII.* Proceedings of the Seventh International Conference on Formal Description Techniques. Chapman & Hall, 1995. To appear.

[74] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International, 1991.

[75] Inmos Ltd. *The Occam Programming Manual.* Prentice-Hall International, 1984.

[76] ISO. Information Processing Systems - Open Systems Interconnection - Basic Reference Model. International Standard 7498, International Standards Organisation, 1984.

[77] ISO. Estelle: A formal description technique based on an extended state transition model. Draft International Standard 9074, International Standards Organisation/IFIP, 1987.

[78] ISO. Information Processing Systems - Open Systems Interconnection - LOTOS : A formal description technique based on the temporal ordering of observational behavior. International Standard 8807, International Standards Organisation/IEC, 1988.

[79] ISO. Information Processing Systems - Open Systems Interconnection - Conformance Testing Methodology and Framework, Part 3: The Tree and Tabular Combined Notation. International Standard 9464, International Standards Organisation, ISO/TC97/SC21, 1991.

[80] ISO. Information Processing Systems - Open Systems Interconnection - Conformance Testing Methodology and Framework, Part 1: General Concepts. International Standard 9464, International Standards Organisation, ISO/TC97/SC21, 1991.

[81] ISO. Revised Text of CD 10731, Information Processing Systems - Open Systems Interconnection - Service Conventions. ISO/IEC JTC 1/SC21 N 6341, International Standards Organisation/IEC, January 1991.

[82] ISO/IEC JTC 1/SC21 WG1. Quality of service framework, working draft # 3. ISO/IEC, Nov. 1993.

[83] ITU-T. Recommendation Z.120, Annex B: Algebraic Semantics of Message Sequence Charts. ITU - Telecommunication Standardization Sector, Geneva, Switzerland, 1995. To appear.

[84] I. Jacobson. *Object Oriented Software Engineering*. 1992.

[85] B. Jonsson, J. Parrow, and B. Pehrson, editors. *Protocol Specification, Testing and Verification, XI*. Proceedings of the IFIP WG 6.1 Eleventh International Symposium on Protocol Specification, Testing and Verification. North Holland, 1992.

[86] G. Karjoth. Generating transition graphs from LOTOS specifications. In *in [50]*, pages 281–294, 1993.

[87] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. PhD thesis, Technical University of Eindhoven, 1989.

[88] A. S. Krishnakumar. Reachability and recurrence in extended finite state machines: Modular vector addition systems. In *[43]*, pages 111–122, 1993.

[89] A. S. Krishnakumar and K. Sabnani. VLSI implementation of communication protocols - a survey. *IEEE Journal on Selected Areas in Communications*, 7(7):1082–1090, September 1989.

[90] J. Kurose. Open issues and challenges in providing quality of service gurantees in high-speed networks. *ACM Computer Communication Review*, pages 6–15, 1993.

[91] R. P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In *[43]*, pages 166–179, 1993.

[92] P. B. Ladkin. Testing properties of reactive systems. Unpublished manuscript, 1994.

[93] P. B. Ladkin and S. Leue. What do Message Sequence Charts Mean? In *[141]*, pages 301–316, 1994.

[94] P. B. Ladkin and S. Leue. Four issues concerning the semantics of Message Flow Graphs. In *[73]*. 1995. To appear.

[95] P. B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 37(9), January 1995.

[96] P.B. Ladkin and B.B. Simons. Compile-time analysis of communicating processes. Technical Report RJ 8488, IBM Almaden Research Center, Nov 1991.

[97] P.B. Ladkin and B.B. Simons. Compile-time analysis of communicating processes. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, pages 248–259. ACM Press, 1992.

[98] P.B. Ladkin and B.B. Simons. Static analysis of concurrent communicating loops. Technical Report RJ 8625, IBM Almaden Research Center, Feb 1992.

[99] P.B. Ladkin and B.B. Simons. Static deadlock analysis for CSP-type communications. In D. Fussell, editor, *Responsive Computer Systems: Toward Integration of Fault-Tolerance and Real Time*. Kluwer, 1994. To appear.

[100] P.B. Ladkin and B.B. Simons. *Static Analysis of Interprocess Communication*. Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.

[101] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[102] K. G. Larsen and A. Skou, editors. *Computer Aided Verification: Proceedings of CAV'91*, volume 575 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[103] J.-Y. Le Boudec. The asynchronous transfer mode: a tutorial. *Computer Network and ISDN Systems*, 24:279–309, 1992.

[104] S. Leue. QoS specification based on SDL/MSC and temporal logic. In G. v. Bochmann, J. de Meer, and A. Vogel, editors, *Proceedings of Workshop on Distributed Multimedia Applications and Quality of Service Verification*, Montreal, Quebec, Canada, May 1994.

[105] S. Leue and Ph. Oechslin. Formalizations and algorithms for optimized parallel protocol implementation. In *Proceedings of the 1994 International Conference on Network Protocols*, pages 178–185. IEEE Computer Society Press, 1994.

[106] S. Leue and Ph. Oechslin. From SDL specifications to optimized parallel protocol implementations, extended abstract. In M. Ito and G. Neufeld, editors, *Proceedings of the 4th International IFIP Workshop on Protocols for High Speed Networks.* Chapman & Hall, 1994. To appear.

[107] S. Leue and Ph. Oechslin. Optimization techniques for parallel protocol implementation. In *Proceedings of the Fourth IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 387–393, Lisbon, Sep. 1993.

[108] M. T. Liu. Protocol engineering. In M. C. Yovitis, editor, *Advances in Computers*, volume 29, pages 79–195. Academic Press, Inc., 1989.

[109] L. Logrippo, R. L. Probert, and H. Ural, editors. *Protocol Specification, Testing and Verification, X.* Proceedings of the IFIP WG 6.1 Tenth International Symposium on Protocol Specification, Testing and Verification. North Holland, 1991.

[110] G. Luo, A. Das, and G. v. Bochmann. Software testing based on SDL specifications with save. *IEEE Transactions on Software Engineering*, 20(1):72–87, 1994.

[111] N. Lynch and F. Vaandrager. Forward and backward simulations - part II: timing-based systems. Technical Report MIT/LCS/TM-487.b, MIT Laboratory for Computer Science, 1993.

[112] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 377–408. ACM Press, August 1990.

[113] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[114] S. Mauw and M.A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4), 1994.

[115] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri-Nets, Applications and Relationship to other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer Verlag, 1987.

[116] P. M. Melliar-Smith. Extending interval logic to real-time systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of the Conference on Temporal Logic in Specifications, 1987*, volume 398 of *Lecture Notes in Computer Science*, pages 224–242. Springer-Verlag, 1989.

[117] N. Meng-Siew. Reasoning with timing constraints in Message Sequence Charts. Master's thesis, University of Stirling, Scotland, U.K., August 1993.

[118] M. Merritt, F. Modugno, and M. R. Tuttle. Time-constrained automata. In *CONCUR 91: 2nd International Conference on Concurrency Theory, Lecture Notes in Computer Science 527*, 1991.

[119] R. Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[120] A. Mitschele-Thiel. On the integration of model-based performance optimization and program implementation. In *4th Workshop on Future Trends of Distributed Computing Systems*, 93.

[121] Ph. Oechslin. Personal Communication, 1994.

[122] Ph. Oechslin. *Implémentation Optimisée de Protocoles à Hauts Débits.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 1995. To appear, in French.

[123] S. W. O'Malley and L. L. Peterson. A highly layered architecture for high-speed networks. In M. J. Johnson, editor, *Protocols for High Speed Networks II*, pages 141–156. Elsevier Science Publishers (North-Holland), 1991.

[124] J. S. Ostroff. Real-time temporal logic decision procedures. In *IEEE Real-Time systems Symposium*, pages 92–101, 1989.

[125] J. S. Ostroff. *Temporal logic for real-time systems.* John Wiley & Sons Inc., 1989.

[126] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec 1986.

[127] K. R. Parker and G. A. Rose, editors. *Formal Description Techniques, IV.* IFIP Transactions C, Proceedings of the Third International Conference on Formal Description Techniques. North-Holland, 1992.

[128] W. Peng and S. Purushothaman. Data flow analysis of communicating finite state machines. *ACM Transactions on Programming Languages and Systems*, 21(3):399–442, 1991.

[129] J. Quemada, J. Mañas, and E. Vazquez, editors. *Formal Description Techniques, III.* Proceedings of the Third International Conference on Formal Description Techniques. North-Holland, 1991.

[130] J. H. Reif and S. A. Smolka. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1):1–31, February 1990.

[131] E. Rudolph. Personal communication, 1992.

[132] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International, 1991.

[133] J. Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-7, SRI International, December 1993.

[134] Mauw. S., M. van Wijk, and T. Winter. A formal semantics of synchronous interworkings. In *[54]*, pages 167–178, 1993.

[135] H. Saito, T. Hasegawa, and Y. Kakuda. Protocol verification system for SDL specifications based on acyclic expansion algorithm and temporal logic. In *In [127]*, pages 511–526, 1992.

[136] B. Selic. Personal Communication, 1994.

[137] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling.* John Wiley & Sons, Inc., 1994.

[138] Siemens AG. *EWSD Softwareentwicklungshandbuch (Software Development Handbook), Kapitel B, Register 6, SDL Diagramme.* Siemens AG, München (Munich), 1988.

[139] J. M. Spivey. *The Z Notation.* Prentice-Hall International, 1989.

[140] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall International, 2nd edition, 1989.

[141] R. L. Tenney, P. D. Amer, and M. Ü. Uyar, editors. *Formal Description Techniques, VI.* IFIP Transactions C, Proceedings of the Sixth International Conference on Formal Description Techniques. North-Holland, 1994.

[142] Y.H. Thia and C.M. Woodside. High-speed OSI protocol bypass algorithm with window flow control. In B. Pehrson, P.Gunningberg, and S. Pink, editors, *Protocols For High-Speed Networks III*, IFIP Transactions C, pages 53–68. Elsevier Publishers B.V. (North-Holland), 1993.

[143] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, chapter 4, pages 132–191. Elsevier Science Publishers B. V. (North-Holland), 1990.

[144] P. A. J. Tilanus. A formalisation of message sequence charts. In O. Faergemand and R. Reed, editors, *SDL '91: Evolving Methods*, pages 273–288. Elsevier Science Publishers B. V. (North-Holland), 1991.

[145] K. J. Turner, editor. *Using Formal Description Techniques*. John Wiley & Sons, 1993.

[146] F. Vaandrager and N. Lynch. Action transducers and timed automata. Technical Report MIT/LCS/TM-480, MIT Laboratory of Computer Science, November 1992. Also in [38].

[147] M. Van Sinderen, L. Ferreira Pires, and C. A. Vissers. Protocol design and implementation using formal methods. *The Computer Journal*, 35(5):478–491, 1992.

[148] H. B. Weinberg and L. D. Zuck. Timed Ethernet: Real-time formal specification of Ethernet. In W. R. Cleaveland, editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 370 – 385. Springer Verlag, 1992.

[149] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[150] C. M. Woodside and R. G. Franks. Alternative software architectures for parallel protocol execution with synchronous IPC. *IEEE/ACM Transactions On Networking*, 1(2):178–186, April 1993.

[151] P. Zumbrunn. Erzeugung von Kontroll- und Datenflussgraphen für SDL-Prozesse. Student project report no. IAM-PR-93534, Department of Computer Science, University of Berne, 1994. In German.

# Part VII

# Appendix

# Appendix A

# Definitions and Notation

**Relations.** Most of our notation is fairly standard, and is somewhat Z-like [139]. Let $f \subseteq R \times R$ denote a binary relation over a set $R$, let $x, y \in R$ and $S$ a set. We define the following *restrictions* and *operators* on a relation $f$.

$$f \rhd S \triangleq \{(a,b) | (a,b) \in f \ \wedge \ b \in S\}$$

$$S \lhd f \triangleq \{(a,b) | (a,b) \in f \ \wedge \ a \in S\}$$

$$(*, y)_f \triangleq f \rhd \{y\}$$

$$(x, *)_f \triangleq \{x\} \lhd f$$

$$\mathit{domain}(f) \triangleq \{a \mid (\exists b \in R)((a,b) \in f)\}$$

$$\mathit{range}(f) \triangleq \{b \mid (\exists a \in R)((a,b) \in f)\}$$

$$\mathit{field}(f) \triangleq \mathit{domain}(f) \cup \mathit{range}(f)$$

In $(*, y)_f$ and $(x, *)_f$, we sometimes omit the reference to the relation $f$ if this is clear from the context. We also extend $\lhd$ and $\rhd$ to $n$-ary relations by restricting to the first, respectively last, elements in an $n$-tuple in $f$ in the obvious way. $(V, E, \mathit{type}, \mathit{labels})$ is a *digraph with node labels* iff $E \subseteq V \times V$, $\mathit{type} : V \rightarrow \mathit{labels}$, and $\mathit{labels} = \mathit{range(type)}$. $(V, E, \mathit{type}, \mathit{labels})$ is a *digraph with edge labels* iff $E \subseteq V \times V$, $\mathit{type} : E \rightarrow \mathit{labels}$, and $\mathit{labels} = \mathit{range(type)}$. A relation $f$ is *functional* if and only if each element in its domain is related to a unique element in its range. For a functional relation $f$ and an $x \in R$ we sometimes write $f(x)$ to denote $\mathit{range}(\{x\} \lhd f)$. A relation $f$ is *injective* if and only if it's functional, and furthermore an element in its range is related to at most one element in its domain. A relation $f$ is *bijective* if and only if it's functional, and furthermore every element in its range is related to exactly one element in its domain. We use $f^+$ to denote the transitive closure of a relation $f$, and $f^*$ to denote the transitive reflexive closure of $f$.

**Digraphs and Trees.** Let $V$ denote a set and let $E \subseteq V \times V$, then we call $T = (V, E)$ a *digraph*. We call $T$ a *tree* if and only if the following additional conditions hold:

- $(\exists v \in V)((E \triangleright \{v\} = \emptyset)) \wedge (\forall w \in V, w \neq v)(E \triangleright \{w\} \neq \emptyset))$ (we call $v$ the *root*),
- $(\forall v, w \in V)((E \triangleright \{v\} = \emptyset) \rightarrow (v, w) \in E^+)$ (all nodes are reachable from the root),
- $E^+ \cap E^* = \emptyset$ (there are no cycles), and
- $(\forall v \in V)(\mid \{v\} \triangleleft E \mid \leq 1)$ (every node except for the root has exactly one predecessor).

Furthermore, for a tree $T = (V, E)$ we define: $\mathit{root}(V, E) \triangleq \{v \in V \mid E \triangleright \{v\} = \emptyset\}$, $\mathit{leaves}(V, E) \triangleq \{v \in V \mid \{v\} \triangleleft E = \emptyset\}$, $\mathit{branchnodes}(V, E) \triangleq \{v \in V \mid (\mid \{v\} \triangleleft E \mid) > 1\}$, and $\mathit{branchedges}(V, E) \triangleq \mathit{branchnodes}(V, E) \triangleleft E$.

**Multi-edged and Labeled Trees.**

- Let $E_1 \ldots E_n \subseteq V \times V$ for $n \geq 1$. Then we call $T = (V, E_1 \ldots E_n)$ a *multi-edged tree* iff $(V, E_1)$ is a tree.
- Let $T = (V, E_1 \ldots E_n)$ a multi-edged tree. Let $D_1 \ldots D_n$ denote sets which are pairwise disjoint from any other set in sight. Let $L_1 \ldots L_n$ denote functional relations with $L_i \subseteq (E_i \times D_i)$. Then we call $T = (V, E_1 \ldots E_n, D_1 \ldots D_n, L_1 \ldots L_n)$ a *multi-edged labeled tree*. We shall slightly abuse notation in that we extend the notations *root*$(T)$ and *leaves*$(T)$ to multi-edged labeled trees, in the obvious way.

**Operations on Trees.**

- Let $T = (V, E)$ denote a tree and let $x \in V$. We define $T' \triangleq \mathit{prune}(T, x)$ iff $V' = V - \mathit{domain}(\{x\} \triangleleft E^+)$ and $E' = E - (E \triangleright \mathit{domain}(\{x\} \triangleleft E^+)$.
- Let $T$ denote a multi-edged labeled tree and let $x \in V$. We define $T' \triangleq \mathit{mlprune}(T, x)$ iff $V' = V - \mathit{domain}(\{x\} \triangleleft E_1^+)$ and the following conditions hold for all $i$: $E_i' = E_i - (E_i \triangleright \mathit{domain}(\{x\} \triangleleft E_1^+)$ and $L_i' = L_i - (\mathit{domain}(\{x\} \triangleleft E_1^+) \triangleleft L_i)$.

**Equivalence Classes.** *Let $R$ be an equivalence relation. Then $[e]_R$ is the equivalence class of $e$ in $R$ , i.e. $[e]_R = \{e' \mid \langle e, e' \rangle \in R\} = e \triangleleft R$.*

**Sequences.** A *sequence* $t = t_0 t_1 t_2 \ldots$ over $A$ is a function $t : X \rightarrow A$:

$$t \ = \ \{i \mapsto t_i \triangleq t(i) \mid i \in X\}, \text{ where } X \in \omega + 1$$

Thus the domain of $t$ is either an initial segment of, or all of, the natural numbers[1].

[1] We use the von Neumann definition of the ordinal numbers, namely that 0 is the empty set, and $n$ is the set of all its predecessors {0, ..., n-1}. $\omega$ is the set of natural numbers. $\omega + 1 = \omega \cup \{\omega\}$.

A sequence is *finite* if its domain is a natural number and *infinite* if its domain is $\omega$. We define $A^* \triangleq \{t \mid t : n \mapsto A \;\wedge\; n < \omega\}$ and $A^\omega \triangleq \{t \mid t : \omega \mapsto A\}$. $A^*$ is the set of finite sequences over $A$, and $A^\omega$ is the set of infinite sequences over $A$. Finally, $A^\infty \triangleq A^* \cup A^\omega$.

**Length and Membership.** We define an operation and a relation of mixed type, $\mathit{length} : A^\infty \to \omega + 1$, and *membership*, $\in\; \subset\; A \times A^\infty$.

$$\mathit{length}(t_0 \ldots t_{n-1}) \triangleq n \textit{ for finite } t$$

$$\mathit{length}(t_0 \ldots t_{n-1} \ldots) \triangleq \omega \textit{ for infinite } t$$

$$a \in t \;\overset{\triangle}{\Leftrightarrow}\; (\exists i \bullet 0 \leq i \leq \mathit{length}(t) \;\wedge\; a = t_i)$$

**Concatenation.** The *concatenation* of two sequences $t$ and $q$, where $t$ is finite, is

$$t^\frown q \triangleq t \cup \{(\mathit{length}(t) + k) \mapsto q(k) \mid k < \mathit{length}(q)\}$$

If $t$ is infinite, we define $t^\frown q \triangleq t$ We often omit the symbol $^\frown$ between the two sequences, and write $tq \triangleq t^\frown q$.

**Representation of Sequences.** We can (and normally do) represent the finite sequence $\{0 \mapsto t_0, ..., (n-1) \mapsto t_{n-1}\}$ by the concatenation $t_0 t_1 ... t_{n-1}$. Extending this to infinite sequences $t = \{0 \mapsto t_0, ..., (n-1) \mapsto t_{n-1}, ......\}$, we write them as $t_0 t_1 ... t_{n-1} ...$

# Appendix B

# Translation of Poem on Page iv

*Misunderstanding of two Surrealists*

> "it is raining"
> she said
> "men in black coats
> are walking by"
> she said
>
> Magritte, however,
> could not really hear her
> any more
> (she said it only years
> after his death)
>
> So he did not hear any more
> her last two words
> and only understood
> "it is raining men in black coats"
> Which he painted

*Erich Fried*

German original version (see page iv) taken from: Erich Fried. *Es ist was es ist. Liebesgedichte, Angstgedichte, Zorngedichte.* Verlag Klaus Wagenbach, Berlin, 1983. Translation into English by Peter Ladkin and Stefan Leue.

## Curriculum Vitae

Stefan Leue

| | |
|---|---|
| **April 21, 1962** | Born in Hamburg, Germany. |
| **August 1968 - July 1972** | Primary School "Schule an den Teichwiesen" at Hamburg, Germany. |
| **August 1972 - August 1981** | Secondary School "Gymnasium Buckhorn" at Hamburg, Germany. |
| **October 1981 - October 1982** | Military Service. |
| **October 1982 - October 1990** | Studies in Computer Science (Informatik) and Economics at the University of Hamburg, Germany. Degree: Masters (Diplom-Informatiker). |
| **Dez. 1990 - Oct. 1991** | Consultant, self-employed. |
| **since November 1991** | Research Assistant (Wissenschaftlicher Assistent) and Doctoral Student at the Department of Computer Science and Applied Mathematics of the University of Berne, Switzerland. |