Constructive formal methods and protocol standardization

Constructive formal methods and protocol standardization

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 2 oktober 2006 om 16.00 uur

door

Arjan Johan Mooij

geboren te Rotterdam

Dit proefschrift is goedgekeurd door de promotor:

prof.dr.ir. J.F. Groote

Copromotor:
dr. J.M.T. Romijn

Constructive formal methods and protocol standardization

Arjan J. Mooij

Promotor: prof.dr.ir. J.F. Groote (Technische Universiteit Eindhoven)
Copromotor: dr. J.M.T. Romijn (Technische Universiteit Eindhoven)

Kerncommissie:
prof.dr. W.H. Hesselink (Rijksuniversiteit Groningen)
prof.dr.ir. J.-P. Katoen (RWTH Aachen University)
dr. S. Mauw (Technische Universiteit Eindhoven)

The work in this thesis is supported by the NWO as a part of project "Improving the Quality of Protocol Standards" (project number 016.023.015).

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2006-14.

Printing: Printservice Technische Universiteit Eindhoven
Cover design: Oranje Vormgevers

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Mooij, Arjan J.

Constructive formal methods and protocol standardization / Arjan Johan Mooij. – Eindhoven : Technische Universiteit Eindhoven, 2006.
Proefschrift. – ISBN 90-386-0794-6. – ISBN 978-90-386-0794-8
NUR 993
Subject headings: programming ; formal methods / software description ; formal proofs / distributed programming / computer networks ; protocols
CR Subject Classification (1998): F.3.1, D.1.3, F.3.2, D.2.4, C.2.2

Preface

Upon completing this thesis, it is tempting to look forward to an exciting future career. At the same time, however, I do realize that there have been a lot of people that enabled me to develop myself this far. As it is easy to get used to such valuable people, I will use this preface to express my gratitude to them.

I like to thank my copromotor Judi Romijn for offering me a position on her research project, for the supervision, and for the discussions on different styles of computer science research. I would like to thank Nicu Goga for the collaboration, and in particular for his interest and concern during the pregnancy leave of Judi. Wieger Wesselink has done a great job in improving my understanding of several techniques by forcing me to explain them properly. Furthermore I like to thank him for his work on the tool implementation that is discussed in Chapter 9 of this thesis. I would also like to thank my promotor Jan Friso Groote for accepting me in his research group, and for allowing me to explore a wide range of topics.

Wim Hesselink, Joost-Pieter Katoen and Sjouke Mauw are thanked for taking part in the committee to judge this thesis. In particular I thank Wim Hesselink and Sjouke Mauw for enabling me to smoothen some irregularities.

Apart from the research leading to this thesis, there was time to work on other subjects. During the last year I have enjoyed the research collaboration with Brijesh Dongol from the University of Queensland. I have also appreciated my local teaching duties; I like to thank these colleagues for their trust and support.

The weekly sessions of the Eindhoven Tuesday Afternoon Club are memorable. In a pleasant and challenging atmosphere, these afternoons have clearly contributed to my skills in (at least) mathematics and problem solving. In particular I am greatly indebted to Wim Feijen, from whom I have learned many things about mathematics and computer science.

Apart from having contact with scientific colleagues, it is indispensable to have some external distractions. In my case, a lot of people have contributed to this, namely by together making music. I have especially valued the atmosphere of mutual inspiration. Finally I would like to thank my parents for providing a safe environment from which I could study and develop myself.

Contents

Chapter 0

Introduction to the thesis

An important feature of many electronic devices is their ability to communicate with other devices. A typical example from the field of consumer electronics is connecting a personal computer to a digital camera or to a network of computers. In particular communication between devices from different manufacturers must be possible. In this way many distributed networks of interconnected devices arise, in which parallelism is inevitable.

To enable these networks to operate successfully, the manufacturers need to agree on the communication protocols to be used. Such protocols are usually described in international protocol standards, which are maintained by organizations like the ANSI, the IEEE, and the ISO. The development of a standard is a long-term effort of a great number of parties, each with a specific interest in the standard.

Developing and reasoning about parallel systems is generally considered to be complicated. Moreover, the quality of many current protocol standards is poor, and hence the proper functioning of these protocols cannot be guaranteed. If after publication of the standard all errors would be detected by each manufacturer, for example by testing their own products, there is still the danger that different parties try to fix the protocol in incompatible ways.

In the last few decades, much computer science research has focussed on formal methods, which are mathematical techniques that can be used to establish correct algorithms and to describe them properly. A wide range of techniques has been developed, and currently the most popular ones are specification, simulation and verification. Specification is used to define an algorithm, simulation is used to explore some behavior of a given algorithm, and verification is used to completely check the correctness of a given algorithm. The primary goal of these particular techniques seems to be finding errors in a given algorithm, while constructive formal techniques that guide the development of correct algorithms receive far less

attention. In this thesis we explore the use of constructive methods.

Formal methods have mainly been applied by academia and to algorithms that have already been developed. In this work we reach out to the industry in order to demonstrate how the use of formal methods can improve the quality of protocol standards that are still under development. At the same time we extract and address theoretical topics that lead to a better applicability of formal methods.

To this end we have worked on several protocol standards and applied various kinds of formal methods during the standardization process. In this thesis we emphasize our theoretical contributions, while we only briefly describe our practical results on the protocol standards. In order to motivate and to explain the issues we have worked on, we also provide short descriptions of the protocol standards.

This thesis consists of two rather independent parts according to the two main standards we have been involved in. Each part contains its own introduction and conclusions. Part I is related to the ISO/IEEE 1073.2 standard for medical device communication, and it addresses the realizability of MSC scenario specifications. Part II is related to the IEEE 1394.1 standard for FireWire bridges, and it addresses distributed spanning tree algorithms for dynamic networks.

Part I

ISO/IEEE 1073.2: Medical Device Communication

Chapter 1

Introduction

In this chapter we introduce Part I of this thesis. This part is based on our work on the protocol standard ISO/IEEE 1073.2, which we describe in Section 1.1. Afterwards, in Section 1.2 we present some basic concepts that will be used in the chapters that follow.

1.1 Introduction to the protocol standard

The ISO/IEEE 1073 Standard for Medical Device Communications is a family of standards, where the substandards in the ISO/IEEE 1073.2 series address the general communication services. In this section we introduce the relevant substandards and describe the issues we have worked on. More details can be found in [MGWB03, MG05].

1.1.1 ISO/IEEE 1073.2.2: base standard

The ISO/IEEE 1073.2.2 standard is the base protocol standard, which enables the communication of patient-related data for the treatment of patients or for the documentation of medical procedures. In the context of a network of medical devices and managerial computer systems, a manager-agent communication system is defined, where the agent usually incorporates a medical device that provides data, and where the manager receives data. Although a manager can communicate with several agents, the protocols are defined for a single manager-agent pair. This standard is based on European pre-standard ENV 13735 [CEN00].

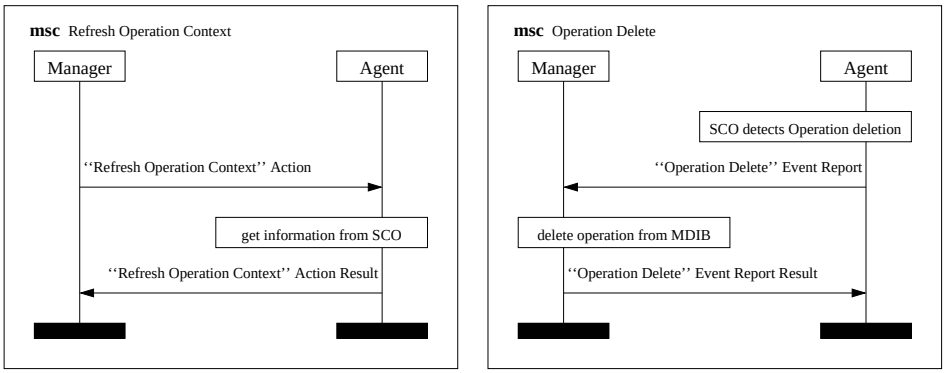


Figure 1.1 Some basic MSCs for remote control

1.1.2 ISO/IEEE 1073.2.3: remote control extension

The base standard supports so-called remote configuration, but it was considered to be too restrictive for performing tasks on a medical device through a communication system. These intended tasks include obtaining medical information, and configuring, programming and operating the device. Therefore the base standard was extended with remote control functionality.

The protocol description in the draft remote control standard ISO/IEEE 1073.2.3 consisted of some scenarios describing typical intended behaviors, like the examples in Figure 1.1, and some accompanying textual descriptions. The scenarios are expressed in a kind of message sequence chart (MSC, [ITU00, Ren99]). MSC is a visual formalism that is widely used in the telecommunication sector, although in academia it is often considered as inadequate. The standard does not contain any description of requirements, i.e. desired properties, on the protocol.

1.1.3 Our contributions

Although communication systems in the medical domain must be reliable under all circumstances, a formal analysis is no common part of the development of this standard. Initially we have analyzed the state transition tables in the base standard, and we have proposed modifications for the problems found, see [MGWB03]. Like the remote control standard, no requirements on the protocols in the base standard were described. Therefore we have primarily considered general properties like absence of deadlock.

In contrast to the base standard, the draft remote control standard did not even contain a formal definition of the protocol, e.g. in the form of state transition

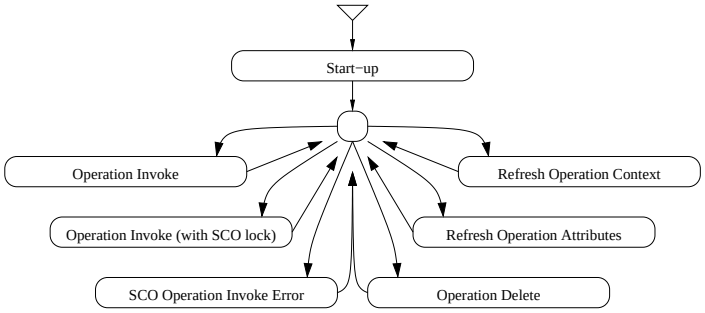


Figure 1.2 Derived high-level MSC for remote control

tables. So before analyzing the remote control standard, we needed to synthesize such a formal definition. In the literature, many algorithms have been proposed for synthesizing a (formal) protocol implementation from a message sequence chart specification. To apply such an algorithm, we have used the textual descriptions in the draft standard to create some additional variations of the basic MSCs, and to derive a high-level MSC that describes the overal structure of the protocol, see Figure 1.2.

Message sequence charts describe behavior from a full-system's perspective, while distributed implementations describe behavior in terms of the individual process instances. Therefore protocol synthesis algorithms mainly project the MSCs on the individual instances. However, this technique does not work properly for MSCs that contain a so-called non-local choice. In case of non-local choice, usually unspecified behavior and even deadlocks are introduced, thus resulting in protocols with undesired behavior. This leads to the realizability problem, viz. whether there exists an implementation with the same behavior as a given MSC specification.

The (high-level) MSC that we have derived for remote control contains a non-local choice. For example, in the first basic MSC from Figure 1.1 the Manager instance has initiative, while in the second basic MSC the Agent instance has initiative. As the two process instances are independent, and both have initiative in a different basic MSC, synthesized implementations admit implied behaviors like the one in Figure 1.3.

In the literature, the problem of implementing MSCs that contain non-local choice is almost completely ignored. Nevertheless, it is a very practical problem as non-local choice is often inevitable. In Chapter 4 we investigate some approaches to this problem, and we propose some ways to slightly modify MSC specifications that contain non-local choice in such a way that they become realizable. These modifications introduce a little additional behavior, but in a controlled way. We also discuss the application of these techniques to the ISO/IEEE 1073.2 standard.

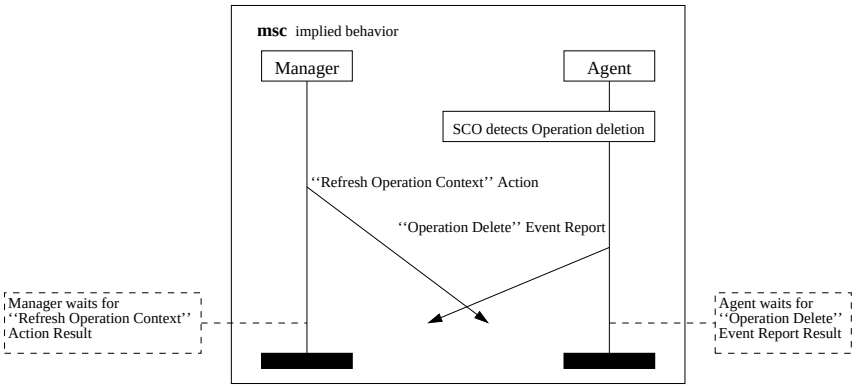


Figure 1.3 Implied behavior in synthesized implementations

Non-local choice is the best-studied realizability criterion. Absence of non-local choice guarantees that there is at most one process instance that determines the choice, but this is not enough to guarantee realizability. Namely, the other process instances must somehow resolve the choice, i.e. the decision must be propagated to these instances. In Chapter 3 we propose and motivate a complete characterization of realizability criteria.

To study these aspects of the realizability problem in a convenient way, we need an appropriate formal model in which both specifications and implementations can be discussed. In Chapter 2 we introduce our framework that is based on the partial order model from [Pra86, KL98]. We also give a denotational semantics of compositional MSC in terms of this model.

The framework from Chapter 2 is used for the technical details in Chapters 3 and 4. Nevertheless the majority of the latter two chapters is comprehensible with just a basic understanding of MSC. In Chapter 5 we conclude Part I of this thesis.

1.2 Preliminaries

In this section we introduce the subset of the graphical Message Sequence Chart (MSC) language that we use. This language is used to describe behaviors of a collection of autonomous process instances that can communicate via asynchronous message communication. Section 2.3 contains a formal semantics of MSC in a textual representation.

1.2.1 Basic MSC

The best-known kind of MSC is basic MSC, see the two examples in Figure 1.1. A basic MSC contains for each process instance a vertical axis, on top of which the name of the instance is printed. Each axis contains a series of events that are only allowed to occur in the order in which they are depicted, from top to bottom. There are three kinds of events:

- *internal action*, which is depicted by a rectangle that contains a description of the action;
- *send* event, which is depicted by an outgoing arrow with an attached name that describes the message to be sent;
- *receipt* event, which is depicted by an incoming arrow with an attached name that describes the message to be received.

Each receipt event can only occur after the corresponding send event. The arrows of corresponding send and receipt events are usually connected, but this is not the case for the MSC extension called compositional MSC.

Sometimes a part of an instance axis is used as a co-region, which denotes that no order between the events in that region is imposed. Many other extensions have been proposed, but we will stick to this manageable subset of the graphical language.

1.2.2 High-level MSC

High-level MSC is used to combine several basic MSCs, see the example in Figure 1.2. It consists of a directed graph in which each node is labeled by a (possibly empty) basic MSC. One of the nodes is designated as the initial node, which is depicted by a triangle. The directions of the edges define the order in which the basic MSCs from the nodes may be executed. Each possible behavior corresponds to the sequential composition of the basic MSCs encountered at a single path through the graph. Hence the nodes with several outgoing edges denote choice.

The sequential composition of two basic MSCs is defined instance-wise, i.e. by connecting the axes that belong to the same process instance. In particular in compositional MSC it is possible that the send and receive event of a single message are located in different basic MSCs. In textual representations, keyword **seq** is used to denote sequential composition, and keyword **alt** is used to denote choice.

Chapter 2

Partial-order framework

In this chapter that is primarily based on [MRW06], we develop a formal framework for compositional MSC [GMP03, MM01] to support our study of realizability in Chapter 3. Compositional MSC is an MSC extension that allows the send event and the receipt event of a single message to be located in different basic MSCs.

Several kinds of semantics have been proposed for MSC specifications (e.g. [MR94, KL98, Hey00, UKM03]), while implementations are typically expressed in terms of finite state machines. To compare specifications and implementations, two different formalisms must then be related, usually via execution traces (in fact a third formalism), see e.g. [AEY03]. We prefer to use one single formalism for both implementations and specifications, and we want to stay close to the MSC specification formalism. Therefore we use a partial order semantics [Pra86] for our study, and sketch the relation with operational formalisms. In addition to the partial order model in [Pra86, KL98], we introduce a way to model deadlocks and a more sophisticated way to deal with communication.

Overview In Section 2.1 we introduce our partial order model, which we extend with communication in Section 2.2. These two sections are rather independent from MSC, but they are the basis of the semantics of compositional MSC in Section 2.3. In Section 2.4 we define the typical way of synthesizing an implementation, and we conclude this chapter in Section 2.5.

2.1 Extended partial order model

In this section we define a partial order model and extend it with deadlocks, to make it suitable for studying realizability criteria.

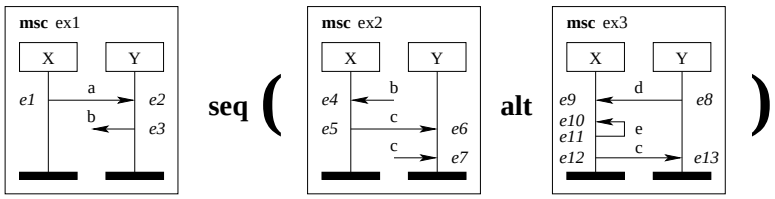


Figure 2.1 Running example

2.1.1 Running example

We illustrate our techniques using a running example. Figure 2.1 contains a (high-level) MSC consisting of the three basic MSCs ex1, ex2 and ex3. It specifies the behavior of process instances X and Y, such that first the behavior of ex1 occurs, followed by either the behavior of ex2 or the behavior of ex3. For reference purposes we have included arbitrary event names (e_1 to e_{13}) in the basic MSCs.

2.1.2 LATERs: LAbeled Transitive Event Relations

As a semantic model of behavior, we introduce the notion of a later, which is an acronym for **la**belled **t**ransitive **e**vent **r**elation. A *later* $(E, <, l)$ is a triple that consists of an event set E, a transitive causality relation $<: < \subseteq E \times E$ and a labeling function $l : E \rightarrow L$ for a given set of labels L.

The behavior of a later is such that any event $e : e \in E$ models a single action with label $l.e$; the event can occur at most once and it may only occur after all events $f : f < e$ have already occurred. As $<$ is not required to be asymmetric, it is not guaranteed that all events can occur. The notion of an event is used to handle multiple occurrences of an action with the same label. Compared to the partial orders in [Pra86], a later is an lposet in which the partial order constraint has been weakened.

In our running example, let laters p_1, p_2 and p_3 correspond to the basic MSCs ex1, ex2 and ex3 in which only the causalities per instance (on each vertical axis) are considered, i.e. without communication. So, $p_1 = (\{e_1, e_2, e_3\}, \{e_2 < e_3\}, l_1)$ and, as we will see later on, $l_1 = \{e_1 \mapsto !(a, X, Y), e_2 \mapsto ?(a, X, Y), e_3 \mapsto !(b, Y, X)\}$. Labeling function l_1 maps event e_1 to the label $!(a, X, Y)$, which we use to denote an action of instance X that sends a message a to instance Y. Similarly, event e_2 is mapped to the label $?(a, X, Y)$, which denotes an action of instance Y that receives a message a from instance X. The structure of p_1 can be visualized as $\boxed{e_1 \qquad e_2 \longrightarrow e_3}$ such that relation $<$ corresponds to the transitive closure of relation $\rightarrow$.

In an interleaved execution model where the events are labeled with atomic actions, the maximal behaviors of a partially ordered later are its linearizations. The linearizations of a later $(E, <, l)$ are the execution traces $e_1 \cdot \ldots \cdot e_n$ such that $\{e_1, \ldots, e_n\} = E$, and for each two indices i and j both $e_i = e_j \;\Rightarrow\; i = j$ and $e_i < e_j \;\Rightarrow\; i < j$. The three linearizations of later p_1 are $e_1 \cdot e_2 \cdot e_3$, $e_2 \cdot e_1 \cdot e_3$, and $e_2 \cdot e_3 \cdot e_1$. We prefer to reason about the (more abstract) laters instead of linearizations, because they are better related to MSC and they avoid decomposing each later into several over-specific total orders. Another advantage is that they can be used to model true concurrency, where events can (partially) overlap.

The most elementary laters are the empty later, with no events, and the singleton laters, with only one event with a label $k : \; k \in L$. We introduce the following abbreviations for them:

$$[\epsilon] = (\emptyset, \emptyset, \emptyset)$$
$$[k] = (\{e\}, \emptyset, \{e \mapsto k\}) \quad \text{for } k : \; k \in L \text{ and arbitrary } e$$

2.1.3 Isomorphism

The event set of a later is abstract in the sense that a consistent renaming of the events yields a later with the same behavior. This is formalized in the following notion of isomorphism. Laters $(E, <, l)$ and $(E', <', l')$ are *isomorphic*, denoted $(E, <, l) \simeq (E', <', l')$, if there is a bijection $\sim$: $\sim \subseteq E \times E'$ such that both

- $(\forall e, e' : e \sim e' : \; l.e = l'.e')$
- $(\forall e, f, e', f' : e \sim e' \;\wedge\; f \sim f' : \; (e < f \;\equiv\; e' <' f'))$

Relation $\simeq$ is an equivalence relation. In what follows we will hardly mention $\simeq$ explicitly, and implicitly assume that where necessary $\simeq$ has been exploited to obtain suitable laters, e.g. ones that are event disjoint. This conforms to the pomset style used in [KL98].

2.1.4 Elementary later operators

We often need to relate events to the process instance in which they occur. We assume a fixed set of *instance names* I, and a function $\phi : L \to I$ that maps labels to the instance in which the actions with that label occur. For a later $(E, <, l)$, [HJ00] uses the slightly different function $\phi' : E \to I$, which can be obtained from our later-independent ϕ as follows: $\phi'.e = \phi.(l.e)$.

To construct larger laters from the elementary laters, we use the following elementary operators on event disjoint laters (i.e. $E_p \cap E_q = \emptyset$):

$$(E_p, <_p, l_p) \parallel (E_q, <_q, l_q) = (E_p \cup E_q,\ <_p \cup <_q,\ l_p \cup l_q)$$

$$(E_p, <_p, l_p) \circ_S (E_q, <_q, l_q) = (E_p \cup E_q,\ <_p \cup <_{\circ_S} \cup <_q,\ l_p \cup l_q)$$
where $<_{\circ_S} = E_p \times E_q$

$$(E_p, <_p, l_p) \circ_W (E_q, <_q, l_q) = (E_p \cup E_q,\ (<_p \cup <_{\circ_W} \cup <_q)^+,\ l_p \cup l_q)$$
where $<_{\circ_W} = \{(e, f) \mid e, f :\ e \in E_p \wedge f \in E_q \wedge \phi.(l_p.e) = \phi.(l_q.f)\}$

Operator $\parallel$ denotes parallel composition, and operators $\circ_S$ and $\circ_W$ denote strong (or pure, or synchronous) and weak (or asynchronous) sequential composition, respectively. These operators are associative and they have unit element $[\epsilon]$. Since parallel composition is also commutative, we can use $\parallel$ as a quantifier.

In our running example, $\phi.(!(a, X, Y)) = X$ and $\phi.(?(a, X, Y)) = Y$. Let laters p_4 and p_5 be defined as $p_4 = p_1 \circ_W p_2$ and $p_5 = p_1 \circ_W p_3$. The structure of p_5 is visualized as $\boxed{e_1 \longrightarrow e_9 \longrightarrow e_{10} \longrightarrow e_{11} \longrightarrow e_{12} \qquad e_2 \longrightarrow e_3 \longrightarrow e_8 \longrightarrow e_{13}}$.

2.1.5 Deadlocks

A later $(E, <, l)$ contains a *deadlock* if there is an event $e :\ e \in E$ such that $e < e$. Conversely, a later is *deadlock-free* if the (transitive) causality relation is a strict partial order, i.e. the conjunction of the following holds:

- irreflexive: $(\forall e ::\ \neg(e < e))$
- asymmetric: $(\forall e, f ::\ \neg(e < f \wedge f < e))$
- transitive: $(\forall e, f, g ::\ e < f \wedge f < g \Rightarrow e < g)$

The definitions of deadlock and deadlock-free are consistent, since asymmetry implies irreflexivity, and transitivity plus irreflexivity implies asymmetry. All laters that can be obtained from the elementary laters using the elementary later operators are deadlock-free.

For example, consider later p_5' (to be defined in Section 2.2) with the following structure: $\boxed{e_1 \longrightarrow e_2 \longrightarrow e_3 \longrightarrow e_8 \longrightarrow e_9 \longrightarrow e_{10} \rightleftarrows e_{11} \longrightarrow e_{12} \longrightarrow e_{13}}$. In this later there is a circular dependency between events e_{10} and e_{11}. From the transitivity of relation $<$ it follows that $e_{10} < e_{10}$, hence e_{10} is a deadlock.

The interpretation of the causality relation is such that the set of events "behind any deadlock" cannot occur either. We define the set of deadlocked events Δ for a later $(E, <, l)$ as follows:

$$\Delta.(E, <, l) = \{f \mid e, f :\ e \in E \wedge f \in E \wedge e < e \wedge e < f\}$$

In our example we obtain $\Delta.p_5' = \{e_{10}, e_{11}, e_{12}, e_{13}\}$, and hence events e_1, e_2, e_3, e_8 and e_9 are the only events that can occur in later p_5'.

2.1.6 Prefix

A natural way to compare laters is to compare their possible behaviors. If all possible behaviors of a later p are contained in the possible behaviors of a later q, we call p a prefix of q. In an interleaved execution model this corresponds to trace inclusion.

To determine whether p is a prefix of q, we only need to consider the deadlock-free part of p. If p is a prefix of q, then (1) p may contain fewer events than q, (2) on this smaller event set, p may contain more causalities than q, (3) q's labeling of events is respected by p, and (4) for each event that is in both p and q, all events that precede the event in q are also in p.

Formally, later p is a *prefix* of later q, to be denoted by $p \preceq q$, if for some laters $(E_p, <_p, l_p) \simeq p$ and $(E_q, <_q, l_q) \simeq q$ the following four conditions hold:

1. $\overline{E_p} \subseteq E_q$
2. $<_q \cap (\overline{E_p} \times \overline{E_p}) \subseteq <_p$
3. $l_p \cap (\overline{E_p} \times L) = l_q \cap (\overline{E_p} \times L)$
4. $(\forall e, f :: e <_q f \wedge f \in \overline{E_p} \Rightarrow e \in \overline{E_p})$

where $\overline{E_p} = E_p \backslash \Delta.(E_p, <_p, l_p)$

In the running example several prefix relations hold, such as $p_1 \preceq p_4$ and $p_1 \preceq p_5$.

As a corollary of $p \preceq q$, we have $\overline{E_p} \subseteq \overline{E_q}$ for $\overline{E_q} = E_q \backslash \Delta.(E_q, <_q, l_q)$. Prefix order $\preceq$ is a pre-order (i.e. reflexive and transitive) with smallest element $[\epsilon]$. Some typical prefixes are $p \preceq p \| q$, $q \preceq p \| q$, $p \preceq p \circ_S q$ and $p \preceq p \circ_W q$. In comparison with [KL98], our definition is more explicit, it can deal with deadlocks, and it allows $<_q \cap (\overline{E_p} \times \overline{E_p})$ to be strictly smaller than $<_p$.

Parallel composition is monotonic in both arguments, while both kinds of sequential composition are only monotonic in their second argument. In general, sequential composition is not monotonic in its first argument. For example, let $p = [\epsilon]$, $q = (\{e\}, \{e < e\}, \{e \mapsto k\})$ and $r = [k]$. Both kinds of sequential composition yield $p \circ r = r$ and $q \circ r \preceq q$. Although $p \preceq q$, we do not have $p \circ r \preceq q \circ r$, because $r \not\preceq q$. This observation has directed our study in Section 3.1.2 towards an action-prefix alike operator instead of a full sequential composition operator.

A special kind of prefix is a *causality extension*:

$$< \subseteq <' \Rightarrow (E, <', l) \preceq (E, <, l)$$

As an example consider later p'_5, which is a causality extension of later p_5.

2.1.7 Projection

To restrict the set of events of a later, we define a projection operator π that restricts a later to the events in process instance i as follows:

$$\pi_i.(E, <, l) = (F, < \cap (F \times F), l \cap (F \times L))$$
$$\text{where } F = \{e \mid e : \ e \in E \ \wedge \ \phi.(l.e) = i\}$$

The relation with parallel composition is $p \ \preceq \ (\|i : i \in I : \ \pi_i.p)$. In general it is not guaranteed that $(\|i : i \in I : \ \pi_i.p) \ \preceq \ p$. For example, consider $p = (\{e_1, e_2\}, \{e_1 < e_2\}, \{e_1 \mapsto k_1, e_2 \mapsto k_2\})$ such that $\phi.k_1 \neq \phi.k_2$. Then $(\|i : i \in I : \pi_i.p) = (\{e_1, e_2\}, \emptyset, \{e_1 \mapsto k_1, e_2 \mapsto k_2\})$, which is not a prefix of p.

Furthermore, projection is monotonic with respect to causality extensions in the sense that:

$$< \subseteq <' \ \Rightarrow \ \ \pi_i.(E, <', l) \ \preceq \ \pi_i.(E, <, l)$$

2.1.8 Sets of laters

Usually a single later cannot describe all possible behavior of a system. Therefore we also study a set of laters (which is the notion of a process in [Pra86]), which represents the set of behaviors of the individual laters (like a choice). We lift each elementary later operator $\oplus$ and the projection operator π as follows:

$$\begin{array}{lcl} P \ \oplus \ Q & = & \{p \ \oplus \ q \mid p, q : \ p \in P \ \wedge \ q \in Q\} \\ \pi_i.P & = & \{\pi_i.p \mid p : \ p \in P\} \end{array}$$

To lift the prefix order $\preceq$, we define order $\sqsubseteq$ as follows:

$$P \sqsubseteq Q \ \equiv \ (\forall p : p \in P : \ (\exists q : q \in Q : \ p \preceq q))$$

Order $\sqsubseteq$ is a pre-order with smallest element $\emptyset$. Like before, parallel composition is monotonic in both arguments, while both kinds of sequential composition are only monotonic in their second argument. Relation $\doteq$ defined as

$$P \doteq Q \ \equiv \ P \sqsubseteq Q \ \wedge \ Q \sqsubseteq P$$

is an equivalence relation. Equivalence $P \doteq Q$ denotes that P and Q have the same sets of deadlock-free prefixes, which means that they are trace equivalent. Thus sets of laters can be interpreted as the delayed choice [BM95] between the individual laters.

2.1.9 Partial synchronization

In Section 4.3 we will use the constraint-oriented style of programming, see e.g. [Bri90]. Therefore we introduce the partial synchronization operator that is known from LOTOS [ISO89], which combines two behaviors by merging certain pairs of events. In terms of laters, it can be defined as follows for a given set of labels K:

$$
\begin{aligned}
&P \parallel_K Q = (\bigcup p, q : p \in P \ \wedge \ q \in Q : \ p \parallel_K q) \\
&p \parallel_K q = \{(E_p \cup E_q, \ <_p \cup <_d \cup <_q, \ l_p \cup l_q) \mid E_p, E_q, <_p, <_d, <_q, l_p, l_q : \\
&\qquad\qquad (E_p, <_p, l_p) \simeq p \ \wedge \ (E_q, <_q, l_q) \simeq q \ \wedge \\
&\qquad\qquad (\forall e : e \in E_p \ \wedge e \in E_q : \ l_p.e = l_q.e \ \wedge \ l_p.e \in K) \ \wedge \\
&\qquad\qquad <_d = \{(e, e) \mid e : \ (e \in E_p \ \wedge \ l_p.e \in K \ \wedge \ e \notin E_q) \ \vee \\
&\qquad\qquad\qquad\qquad (e \in E_q \ \wedge \ l_q.e \in K \ \wedge \ e \notin E_p) \qquad \} \qquad \}
\end{aligned}
$$

This definition is similar to the definition of parallel composition. The event disjointness requirement on laters only holds for the events with labels that are not in K. The events with a label from K must be synchronized, and hence we require that if such an event is present in both E_p and E_q, then it has the same label in l_p and l_q. If such an event is not present in both E_p and E_q, then it becomes a deadlock event via $<_d$.

Notice that the result of the partial synchronization of single laters is a set of laters. In many cases it could be expressed as a single later, but not in cases like the following:

- $p = (\{e_1, e_2, e_3, e_4\}, \{e_1 < e_3, e_2 < e_4\}, \{e_1 \mapsto a, e_2 \mapsto a, e_3 \mapsto b, e_4 \mapsto c\})$
- $q = (\{e_3, e_4, e_5\}, \{e_5 < e_3, e_5 < e_4\}, \{e_3 \mapsto b, e_4 \mapsto c, e_5 \mapsto a\})$

Depending on whether event e_5 is matched with e_1 or with e_2, either events e_2 and e_4 or events e_1 and e_3 cannot occur respectively. Hence the result of $p \parallel_{\{a,b,c\}} q$ cannot be expressed in terms of a single later.

Furthermore, notice that $P \parallel_\emptyset Q \ \doteq \ P \parallel Q$. Recalling that L denotes the set of all labels, we even have $P \parallel_L P \ \doteq \ P$ thanks to considering trace equivalence. Since $\parallel_K$ on sets of laters is associative and commutative, we can use $\parallel_K$ as a quantifier.

2.2 Asynchronous communication

In this section we develop an operator that introduces in a later the causalities that correspond to asynchronous message communication. This is the most commonly used communication model, although [EMR02] investigate the consequences of other models. To model distributed systems with communication via message passing, some labels are used to denote sending or receiving a message. The

most liberal causalities are obtained by matching sends and receipts in their order of occurrence. This does not require that messages with identical names are communicated in FIFO order.

2.2.1 Label-wise trichotomy

To match events properly, we need to determine the order in which events with identical labels occur. For simplicity reasons, we assume for each label that the events with that label are totally ordered; at least, in the deadlock-free part of the later. Since this deadlock-free part is strict partially ordered, we only need trichotomy (or comparability) for events with identical labels. For notational convenience, we require this property for the whole later and for all labels.

The *label-wise trichotomy* property T is defined as follows:

$$T.P \;\equiv\; (\forall p : p \in P :\; T.p)$$
$$T.(E, <, l) \;\equiv\; (\forall e, f : l.e = l.f :\; e = f \;\vee\; e < f \;\vee\; f < e)$$

As we will see in Section 2.3, this only imposes a few, acceptable restrictions on MSCs. This property is maintained under causality extensions and event restrictions, it holds for the elementary laters, and it is maintained under sequential composition; only for a parallel composition $(E_p, <_p, l_p) \parallel (E_q, <_q, l_q)$ label-disjointness is required, i.e. $(\forall e, f : e \in E_p \;\wedge\; f \in E_q :\; l_p.e \neq l_q.f)$.

2.2.2 Communication causalities

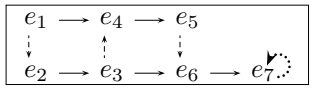
We define operator $\Gamma.p$, which introduces the communication causalities in a later p. For compositional MSC, we must also address communication between two sequentially composed laters. Therefore we introduce a parameter t of type later, which denotes the history, i.e. the entire preceding behavior of later p. The default value of parameter t is $[\epsilon]$.

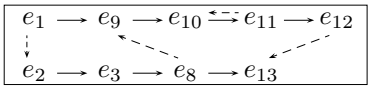
For each message m, we must ensure that each receipt event (with label $?m$) is preceded by the corresponding send event (with label $!m$). In case there are more receive events than send events, these remaining receipt events are turned into deadlocks. Thus we obtain (provided $T.t$ and $T.P$ hold):

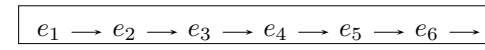
$$\Gamma^t.P = \{\Gamma^t.p \mid p :\ p \in P\}$$
$$\Gamma^t.(E, <_b, l) = (E,\ (<_b \cup <_c)^+ \cup <_d,\ l)$$
where $<_c = <'_c \cap (E \times E)$ and $<_d = <'_d \cap (E \times E)$
and $(E', <', l') = t \circ_W (E, <_b, l)$ and $\overline{E'} = E' \backslash \Delta.(E', <', l')$
and $<'_c = \{(e, f) \mid e, f, m :\ e \in \overline{E'} \wedge f \in \overline{E'} \wedge l'.e = !m \wedge l'.f = ?m \wedge (\#g :: g <' e \wedge l'.g = !m) = (\#g :: g <' f \wedge l'.g = ?m)\}$
and $<'_d = \{(f, f) \mid f, m :\ f \in \overline{E'} \wedge l'.f = ?m \wedge (\#g :: g \in \overline{E'} \wedge l'.g = !m) \leq (\#g :: g <' f \wedge l'.g = ?m)\}$

In this definition, first an auxiliary later $(E', <', l')$ is computed as the sequential composition of t and $(E, <_b, l)$. Then causalities $<'_c$ are defined for the matching communications, and causalities $<'_d$ are defined for the deadlocked receipt events. Finally, only the causalities on events E (i.e. not on events from previous behavior t) are added to later $(E, <_b, l)$.

For the running example, we define later $p'_4 = \Gamma^{[\epsilon]}.p_4$ and $p'_5 = \Gamma^{[\epsilon]}.p_5$. When visualizing p'_4 and p'_5, we add the additional communication causalities according to $<'_c$ with dashed arrows, and the additional deadlock causality for unmatched receipts ($<'_d$) with a dotted arrow as follows:

p'_4: 

p'_5: 

For p'_4 this then boils down to: 

. For p'_5, the result was already visualized in Section 2.1.

Since Γ is a causality extension, it maintains predicate T. However, Γ can introduce deadlocks. The following are some closure properties of Γ:

$$\begin{array}{ll} (\textit{shrinking}) & \Gamma^t.p \preceq p \\ (\textit{idempotence}) & \Gamma^t.p = \Gamma^t.(\Gamma^t.p) \\ (\textit{monotonicity}) & p \preceq q \ \Rightarrow\ \Gamma^t.p \preceq \Gamma^t.q \end{array}$$

These properties can even be generalized to sets of laters.

Consider the three laters x, y and z. If x and z are label-disjoint, then condition $\Gamma^t.(x \| z) \preceq \Gamma^t.x$ denotes that each event from later z is contained in the set of deadlocked events $\Delta.(\Gamma^t.(x \| z))$ of $\Gamma^t.(x \| z)$. For this kind of expressions, we have the following rules:

- *multiple deadlock extension*, provided x and $y \| z$ are label-disjoint:

$$\Gamma^t.(x \| y) \preceq \Gamma^t.x \ \wedge\ \Gamma^t.(x \| z) \preceq \Gamma^t.x \quad \equiv \quad \Gamma^t.(x \| y \| z) \preceq \Gamma^t.x$$

- *elimination*, provided $x \| y$ and z are label-disjoint:

$$\Gamma^t.(x \| y \| z) \preceq \Gamma^t.(x \| y) \quad \Rightarrow \quad \Gamma^t.(x \| z) \preceq \Gamma^t.x$$

2.2.3 History parameter

History parameter t of Γ is necessary for our study of sequential composition in Section 3.1. There we exploit the following important property:

$$\Gamma^t.(\{p\} \circ_W Q) \;\doteq\; \Gamma^t.(\{p\} \circ_W \Gamma^{t \circ_W p}.Q)$$

which can be proved by splitting $\doteq$ in its two directions:

$$\begin{array}{ll}
 & \Gamma^t.(\{p\} \circ_W \Gamma^{t \circ_W p}.Q) \;\sqsubseteq\; \Gamma^t.(\{p\} \circ_W Q) \\
\Leftarrow & \quad \{\text{monotonicity of } \Gamma\} \\
 & \{p\} \circ_W \Gamma^{t \circ_W p}.Q \;\sqsubseteq\; \{p\} \circ_W Q \\
\Leftarrow & \quad \{\text{monotonicity of } \circ_W\} \\
 & \Gamma^{t \circ_W p}.Q \;\sqsubseteq\; Q \\
\equiv & \quad \{\text{shrinking } \Gamma\} \\
 & \mathit{true}
\end{array}$$

$$\begin{array}{ll}
 & \Gamma^t.(\{p\} \circ_W Q) \;\sqsubseteq\; \Gamma^t.(\{p\} \circ_W \Gamma^{t \circ_W p}.Q) \\
\equiv & \quad \{\text{idempotence of } \Gamma\} \\
 & \Gamma^t.(\Gamma^t.(\{p\} \circ_W Q)) \;\sqsubseteq\; \Gamma^t.(\{p\} \circ_W \Gamma^{t \circ_W p}.Q) \\
\Leftarrow & \quad \{\text{monotonicity of } \Gamma\} \\
 & \Gamma^t.(\{p\} \circ_W Q) \;\sqsubseteq\; \{p\} \circ_W \Gamma^{t \circ_W p}.Q \\
\Leftarrow & \quad \{\text{calculus}\} \\
 & (\forall q : q \in Q : \Gamma^t.(p \circ_W q) \;\preceq\; p \circ_W \Gamma^{t \circ_W p}.q)
\end{array}$$

For the remaining $\preceq$, note that the event sets and the labeling are identical, and hence we only need to show that the left-hand side contains more causalities than the right-hand side. Since $\circ_W$ is associative, $(E', <', l')$ is identical in both Γ's. Since the events of q are contained in the events of $p \circ_W q$, the orders introduced by Γ in the right term are a subset of the orders introduced by Γ in the left term.

2.3 Semantics of compositional MSC

Using the preceding concepts, we define a semantics of compositional MSC as an extension of the MSC semantics of [KL98]. For simplicity reasons, we delay the introduction of the communication causalities; in Section 3.1 we will show how they can be introduced earlier (like in [KL98]). We start by giving the semantics of basic MSC, then the semantics of high-level MSC, and finally we complete this semantics by including the communication causalities.

2.3.1 Basic MSC

A common representation of basic MSCs is a graphical one like in Figure 2.1, but they can also be transformed into a textual representation. The semantics (without communication) of a basic MSC B in the instance-oriented textual representation [Ren99] is defined as a later $M_{bmsc}[\![B]\!]$ as follows:

$$\begin{array}{rcl} M_{bmsc}[\![\langle\,\rangle]\!] & = & [\epsilon] \\ M_{bmsc}[\![\textbf{inst}\ i; S\ \textbf{endinst}; B]\!] & = & M_{inst}[\![S]\!](i) \parallel M_{bmsc}[\![B]\!] \end{array}$$

As communication is not yet considered, the individual process instances are addressed independently, and combined using parallel composition. In turn, later $M_{inst}[\![S]\!](i)$ considers the description S of a single instance i:

$$\begin{array}{rcl} M_{inst}[\![\langle\,\rangle]\!](i) & = & [\epsilon] \\ M_{inst}[\![a; S]\!](i) & = & M_{inst}[\![a]\!](i)\ \circ_S\ M_{inst}[\![S]\!](i) \\ \\ M_{inst}[\![\textbf{co}\ \langle\,\rangle\ \textbf{endco}]\!](i) & = & [\epsilon] \\ M_{inst}[\![\textbf{co}\ a; C\ \textbf{endco}]\!](i) & = & M_{inst}[\![a]\!](i) \parallel M_{inst}[\![\textbf{co}\ C\ \textbf{endco}]\!](i) \\ \\ M_{inst}[\![\textbf{in}\ n\ \textbf{from}\ j]\!](i) & = & [?(n, j, i)] \\ M_{inst}[\![\textbf{out}\ n\ \textbf{to}\ j]\!](i) & = & [!(n, i, j)] \\ M_{inst}[\![\textbf{local}\ b]\!](i) & = & [b(i)] \end{array}$$

The events per instance are combined using sequential composition in their order of occurrence, and the events in co-regions are combined using parallel composition. Finally the single events are mapped to elementary laters.

Function ϕ, which maps events to instances, can then be defined as follows: $\phi.(?(n, j, i)) = i$, $\phi.(!(n, i, j)) = i$ and $\phi.(b(i)) = i$. By construction, for each basic MSC B the corresponding later $M_{bmsc}[\![B]\!]$ is a strict partial order.

To ensure that predicate T is satisfied, we assume that no instance name occurs more than once per bMSC [Ren99], and we require that in each co-region the events are label disjoint. The interest in co-regions is usually very limited (they are completely excluded in [HJ00, GMP03]), so this is no severe restriction. The unrealistic assumption that for each message name there is at most one send event and at most one receipt event per bMSC [KL98], is not required here.

2.3.2 High-level MSC

High-level MSCs are often represented by a finite directed graph, in which each node is labeled with a basic MSC, and in which one node is designated as initial node. Each possible behavior corresponds to the sequential composition of the basic MSCs encountered at a single path through the graph. We prefer the graph

to be normalized such that if a node has more than one outgoing edge, then the basic MSC associated with the node is the empty one. In this way the choices are made explicit in nodes without an associated basic MSC.

This graphical representation of high-level MSC can also be transformed into a textual representation. The semantics (without communication) of high-level MSC A in textual representation is defined as a set of laters $M_{hmsc}[\![A]\!]$ as follows:

$$\begin{array}{rcl} M_{hmsc}[\![\textbf{empty}]\!] & = & \{[\epsilon]\} \\ M_{hmsc}[\![\textbf{msc}\ name; B\ \textbf{endmsc}]\!] & = & \{M_{bmsc}[\![B]\!]\} \\ M_{hmsc}[\![A\ \textbf{seq}\ B]\!] & = & M_{hmsc}[\![A]\!] \circ_W M_{hmsc}[\![B]\!] \\ M_{hmsc}[\![A\ \textbf{alt}\ B]\!] & = & M_{hmsc}[\![A]\!] \cup M_{hmsc}[\![B]\!] \end{array}$$

By construction, each later in $M_{hmsc}[\![...]\!]$ is a strict partial order, and satisfies predicate T. Following the standardized semantics of MSC, we consider *weak* sequential composition and *delayed* choice. The use of delayed choice is sometimes referred to as the wait-and-see approach.

We do not explicitly address iteration, since it can be expressed via least fixed points and sequential composition. Sometimes the parallel composition of high-level MSCs, denoted by **par**, is also considered. Its semantics can easily be expressed in terms of operator $\|$ on sets of laters, but we will not consider it in our current study.

2.3.3 MSC

Finally we introduce the causalities imposed by communication:

$$\begin{array}{l} M_{msc}[\![A]\!] = M_{msc}^{[\epsilon]}[\![A]\!] \\ M_{msc}^{t}[\![A]\!] = \Gamma^t.M_{hmsc}[\![A]\!] \end{array}$$

This is a proper definition since $M_{hmsc}[\![A]\!]$ satisfies predicate T. By construction, predicate T also holds for $M_{msc}^{t}[\![A]\!]$. Note that the application of Γ^t may introduce deadlocks, which violate the strict partial order property. This illustrates one of the reasons for our extended partial order semantics.

Using the example laters from Sections 2.1 and 2.2, the semantics of the MSC in Figure 2.1 corresponds to $\Gamma^{[\epsilon]}.(\{p_1\} \circ_W (\{p_2\} \cup \{p_3\}))$, which simplifies via $\{\Gamma^{[\epsilon]}.(p_1 \circ_W p_2), \Gamma^{[\epsilon]}.(p_1 \circ_W p_3)\}$ into $\{p_4', p_5'\}$. These two laters represent the possibility of either performing ex1 followed by ex2, or ex1 followed by ex3.

In [GMP03] there is a restriction that receive events in bMSCs may not be matched to send events in future bMSCs. In [MM01] an extension is proposed that drops this restriction. We consider the extension, since the original restriction conflicts with elegant rules, like sequential composition of two bMSCs being equal to simply connecting the instance axis [Ren99].

2.4 Implementations

In this section we define the usual way of obtaining an implementation for a given specification. The difference between them is that a specification describes behavior in terms of all process instances, while an implementation describes behavior in terms of each individual instance. Thus an implementation for an instance can be represented by a set of laters that contain events of that instance only.

Many synthesis algorithms have been proposed to generate an implementation. These algorithms are very similar, although they differ in the formalism that is used for the transformation (process algebra, automata theory, etc.) and the kind of output that is generated (Petri-net, state chart, etc.). The basic idea is to decompose the specification according to the instances. The joint execution behavior of an implementation is obtained by recomposing the instances.

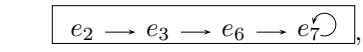
We do not consider the unusual implementation with message parameters proposed in [Gen05], which effectively boils down to renaming the messages and shifting the moments of choice. In such an implementation, additional parameters in a request message are sometimes used to fix the choice that should made by the receiver of the request.

2.4.1 Decomposition

The typical decomposition D of a set of laters M to its instances is:

$$D.M \;=\; \{i \mapsto \pi_i.M \mid i :\; i \in I\}$$

In this function, each instance name is mapped to the corresponding projection of M. Since projection is an event restriction, each projection of M satisfies predicate T if M itself satisfies predicate T.

For our running example, the decomposition of the laters, $D.\{p'_4, p'_5\}$, yields the following: $\{\, X \mapsto \{\, \boxed{e_1 \longrightarrow e_4 \longrightarrow e_5}, \boxed{e_1 \longrightarrow e_9 \longrightarrow e_{10} \leftrightarrows e_{11} \longrightarrow e_{12}} \,\},\; Y \mapsto \{$  $\}\,\}$.

Let us briefly investigate what might be lost by decomposition. For a singleton set $\{(E, <, l)\}$, note that E and l are partitioned per instance, and hence only the causalities between different instances are lost. For each later in a larger set M, also the link between its projections in the different instances is lost.

2.4.2 Recomposition

To study the joint execution behavior of the decompositions, the decomposition has to be recomposed. Using the definition from the previous section, the typical

recomposition R of a decomposition with history t becomes:

$$R^t.\{i \mapsto \pi_i.M \mid i : \ i \in I\} \ = \ \Gamma^t.(\|i : i \in I : \ \pi_i.M)$$

This is a proper definition provided $T.M$ holds, since T is maintained under parallel composition with disjoint labels. The projections are label-disjoint, since for each label k all events with that label belong to one instance, viz. $\phi.k$.

2.4.3 Recomposition and decomposition

Combining the definitions of recomposition and decomposition, we obtain for any set of laters M and history t the expression

$$(R^t \circ D).M \ = \ \Gamma^t.(\|i : i \in I : \ \pi_i.M)$$

where $\circ$ denotes function composition.

Given an MSC specification A and a history t, the behavior of this specification is expressed by $M^t_{msc}[\![A]\!]$ and the behavior of its usual implementation is expressed by $(R^t \circ D).M^t_{msc}[\![A]\!]$.

2.4.4 Monotonicity

We stress that $R^t \circ D$ is not monotonic with respect to $\sqsubseteq$. However, $R^t \circ D$ is monotonic with respect to causality extensions like Γ^t, for each set of laters M and history t:

$$(R^t \circ D).(\Gamma^t.M) \ \sqsubseteq \ (R^t \circ D).M$$

which can be proved as follows:

$$\begin{array}{ll}
 & (R^t \circ D).(\Gamma^t.M) \ \sqsubseteq \ (R^t \circ D).M \\
\equiv & \quad \{\text{definition of } R^t \circ D\} \\
 & \Gamma^t.(\|i : i \in I : \ \pi_i.(\Gamma^t.M)) \ \sqsubseteq \ \Gamma^t.(\|i : i \in I : \pi_i.M) \\
\Leftarrow & \quad \{\text{monotonicity of } \Gamma\} \\
 & (\|i : i \in I : \ \pi_i.(\Gamma^t.M)) \ \sqsubseteq \ (\|i : i \in I : \pi_i.M) \\
\Leftarrow & \quad \{\text{property of } \|\} \\
 & (\forall i : i \in I : \ \pi_i.(\Gamma^t.M) \ \sqsubseteq \ \pi_i.M) \\
\Leftarrow & \quad \{\text{calculus}\} \\
 & (\forall i, m : i \in I \wedge m \in M : \ \pi_i.(\Gamma^t.m) \ \preceq \ \pi_i.m) \\
\equiv & \quad \{\text{monotonicity of } \pi \text{ with respect to causality extension } \Gamma^t\} \\
 & \mathit{true}
\end{array}$$

2.4.5 Relation with operational formalisms

Using our later representation, implementations in operational formalisms can easily be obtained. In an interleaved execution model where the labels denote

atomic actions, the set of maximal behaviors of a single later are the linearizations of the maximal deadlock-free prefix. The set of maximal behaviors of a set of laters is the union of the maximal behaviors of the individual laters.

In turn, linearizations can easily be transformed to process algebraic expressions using the delayed choice operator [BM95]. The implementation of our running example corresponds to the following CSP-style implementation:

$$\begin{array}{lllll} X: & !a & \cdot (?b \cdot !c & + ?d &) \\ Y: & ?a \cdot !b & \cdot (?c & + !d \cdot ?c) & \end{array}$$

Notice that there are no actions for events e_7, e_{10}, e_{11} and e_{12} since these events cannot occur, i.e. they are guaranteed to be behind a deadlock. As we consider trace equivalence, the process algebra expression contains no deadlock δ.

2.5 Conclusions

We have argued that for the study of realizability one single formalism should be used to express both MSC specifications and their implementations. Instead of adopting the usual notion of execution traces for this purpose, we prefer to use a more abstract formalism that is closer to MSC. Therefore we have extended the notion of a labeled partially-ordered set, also known as an lposet, into a so-called later that allows deadlocks.

It is further work to reconsider the symbols that are used for the operators. In particular the symbol $\circ_W$ for sequential composition is in fact too large, and the use of a comma in the usual set notation like $\{p\,,\,q\}$ is too small to denote choice. The last series of expressions in Section 4.3 also illustrates this issue.

In our framework both distributed implementations and MSC specifications can be conveniently expressed. In particular for the specifications, we have developed a denotational semantics for compositional MSC. In Chapter 3 we will use this framework to study realizability.

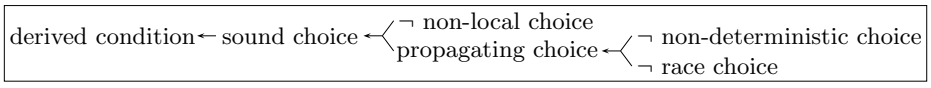
Chapter 3

Realizability criteria

In this chapter that is primarily based on [MGR05, MRW06], we deal with realizability criteria for compositional MSC. A specification is called realizable if there exists an implementation that is trace equivalent to it. We propose a complete classification of realizability criteria for the choice construct. The best-known criterion is non-local choice [BAL97], but also various other criteria [Hél01, HJ00, Gen05] have been proposed to determine the realizability of a given MSC. In addition to non-local choice, we define two classes of problems related to the propagation of choice, and we use them to discuss some related work.

On the one hand, many existing realizability criteria seem to be tricky formalizations of intuitions about realizability. On the other hand, theoretical work like [AEY05] investigates the decidability and worst-case time complexity of checking whether an MSC is realizable, but it provides no practical criteria. In contrast to both approaches, we start by studying under what circumstances specifications are trace equivalent to their implementations, and by formally deriving a realizability condition that is both necessary and sufficient. Based on this condition, we derive our classification of realizability criteria for compositional MSC.

The following is an overview of the many conditions that will be introduced:



Overview Section 3.1 is the most technical section, in which we study under what conditions specifications and implementations are trace equivalent. These conditions are used to derive a complete classification of realizability criteria in Section 3.2. In Section 3.3 we use these criteria to point out some errors in related work, and we conclude this chapter in Section 3.4.

3.1 Realizability problem

In this section, we study whether compositional MSC specifications are realizable, i.e. whether the behavior of compositional MSC specifications is trace equivalent to the behavior of their implementations. Using the notions from Chapter 2, this property can be expressed as: for each MSC A and history t

$$M^t_{msc}[\![A]\!] \;\doteq\; (R^t \circ D).M^t_{msc}[\![A]\!]$$

In what follows we will investigate for which MSCs this property holds. This will lead to derived condition ▲ in Section 3.1.2, which we require for each choice within the MSC.

Using the definition of $\doteq$, we study this property by splitting $\doteq$ into $\sqsubseteq$ and $\sqsupseteq$.

3.1.1 Implementation contains specification

We first show that the specification is contained in the implementation, i.e. for each MSC A and history t:

$$M^t_{msc}[\![A]\!] \;\sqsubseteq\; (R^t \circ D).M^t_{msc}[\![A]\!]$$

It can be proved as follows:

$$\begin{array}{ll} & (R^t \circ D).M^t_{msc}[\![A]\!] \\ = & \quad \{\text{definition of } R^t \circ D\} \\ & \Gamma^t.(\|i : i \in I :\ \pi_i.M^t_{msc}[\![A]\!]) \\ \sqsupseteq & \quad \{\text{property of } \pi \text{ and } \|\text{; monotonicity of } \Gamma\} \\ & \Gamma^t.M^t_{msc}[\![A]\!] \\ = & \quad \{\text{definition of } M^t_{msc}[\![A]\!]\text{; idempotence of } \Gamma\} \\ & M^t_{msc}[\![A]\!] \end{array}$$

3.1.2 Specification contains implementation

In the remainder we derive conditions under which the implementation is contained in the specification, i.e. for each MSC A and history t:

$$(R^t \circ D).M^t_{msc}[\![A]\!] \;\sqsubseteq\; M^t_{msc}[\![A]\!]$$

We will set up an inductive argument based on the structure of the high-level MSC, assuming that the following rewrite rules have been applied:

$$\begin{array}{lcl} (\mathbf{empty})\ \mathbf{seq}\ C & \rightarrow & C \\ (A\ \mathbf{seq}\ B)\ \mathbf{seq}\ C & \rightarrow & A\ \mathbf{seq}\ (B\ \mathbf{seq}\ C) \\ (A\ \mathbf{alt}\ B)\ \mathbf{seq}\ C & \rightarrow & (A\ \mathbf{seq}\ C)\ \mathbf{alt}\ (B\ \mathbf{seq}\ C) \end{array}$$

These rules do not change the occurrences of choice, but they ensure that the first argument of sequential composition is just a single basic MSC.

Using the property of Γ and $\circ_W$ in Section 2.2, we derive an alternative characterization of $M^t_{msc}[\![...]\!]$ in which communication is addressed earlier (like in [KL98]):

$$\begin{array}{ll}
M^t_{msc}[\![\textbf{msc}\ name; A\ \textbf{endmsc}]\!] & = M^t_{msc}[\![\textbf{msc}\ name; A\ \textbf{endmsc seq empty}]\!] \\
M^t_{msc}[\![\textbf{empty}]\!] & = \{[\epsilon]\} \\
M^t_{msc}[\![\textbf{msc}\ name; A\ \textbf{endmsc seq}\ B]\!] & \doteq \Gamma^t.(\{M_{bmsc}[\![A]\!]\} \circ_W M^{t \circ_W M_{bmsc}[\![A]\!]}_{msc}[\![B]\!]) \\
M^t_{msc}[\![A\ \textbf{alt}\ B]\!] & = M^t_{msc}[\![A]\!] \cup M^t_{msc}[\![B]\!]
\end{array}$$

Case: empty

This is the base case, which has a very simple proof:

$$\begin{array}{ll}
 & (R^t \circ D).M^t_{msc}[\![\textbf{empty}]\!] \\
= & \quad \{\text{alternative characterization of } R^t \circ D\} \\
 & (R^t \circ D).\{[\epsilon]\} \\
= & \quad \{\text{calculus}\} \\
 & \{[\epsilon]\} \\
= & \quad \{\text{alternative characterization of } R^t \circ D\} \\
 & M^t_{msc}[\![\textbf{empty}]\!]
\end{array}$$

Case: sequential composition

This inductive case can be proved as follows:

$$\begin{array}{ll}
 & (R^t \circ D).M^t_{msc}[\![\textbf{msc}\ name; A\ \textbf{endmsc seq}\ B]\!] \\
\doteq & \quad \{\text{alternative characterization of } R^t \circ D\} \\
 & (R^t \circ D).(\Gamma^t.(\{M_{bmsc}[\![A]\!]\} \circ_W M^{t \circ_W M_{bmsc}[\![A]\!]}_{msc}[\![B]\!])) \\
\sqsubseteq & \quad \{\text{monotonicity of } (R^T \circ D)\} \\
 & (R^t \circ D).(\{M_{bmsc}[\![A]\!]\} \circ_W M^{t \circ_W M_{bmsc}[\![A]\!]}_{msc}[\![B]\!]) \\
\sqsubseteq & \quad \{\bullet \text{ see below}\} \\
 & \Gamma^t.(\{M_{bmsc}[\![A]\!]\} \circ_W (R^{t \circ_W M_{bmsc}[\![A]\!]} \circ D).M^{t \circ_W M_{bmsc}[\![A]\!]}_{msc}[\![B]\!]) \\
\doteq & \quad \{\text{induction hypothesis, monotonicity of } \Gamma \text{ and } \circ_W\} \\
 & \Gamma^t.(\{M_{bmsc}[\![A]\!]\} \circ_W M^{t \circ_W M_{bmsc}[\![A]\!]}_{msc}[\![B]\!]) \\
\doteq & \quad \{\text{alternative characterization of } R^t \circ D\} \\
 & M^t_{msc}[\![\textbf{msc}\ name; A\ \textbf{endmsc seq}\ B]\!]
\end{array}$$

The step marked $\bullet$ follows from the following rule, where m denotes a later that does not order events in different instances, and M denotes a set of laters:

$$(R^t \circ D).(\{m\} \circ_W M) \ \doteq\ \Gamma^t.(\{m\} \circ_W (R^{t \circ_W m} \circ D).M)$$

This rule can be proved as follows:

$$
\begin{array}{ll}
 & (R^t \circ D).(\{m\} \circ_W M) \\
= & \quad \{\text{definition of } R \circ D\} \\
 & \Gamma^t.(\|i : i \in I :\ \pi_i.(\{m\} \circ_W M)) \\
= & \quad \{\text{distribution}\} \\
 & \Gamma^t.(\|i : i \in I :\ \pi_i.\{m\}\ \circ_W\ \pi_i.M) \\
= & \quad \{\text{distribution}\} \\
 & \Gamma^t.((\|i : i \in I :\ \pi_i.\{m\})\ \circ_W\ (\|i : i \in I :\ \pi_i.M)) \\
= & \quad \{\text{use that } m \text{ does not order events in different instances}\} \\
 & \Gamma^t.(\{m\} \circ_W (\|i : i \in I :\ \pi_i.M)) \\
\doteq & \quad \{\text{property of } \Gamma \text{ and } \circ_W\} \\
 & \Gamma^t.(\{m\} \circ_W \Gamma^{t \circ_W m}.(\|i : i \in I :\ \pi_i.M)) \\
= & \quad \{\text{definition of } R \circ D\} \\
 & \Gamma^t.(\{m\} \circ_W (R^{t \circ_W m} \circ D).M)
\end{array}
$$

This proof exploits that sequential composition is *weak*. In view of the graphical syntax of MSC, it would be more natural to define sequential composition as strong. However, the above rule only holds for weak sequential composition. If we would start to replace $\circ_W$ by $\circ_S$ from the top of the above proof, then after the second step we get a term in which $\circ_W$ and $\circ_S$ are equivalent, and after the third step we get stuck as we need $\circ_W$ again. Although this does not prove that strong sequential composition is infeasible, it is at least an indication that weak sequential composition might be the strongest one that is realizable.

Case: basic MSC

This case is a corollary of case sequential composition:

$$
\begin{array}{ll}
 & (R^t \circ D).M^t_{msc}[\![\textbf{msc}\ name; A\ \textbf{endmsc}]\!] \\
= & \quad \{\text{alternative characterization of } R^t \circ D\} \\
 & (R^t \circ D).M^t_{msc}[\![\textbf{msc}\ name; A\ \textbf{endmsc seq empty}]\!] \\
\doteq & \quad \{\text{use case: sequential composition}\} \\
 & M^t_{msc}[\![\textbf{msc}\ name; A\ \textbf{endmsc seq empty}]\!] \\
= & \quad \{\text{alternative characterization of } R^t \circ D\} \\
 & M^t_{msc}[\![\textbf{msc}\ name; A\ \textbf{endmsc}]\!]
\end{array}
$$

Case: choice

This inductive case can be proved as follows:

$$
\begin{array}{ll}
 & (R^t \circ D).M^t_{msc}[\![A \textbf{ alt } B]\!] \\
= & \quad \{\text{alternative characterization of } R^t \circ D\} \\
 & (R^t \circ D).(M^t_{msc}[\![A]\!] \cup M^t_{msc}[\![B]\!]) \\
\sqsubseteq & \quad \{\blacktriangle \text{ see below}\} \\
 & (R^t \circ D).M^t_{msc}[\![A]\!] \ \cup \ (R^t \circ D).M^t_{msc}[\![B]\!] \\
\doteq & \quad \{\text{induction hypothesis (twice)}\} \\
 & M^t_{msc}[\![A]\!] \ \cup \ M^t_{msc}[\![B]\!] \\
= & \quad \{\text{alternative characterization of } R^t \circ D\} \\
 & M^t_{msc}[\![A \textbf{ alt } B]\!]
\end{array}
$$

So a choice between *realizable* MSCs A and B, given a history t, is realizable if the following condition ▲ holds:

$$(R^t \circ D).(M^t_{msc}[\![A]\!] \cup M^t_{msc}[\![B]\!]) \ \sqsubseteq \ (R^t \circ D).M^t_{msc}[\![A]\!] \ \cup \ (R^t \circ D).M^t_{msc}[\![B]\!]$$

This condition is both sufficient and necessary. If it does not hold, implementations contain additional behavior, which is usually called *implied* behavior.

Note that this condition reflects that the core implementation problem is that one collective choice is specified (see the right-hand side), while it must be implemented in a distributed way (see the left-hand side). Since this condition does not hold for each two MSCs A and B, we will study it in more detail in Section 3.1.3.

3.1.3 Sound choice

Before discussing our characterization of realizability criteria in Section 3.2, we first strengthen derived condition ▲ from Section 3.1.2 into a more convenient one for this purpose. Using the definition of $R^t \circ D$, it is equivalent to:

$$\Gamma^t.(\|i :: \pi_i.(M^t_{msc}[\![A]\!] \cup M^t_{msc}[\![B]\!])) \ \sqsubseteq \ \Gamma^t.(\|i :: \pi_i.M^t_{msc}[\![A]\!]) \ \cup \ \Gamma^t.(\|i :: \pi_i.M^t_{msc}[\![B]\!])$$

Or formulated differently, for each function $f :: [I \to (M^t_{msc}[\![A]\!] \cup M^t_{msc}[\![B]\!])]$ representing per instance a chosen later, (at least) one of the following holds (where g and h denote functions):

$$(\exists g : g :: [I \to M^t_{msc}[\![A]\!]] : \ \Gamma^t.(\|i :: \pi_i.f_i) \ \preceq \ \Gamma^t.(\|i :: \pi_i.g_i))$$

$$(\exists h : h :: [I \to M^t_{msc}[\![B]\!]] : \ \Gamma^t.(\|i :: \pi_i.f_i) \ \preceq \ \Gamma^t.(\|i :: \pi_i.h_i))$$

Checking this condition is quite involved in practice, since arbitrary combinations of projected laters f (i.e. from both $M^t_{msc}[\![A]\!]$ and $M^t_{msc}[\![B]\!]$) need to be considered. To reduce the number of combinations, we strengthen this condition in two steps. We first concentrate on the term in the first existential quantification:

$$
\begin{array}{ll}
 & \Gamma^t.(\|i :: \pi_i.f_i) \preceq \Gamma^t.(\|i :: \pi_i.g_i) \\
\Leftarrow & \quad \{ \text{ common design decision: monotonicity } \} \\
 & \Gamma^t.(\|i :: \pi_i.f_i) \preceq \Gamma^t.(\|i : \pi_i.f_i \preceq \pi_i.g_i : \pi_i.g_i) \\
\Leftarrow & \quad \{ \text{ adapt left-hand side to right-hand side: domain split; monotonicity } \} \\
 & \Gamma^t.((\|i : \pi_i.f_i \preceq \pi_i.g_i : \pi_i.g_i) \parallel (\|i : \pi_i.f_i \not\preceq \pi_i.g_i : \pi_i.f_i)) \\
 & \quad \preceq \Gamma^t.(\|i : \pi_i.f_i \preceq \pi_i.g_i : \pi_i.g_i) \\
\equiv & \quad \{ \text{ property of } \Gamma\text{: multiple deadlock extension rule } \} \\
 & (\forall j : \pi_j.f_j \not\preceq \pi_j.g_j : \\
 & \quad \Gamma^t.((\|i : \pi_i.f_i \preceq \pi_i.g_i : \pi_i.g_i) \parallel \pi_j.f_j) \preceq \Gamma^t.(\|i : \pi_i.f_i \preceq \pi_i.g_i : \pi_i.g_i)) \\
\Leftarrow & \quad \{ \text{ property of } \Gamma\text{: elimination rule } \} \\
 & (\forall j : \pi_j.f_j \not\preceq \pi_j.g_j : \\
 & \quad \Gamma^t.((\|i : i \neq j : \pi_i.g_i) \parallel \pi_j.f_j) \preceq \Gamma^t.(\|i : i \neq j : \pi_i.g_i))
\end{array}
$$

To continue, let us abbreviate $\Gamma^t.((\|i : i \neq j : \pi_i.g_i) \| \pi_j.f_j) \preceq \Gamma^t.(\|i : i \neq j : \pi_i.g_i)$ as $P.g.j.f_j$. Then we can concentrate on the remainder as follows:

$$
\begin{array}{ll}
 & (\forall f :: \\
 & \quad (\exists g :: (\forall j : \pi_j.f_j \not\preceq \pi_j.g_j : P.g.j.f_j)) \vee \\
 & \quad (\exists h :: (\forall j : \pi_j.f_j \not\preceq \pi_j.h_j : P.h.j.f_j))) \\
\Leftarrow & \quad \{ \text{ disturb symmetry between the instances } \} \\
 & (\forall f :: (\exists k :: \\
 & \quad (\exists g :: \pi_k.f_k \preceq \pi_k.g_k \wedge (\forall j : \pi_j.f_j \not\preceq \pi_j.g_j : P.g.j.f_j)) \vee \\
 & \quad (\exists h :: \pi_k.f_k \preceq \pi_k.h_k \wedge (\forall j : \pi_j.f_j \not\preceq \pi_j.h_j : P.h.j.f_j)))) \\
\equiv & \quad \{ \text{ case } j = k \text{ follows from the left conjunct } \} \\
 & (\forall f :: (\exists k :: \\
 & \quad (\exists g :: \pi_k.f_k \preceq \pi_k.g_k \wedge (\forall j : \pi_j.f_j \not\preceq \pi_j.g_j \wedge j \neq k : P.g.j.f_j)) \vee \\
 & \quad (\exists h :: \pi_k.f_k \preceq \pi_k.h_k \wedge (\forall j : \pi_j.f_j \not\preceq \pi_j.h_j \wedge j \neq k : P.h.j.f_j)))) \\
\Leftarrow & \quad \{ \text{ use tautology } (\forall f, k :: \\
 & \qquad (\exists g :: \pi_k.f_k \preceq \pi_k.g_k \wedge (\forall j : \pi_j.f_j \not\preceq \pi_j.g_j : \{\pi_j.f_j\} \not\sqsubseteq \pi_j.M^t_{msc}[\![A]\!])) \vee \\
 & \qquad (\exists h :: \pi_k.f_k \preceq \pi_k.h_k \wedge (\forall j : \pi_j.f_j \not\preceq \pi_j.h_j : \{\pi_j.f_j\} \not\sqsubseteq \pi_j.M^t_{msc}[\![B]\!]))) \} \\
 & (\forall f :: (\exists k :: \\
 & \quad (\forall g, j : \{\pi_j.f_j\} \not\sqsubseteq \pi_j.M^t_{msc}[\![A]\!] \wedge j \neq k : P.g.j.f_j) \wedge \\
 & \quad (\forall h, j : \{\pi_j.f_j\} \not\sqsubseteq \pi_j.M^t_{msc}[\![B]\!] \wedge j \neq k : P.h.j.f_j))) \\
\Leftarrow & \quad \{ \text{ quantifier shunting } \} \\
 & (\exists k :: (\forall j : j \neq k : \\
 & \quad (\forall f, g : \{\pi_j.f_j\} \not\sqsubseteq \pi_j.M^t_{msc}[\![A]\!] : P.g.j.f_j) \wedge \\
 & \quad (\forall f, h : \{\pi_j.f_j\} \not\sqsubseteq \pi_j.M^t_{msc}[\![B]\!] : P.h.j.f_j))) \\
\equiv & \quad \{ \text{ dummy renaming } \} \\
 & (\exists k :: (\forall j : j \neq k : \\
 & \quad (\forall g, n : n \in \pi_j.M^t_{msc}[\![B]\!] \wedge \{n\} \not\sqsubseteq \pi_j.M^t_{msc}[\![A]\!] : P.g.j.n) \wedge \\
 & \quad (\forall h, m : m \in \pi_j.M^t_{msc}[\![A]\!] \wedge \{m\} \not\sqsubseteq \pi_j.M^t_{msc}[\![B]\!] : P.h.j.m)))
\end{array}
$$

Thus we obtain what we call the *sound choice* property for a choice between MSCs A and B given a history t: there exists an instance k such that for each instance $j : j \neq k$ both

- $(\forall\ g :: [I \to M^t_{msc}[\![A]\!]],\ n :\ n \in \pi_j.M^t_{msc}[\![B]\!] \ \wedge\ \{n\} \not\sqsubseteq \pi_j.M^t_{msc}[\![A]\!]$:

$$\Gamma^t.((\|i : i \neq j :\ \pi_i.g_i)\ \|\ n)\ \preceq\ \Gamma^t.(\|i : i \neq j :\ \pi_i.g_i)\ )$$

- $(\forall\ h :: [I \to M^t_{msc}[\![B]\!]],\ m :\ m \in \pi_j.M^t_{msc}[\![A]\!] \ \wedge\ \{m\} \not\sqsubseteq \pi_j.M^t_{msc}[\![B]\!]$:

$$\Gamma^t.((\|i : i \neq j :\ \pi_i.h_i)\ \|\ m)\ \preceq\ \Gamma^t.(\|i : i \neq j :\ \pi_i.h_i)\ )$$

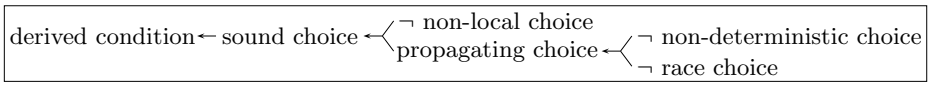
Here functions g and h represent a chosen later per instance. Later $n : n \in \pi_j.M^t_{msc}[\![B]\!] \ \wedge\ \{n\} \not\sqsubseteq \pi_j.M^t_{msc}[\![A]\!]$ denotes a later from MSC B that is no prefix of any later from MSC A. Note that behaviors occurring both in MSC A and MSC B are not problematic for the choice between A and B. The $\preceq$-term expresses that later n (or later m) cannot perform any behavior. Instance k and condition $j \neq k$ ensure that some instance may have initiative.

The choice in our running example is not a sound choice, as can be pointed out by considering both options for k. For $k = X$, we can choose $n = \pi_Y.(\Gamma^{p1}.p3)$ and $g_X = \Gamma^{p1}.p2$, which violate the first $\preceq$ term; and similarly for $k = Y$. We will discuss it in more detail using the non-local choice criterion in Section 3.2.

Notice that instead of considering arbitrary combinations of projected laters, on the left-hand side of the $\preceq$ in this condition, the combinations of projected laters contain only one later n from B, or only one later m from A respectively. Finally we stress that this condition is stronger than condition ▲.

3.2 Classification of realizability criteria

In this section we present our classification of realizability criteria based on the sound choice property from Section 3.1.3. If each choice in a specification is a sound choice, then the specification is trace equivalent to its usual implementation, and hence the specification is realizable. In the remainder of this section we will motivate and define our realizability criteria. For reasons of overview, we first schematically depict how our criteria are classified in comparison with sound choice and derived condition ▲ from Section 3.1.2:



Our criteria are independent, and examples of MSCs with some of these properties can be generated using the MSCs in Figure 3.1, viz. by considering a choice between MSC msc_base and the MSCs corresponding to the particular criteria of interest.

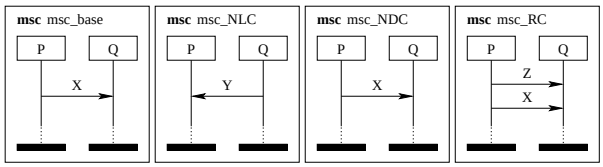


Figure 3.1 Basic MSCs to illustrate our classification

3.2.1 Non-local choice

From the perspective of a single process instance, the first question in a choice is whether the instance should initiate some behavior or it should just wait to receive a message that must be sent after the choice. An instance has initiative in an MSC if some first event of the instance is labeled with either an internal action, or sending a message, or receiving a message that was sent before the choice. A choice between two MSCs is local if at most one instance has initiative in these MSCs; otherwise several instances can independently start executing different MSCs. This is sometimes called cross-talk [KM00], and it typically leads to implied behaviors.

Non-local choice follows naturally from sound choice, and in particular from its $\preceq$-terms. Observe that a later n is likely to be problematic if for each label-disjoint later x we have $\Gamma^t.(x \| n) \not\preceq \Gamma^t.x$. Using the elimination rule for Γ, this condition follows from $\Gamma^t.n \not\preceq [\epsilon]$, which means that projected later n has initiative. Due to condition $j \neq k$ in the definition of sound choice, only instance k may have initiative, i.e. no two different instances, say i and j, may have initiative. This leads to the *non-local choice* (NLC) criterion for a choice between MSCs A and B given a history t:

$$\begin{array}{ll} (\exists i, j, m, n :: \ i \neq j & \wedge\, m \in \pi_i.M^t_{msc}[\![A]\!] \ \wedge\ \{m\} \not\sqsubseteq \pi_i.M^t_{msc}[\![B]\!] \ \wedge\ \Gamma^t.m \not\preceq [\epsilon] \\ & \wedge\, n \in \pi_j.M^t_{msc}[\![B]\!] \ \wedge\ \{n\} \not\sqsubseteq \pi_j.M^t_{msc}[\![A]\!] \ \wedge\ \Gamma^t.n \not\preceq [\epsilon] \quad) \end{array}$$

The difference with other variants of non-local choice in [BAL97, HJ00] is in our first two conjuncts on both m and n, where we ensure that sound choice is violated. The choice in the running example of Chapter 2 is non-local, since due to events e_4 and e_8 both X and Y have initiative.

3.2.2 Propagating choice

Absence of non-local choice is not sufficient to guarantee sound choice. It does guarantee that there is at most one process instance that determines the choice, viz. instance k in the definition of sound choice. The other instances j have no initiative and hence their chosen laters n are characterized by $\Gamma^t.n \preceq [\epsilon]$. What

remains to guarantee sound choice is that the other instances can resolve the choice using their first event, which is characterized by the *propagating choice* property for a choice between MSCs A and B given a history t: for each instance j both

- $\forall\ g :: [I \to M^t_{msc}[\![A]\!]],\ n :\ n \in \pi_j.M^t_{msc}[\![B]\!] \ \wedge\ \{n\} \not\sqsubseteq \pi_j.M^t_{msc}[\![A]\!] \ \wedge\ \Gamma^t.n \preceq [\epsilon]$:

$$\Gamma^t.((\|i : i \neq j :\ \pi_i.g_i)\ \|\ n)\ \preceq\ \Gamma^t.(\|i : i \neq j :\ \pi_i.g_i)$$

- $\forall\ h :: [I \to M^t_{msc}[\![B]\!]],\ m :\ m \in \pi_j.M^t_{msc}[\![A]\!] \wedge \{m\} \not\sqsubseteq \pi_j.M^t_{msc}[\![B]\!] \wedge \Gamma^t.m \preceq [\epsilon]$:

$$\Gamma^t.((\|i : i \neq j :\ \pi_i.h_i)\ \|\ m)\ \preceq\ \Gamma^t.(\|i : i \neq j :\ \pi_i.h_i)$$

The issue of propagation has not yet really been recognized, and as a consequence it is frequently ignored.

3.2.3 Non-deterministic choice

Propagating choice is an important property, but it is not easy to check for a given MSC. Therefore we are interested in a simple criterion that violates it. Consider a process instance without initiative, i.e. all first events are real receipts. Once a matching message arrives, a question is whether this is sufficient to derive the decision made about the choice. This is clearly not the case if the MSCs have a common first receipt.

A simple case that violates propagating choice is when the MSCs contain behaviors m and n that are different, although they share a common prefix p, i.e. $p \preceq m$ and $p \preceq n$. In case such a prefix p starts with a receipt behavior, instance j cannot resolve the choice using one of its initial events. This is characterized by the *non-deterministic choice* (NDC) criterion for a choice between MSCs A and B given a history t:

$$\begin{aligned}
(\exists j, m, n, p ::\ & p \preceq m\ \wedge\ p \preceq n\ \wedge \\
& m \in \pi_j.M^t_{msc}[\![A]\!]\ \wedge\ \{m\} \not\sqsubseteq \pi_j.M^t_{msc}[\![B]\!]\ \wedge\ \Gamma^t.m \preceq [\epsilon] \\
& \wedge\ n \in \pi_j.M^t_{msc}[\![B]\!]\ \wedge\ \{n\} \not\sqsubseteq \pi_j.M^t_{msc}[\![A]\!]\ \wedge\ \Gamma^t.n \preceq [\epsilon] \\
& \wedge\ (\exists g, h :\ g :: [I \to M^t_{msc}[\![A]\!]]\ \wedge\ h :: [I \to M^t_{msc}[\![B]\!]] : \\
& \qquad (\Gamma^t.((\|i : i \neq j :\ \pi_i.g_i)\ \|\ p)\ \not\preceq\ \Gamma^t.(\|i : i \neq j :\ \pi_i.g_i) \\
& \qquad\ \vee\ \Gamma^t.((\|i : i \neq j :\ \pi_i.h_i)\ \|\ p)\ \not\preceq\ \Gamma^t.(\|i : i \neq j :\ \pi_i.h_i))\ )\ )
\end{aligned}$$

This criterion can be made more syntactic by weakening the inner existential quantification into condition $p \not\preceq [\epsilon]$.

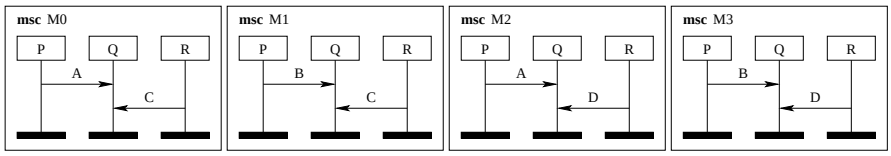


Figure 3.2 Non-local choice without implied behaviors

3.2.4 Race choice

Absence of non-deterministic choice can easily be checked, but it is not enough to guarantee propagating choice. It does guarantee that each process instance can resolve the choice when no initiating receipt event can end up receiving a message intended for a non-initial receipt event in another MSC. In case messages *arrive* in a different order than in which their *receipt* is specified in the bMSC, the instance may incorrectly derive which decision has been made. In contrast to non-deterministic choice, this property takes into account the asynchrony of the communication. So the first message receipt in one MSC, may actually have been sent according to another MSC in which the receipt is not the first event of the recipient. These cases are characterized by the *race choice* (RC) criterion for a choice between MSCs A and B given a history t:

$$
\begin{aligned}
(\exists j ::\ & (\exists g, n ::\ g :: [I \to M^t_{msc}[\![A]\!]] \\
& \quad \wedge n \in \pi_j.M^t_{msc}[\![B]\!] \ \wedge\ \{n\} \not\sqsubseteq \pi_j.M^t_{msc}[\![A]\!] \ \wedge\ \Gamma^t.n \preceq [\epsilon] \\
& \quad \wedge \Gamma^t.((\|i : i \neq j :\ \pi_i.g_i)\ \|\ n) \ \not\preceq\ \Gamma^t.(\|i : i \neq j :\ \pi_i.g_i) \\
& \quad \wedge (\forall p : p \preceq n \ \wedge\ \{p\} \sqsubseteq \pi_j.M^t_{msc}[\![A]\!] : \\
& \qquad \Gamma^t.((\|i : i \neq j :\ \pi_i.g_i)\ \|\ p) \ \preceq\ \Gamma^t.(\|i : i \neq j :\ \pi_i.g_i)\ )\ ) \\
& \vee (\exists h, m ::\ h :: [I \to M^t_{msc}[\![B]\!]] \\
& \quad \wedge m \in \pi_j.M^t_{msc}[\![A]\!] \ \wedge\ \{m\} \not\sqsubseteq \pi_j.M^t_{msc}[\![B]\!] \ \wedge\ \Gamma^t.n \preceq [\epsilon] \\
& \quad \wedge \Gamma^t.((\|i : i \neq j :\ \pi_i.h_i)\ \|\ m) \ \not\preceq\ \Gamma^t.(\|i : i \neq j :\ \pi_i.h_i) \\
& \quad \wedge (\forall p : p \preceq m \ \wedge\ \{p\} \sqsubseteq \pi_j.M^t_{msc}[\![B]\!] : \\
& \qquad \Gamma^t.((\|i : i \neq j :\ \pi_i.h_i)\ \|\ p) \ \preceq\ \Gamma^t.(\|i : i \neq j :\ \pi_i.h_i)\ )\ )\ )
\end{aligned}
$$

This definition of race choice boils down to a choice that is not a propagation choice and not a non-deterministic choice.

3.3 Related literature

In this section we discuss various related issues from the literature and we point out some errors in related work.

3.3.1 Definitions of non-local choice

A frequently-referenced paper for the definition of non-local choice is [BAL97]. Although much literature suggests the equivalence of the various definitions in [BAL97], we show that they are inconsistent. The informal introduction contains the following description:

> "When the wait-and-see strategy can be used to resolve a non-determinism within each process, we call the branching a *local branching choice*. Otherwise, when explicit synchronization between the processes is necessary to resolve a non-determinism, we call the branching a *non-local branching choice*."

Recall from Section 2.3 that the wait-and-see strategy boils down to the use of delayed choice. After introducing a formal semantic definition and a syntactic characterization (similar to ours), the following explanation of the syntactic version is given:

> "An MSC specification has no non-local branching choice iff at each of its branching points, the first events in all basic MSCs are sent by the same process."

Usually this last version is used to define and to detect non-local choice, while the first one is used as a convenient property about implementations. However, it is easy to see that these two definitions are different by studying a choice between the MSCs msc_base and msc_RC from Figure 3.1. Since process instance P is the only instance that has initiative, it is local according to the second definition. Then according to the first definition all non-determinism should be resolved, but process instance Q shows the contrary.

3.3.2 Implied scenarios

Implied scenarios are scenarios that are not contained in the MSC specification, but that are contained in implementations of the MSC. Although implied scenarios can result from propagation problems, only the relation with non-local choices (according to the syntactic definition of [BAL97]) has been studied. In [Uch03] the following two observations are made:

1. "Non-local choices are implied scenarios;"
2. "nevertheless the converse is not the case."

In contrast, [Muc03] makes the following two observations:

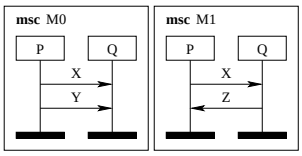


Figure 3.3 Hidden non-local choice

3. "Notice that a non-local choice is not enough to have an implied scenario."

4. "To have an implied scenarios these conditions hold: i) there is a non-local branching choice in the MSC specification so that ii) ..."

There are two contradictions here. Observation 3 falsifies erroneous observation 1, which can be shown by a choice between the two MSCs from Figure 3.2, where more than one process instance has initiative but no implied behaviors result.

In turn, observation 2 falsifies erroneous observation 4, which can be shown by a choice between the MSCs msc_base and msc_RC from Figure 3.1. Implementations of this example, without non-local choice, contain implied scenarios with the prefix $!Z \cdot !X \cdot ?X$; another example can be found in [Uch03]. Nevertheless observation 4 is the basis of the procedure in [Muc03] for detecting implied behavior.

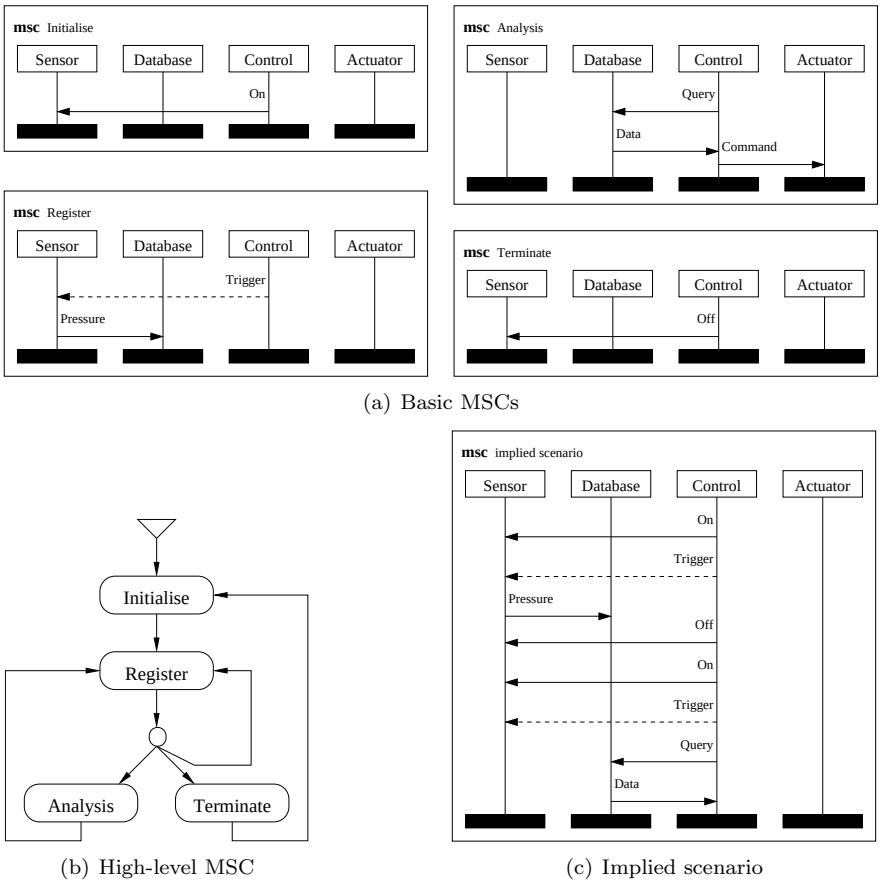
3.3.3 Delayed choice

The widely accepted solution to non-deterministic choice is to use delayed choice semantics instead of ordinary choice semantics. Since this solution is effective quite often (though not always), it has become part of the MSC standard. Sometimes, it can even eliminate non-local choice by factoring out a common non-local prefix of the MSCs after which a local choice remains.

However, we could not find any warning for the following possible side-effect. Namely, delayed choice can also expose non-local choice, e.g. in a choice between the MSCs from Figure 3.3. Although the choice itself is local, after applying delayed choice it becomes non-local.

3.3.4 Boiler example

A popular example to illustrate methods for detecting implied scenarios is a boiler system [UKM01, Muc02], see Figures 3.4(a) and 3.4(b) without the dashed trigger messages. This example contains non-local choice, since in the choice both the Sensor and the Control can initiate behaviors. Implementations of this system contain implied scenarios like the one in Figure 3.4(c).



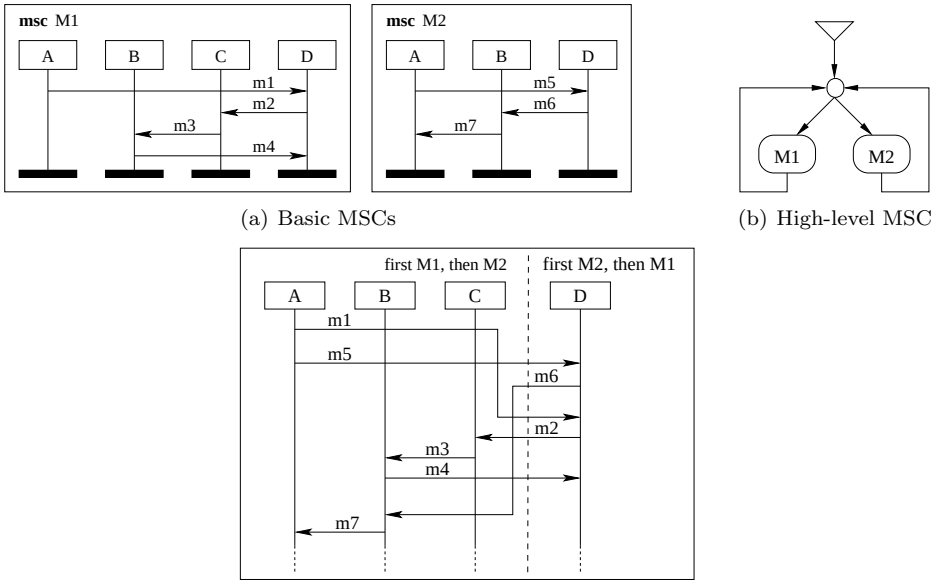
(a) Basic MSCs

(b) High-level MSC

(c) Implied scenario

Figure 3.4 Boiler example

We wondered whether these implied scenarios are really caused by non-local choice. As an experiment, the choice can be made local by introducing a trigger message from the the Control to the Sensor before the the Sensor may initiate an event in a basic MSC. Then, although the choice becomes local, there are still the same kind of implied scenarios. So we prefer to conclude that the *real* problem in this example is propagation of the choice to the Sensor; more specifically the choice is a race choice.



(a) Basic MSCs

(b) High-level MSC

(c) Implied behavior

Figure 3.5 Reconstructible choice example from [HJ00]

3.3.5 Reconstructible choice

In [HJ00] the reconstructible choice criterion is proposed in order to guarantee realizability, and it is mentioned explicitly that the communication channels are not assumed to be order preserving. However, this claim on reconstructability contradicts their example of a reconstructible MSC in [HJ00, Figure 15]. We have copied the MSC for this example into Figures 3.5(a) and 3.5(b).

Implementations allow behaviors that start as depicted in Figure 3.5(c), where process instance D starts to execute MSC M_2 followed by MSC M_1, while the other instances start to execute MSC M_1 followed by MSC M_2. However, prefix $!m_1 \cdot !m_5 \cdot ?m_5$ shows that this behavior is not part of the specified behavior. In terms of our classification, the choice is not properly propagated to instance D; more specifically the choice is a race choice.

3.4 Conclusions

We have studied the realizability problem for compositional MSC, especially in relation with choice. Although realizability can be determined by comparing a

given MSC specification with its synthesized implementation, it is more effective to have realizability criteria that can be applied to specifications.

Instead of inventing criteria based on some intuition about realizability, we have shown that it pays off to constructively derive necessary formal conditions. Thus we have derived a complete classification of realizability criteria for compositional MSC. The need for such a classification is clearly indicated by the many errors we have found in related work in the literature.

A line of further work is to derive other realizability criteria, because a choice that is no sound choice may still be realizable, and because more syntactic criteria would be easier to check for a given MSC and even to automate. It would also be interesting to formally relate some other proposed criteria to the realizability problem. Finally the realizability of other MSC constructs may be studied, of which parallel composition is a challenging one.

Chapter 4

Realizing non-local choice

In this chapter that is primarily based on [MG05, MGR05], we study how to obtain a reasonable implementation for MSCs that contain non-local choice. Among the realizability criteria from Chapter 3, non-local choice is the best-known criterion, but hardly any solution has been developed and there is definitely no standard implementation. Nevertheless it is an important issue, since non-local choice is almost inevitable in MSC specifications of distributed systems with autonomous computational units.

We propose a new direction to implement MSCs with non-local choice such that a little more behavior is introduced, but in a controlled way. We consider both a solution for systems with two process instances, and a solution for arbitrary numbers of instances. Both techniques require some practical assumptions on the MSC specification. Finally we apply these techniques to our case studies.

Overview In Section 4.1 we discuss several views on non-local choice, and we motivate our technique described in Section 4.2 and its generalization described in Section 4.3. Then in Section 4.4 we discuss the impact of these techniques on the case studies, and Section 4.5 contains the conclusions.

4.1 Views on non-local choice

In this section we discuss some ways to address the best-known choice problem from Chapter 3, viz. non-local choice.

4.1.1 Traditional approaches

For most MSCs that contain non-local choice, there exists no implementation with exactly the same behavior. Therefore such MSCs are often considered as erroneous, and hence methods are needed that either detect them syntactically (e.g. [BAL97]), or detect them by generating the resulting implied behaviors (e.g. [Muc03, UKM03]). However, these approaches do not address how to implement a given MSC that contains non-local choice.

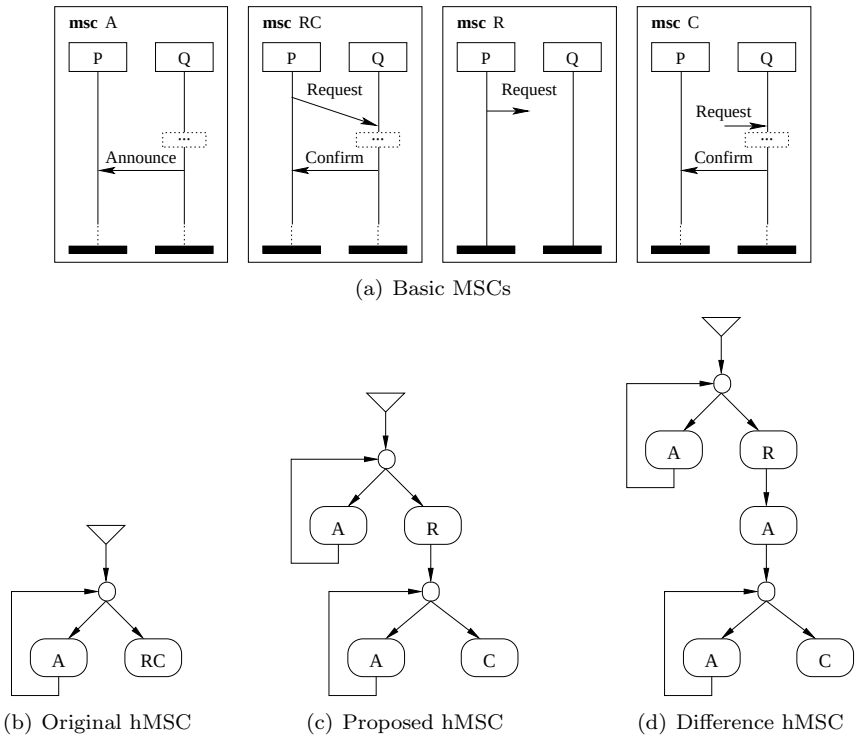
An obvious approach to overcome the problems resulting from non-local choice is to generate all implied behaviors and explicitly include them in the specification. Although implementations of the resulting specification contain no implicit extra behaviors, the MSC specification has become more complicated, which is definitely not desired from a practical point of view.

The problems with non-local choice can also be seen as implementation issues, and hence they should not even be addressed in a specification. To obtain an implementation, some coordination protocol needs to be introduced (e.g. [LL97]) which usually requires additional messages. Although this may lead to a nice layered design, the specific coordination protocol should be made explicit in order to obtain consistent implementations. In some cases non-local choice can be solved using synchronous communication, but implementing this in terms of asynchronous communication usually yields a bad performance.

4.1.2 Our approach

The source of the problems with non-local choice is that the implementations are distributed. Since the process instances are independent computational units, a coordination problem arises when the instances *together* need to make a transition in the high-level MSC. Nowadays this problem is mainly observed for choice, but in fact it also arises for strong sequential composition of MSCs. The latter issue has been solved by defining its semantics to be *weak* sequential composition (see also a remark in Section 3.1), which usually corresponds to the intentions of the developer of the MSC. For choice the use of delayed choice semantics is not sufficient.

Suppose that during execution all process instances have reached a given non-local choice. In absence of a coordination protocol, it can usually not be avoided that the instances initiate the execution of several different MSCs. This means that *implementations* of choice should allow, to some degree, parallel execution of the basic MSCs. Of course, the amount of additional parallel behavior should be minimal, and as soon as possible the behavior should converge to the behavior of a conventional (or synchronous) choice. Sections 4.2 and 4.3 describe this approach in more detail.



(a) Basic MSCs

(b) Original hMSC (c) Proposed hMSC (d) Difference hMSC

Figure 4.1 Simple MSC patterns for two processes

4.2 A solution for two processes

In this section we consider a system consisting of two process instances, say P and Q. For such a system we propose a way to implement an MSC that specifies a choice that is non-local but propagating. Our proposal is such that a little more behavior is introduced, but in a controlled way.

4.2.1 Simple pattern

Our approach assumes that the MSC specification has a certain shape, which we will explain using the MSCs in Figure 4.1. In particular the basic MSCs in Figure 4.1(a) assume that there are three different kinds of messages, viz. Request, Confirm and Announce, that do not occur in the remainder of basic MSC A. Notice that basic MSCs A, RC and C may contain some internal actions in instance Q, and their depicted behavior may be followed by some additional behavior.

For the specification we consider the high-level MSC from Figure 4.1(b), which is a frequently occurring pattern. The choice is non-local, and usual implementations of this choice contain unspecified behaviors that result in deadlocks.

To obtain an implementation, we propose to break the symmetry of the choice. More specifically, we propose to implement it as the high-level compositional MSC from Figure 4.1(c), where MSC RC has been split into basic MSCs R and C.

Then the basic MSCs can be interpreted as asymmetric negotiation scenarios. If instance P receives an Announce message from instance Q, basic MSC A is executed. Even after sending the Request message in basic MSC RC, i.e. in basic MSC R, instance P may still receive Announce messages from instance Q that indicate that basic MSC A must be executed. Once the Confirmation message arrives, execution of basic MSC RC can be completed by executing basic MSC C. Thus the Request message may be sent in parallel with parts of basic MSC A. Notice that the behavior of instance Q is not changed in this way, and that instance Q behaves as a kind of arbiter in the sense that it determines the order in which basic MSCs A and RC are executed.

In this implementation, the second choice is local for arbiter Q, and the first choice is local for instance P as the behavior of instance Q in the left alternative is contained in the right alternative. Nevertheless, the implementation may contain race choice if MSC A contains no message from instance P to instance Q. If MSC A contains a message from instance P to instance Q, the extra behaviors are depicted in Figure 4.1(d). Notice that MSC A can be executed several times between sending and receiving a Request message. The extra behaviors that we have introduced, start with behavior that leads to a deadlock in usual implementations.

4.2.2 Related approach

In [GY84] another implementation is proposed. They break the symmetry of the choice by calling the two process instances ‘winner’ and ‘loser’ respectively. When the instances detect interference between behaviors initiated by different instances, the synchronization between them is restored by discarding the behavior initiated by the loser. So their implementation also slightly deviates from the original specified behavior, but in a different way than our solution. They propose to ignore the additional behavior, while we store it for later use. In our case studies, their modification is not acceptable.

4.2.3 Generalized pattern

The pattern from Section 4.2.1 for a system with two process instances addresses only binary choice. In this section we sketch how it can be generalized to arbitrary choice. In the high-level MSC the successor MSCs of each non-local choice must

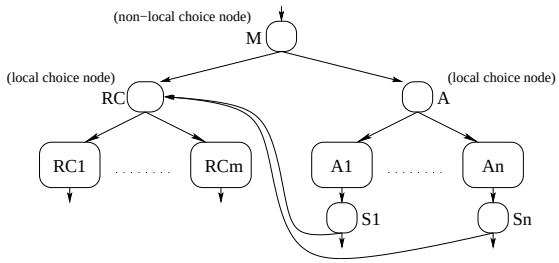


Figure 4.2 High-level MSC pattern

be partitioned into RC MSCs and A MSCs, see Figure 4.2. Each RC_i denotes a renaming of basic MSC RC from Figure 4.1(a), and each A_i denotes a renaming of a basic MSC A. The renaming must be such that all Announce messages are different, and that all Request messages are different. For convenience reasons, we have introduced a node RC and a node A, which are empty nodes that group the RC and A MSCs together.

To apply our technique for solving non-local choice, we must ensure that when instance P has sent a Request message, it still makes sense for instance Q to receive the message after execution of an A MSC. This is guaranteed if after any A MSC, i.e. in any node S, node RC is reachable. In addition, via the open outgoing edge of any node S also extra RC MSCs and an arbitrary series of A MSCs may be reachable.

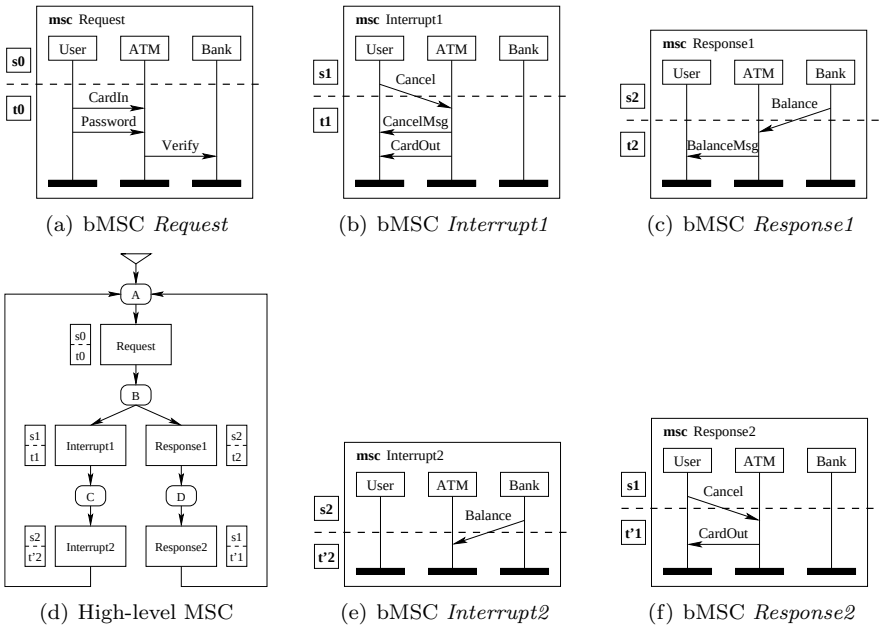
4.3 A solution for arbitrarily many processes

In this section we extend our approach for non-local choice from Section 4.2 to systems with an *arbitrary* number of process instances, and to MSCs in which several instances have initiative. We first address the MSC pattern we assume, followed by a description of our proposed implementation. Finally we relate it to a proposed MSC extension.

4.3.1 Running example

To illustrate our approach, we use a simplified version of the well-known ATM example [UKM03]. We have restricted it to its core non-local choice problem, as depicted in Figure 4.3. For later use, the MSCs contain some extra annotations; in particular the basic MSCs have been split by a horizontal line.

We briefly explain the functionality of this simplified ATM. In node A some re-



(a) bMSC *Request* (b) bMSC *Interrupt1* (c) bMSC *Response1*

(d) High-level MSC (e) bMSC *Interrupt2* (f) bMSC *Response2*

Figure 4.3 Simplified ATM example

petitive behavior is started with MSC *Request*. It consists of inserting a card and entering a password, followed by verifying the bank account. Then node B is reached, in which the user can choose to:

- interrupt and cancel the account verification, which corresponds to: MSC *Interrupt1* $\rightarrow$ node C $\rightarrow$ MSC *Interrupt2* $\rightarrow$ node A;
- wait for a balance report and press the cancel button to end the session: MSC *Response1* $\rightarrow$ node D $\rightarrow$ MSC *Response2* $\rightarrow$ node A.

4.3.2 Pattern

As discussed in Section 2.3, we assume the high-level MSC to be such that the choices are made explicit in (choice) nodes without an associated basic MSC. Thus we can easily interpret the high-level MSC as a graph in which the edges are labeled with (concatenated) basic MSCs, while the nodes contain no basic MSC anymore.

To apply our approach, each basic MSC must be split into a (preferably small) *front* part that may be executed in parallel, and the remaining *tail* part that will be part of a real choice. To solve non-local choice, in each node the choice between

the tails of the successor MSCs must be a local choice. This can be achieved as follows:

1. *choose an instance to become the "arbiter"*, which is typically a process instance that has in each basic MSC an event that occurs in an early stage;
2. *split the basic MSCs in the edges* into a front and a tail, such that for each node with more than one outgoing edge, the arbiter is the only instance with initiative in the tails of the edges. (If a tail is empty for an instance, also the tails of successor MSCs are involved in deciding whether the instance has initiative.)

The basic MSCs must be split in such a way that also the following conditions hold:

1. for each node and for each two of its outgoing edges e and f with different fronts, the node reached via edge e is no terminal node and it has an outgoing edge with the same front as edge f;
2. for each two edges e and f (from any two nodes) with different fronts, the event labels in the front of edge e do not occur in the front of edge f;
3. for each two edges e and f (from any two nodes), the event labels in the front of edge e do not occur in the tail of edge f (nor in the tail of edge e).

The first condition reflects that additional front behaviors, which are the additional parallel behaviors, can indeed be used in subsequent nodes. If it does not hold, a wrong arbiter might have been chosen. It is also possible that the MSC lacks some unavoidable behavior. To illustrate this, in our running example it is tempting to omit basic MSC *Interrupt2*. Thus this condition can constructively help to improve the MSC specification without studying its implementation.

The motivation for the last two conditions is quite technical and it will be discussed upon their use. Although our pattern contains some restrictions, it includes the patterns from Section 4.2.

Running example

Let us apply this pattern to our running example. All choices have been made explicit in the empty nodes A, B, C and D, and the only node with more than one outgoing edge, viz. node B, suffers from non-local choice.

To check the pattern, an arbiter must be chosen. Using the heuristics from the first step, instance *ATM* should be an appropriate arbiter. The next step is to split the basic MSCs according to this arbiter. This is depicted by horizontal dashed lines

in the basic MSCs in Figure 4.3. This way of splitting fulfills the first condition, e.g. in node B after MSC *Interrupt1* it is possible to execute the front of MSC *Response1* namely as front of MSC *Interrupt2*. The last two conditions also turn out to hold.

For reference purposes, we introduce names s_0, s_1, s_2, t_0, t_1, t_2, t'_1 and t'_2 for the basic MSC parts as indicated in Figure 4.3. For example, using $M_{inst}[\![\ldots]\!]$ from Section 2.3, we obtain for process instance *User* in terms of laters:

$s_0 =$	$[\epsilon]$	$t_0 =$	$[!CardIn] \circ_W [!Password]$		
$s_1 =$	$[!Cancel]$	$t_1 =$	$[?CancelMsg] \circ_W [?CardOut]$	$t'_1 =$	$[?CardOut]$
$s_2 =$	$[\epsilon]$	$t_2 =$	$[?BalanceMsg]$	$t'_2 =$	$[\epsilon]$

4.3.3 Implementation

In this section, we focus on the implementation for a single process instance, since the implementations for the other instances can be defined analogously. We will use techniques from constraint-oriented programming, and in particular the partial synchronization operator defined in Section 2.1.

We use V to denote the set of nodes in the high-level MSC, and E to denote the set of labeled edges. More specific, we represent each edge as a four-tuple $(v, m, n, w) \in E$ as follows: the edge is directed from node v to node w, and m and n are the front and the tail respectively of the corresponding basic MSC projected on the process instance to be implemented.

Our implementation is described in Figure 4.4, where the smallest solution of $I.v$ denotes the implementation for the process instance, given that v is the initial node of the high-level MSC. For $I.v$ we only consider the initial node v, while we consider all nodes v for $I_i.v$ and $I_a.v$, which are to be discussed next. Recall from Section 2.1 that $\|$ denotes parallel composition, $\circ_W$ denotes (weak) sequential composition, and $\cup$ denotes (delayed) choice.

The term $I.v$ is defined as the synchronized execution of the term $I_i.v.m$ for each individual front m in the MSC. Each term $I_i.v.m$ expresses where front m may be executed in relation with all tails, as described below. The intended synchronized execution is a kind of parallel composition that synchronizes the tails. Therefore we use the partial synchronization operator $\|_K$, which only synchronizes the events with labels from set K. For this set K we use the labels of the events in the tails. To apply this operator successfully, we exploit the last two (technical) conditions mentioned above.

The term $I_i.v.m$ describes the implementation in node v with respect to the *inactive* front m. If m is the front of a successor basic MSC of node v, its execution can be started and hence it becomes active. Otherwise it remains inactive, and a usual choice is performed on the tails of the successor basic MSCs of node v.

$$
\begin{array}{lcl}
I.v & = & (\|_K\ m : (\exists v', n', w' :: (v', m, n', w') \in E) :\ I_i.v.m) \\
\\
I_i.v.m & = & \left\{ \begin{array}{l} (\exists n, w :: (v, m, n, w) \in E) \quad : I_a.v.m.[\epsilon] \\ (\forall n, w :: (v, m, n, w) \notin E) \quad : \\ \qquad (\bigcup m', n', w' : (v, m', n', w') \in E :\ \{n'\}\ \circ_W\ I_i.w'.m) \end{array} \right. \\
\\
I_a.v.m.p & = & (\bigcup m', n', w' : (v, m', n', w') \in E : \\
 & & \qquad \left\{ \begin{array}{lcl} m \neq m' & : & I_a.w'.m.(p\ \circ_W\ n') \\ m = m' & : & \{(p \parallel m')\ \circ_W\ n'\}\ \circ_W\ I_i.w'.m \end{array} \right) \\
\end{array}
$$

Figure 4.4 Formalized implementation for a single process instance

The term $I_a.v.m.p$ describes the implementation in node v with respect to the *active* front m. The additional parameter p is used to accumulate the sequence of executions of tails since front m became active. In case the tail of a basic MSC with front m is executed, then it is required that front m was executed along the path p to node v, which is expressed by the term $(p \parallel m')$.

Running example

To illustrate this approach on our ATM example, we first apply it to the high-level description in terms of s and t. Afterwards, the specific details can be substituted to obtain the final implementations. First we give the instantiations of some of the formulas:

$I.A$	$=$	$I_i.A.s_0\ \|_K\ I_i.A.s_1\ \|_K\ I_i.A.s_2$
$I_i.A.s_1$	$=$	$\{t_0\}\ \circ_W\ I_i.B.s_1$
$I_i.B.s_1$	$=$	$I_a.B.s_1.[\epsilon]$
$I_a.B.s_1.[\epsilon]$	$=$	$\{([\epsilon]\|\|s_1) \circ_W t_1\}\ \circ_W\ I_i.C.s_1\ \cup\ I_a.D.s_1.([\epsilon] \circ_W t_2)$
$I_i.C.s_1$	$=$	$\{t_2'\}\ \circ_W\ I_i.A.s_1$
$I_a.D.s_1.([\epsilon] \circ_W t_2)$	$=$	$\{(([\epsilon] \circ_W t_2) \parallel s_1)\ \circ_W\ t_1'\}\ \circ_W\ I_i.A.s_1$

After simplification we obtain the following implementation per process instance:

$I.A$	$=$	$I_i.A.s_0 \|_K I_i.A.s_1 \|_K I_i.A.s_2$
$I_i.A.s_0$	$=$	$\{s_0\ \circ_W\ t_0\}\ \circ_W\ \{t_1 \circ_W t_2',\ t_2 \circ_W t_1'\}\ \circ_W\ I_i.A.s_0$
$I_i.A.s_1$	$=$	$\{t_0\}\ \circ_W\ \{s_1 \circ_W t_1 \circ_W t_2',\ (t_2\|\|s_1) \circ_W t_1'\}\ \circ_W\ I_i.A.s_1$
$I_i.A.s_2$	$=$	$\{t_0\}\ \circ_W\ \{(t_1\|\|s_2) \circ_W t_2',\ s_2 \circ_W t_2 \circ_W t_1'\}\ \circ_W\ I_i.A.s_2$

As an aid in understanding this solution, note that $I.A$ corresponds to the Petri-net in Figure 4.5. By substituting the specifics and eliminating the partial synchronization operator, the following final implementations are obtained:

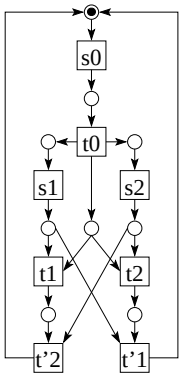


Figure 4.5 Petri-net for a single process instance in our running example

$$
\begin{array}{lcl}
I^{User} & = & \{[!CardIn] \circ_W [!Password]\} \circ_W \\
 & & \quad \{\ [!Cancel] \circ_W [?CancelMsg]\,, \\
 & & \quad\ \ [!Cancel] \circ_W [?BalanceMsg]\,, \\
 & & \quad\ \ [?BalanceMsg] \circ_W [!Cancel] \\
 & & \quad \} \circ_W \{[?CardOut]\} \circ_W I^{User} \\
I^{ATM} & = & \{[?CardIn] \circ_W [?Password] \circ_W [!Verify]\} \circ_W \\
 & & \quad \{\ [?Cancel] \circ_W [!CancelMsg] \circ_W [!CardOut] \circ_W [?Balance]\,, \\
 & & \quad\ \ [?Balance] \circ_W [!BalanceMsg] \circ_W [?Cancel] \circ_W [!CardOut] \\
 & & \quad \} \circ_W I^{ATM} \\
I^{Bank} & = & \{[?Verify] \circ_W [!Balance]\} \circ_W I^{Bank}
\end{array}
$$

The implementation for the *ATM* (which is the arbiter) and for the *Bank* are the usual ones. The possible behavior for the *User* has been extended, but it is intuitive in relation to Figure 4.3. In particular after pressing *Cancel*, the user can get a *BalanceMsg* instead of a *CancelMsg*, which resolves the potential deadlock.

4.3.4 Relation with compositional MSC

Another example of our approach is a producer-consumer system, which could be naturally specified using the MSC in Figure 4.6. The choice in this MSC is non-local, but our approach can provide an implementation, where instance P is the arbiter. This implementation corresponds to the behavior that is used in [MM01] to advocate the MSC extension that is called compositional MSC. Although the version in terms of compositional MSC is more precise for an implementation, our specification is simpler and more intuitive for a system specification and it still allows an implementation using the technique presented in this section.

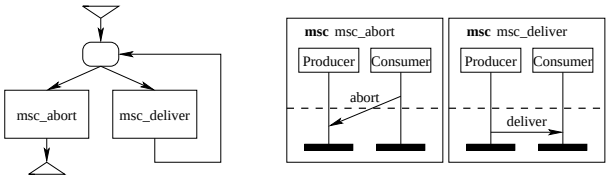


Figure 4.6 Producer-consumer example

4.4 Relation with the case studies

In this section we relate the implementation technique from Section 4.2 to the protocol standards we have worked on.

4.4.1 Base standard

In [MGWB03] we reported on our analysis of the communication protocols in the base standard, and in particular ENV 13735. We analyzed and extended its draft state tables, and we proposed modifications to overcome the problems found. The main kind of problems were:

- inconsistencies (e.g. nomenclature) between the manager and the agent;
- interference between typical scenarios.

Given our work on MSCs, it is interesting to study the last problem in an MSC context. For illustration purposes, we have included in Figure 4.7 a rough sketch of an MSC specification for this protocol. This specification contains non-local choices, and the interference problems we have found seem to be related to this. In fact the state transition tables already contained incomplete fragments of our implementation techniques.

4.4.2 Remote control extension

We have developed our techniques in order to apply them to the non-local choice in the remote control package. From Figures 1.1 and 1.2 it follows that the agent should become the arbiter. Then the deadlock scenario of Figure 1.3 can be avoided by continuing the deadlock behavior as depicted in Figure 4.8. The developers of the remote control package have agreed with such a solution, and it is used as the basis of the state transition tables that will be included in the standard.

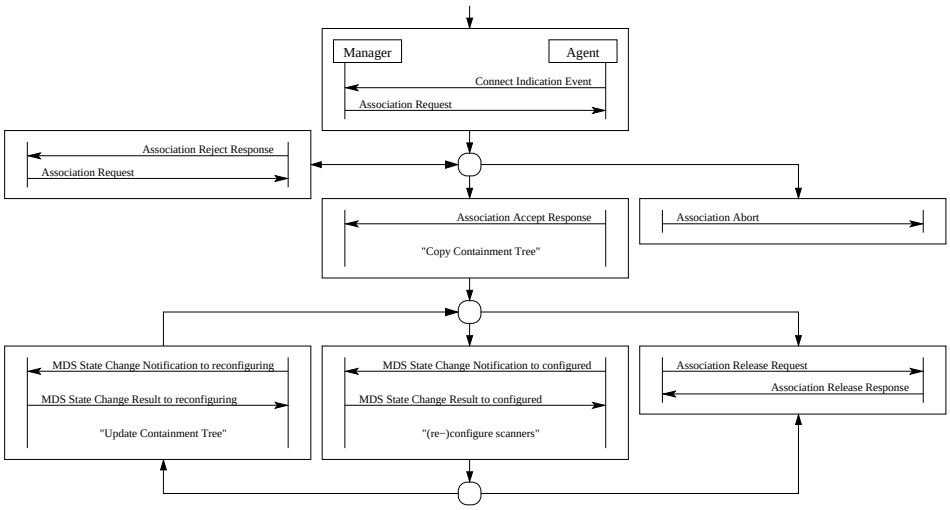


Figure 4.7 Sketch of an MSC specification for ENV13735

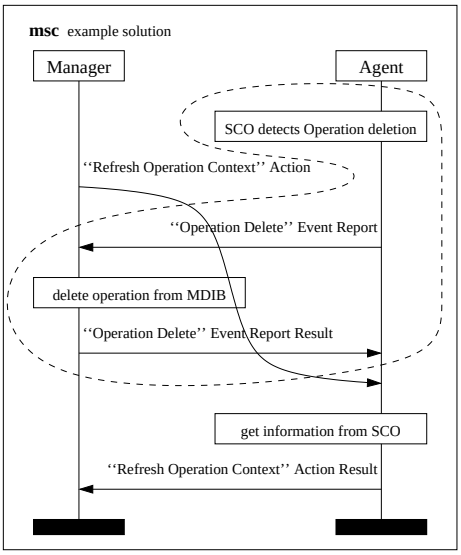


Figure 4.8 New behavior for remote control

4.4.3 Health-Level Seven

In [WGMS05] we reported on our analysis of parts of the Health Level Seven (HL7) standard. This is an ANSI standard that provides a comprehensive framework for

electronic health information. The most-widely used HL7 specification is called Infrastructure Management, which facilitates health-care applications to exchange key sets of clinical and administrative data. Non-local choice also played a role in this standard, and our techniques could be applied again. However, as a result of our work, this standardization committee has even started to work on a proposal to fundamentally simplify the interaction framework.

4.5 Conclusions

We have focused on the best-known realizability problem, viz. non-local choice. In the literature, MSCs with non-local choice are often simply classified as erroneous MSCs. However, non-local choice is often present in practical MSCs. Because no existing solution could be employed successfully to our case studies, we have proposed a new direction.

We have proposed alternative implementations that allow a little more behavior than the usual ones. In particular the extra behaviors have been introduced in the deadlock states of usual implementations. Upon application of these techniques, the extra behaviors must be validated by the designers. In contrast, the approach to eliminate deadlocks that is described in Section 6.2.5 does not suffer from this problem. We have successfully applied our techniques to the ISO/IEEE 1073.2 standard, and the additional behavior turned out to correspond to the intuition of the developers. As a recognition of this work, our names have been added to the list of authors of these standards.

Further work is to explore additional properties of our approach, including whether there exist implementations that are even closer to the original specification. It would also be interesting to investigate whether ignoring the additional behavior as in [GY84] can be integrated, and how propagation problems can be addressed. For the analysis of our proposal for arbitrary numbers of components, our theory about realizability in Chapter 3 must at least be extended with parallel composition, which in turn will require additional properties of laters in Chapter 2.

Chapter 5

Conclusions

Many protocols are developed by reasoning about a couple of typical scenarios, which makes it easy to overlook situations in which these scenarios interfere. The realizability problems we have addressed are instances of this phenomenon that can even lead to deadlocks. Further efforts in developing and transferring proper techniques for constructing protocols are clearly necessary.

Our work has directly influenced and improved the protocols in the ISO/IEEE 1073.2 standard for medical device communication. We started with verifying and adjusting some existing state transition tables, and later on we also created state transition tables from a couple of typical scenarios. As a recognition of this work, our names have been added to the list of authors of these standards. In addition, this practical work has served as inspiration to several theoretical contributions.

In Chapter 2 we have argued that the relation between MSC specifications and their implementations should be studied using one single formalism. Therefore we have proposed a framework based on partial orders, which is more abstract than the usual notion of execution traces. In Chapter 3 we have demonstrated its use by constructively deriving a complete classification of realizability criteria for compositional MSC.

The MSC specifications in our case studies were not realizable due to non-local choice, but the problem of how to systematically obtain proper implementations for such MSC specifications has largely been ignored in the literature. In Chapter 4 we have proposed a new approach to constructively address the problem of implementing specifications that contain non-local choice. Our approach introduces a little more behavior, and it turns out to be appropriate for our case studies.

Part II

IEEE 1394.1: FireWire Bridges

Chapter 6

Introduction

In this chapter we introduce Part II of this thesis. This part is based on our work on the protocol standard IEEE 1394.1, which we describe in Section 6.1. Afterwards, in Section 6.2 we present some basic concepts that will be used in the chapters that follow.

6.1 Introduction to the protocol standard

The development of the IEEE 1394.1 Standard for High Performance Serial Bus Bridges [IEE05] had been initiated in 1996, and it was finished by the end of 2004. It is developed on top of the IEEE 1394 Standard for a High Performance Serial Bus [IEE96]. The latter standard is also known as the FireWire standard, and it has been studied extensively. In this section we introduce these two standards and describe the issues we have worked on. More details can be found in [Rom99, vLRG03].

6.1.1 IEEE 1394: underlying standard

The IEEE 1394 standard enables efficient communication between connected devices that are physically close to each other. To this end the IEEE 1394 standard defines a bus as a non-empty collection of connected IEEE 1394 devices, but the maximum number of devices per bus is limited. Each device is a computational unit that has a unique identity and that belongs to exactly one bus. A total order on these device identities is given.

This standard defines how the devices on a single bus can communicate with each other in terms of communication between connected devices, see Figure 6.1. The

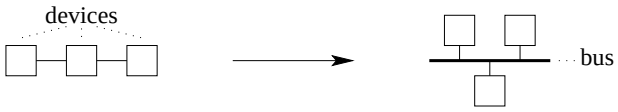


Figure 6.1 IEEE 1394 bus

available types of message communication include unreliable broadcast and reliable point-to-point communication.

The buses are dynamic in the sense that topology changes can occur, primarily by connecting and disconnecting wires. These topology changes can be decomposed into the following elementary topology changes:

- creating a new device: a corresponding singleton bus is created;
- connecting two devices: the corresponding buses are merged;
- disconnecting two devices: the corresponding bus may be split into two buses;
- deleting a device on a singleton bus: the corresponding bus is deleted.

Upon a topology change, all devices on the related buses are notified that the topology has changed. This notification does not include any information about the specific topology changes that have occurred. After a topology change, the devices on the bus execute a protocol to determine the new topology of the bus.

6.1.2 IEEE 1394.1: intended extension

To lift the limitation on the number of devices that can communicate with each other, and to enable efficient communication between connected devices that are not physically close to each other, the IEEE 1394 standard was extended. This extension must be transparent to the ordinary IEEE 1394 devices, in the sense that these devices must be able to continue functioning on their bus without any modification.

To this end, the IEEE 1394.1 standard introduces bi-directional bridges to interconnect pairs of buses, in a similar way as the IEEE 802.1 MAC Bridge Standard. Such an IEEE 1394.1 bridge consists of two portals, which are special devices on the buses that are connected by the bridge, see Figure 6.2. Apart from the usual communication capabilities on the buses, the two portals of each bridge can also directly communicate with each other.

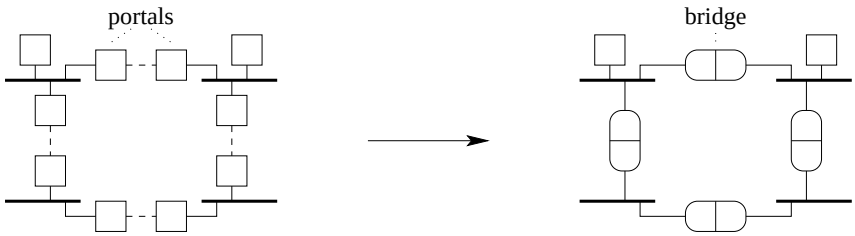


Figure 6.2 IEEE 1394.1 bridges

6.1.3 Abstractions

To conveniently study the latter standard, we need some abstractions with respect to views like Figure 6.3(a). Since the IEEE 1394.1 extension must be transparent to the ordinary IEEE 1394 devices, we abstract from the ordinary devices and assume that each portal can communicate with all portals at its bus. Similarly, we do not consider buses without any bridge portal. Moreover, we abstract from the details of the connection between the two portals of every single bridge.

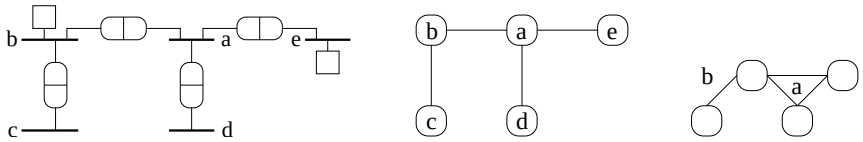
To get closer to the realm of normal graph theory, we want to abstract from the portals. Thus we can consider a network with buses as nodes and bridges as edges as depicted in Figure 6.3(b). Since the portals are the computational units, the nodes and the edges become (parallel) computational units. Each edge gets a unique identity, it can maintain persistent (with respect to topology changes) data, and it can communicate with the nodes it connects; and each node can communicate with its incident edges.

This is our main abstraction, although it is unconventional in the sense that data can only be persistently stored in the edges. In Chapter 7 we use the different abstraction that is also used in [Per85], viz. the line graph (or interchange graph) of the previous abstract network, as depicted in Figure 6.3(c). In this abstraction each bridge is represented by a node, and each bus is represented by a (possibly empty) series of edges.

Both abstractions yield a dynamic network in which the nodes can locally detect whether additions and removals of any connected edge have occurred. This can be used to detect topology changes efficiently, and it avoids the need for a self-stabilizing algorithm [Dij74, Gär03] with its inherent performance drawbacks.

6.1.4 Net update: maintaining a spanning tree

The IEEE 1394.1 standard enables the communication of messages between devices on different buses by assigning to each bus a unique identity and by constructing



(a) Explicit portals (b) Network (c) Line graph

Figure 6.3 Graph abstractions

routing tables. This functionality is based on a distributed algorithm that is called net update, which has to maintain a rooted directed spanning tree on the network of buses and bridges.

A spanning tree of a connected undirected graph is a connected acyclic subgraph that contains all the nodes. In addition, the edges of the spanning tree must be directed towards one single node, which is the root of the tree. Phrased differently, a rooted directed spanning tree of an undirected connected network is a directed subgraph containing all nodes such that

- each node has at most one outgoing edge in the subgraph;
- the subgraph is acyclic;
- the subgraph is connected.

The spanning tree must be maintained under topology changes, which requires a kind of plug-and-play or self-configuration property. As the topology changes may disturb the tree, the requirement is that if (during a sufficiently large period of time) no more topology changes occur, eventually a spanning tree is computed. In comparison with the IEEE 802.1 standard, the IEEE 1394.1 standard should benefit from the IEEE 1394 property that all devices on a bus are notified whenever a device (e.g. a bridge portal) is added to or removed from the bus.

To illustrate the potential of the topology changes on maintaining a spanning tree, we briefly illustrate them using the four example networks in Figure 6.4. These networks, which can correspond to both abstractions discussed before, contain the four nodes a, b, c and d. The (undirected) edges in the networks are represented by lines (including the arrows) between the nodes, and the (directed) edges in the spanning tree are represented by arrows between the nodes. The transitions between the networks denote some topology changes and a way to maintain the spanning tree.

When an edge is added to the network, it might be necessary to extend the tree; if this is necessary it suffices to extend the tree with that edge and possibly turn around some other edges (see cases 1 and 2). Removal of an edge from the network

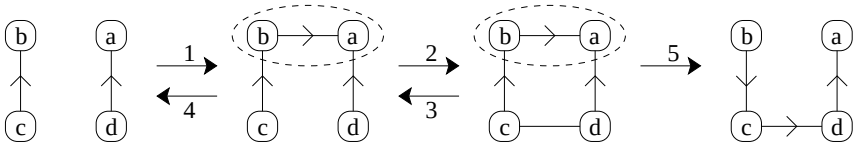


Figure 6.4 Example topology changes

that is no part of the tree does not involve a change in the tree (see case 3). Removal of an edge that is a part of the tree involves removing that edge from the tree, therewith splitting a tree into two trees (see case 4). However, if these two trees are in the same component of the network, the tree must also be extended and possibly modified (see case 5). Note that nodes a and b cannot locally detect whether the removal of the edge between nodes a and b is a case 4 or a case 5 removal (see the dashed lines). In general this also holds for the addition of an edge (case 1 or 2).

6.1.5 Some related spanning tree work

The *computation* of minimum spanning trees has been extensively studied. A *minimum* spanning tree of a graph with weighted edges is a spanning tree with minimum weight. According to [BG92], a distributed algorithm for that purpose exists for graphs in which the minimum spanning tree is uniquely determined, and this is the case if all edge weights are distinct. Furthermore, [GHS83] states that for graphs with neither distinct edge weights nor distinct node identities, no such a distributed algorithm exists that uses a bounded number of messages.

Many algorithms for minimum spanning trees are based on one of the two classical algorithms for computing spanning trees: Prim-Dijkstra's algorithm and Kruskal's algorithm. In Prim-Dijkstra's algorithm a single tree is formed. Initially a root node is chosen; then the algorithm successively adds an edge with minimal weight that extends the tree with a node that is not yet in the tree.

Kruskal's algorithm builds fragments of the spanning tree which are themselves spanning trees. Initially each node is a singleton fragment; then the algorithm successively adds an edge with minimal weight that combines two different fragments. Prim-Dijkstra's algorithm has a more centralized nature than Kruskal's algorithm. Because in practice distributivity is a desired property, Kruskal's algorithm is usually preferred over Prim-Dijkstra's algorithm, and many distributed algorithms have been derived from it, e.g. [GHS83].

In contrast to the computation of spanning trees, we need to address distributed *maintenance* of spanning trees under topology changes. In case the topology changes are restricted to additions only, most distributed computing algorithms

can be used thanks to their incremental nature. In [AAG87] a general algorithm structure is proposed that extends any algorithm on static networks to dynamic networks, by initiating upon each topology change another algorithm that resets the static algorithm to an initial state.

In contrast to minimum spanning trees, we only need *arbitrary* spanning trees. The most-used distributed algorithm for maintenance of such spanning trees is part of the IEEE 802.1 standard, which is described in [Per85, Per00]. It uses the line graph abstraction, but also a procedure to convert the spanning tree into a spanning tree of the original network is described. To compute (and maintain) the spanning tree, every few seconds each bridge broadcasts its unique identity. The bridge with the minimal identity is elected as the root of the tree; then the tree is constructed by including for each bridge a shortest path to the root. To make the computed spanning tree unique, bridge identities are used to break ties in case there are multiple shortest paths to the root. Even after the spanning tree has been computed, the algorithm continues to periodically send messages in order to detect topology changes and to maintain the spanning tree. This is not a desired property for IEEE 1394.1, and we will further address this issue in Chapter 7.

6.1.6 Our contributions

Our work on this standard was prompted by the problems of the standardization committee in getting net update correct. In particular the possibility that buses are split, has led to many consecutive net update proposals that only differ in some of the details. The net update proposals describe the distributed spanning tree algorithm in terms of the portals, typically including some detailed fragments of C code. To get a grip on this (kind of) algorithm, we apply various abstractions to the underlying network. In Section 6.1.3, we have already described the basic abstractions, but we will also abstract from various low-level algorithms that are available (or can be implemented) within one bus or bridge.

The easiest and most obvious solution to their problems would be to use a known spanning tree algorithm. However, such algorithms would not exploit the topology change detection mechanisms from the IEEE 1394 standard as required. We have explored two other directions: First, in Chapter 7, we present a new distributed algorithm for computing and maintaining an arbitrary spanning tree in a dynamic network in which each node has a unique identity. This algorithm is based on IEEE 802.1, but it has not been adopted by the standardization committee, because they did not want to make rigorous changes anymore. This is also the reason that our efforts on this algorithm are limited.

The second approach is to try to fix the net update proposals. Attempts to analyze and to prove the correctness of industrial algorithms are frequently based on formal verification techniques like model checking, which require limited human input. In [vLRG03] a model checker is used to verify some of the net update proposals. A

well-known problem in model checking is the so-called explosion of the state space of the system, which mainly occurs in systems with many interacting processes, and systems containing data. The algorithm we want to analyze fits both profiles, which explains the limited results of [vLRG03]: some errors have been detected, but a full verification of the proposed fixes is still infeasible, see also [Vor04, Huo05].

A typical problem of verifying a given algorithm against a specification, is that the information about how the algorithm was intended to fulfill the specification is not available. Instead of such an a-posteriori verification, in Chapter 8, we formally reconstruct a version of net update from its specification. Such a formal derivation shows how the requirements influence the algorithm under development; and as an important side-effect it provides a correctness proof. In Chapter 9, we develop some tool-support for the programming method that we have used. Finally in Chapter 10 we conclude Part II of this thesis.

6.2 Preliminaries

In this section we summarize the main formal techniques [FvG99] that we use.

6.2.1 Processes, actions and assertions

A parallel program consists of a (possibly dynamic) collection of components. Execution of a single component results in a process consisting of a sequence of atomic actions; execution of a parallel program results in a fair interleaving of these processes. Thus an atomic action is an action that is guaranteed to be executed without interference of any other action. A control point (or an interleaving point) in a component is a location between two consecutive atomic actions of the component. A component is said to be "at a control point" if execution of the component so far ended at the control point.

An annotated program is a program that is annotated with assertions. An assertion is a predicate on the state of the system and it is located at a control point. An assertion at a component's control point is correct if the state of the system satisfies the assertion whenever the component is at the control point. A correctly-annotated program (or a proof outline) is a program in which all assertions are correct. Usually an assertion P is denoted as $\{P\}$, and an atomic statement S with pre-assertion P is denoted as $\{P\}\ S$.

6.2.2 Programming language

The components are described using the following language constructions, based on the Guarded Command Language (GCL):

- **skip** : the *empty statement*;
- $x, y := E, F$: a *multiple assignment*, which first evaluates expressions E and F, and then assigns their values to variables x and y respectively;
- x: $P.x$: a *non-deterministic assignment*, which non-deterministically assigns variable x a value X satisfying $P.X$;
- $S; T$: *sequential composition*, which first executes statement S and then executes statement T;
- **if** $B_0 \rightarrow S_0$ [] $B_1 \rightarrow S_1$ **fi** : *alternative construct*, which, once one of the guards B_0 or B_1 holds, executes one of the statements S_0 or S_1 for which the corresponding guard holds;
- **do** $B \rightarrow S$ **od** : *repetitive construct*, which repeatedly evaluates guard B and executes statement S, until the guard evaluates to *false*;
- **par** x: $P.x \rightarrow S.x$ **rap** : *parallel composition*, which executes statement $S.x$ for all x: $P.x$ in parallel (where P must be stable during the execution).

The usual atomic statements are the skip statement, assignment statements and the evaluations of guards. Larger atomic statements can be denoted by placing a series of statements within atomicity brackets $\langle ... \rangle$. In a final implementation, all atomic statements must be implementable. We will use **await** (B) as an abbreviation of **if** $B \rightarrow$ **skip fi**.

6.2.3 Hoare triples and the theory of Owicki/Gries

A basic notion for the correctness of assertions is a Hoare triple [Hoa69]. A Hoare triple $\{P\}\ S\ \{Q\}$ is a boolean that has the value *true* if and only if each terminating execution of statement S that starts from a state satisfying predicate P is guaranteed to end up in a final state satisfying predicate Q. This definition expresses *partial correctness*, since termination is not considered.

Hoare triples for atomic statements are usually defined using weakest liberal preconditions. The weakest liberal precondition (*wlp* for short) of a statement S is a predicate transformer, to be denoted by $wlp.S$. The $wlp.S$ of a predicate Q, to be denoted by $wlp.S.Q$, is the weakest precondition P such that $\{P\}\ S\ \{Q\}$ is a correct Hoare triple. More formally $\{P\}\ S\ \{Q\}\ \equiv\ [\ P \Rightarrow wlp.S.Q\ ]$, in which the square brackets are a shorthand for "for all states", i.e. a universal quantifier binding all free variables. The following two properties of Hoare triples $\{P\}\ S\ \{Q\}$ are particularly important: they are anti-monotonic in P, and universally conjunctive in Q:

$$[\ X \Rightarrow Y\ ] \wedge \{Y\}\ S\ \{Q\}\ \Rightarrow\ \{X\}\ S\ \{Q\}$$

$$\{P\}\ S\ \{(\forall x :: Q.x)\}\ \equiv\ (\forall x ::\ \{P\}\ S\ \{Q.x\})$$

Composite statements are usually flattened into atomic actions using small theorems. E.g., a selection statement $\{P\}$ **if** $B \rightarrow \{Q\}\ S$ **fi** $\{R\}$ with inner assertion Q is flattened into an atomic evaluation of guard B (with proof obligation $[\ P \wedge B\ \Rightarrow\ Q\ ]$) and a statement $\{Q\}\ S\ \{R\}$.

Partial correctness

For the partial correctness of an annotation we use the Owicki/Gries theory [OG76] in the terminology of [FvG99]. It states that an assertion P in a component is correct whenever the following two conditions hold:

- local correctness is guaranteed, i.e. if P is an initial assertion then P is implied by the precondition of the program, and if P is preceded by atomic action $\{Q\}\ S$ then P is established by that action, i.e. $\{Q\}\ S\ \{P\}$ is a correct Hoare triple; and
- global correctness (or maintenance, or interference freedom) under each atomic action $\{Q\}\ S$ in the other components is guaranteed, i.e. $\{P \wedge Q\}\ S\ \{P\}$ is a correct Hoare triple.

Invariants

We consider three kinds of invariants. A *repetition invariant* is an assertion that is placed at the control point of a repetition and at the last control point of each body of the repetition. For local correctness it must be established by the statement preceding the repetition, and it must be (re-)established by each body of the repetition. Global correctness only needs to be proved once.

An *invariant of a parallel composition* is an assertion that is placed at the control point of the parallel composition and at all control points within. It is correct if it is locally correct at the control point of the parallel composition, maintained under each atomic statement within the parallel composition, and globally correct under each statement outside the parallel composition. A special case is a *system invariant*, which is an assertion that is placed at each control point. It is correct if it is implied by the precondition of the program, and it is maintained by each atomic statement.

6.2.4 Method of Feijen/van Gasteren

Before a program's correctness can be proved using the theory of Owicki/Gries, a full annotation must have been invented. To start verifying an annotation before

the full annotated program has been developed, rely-guarantee methods (see e.g. [XdRH97]) have been proposed. These methods allow to verify each single component based on a rely-guarantee abstraction of the other components. However, during program development it is likely that such abstractions are not available. Others focus on constructing the annotation (together with the program) and on verifying parts of it as soon as possible. In what follows we describe the programming method of Feijen/van Gasteren [FvG99].

The method of Feijen/van Gasteren addresses the construction of parallel programs hand-in-hand with a suitable annotation and correctness proof. Being based on the style of [Dij76], assertions play an important role. We first summarize some conventions. Multiple assertions can be placed at a control point. Such a sequence of assertions denotes their conjunction, and the assertions are called *co-assertions*. Since Hoare triples $\{P\}\ S\ \{Q\}$ are conjunctive in Q, the correctness of individual co-assertions can be proved independently. A *queried assertion* is an assertion which correctness has not yet been proved. Usually a queried assertion Q is denoted as $\{?Q\}$.

Program development starts by expressing the specification of the program in terms of a preliminary program, called the computation proper, and some queried assertions, like post-conditions. Then, one-by-one, all queried assertions must become correct assertions (as described below). The proof obligations for local correctness give rise to a typical style in which programs are constructed from the required assertions towards the initial control point. When all assertions (including those related to the original specification) are correct assertions, the developed program is correct with respect to the specification.

If a queried assertion's correctness (in the current annotated program) cannot yet be proved, there are mainly two solutions (which can also be combined):

- introduce additional queried assertions in the current annotation;
- modify the program.

An important issue is whether these two steps can endanger correctness of the prior assertions. Since Hoare triples $\{P\}\ S\ \{Q\}$ are anti-monotonic in P, introducing additional assertions cannot endanger the correctness of the prior assertions. However, modifying the program may turn all correct assertions into queried assertions again. The typically-used modifications of the program are inserting statements (for local correctness) and changing atomic actions.

In the common case that the changes of the atomic actions can only reduce the program's behavior, correctness of the annotation is maintained. This leads to a style in which non-determinism is used as a way to postpone design decisions. For example, as long as the execution order of a series of statements does not matter, they can be executed in parallel.

6.2.5 Safety, termination, and deadlock freedom

Although progress is not formally addressed by [OG76] and [FvG99], it needs to be considered to obtain a suitable program. In this thesis we apply the usual ad-hoc approach to progress, although recently in [DM06] we have developed a more rigorous approach to address it.

We typically start by developing an initial program that is partially correct, i.e. "no bad thing can happen". After this safety property, we deal with progress, which consists of termination and deadlock freedom. To maintain safety while modifying the program for progress, we will only restrict the possible behavior. Typically the possible behavior of the program is reduced by reducing the amount of non-determinism, i.e. by strengthening the guards of an alternative composition.

We first address termination (typically using variant or bound functions), because in contrast to deadlock-freedom, it is also maintained by restricting the possible behavior afterwards. The last thing we address is deadlock-freedom, again by restricting the behavior, i.e. by avoiding that a deadlock state will be reached. The only statements that can cause a deadlock are the blocking statements, viz. selection statements and non-deterministic assignments. A selection statement is non-blocking whenever its pre-assertion guarantees that (at least) one of its guards holds. Similarly, a non-deterministic assignment is non-blocking whenever its pre-assertion guarantees that there exists a suitable value to be assigned.

6.2.6 Lemmas

In this section we provide two little useful lemmas that we have developed.

Proof reduction for parallel compositions

Consider a parallel program with a component in which the following annotated program fragment occurs:

$$
\begin{array}{|l}
\hline
\{A\} \\
\textbf{par}\ x : P.x\ \rightarrow \\
\qquad \{B.x\} \\
\qquad \ldots \\
\qquad \{C.x\} \\
\textbf{rap} \\
\{D\} \\
\hline
\end{array}
$$

- If assertion $B.x$ (for all $x : P.x$) is globally correct, and assertion A is such that $[\ A \equiv (\forall x : P.x : B.x)\ ]$, then assertion $B.x$ (for all $x : P.x$) is locally correct (by definition). Moreover assertion A is globally correct as each

statement that would violate A, would also violate $B.x$ (for some $x : P.x$), but the latter is impossible as $B.x$ is globally correct.

- If assertions A and $C.x$ (for all $x : P.x$) are globally correct, and assertion D is locally correct, then assertion D is globally correct. Nevertheless, for the global correctness *proof* of assertion D, it may be necessary to strengthen assertion D using A and $C.x$ (for all $x : P.x$).

Program transformation

Consider a parallel program that contains variables x, y and A, and in which all annotation has been removed. We assume that variables x and y are local variables that can only be accessed by the single component in which the following program fragment occurs:

$$\boxed{\begin{array}{l} x := A \ ; \\ y := A \end{array}}$$

During some executions of this program fragment, no statements are executed in between these two assignments. Hence correctness of the program is maintained by executing these two assignments atomically. Then the assignment to y is equivalent to the assignment $y := x$.

$$\boxed{\begin{array}{l} \langle \ x := A \ ; \\ \quad y := x \quad \rangle \end{array}}$$

As variables x and y are not accessible by other components, no component can distinguish between this statement and the following program fragment in which the atomicity brackets have been omitted:

$$\boxed{\begin{array}{l} x := A \ ; \\ y := x \end{array}}$$

Chapter 7

A spanning tree algorithm for dynamic networks

In this chapter that is based on [MGW04], we present a distributed algorithm for maintaining an arbitrary spanning tree in a dynamic network. The goal is to make the algorithm in the IEEE 802.1 standard applicable to dynamic networks. Our algorithm can serve as a simpler alternative for the spanning tree algorithm in the IEEE 1394.1 standard, but it can also be used in more general dynamic networks.

We use the line graph abstraction from Section 6.1.3 and assume the nodes to be computational units that can store data and that have a unique identity. We assume that a total order on these node identities is given, and that the edges can be used as buffered communication channels. Upon adding or removing an edge, the corresponding communication channel is reset. For simplicity of the nomenclature, we restrict the topology changes to additions and removals of edges such that between any two nodes there is at most one edge, and such that there are no self-loops.

Overview In Section 7.1 we sketch the algorithm in an informal way. A formal treatment can be found in Section 7.2. Section 7.3 gives the conclusions.

7.1 Algorithm

Before describing our algorithm, we first refine the specification from Section 6.1.4. The basis of our spanning tree algorithm will be an algorithm that deals with additions of edges only, which are usually the simplest kind of topology changes. Removals of edges will be addressed by superimposing another algorithm, inspired

by [AAG87]. Finally, we give an example, and discuss some properties of the algorithm. As a convention we use variables v and w for nodes.

7.1.1 Refined specification

The algorithm must compute a spanning tree whenever (during a sufficiently large period of time) no more topology changes occur. As the algorithm must be fully-distributed, we refine the requirements of a spanning tree from Section 6.1.4 into local requirements that only refer to nodes and their direct neighbors in the network. For that purpose additional variables need to be introduced in the nodes. Like [AG94], we introduce three local variables per node v with the following requirements:

- $parent.v$ of type node identity. It encodes the tree and it ensures that each node has at most one parent. It indicates the neighbor (if any) of the node that is its direct parent in the tree. In case a node has no parent, i.e. it is a root, then $parent.v = v$.
- $dist.v$ of type natural. It ensures acyclicity of the tree by requiring that $dist.(parent.v) < dist.v$ if v is not a root, thus exploiting that order $<$ on the naturals is irreflexive and transitive. Intuitively, $dist.v$ can be interpreted as an upperbound on the distance from node v to the root of the tree.
- $root.v$ of type node identity. It ensures that the tree is spanning, i.e. any two neighbor nodes belong to the same tree. For that purpose a unique identity is assigned to each tree. Using that nodes have unique identities, we identify trees by the identity of their sole root. So we require $root.v = v$ if v is a root, and $root.v = root.w$ for each neighbor w of node v.

Notice that the spanning tree requirements from Section 6.1.4 are fulfilled if these requirements are established.

In [AG94] it is assumed that each node can atomically perform an operation on its own variables and inspect the variables of a neighbor node. This provides a convenient way of abstracting from the message communication, and we will use a related abstraction in Chapter 8. For the algorithm in the current chapter, the use of this abstraction yields an implementation with a bad performance, and hence we will directly consider the way in which neighbor nodes communicate their $(root, dist)$ value to each other, viz. using messages.

7.1.2 Additions of edges

Assuming that initially the network contains no edges, we initialize each node v such that $parent.v = v \wedge root.v = v \wedge dist.v = 0$, which establishes the requirements

above. The remaining algorithm for each node v deals with additions of edges:

```
do true →
    if an edge between v and w has been added →
        send a message (root.v, dist.v) to neighbor w
    [] a message from a neighbor w has arrived →
        receive a message (r, d) from neighbor w ;
        if (r, d + 1) < (root.v, dist.v) →
            parent.v, root.v, dist.v := w, r, d + 1 ;
            send a message (root.v, dist.v) to all neighbors except w
        [] (root.v, dist.v) ≤ (r, d + 1) → skip
        fi
    fi
od
```

When the node detects that an incident edge has been added, it sends a message over the edge. Upon receipt of a message (r, d) from node w, node v can assign to $(parent.v, root.v, dist.v)$ the value $(w, r, d + 1)$ such that node w becomes its parent. To guarantee stabilization, this assignment is only performed if the node's $(root, dist)$ value decreases by this assignment, i.e. its *root* becomes smaller, or its *root* remains equal and its *dist* becomes smaller. Thus the neighbor with the smallest known $(root, dist)$ is chosen as parent. If the node performs the assignment, it informs all neighbors by sending a message to them.

The $(root, dist)$ value of a node never increases, and handling an arrived message results in a decrease of this value or a decrease in the number of messages in the system. Hence this algorithm stabilizes if (during a sufficiently large period of time) no more edges are added.

7.1.3 Removals of edges

To deal with removals of edges, we will use a variant of the reset technique from [AAG87]. Apart from the start criterion and the part of the network that needs to be reset, we refer to [AAG87] for the details of the reset procedure. In [AAG87], a reset is initiated upon each topology change, and it is propagated only over the edges that have been used for communication.

Since our algorithm can easily deal with additions of edges, we only use the reset technique in case an edge is removed. We consider the main guarded commands above as atomic statements, and when a node detects that an edge has been removed, it initiates a reset. Thus we use the technique in [AAG87] as a way to abstract from all topology changes apart from the ones we can easily deal with.

Exploiting that the messages only contain information about neighbor nodes, the refined specification suggests that a node does not need to initiate a reset if the edge did not lead to its parent. If a node detects that an edge leading to its parent

has been removed, and there is another known (by storing per edge the message with the smallest value) edge that leads to a node with a smaller (*root*, *dist*)-value, that node can become its parent, and no reset is necessary. Otherwise, the node must reset itself and its incident edges. If a node resets itself, the children of the node must also be reset. So the reset only needs to be propagated over the child relation (the inverse of the parent relation).

7.1.4 Example

To get an operational idea of how this algorithm can behave, Figure 7.1 describes some possible behavior of the algorithm. It is based on the topology changes that were described in Figure 6.4. We define the following strict total order on the node-identities: $a < b < c < d$. The three-tuples represent (*parent*, *root*, *dist*) values, and the messages are labeled with their (r, d) content. For the topology changes the following kind of abbreviations are used: "add $a \sim d$" for the "addition of an edge between nodes a and d", and "rem $a \sim d$" for the "removal of an edge between nodes a and d". The details of the reset are hidden in dashed boxes.

7.1.5 Discussion

Our algorithm stabilizes if no more topology changes occur. Once all resets have terminated, the node with the minimal identity stabilizes. Stabilized nodes will not change the value of their variables nor send any message. When all messages sent by this node have been received by its direct neighbors, these neighbors will also stabilize; and so on. Thus the algorithm stabilizes in a time proportional to the diameter of the network, like in [Per85].

Thus if during a sufficiently large period of time no more topology changes occur, no messages are being communicated. In contrast, the self-stabilizing algorithm in [Per85] needs regular message communication for the *detection* of topology changes. If only a relatively small number of topology changes occur, our algorithm uses fewer messages than [Per85]. Some initial messages of the algorithm for additions of edges can even be integrated with the reset protocol of [AAG87]. Moreover, the size of our messages is smaller than the ones used in [Per85].

If a node is not involved in a reset, it can possibly do some higher-level functionality (e.g. forwarding data) using the currently existing tree. The waiting time, i.e. the period during which it cannot do so, is the duration of the reset. In our algorithm this is *dynamically* determined for each node that is involved, since we use the reset from [AAG87]. In [Per85] and [AG94], the waiting time depends on the (theoretical) maximum number of nodes in the network. Such a *static* waiting time usually involves a larger overhead than a dynamic one.

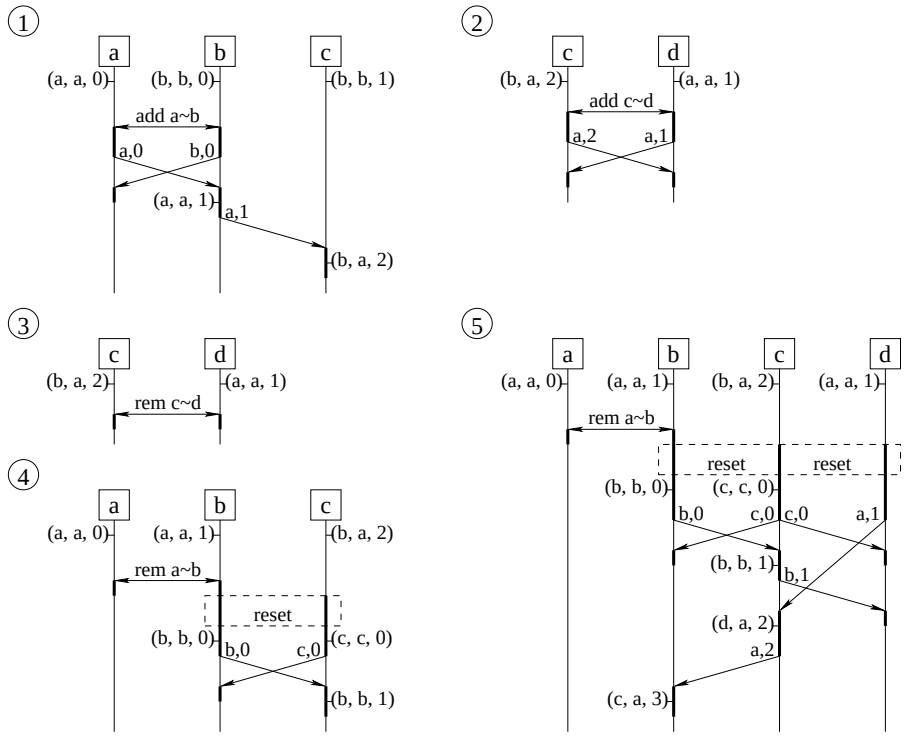


Figure 7.1 Example behavior

7.2 Proof

In this section we show how the correctness of the algorithm can be proved. We will first provide invariants for the algorithm for additions of edges only. Afterwards we will use these invariants to discuss the various conditions related to removals of edges.

7.2.1 Refined specification

To formalize the refined specification, we introduce a binary irreflexive symmetric relation $\sim$, such that $v \sim w$ denotes that v and w are neighbors in the network. Then the refined specification can be formalized as (the conjunction of):

0: $(\forall v : v \neq \mathit{parent}.v : \ v \sim \mathit{parent}.v)$

1: $(\forall v : v \neq \mathit{parent}.v : \ \mathit{dist}.(\mathit{parent}.v) < \mathit{dist}.v)$

2: $(\forall v : \mathit{parent}.v = v : \ \mathit{root}.v = v)$

3: $(\forall v, w : v \sim w : \ \mathit{root}.v \leq \mathit{root}.w)$

In the last requirement one might expect the term $\mathit{root}.v = \mathit{root}.w$ instead of the term $\mathit{root}.v \leq \mathit{root}.w$. Thanks to the symmetry of $\sim$, these are equivalent, but our choice is formally weaker and simplifies the rest of this treatment.

Before showing that the algorithm stabilizes if all edge additions have been detected and all messages have been received, we show that the algorithm is partially correct, i.e. it establishes the requirements upon stabilization.

7.2.2 Partial correctness

We first weaken the requirements into a set of system invariants such that upon stabilization the requirements are implied. For that purpose we introduce some formalizations for topology changes and communication. We introduce a boolean variable $S.v.w$ to denote that an edge $v \sim w$ has been added, but node v has not yet detected this. We use $C_{w \to v}$ to denote the bag of messages communicated by node w to node v. Furthermore we use $C \ominus m$ to denote a bag C from which an element m has been removed, and $|C|$ to denote the number of elements in bag C.

Requirements 0 and 2 do not need to be weakened as they contain variables of one node only. Requirement 3 needs to be established by message communications and therefore we weaken it with a disjunct that expresses that a relevant message is communicated to node v (see below). To keep requirement 1 simple, we just rewrite it by exploiting requirement 0, and requirement 3 for the two pairs $(\mathit{parent}.v, v)$ and $(v, \mathit{parent}.v)$.

In turn, the weakening of requirement 3 can be endangered by addition of an edge. Since node w can easily restore it using message communication, we weaken requirement 3 also with a disjunct $S.w.v$. Thus we obtain the following system invariants:

I_0: $(\forall v : v \neq \mathit{parent}.v : \ v \sim \mathit{parent}.v)$

I_1: $(\forall v : v \neq \mathit{parent}.v : \ (\mathit{root}.(\mathit{parent}.v), \mathit{dist}.(\mathit{parent}.v)) < (\mathit{root}.v, \mathit{dist}.v))$

I_2: $(\forall v : \mathit{parent}.v = v : \ \mathit{root}.v = v)$

I_3: $(\forall v, w : v \sim w :$

$$\mathit{root}.v \leq \mathit{root}.w \ \vee \ (\exists r', d' : (r', d') \in C_{w \to v} : \ r' \leq \mathit{root}.w) \ \vee \ S.w.v)$$

Note that these invariants imply the original requirements when there are no more messages in the system, and all edge additions have been detected. To simplify

the proof that our algorithm maintains these invariants, we annotate it with intermediate assertions. To avoid extra auxiliary variables, we split the statement "receive a message" from a communication channel into "read the message" and "remove the message" from the communication channel. By reading a message, a node can obtain the contents of the message, and by removing the message it is removed from the communication channel.

do *true* →
  ⟨ **if** $S.v.w$ →
    send a message $(root.v, dist.v)$ to neighbor w ;
    $S.v.w := false$
  [] a message from a neighbor w has arrived →
    read a message (r, d) from neighbor w ;
    $\{v \sim w\}\{(root.w, dist.w) \leq (r, d)\}$
    $\{root.v \leq root.w \ \vee \ (\exists r', d' : (r', d') \ \in \ C_{w \to v} : \ r' \leq root.w) \ \vee \ S.w.v\}$
    if $(r, d+1) < (root.v, dist.v)$ →
      $\{v \sim w\}\{(root.w, dist.w) < (r, d+1)\}\{(r, d+1) < (root.v, dist.v)\}$
      $\{r \leq root.w \ \vee \ (\exists r', d' : (r', d') \ \in \ C_{w \to v} \ominus (r, d) : \ r' \leq root.w) \ \vee \ S.w.v\}$
      send a message $(r, d+1)$ to all neighbors except w ;
      $parent.v, root.v, dist.v := w, r, d+1$
    [] $(root.v, dist.v) \leq (r, d+1)$ → **skip**
    fi ;
    $\{root.v \leq root.w \ \vee \ (\exists r', d' : (r', d') \ \in \ C_{w \to v} \ominus (r, d) : \ r' \leq root.w) \ \vee \ S.w.v\}$
    remove a message (r, d) from node w
  fi ⟩
od

Additional required invariant:

I_4: $(\forall d, r, v, w : (r, d) \in C_{w \to v} : \ (root.w, dist.w) \leq (r, d))$

Maintenance of invariant I_4 is guaranteed using the descendence of the $(root, dist)$ values. All invariants in this section can be initialized by requiring initially that for each node v we have $(parent.v, root.v, dist.v) = (v, v, 0)$, and by having no edges nor messages.

7.2.3 Stabilization

To prove stabilization we impose a well-founded function on the state of the system such that it is a variant function for the algorithm, i.e. it decreases in each execution of the body of the repetition. We impose as variant function the four-tuple $[(\#v, w :: \ S.v.w), (\sum v :: \ root.v), (\sum v :: \ dist.v), (\sum v, w :: \ |C_{v \to w}|)]$ with the lexicographical order, in which # is used as "the number of"-quantifier. We have used addition of node identities as an abbreviation of concatenation with the lexicographical order. We assume that the *dist*-values are natural numbers.

The first guarded command decreases it via assignment $S.v.w := \mathit{false}$. In the second guarded command, it is decreased by either decreasing $(\mathit{root}.v, \mathit{dist}.v)$ in the first guarded command of the inner selection, or by decreasing $|C_{w \to v}|$ after the second guarded command of the inner selection.

7.2.4 Removals of edges

Finally we discuss the influence of edge removals. Using the invariants, we can derive various conditions that were just mentioned earlier. Removals of edges can only endanger invariant I_0. It can easily be restored if there is a known (from previously received messages) neighbor with a smaller $(\mathit{root}, \mathit{dist})$ than v, namely by using that neighbor as the parent. Otherwise, the invariant can be restored by an assignment $\mathit{parent}.v := v$, which in turn requires an assignment $\mathit{root}.v := v$ for invariant I_2. This assignment requires that the incident edges are reset for invariant I_3 and I_4. For invariant I_1 the current children of the node must also possibly be reset, so the reset must propagate over the child relation.

7.3 Conclusions and further work

We have presented an algorithm for computing and maintaining a spanning tree in a dynamic network. In comparison with the self-stabilizing algorithm in IEEE 802.1, our algorithm exploits properties of dynamic networks like IEEE 1394.1. Thus the algorithm improves on performance, and in the waiting time in which data cannot be forwarded through the network. Furthermore, the algorithm does not require any configuration or tuning of network dependent parameters.

To simplify the nomenclature, we have imposed some restrictions on the network. It is further work to eliminate them, although we do not expect any serious changes of the algorithm. The reason is that multiple edges can be addressed by explicitly referring to edges instead of to pairs of neighbor nodes, and that self-loops are just a local issue.

To conveniently develop this spanning tree algorithm, we have used a variant of the reset technique from [AAG87] to abstract from some topology changes. We have first developed an algorithm that can deal with most of the topology changes, and then we have addressed the remaining topology changes using a reset. Thus we do not need to get into the complications of implementing a reset on a dynamic network. As we will show in Chapter 8, the spanning tree algorithm in IEEE 1394.1 has a similar structure as the spanning tree algorithm in this chapter.

Chapter 8

A formal reconstruction of net update

In this chapter that is based on [MW03], we formally reconstruct a version of the spanning tree algorithm of net update, starting from its specification. In contrast to the alternative algorithm proposed in Chapter 7, in the current chapter we try to stay close to the net update proposals. In the IEEE 1394.1 standard, algorithms are described as low-level implementations for the portals. Attempts to formalize and to analyze algorithms are often based on such an implementation-level description, see e.g. [vLRG03]. However, in order to understand the essence of the algorithm, the portals are not the right entities to start reasoning about.

Since net update is closely related to spanning trees, we prefer to abstract from the portals and from the message communication. Thus we can reason about a graph and about communication via shared variables. From Section 6.1.3 we use the abstraction in which buses become nodes and bridges become edges.

In this graph context we develop a distributed spanning tree algorithm that is strikingly similar to net update. More specifically, we systematically derive a version of net update starting from its requirements. Such a derivation shows how the requirements influence the algorithm under construction; and as a side-effect it provides a correctness proof. The crucial parts have also been checked using the theorem prover PVS, which is discussed in Chapter 9. Our derivation is not intended to address methodological issues and heuristics, but it is a way to get a grip on net update. That is, the high-level design decisions are primarily motivated by the net update proposals.

Finally we use a transformational approach to decompose the implementations of the buses and bridges into implementations of the underlying portals. This is also the place where the message communication is explicitly introduced, which is in a

much later stage than using the approach from Chapter 7. The algorithm we thus obtain is an algorithm that is distributed at two levels.

Overview In Section 8.1 we introduce some notations. Then in Section 8.2 we develop an abstract algorithm, which in turn is implemented in Section 8.3. Finally Sections 8.4 and 8.5 evaluate the results and contain the conclusions.

8.1 Notations

We introduce some nomenclature, notations and abbreviations for the two graphs that we consider, namely an undirected graph for the network, and a directed subgraph of this graph for the spanning tree. As a convention we use variables u, v and w for nodes, and variables e, f, g and h for edges.

An edge (with identity) e between nodes u and v in the undirected graph is denoted by $e : u \sim_e v$, and an edge e from node u to node v in the directed subgraph is denoted by $e : u \rightarrow_e v$. Each edge from the undirected graph occurs in the directed subgraph in at most one direction. An edge of the undirected graph that is not in the directed subgraph is called a muted edge; for an edge $e : u \sim_e v$ we correspondingly define $mut.e \equiv \neg(u \rightarrow_e v \ \vee \ v \rightarrow_e u)$.

Frequently we want to indicate for a node all incident edges or all self-loops in the graph, or all outgoing edges, all incoming edges or all muted edges in the subgraph. As an abbreviation we introduce for each node v the sets $edge.v$ and $loop.v$, and $out.v$, $in.v$ and $mut.v$ respectively. Notice that $mut.e$ denotes whether edge e is muted, and that $mut.v$ denotes the set of muted edges incident to node v.

We introduce the following atomic operations for an edge $e : v \sim_e w$:

$$
\begin{array}{lcl}
\textbf{mute } e : \ v \rightarrow_e w & \equiv & out.v, mut.v \ := \ out.v \backslash \{e\}, mut.v \cup \{e\} \ \| \\
 & & in.w, mut.w \ := \ in.w \backslash \{e\}, mut.w \cup \{e\} \\
\textbf{unmute } e : \ mut.e \text{ as } v \rightarrow_e w & \equiv & mut.v, out.v \ := \ mut.v \backslash \{e\}, out.v \cup \{e\} \ \| \\
 & & mut.w, in.w \ := \ mut.w \backslash \{e\}, in.w \cup \{e\} \\
\textbf{turn } e : \ v \rightarrow_e w & \equiv & \textbf{mute } e \ ; \ \textbf{unmute } e \text{ as } w \rightarrow_e v
\end{array}
$$

8.2 Abstract algorithm

In this section we reconstruct a version of net update for a graph abstraction. We discuss the topology changes in Section 8.2.5, and ignore them until then.

We start by massaging the specification from Section 6.1.4 into a more appropriate shape. Then we develop an initial version of the algorithm that is *partially correct*, i.e. whenever the algorithm stabilizes all requirements are fulfilled. Afterwards

we ensure that the algorithm *stabilizes* by reducing the possible behavior of the algorithm. Finally we prevent that (unwanted) *deadlocks* occur, again by reducing the possible behavior.

While developing the algorithm, we regularly focus on fragments of the algorithm and its annotation, and temporarily omit the rest. For the sake of completeness, Section 8.2.6 contains a fully-annotated version of the whole algorithm. Although we treat the algorithm in a non-operational way, one example behavior has been included in Section 8.2.8.

8.2.1 Refined Specification

The algorithm must compute a spanning tree whenever (during a sufficiently large period of time) no more topology changes occur. As the algorithm must be fully-distributed, we refine the requirements of a spanning tree from Section 6.1.4 into local requirements that only refer to nodes (or edges) and their direct neighbors in the network. For that purpose additional variables need to be introduced in the nodes or edges. As argued in Section 6.1.3 the nodes cannot store persistent data, and hence we only introduce variables in the edges.

To ensure that each node v has at most one outgoing edge, we require

0: $(\forall v :: |out.v| \leq 1)$

To guarantee acyclicity, we associate with each edge e a natural variable[1] $dist.e$. Exploiting irreflexivity and transitivity of order $<$ on the naturals, we require

1: $(\forall v :: (\forall f, g : f \in out.v \ \wedge \ g \in in.v : \ dist.f < dist.g))$

Intuitively, $dist.v$ can be interpreted as an upperbound on the distance from node v to the root of the tree.

For connectivity, we use that two nodes in a tree are connected if and only if they belong to the same tree. So if each node stores the unique identity of the tree it belongs to, we could require that each two neighbor nodes in the graph store the same tree identity. For a unique identity of a tree, we could exploit that the tree is rooted by using the unique identity of its root node. However, this is impossible as nodes cannot store persistent data.

Therefore we distribute these stored tree identities to the edges and require that all incident edges of a node store the same identity (see 2 below). Since root nodes have no unique identity, we use the identity of one[2] of its incoming edges or self-loops unless it has no edges (see 3 below). Note that since a muted edge

[1] Net update jargon: *hops to prime*.
[2] Net update jargon: the *prime* portal of the tree.

is symmetric with respect to the nodes it connects, we cannot use the identity of a muted non-self-loop as a unique identity of a root node. So we associate with each edge e a variable *root.e* of type edge identity, and require:

2: $(\forall v :: (\forall f, g : f \in edge.v \ \wedge \ g \in edge.v : \ root.f = root.g))$

3: $(\forall v : edge.v \neq \emptyset \ \wedge \ out.v = \emptyset : (\exists f : f \in in.v \cup loop.v : \ root.f = f))$

Notice that the spanning tree requirements from Section 6.1.4 are fulfilled if these four requirements are established.

8.2.2 Partial correctness

In this section we define the overall shape of the algorithm by developing an initial, but partially-correct, version of the algorithm. In general, numerous algorithms can be developed for a given specification, and hence many design decisions will be made in this section. To stay close to net update, we include some short high-level descriptions of the way net update is supposed to behave before we derive the corresponding part of the algorithm.

In net update, each node locally tries to establish the requirements related to the node. However, while a node establishes its own requirements, some requirements related to its neighbors may be endangered. Nodes signal[3] their neighbors when some of their requirements may be violated, such that they can re-establish them. Net update does not terminate, but it *stabilizes* when no node is signaled anymore and all nodes have established their requirements.

To formalize the ideas in this description, we explicitly associate with each node v its requirements $Q.v$. For later use, we slightly rephrase requirements 0 and 3 to make them more homogeneous. We define $Q.v$ as the conjunction of

$$\begin{array}{lcl} Q.v.0 & \equiv & (\forall f, g : f \in out.v \ \wedge \ g \in out.v : \ f = g) \\ Q.v.1 & \equiv & (\forall f, g : f \in out.v \ \wedge \ g \in in.v : \ dist.f < dist.g) \\ Q.v.2 & \equiv & (\forall f, g : f \in edge.v \ \wedge \ g \in edge.v : \ root.f = root.g) \\ Q.v.3 & \equiv & edge.v = \emptyset \ \vee \ (\exists f :: \ f \in out.v) \ \vee \ (\exists f : f \in in.v \cup loop.v : \ root.f = f) \end{array}$$

Notice that the original requirements are fulfilled if $Q.v$ holds for all nodes v. For nodes $v : edge.v = \emptyset$, requirement $Q.v$ reduces to *true*. Since such nodes cannot communicate with other nodes or edges, we will not consider them any further.

To formalize the signals, we associate with each combination of a node v and an edge $f : f \in edge.v$ a boolean variable $sig.v_f$ to indicate whether node v has been signaled via edge f. Then we are heading for an algorithm, for each node $v : edge.v \neq \emptyset$, of the following shape:

[3] Net update jargon: a *bus reset* (and hence net update) is initiated on the neighbor bus.

```
do true →
     par f : f ∈ edge.v  →  sig.v_f := false rap ;
     ...
      {? Q.v ∨ (∃f : f ∈ edge.v : sig.v_f)}
     await( (∃f : f ∈ edge.v : sig.v_f) )
od
```

Observe that this is just a partial algorithm in the sense that we still have to fill in the gap "..." in such a way that the queried assertion becomes a correct assertion. Each node v starts to reset variable $sig.v_f$ for all incident edges, and then it executes the gap to establish $Q.v$ unless it gets signaled. Afterwards, once it has been signaled, it starts over again. Upon stabilization, in each node the (currently) queried assertion holds and the negation of the **await**-guard holds. Hence $(\forall v :: Q.v)$ holds, which fulfills the specification.

Note that the queried assertion can be made correct by inserting an assignment $sig.v_f := true$ for some edge $f : f \in edge.v$. However, such an assignment is likely to endanger stabilization. Later on we will deal with stabilization in detail, but for the moment we will already avoid such assignments.

In net update, the (currently) queried assertion is established in two phases, which turns out to be related to the internal structure of condition $Q.v$. Observe that all terms in $Q.v.0$, $Q.v.1$ and $Q.v.2$ are about *pairs* of edges, while $Q.v.3$ can be witnessed by *one* edge. In net update, first a witness edge[4] is elected for $Q.v.3$, and then based on this edge $Q.v.0$, $Q.v.1$ and $Q.v.2$ are established.

The queried assertion in the last program fragment contains many dependencies between variables of different edges, which is not a convenient basis for developing an algorithm in which the data is distributed. Therefore we strengthen this assertion into an assertion $R.v$ in which each term refers to the variables of at most one edge. Like in Section 8.2.1, this requires the introduction of additional variables. Since we are dealing with an intermediate assertion, such variables can also be introduced in the nodes, see also Section 8.2.5.

For the parts related to conditions $Q.v.2$ and $Q.v.0$, we exploit the transitivity and the symmetry of $=$. We strengthen these parts into conjuncts $R.v.0$ and $R.v.1$ of $R.v$ by introducing in each node v fresh local variables r and e of type edge identity:

$$R.v.0 \;\equiv\; (\forall f : f \in edge.v :\; sig.v_f \;\vee\; root.f = r)$$
$$R.v.1 \;\equiv\; (\forall f : f \in out.v :\; sig.v_f \;\vee\; f = e)$$

Because the parts related to condition $Q.v.1$ are asymmetric, we distinguish between outgoing and incoming edges. We strengthen these parts into conjuncts $R.v.2$ and $R.v.3$ by introducing in each node v a fresh local natural variable d:

[4] Net update jargon: an *alpha* portal, which leads towards the prime portal of the tree.

$$R.v.2 \;\equiv\; (\forall f : f \in \mathit{out.v} :\; \mathit{sig.v}_f \;\vee\; \mathit{dist.}f \leq d)$$
$$R.v.3 \;\equiv\; (\forall f : f \in \mathit{in.v} :\; \mathit{sig.v}_f \;\vee\; d < \mathit{dist.}f) \;\vee\; (\forall f : f \in \mathit{out.v} :\; \mathit{sig.v}_f)$$

The second disjunct in $R.v.3$ may be left out, but our version is weaker. Similarly we could also introduce a disjunct $(\forall f : f \in \mathit{in.v} :\; \mathit{sig.v}_f)$ in $R.v.2$, but our asymmetric combination turns out to simplify the rest of our derivation. To really decouple the variables of different edges, we can use $R.v.1$ to rewrite $R.v.2$ and strengthen $R.v.3$ into

$$R.v.2 \;\equiv\; \mathit{sig.v}_e \;\vee\; e \notin \mathit{out.v} \;\vee\; \mathit{dist.}e \leq d$$
$$R.v.3 \;\Leftarrow\; \begin{cases} (\mathit{sig.v}_e \;\vee\; e \notin \mathit{out.v} \;\vee\; e \notin \mathit{in.v}) \;\wedge \\ (\forall f : f \in \mathit{in.v} :\; \mathit{sig.v}_f \;\vee\; d < \mathit{dist.}f \;\vee\; f = e) \end{cases}$$

What remains are the parts related to condition $Q.v.3$. Since we are considering a node $v : \mathit{edge.v} \neq \emptyset$, we can eliminate the first disjunct. For simplicity reasons, we strengthen the remaining parts into conjunct $R.v.4$ by using variable e as a witness:

$$R.v.4 \;\equiv\; \mathit{sig.v}_e \;\vee\; e \in \mathit{out.v} \;\vee\; (e \in \mathit{in.v} \cup \mathit{loop.v} \;\wedge\; \mathit{root.}e = e)$$

Note that $R.v$ implies the queried assertion in node v, and hence it is sufficient to turn $R.v$ into a correct assertion.

We first consider the local correctness of queried assertion $R.v$. Conditions $R.v.3$ and $R.v.4$ suggest that we should consider the conditions on edge e separately. The conditions for the other edges are such that they can be established independently. More specifically, we insert a parallel composition that establishes for each edge $f : f \neq e$ the conjuncts in $R.v$ about edge f, and we require the conjuncts in $R.v$ about edge e as invariants of the parallel composition. Note that in particular these invariants must be established as pre-assertion of the parallel composition.

The global correctness of assertion $R.v$ will follow from the global correctness of the assertions and invariants that we have introduced for its local correctness (see Section 6.2.6). So we are developing a program fragment of the following shape:

```
...
 {inv ? sig.v_e ∨ root.e = r} {inv ? sig.v_e ∨ e ∉ out.v ∨ dist.e ≤ d}
 {inv ? sig.v_e ∨ e ∉ out.v ∨ e ∉ in.v}
 {inv ? sig.v_e ∨ e ∈ out.v ∨ (e ∈ in.v ∪ loop.v ∧ root.e = e)}
par f, u : u ∼_f v ∧ f ≠ e →
     ...
      {? sig.v_f ∨ root.f = r} {? sig.v_f ∨ f ∉ out.v} {? sig.v_f ∨ f ∉ in.v ∨ d < dist.f}
rap
 {R.v}
```

We first deal with the local correctness of the last series of queried assertions. For simplicity reasons, we insert one large atomic statement in the parallel composition. We first discuss some ways to establish the individual queried assertions, and then we combine them into alternatives of an **if**-statement. At this point we can hardly decide which alternatives are necessary, but fortunately we can introduce

as many alternatives as we like, since alternatives can safely be eliminated later on (upon need).

Recall that we do not want to establish these assertions using an assignment to $sig.v_f$; but if $sig.v_f$ already holds, a **skip** is sufficient to fulfill all of them. Alternatively, we can fulfill the first assertion by inserting an assignment $root.f := r$. For the second assertion we can insert a statement $\{f \in out.v\}$ **mute** f or $\{f \in out.v \wedge u \neq v\}$ **turn** f. To establish the last assertion we can insert an assignment $\{f \in in.v\}$ $dist.f := d+1$. Although for this last assertion we could also mute incoming edges, this turns out to only complicate the rest of the derivation. Because it is important that edges can also be unmuted, we also consider the statement $\{f \in mut.v\}$ **unmute** f as $u \rightarrow_f v$.

For the global correctness of the last series of queried assertions, we consider each statement in node v that can possibly endanger these assertions in a node $u : u \neq v$. To prevent such a violation, we accompany these statements with suitable assignments that evaluate to $sig.u_f := true$. For example, for the first assertion each assignment that assigns a value r to $root.f$ is extended with an assignment $sig.u_f := sig.u_f \vee (root.f \neq r \wedge u \neq v)$. Thus we obtain:

```
...
  {inv ? sig.v_e ∨ root.e = r} {inv ? sig.v_e ∨ e ∉ out.v ∨ dist.e ≤ d}
  {inv ? sig.v_e ∨ e ∉ out.v ∨ e ∉ in.v}
  {inv ? sig.v_e ∨ e ∈ out.v ∨ (e ∈ in.v ∪ loop.v ∧ root.e = e)}
par f, u : u ∼_f v ∧ f ≠ e →
   ⟨ if f ∈ out.v →
          root.f, sig.u_f := r, sig.u_f ∨ (root.f ≠ r ∧ u ≠ v) ;
          mute f
     [] f ∈ out.v ∧ u ≠ v →
          root.f, dist.f, sig.u_f := r, d + 1, true ;
          turn f
     [] f ∈ in.v ∧ u ≠ v →
          root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r
     [] f ∈ mut.v ∧ u ≠ v →
          root.f, dist.f, sig.u_f := r, d + 1, true ;
          unmute f as u →_f v
     [] f ∈ mut.v →
          root.f, sig.u_f := r, sig.u_f ∨ (root.f ≠ r ∧ u ≠ v)
     [] sig.v_f →
          skip
     fi ⟩
      {sig.v_f ∨ root.f = r} {sig.v_f ∨ f ∉ out.v} {sig.v_f ∨ f ∉ in.v ∨ d < dist.f}
rap
```

Note that each guarded command establishes the post-assertion within the parallel composition, and that the guards of the selection statement cover all cases. In what follows we will refer to these guarded commands in terms like muting an outgoing edge, turning an outgoing edge, maintaining an incoming edge, unmuting a muted edge and maintaining a muted edge respectively.

We continue with the remaining queried invariants. Their maintenance under the statement in the parallel composition in node v is guaranteed, thanks to condition $f \neq e$. Maintenance of these invariants in a node $u : u \neq v$ under the statements in

node v is guaranteed for the first and the third invariant. For the second invariant, upon maintaining an incoming edge, variable $sig.u_f$ must also become *true* in case $dist.f$ increases, i.e. if $dist.f \leq d$. For the last invariant, upon muting an outgoing edge, variable $sig.u_f$ must also become *true* in case $(root.f = f \ \wedge \ u \neq v)$ holds.

For the local correctness of the first two invariants we insert an assignment to local variables r and d that establishes their last disjunct. For the local correctness of the other two invariants we require them as pre-assertions of this assignment:

```
...
  {? sig.v_e ∨ e ∉ out.v ∨ e ∉ in.v}
  {? sig.v_e ∨ e ∈ out.v ∨ (e ∈ in.v ∪ loop.v ∧ root.e = e)}
r, d := root.e, dist.e ;

  {inv sig.v_e ∨ root.e = r} {inv sig.v_e ∨ e ∉ out.v ∨ dist.e ≤ d}
  {inv sig.v_e ∨ e ∉ out.v ∨ e ∉ in.v}
  {inv sig.v_e ∨ e ∈ out.v ∨ (e ∈ in.v ∪ loop.v ∧ root.e = e)}
par f, u : u ∼_f v ∧ f ≠ e →
   ⟨ if f ∈ out.v →
          root.f, sig.u_f := r, sig.u_f ∨ ((root.f ≠ r ∨ root.f = f) ∧ u ≠ v) ;
          mute f
     [] f ∈ in.v ∧ u ≠ v →
          root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r ∨ dist.f ≤ d
     ...
     fi ⟩
rap
```

The global correctness of these queried assertions follows from the arguments given for maintenance of the invariants. For their local correctness, an assignment to local variable e needs to be introduced. We will establish the two assertions in different ways. For the second assertion, we insert an assignment that selects an edge e that fulfills the assertion. For the first assertion, we generalize the assertion to all edges of the node, and require it as a pre-assertion of this assignment.

```
...
  {? (∀f : f ∈ edge.v : sig.v_f ∨ f ∉ out.v ∨ f ∉ in.v)}
e: e ∈ out.v ∨ (e ∈ in.v ∪ loop.v ∧ root.e = e)
```

Notice that it is not yet guaranteed that such an edge e exists, but we postpone this issue to Section 8.2.4.

The global correctness of the queried assertion is guaranteed by the same argument as before. Since this assertion is a post-assertion of the assignments $sig.v_f := false$, we establish its local correctness by dropping disjunct $sig.v_f$ and requiring the equivalent condition $loop.v \subseteq mut.v$ as a repetition invariant.

This repetition invariant is globally correct as it cannot be endangered by other nodes. Its maintenance under the body of the repetition is also guaranteed, since each atomic statement in node v maintains it. Initialization of this invariant can be established by inserting mute statements before the repetition. After using $loop.v \subseteq mut.v$ to simplify some statements, and eliminating the annotation, we thus obtain:

```
par f : f ∈ loop.v  →
   ⟨ if f ∉ mut.v → mute f   []   f ∈ mut.v → skip fi ⟩
rap ;

do true →
     par f : f ∈ edge.v  →  sig.v_f := false rap ;

     e: e ∈ out.v ∨ (e ∈ in.v ∪ loop.v ∧ root.e = e) ;
     r, d := root.e, dist.e ;

     par f, u : u ∼_f v ∧ f ≠ e  →
        ⟨ if f ∈ out.v →
               root.f, sig.u_f := r, sig.u_f ∨ root.f ≠ r ∨ root.f = f ;
               mute f
          [] f ∈ out.v →
               root.f, dist.f, sig.u_f := r, d + 1, true ;
               turn f
          [] f ∈ in.v →
               root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r ∨ dist.f ≤ d
          [] f ∈ mut.v ∧ u ≠ v →
               root.f, dist.f, sig.u_f := r, d + 1, true ;
               unmute f as u →_f v
          [] f ∈ mut.v →
               root.f, sig.u_f := r, sig.u_f ∨ (root.f ≠ r ∧ u ≠ v)
          [] sig.v_f →
               skip
          fi ⟩
     rap ;

     await( (∃f : f ∈ edge.v : sig.v_f) )
od
```

Thus we have obtained a partially-correct algorithm, for which we have not yet guaranteed that it stabilizes nor that it is deadlock-free.

8.2.3 Stabilization

In this section we modify the algorithm such that it stabilizes. As described in Section 6.2.5, we ensure that partial correctness is maintained by only strengthening the guards. In what follows, we impose a well-founded function on the state of the system, and ensure that its value is descending (i.e. it never increases) and it decreases regularly.

We propose a function that consists of three parts. Upon stabilization we have $\neg sig.v_f$ for each edge $f : f \in edge.v$, so we head for a function that decreases under an assignment $\{sig.v_f\}$ $sig.v_f := false$. Since for a spanning tree we need in fact a minimum number of (non-muted) edges, we want a function that decreases under **mute**-statements. Furthermore, for such an algorithm it turns out to be (at least) convenient to introduce a total order $\leq$ on the edge identities, which we use to choose a function that decreases if $root.f$ for an edge f decreases. Combining these ingredients, we impose as a variant function the three-tuple $[(\sum f :: root.f), (\#f :: \neg mut.f), (\#v, f :: sig.v_f)]$ with the lexicographical order, in which # is used as "the number of"-quantor. We have used addition of edge identities as an abbreviation of concatenation with the lexicographical order. The

variant function is well-founded since each edge has one unique identity, and we assume the network to be finite.

Then we need to ensure that this function decreases regularly. Note that upon passing the **await** statement, guard $(\exists f : f \in edge.v : sig.v_f)$ holds stably up to the parallel composition with assignments $sig.v_f := false$, which then decreases the variant function. So after one execution of the body of the repetition, each further execution yields a decrease of the function.

What remains is to ensure that this function is descending. The only statement that may increase it is the large atomic statement in the parallel composition. For each assignment $root.f := r$ we must require pre-assertion $r \leq root.f$; and for unmuting an edge we must even require pre-assertion $r < root.f$. For each assignment that evaluates to $sig.v_f := true$ we also require $r < root.f$, except if it is combined with a mute statement. For updating an incoming edge this boils down to a combined pre-assertion $(r, d) < (root.f, dist.f)$, which can also be used to simplify the assignment to $sig.u_f$. Since these pre-assertions are part of an atomic region, we directly strengthen the preceding guards of the selection. Thus we obtain:

```
par f : f ∈ loop.v →
   ⟨ if f ∉ mut.v → mute f   []   f ∈ mut.v → skip fi ⟩
rap ;

do true →
     par f : f ∈ edge.v → sig.v_f := false rap ;

     e: e ∈ out.v ∨ (e ∈ in.v ∪ loop.v ∧ root.e = e) ;
     r, d := root.e, dist.e ;

     par f, u : u ∼_f v ∧ f ≠ e →
        ⟨ if f ∈ out.v ∧ r ≤ root.f →
               root.f, sig.u_f := r, sig.u_f ∨ root.f ≠ r ∨ root.f = f ;
               mute f
          [] f ∈ out.v ∧ r < root.f →
               root.f, dist.f, sig.u_f := r, d + 1, true ;
               turn f
          [] f ∈ in.v ∧ (r, d) < (root.f, dist.f) →
               root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r
          [] f ∈ mut.v ∧ r < root.f ∧ u ≠ v →
               root.f, dist.f, sig.u_f := r, d + 1, true ;
               unmute f as u →_f v
          [] f ∈ mut.v ∧ r ≤ root.f →
               root.f, sig.u_f := r, sig.u_f ∨ (root.f ≠ r ∧ u ≠ v)
          [] sig.v_f →
               skip
          fi ⟩
     rap ;

     await( (∃f : f ∈ edge.v : sig.v_f) )
od
```

Thus we have obtained a partially-correct and stabilizing algorithm, which is not yet guaranteed to be deadlock-free. From this version of the algorithm we have for each $f, u, v : u \sim_f v$ the following three important properties:

Descendence	: Both $(root.f, dist.f)$ and $root.f$ are descending in time.
Direction	: Every statement in node v that affects f ensures $f \notin out.v$.
Signalling	: Every change of $root.f$ by node $u : u \neq v$ also ensures $sig.v_f$.

8.2.4 Deadlock freedom

In this section we ensure that there are no unwanted deadlocks. As described in Section 6.2.5, we ensure that partial correctness and stabilization are maintained by only strengthening the guards. Since stabilization is in fact a desired deadlock and it is achieved by the **await**-statements, we will not consider these statements.

In what follows, we ensure that the other statements are non-blocking by requiring some pre-assertions. Dealing with these assertions turns out to be quite technical, because many assertions need to be introduced and because some of the proofs are complicated. The correctness of these proofs has been checked using the theorem prover PVS as described in Section 9.4.

To ensure that the **if**-statement within the parallel composition is non-blocking, we must require the disjunction of its guards as pre-assertion:

```
par f, u : u ∼_f v ∧ f ≠ e →
      {? sig.v_f ∨ r ≤ root.f} {? sig.v_f ∨ f ∉ in.v ∨ (r, d) < (root.f, dist.f)}
   ⟨ if ... fi ⟩
rap
```

Their global correctness is guaranteed by the signalling property and the direction property ($f \notin in.v$) respectively. For their local correctness, we also require them (for all $f : f \in edge.v \wedge f \neq e$) as pre-assertion of the parallel composition.

The global correctness of these pre-assertions of the parallel composition is guaranteed by construction (see Section 6.2.6). Since these pre-assertions are also required post-assertions of assignment $r, d := root.e, dist.e$, their local correctness is guaranteed by requiring the following pre-assertions of this assignment:

```
  {? (∀f : f ≠ e ∧ f ∈ edge.v : sig.v_f ∨ root.e ≤ root.f)}
  {? (∀f : f ≠ e ∧ f ∈ in.v : sig.v_f ∨ (root.e, dist.e) < (root.f, dist.f))}
r, d := root.e, dist.e
```

The global correctness of these assertions is guaranteed by the descendence property of edge e, and the signalling and direction properties of edge f respectively. Since these assertions are required post-assertions of the selection of an edge e, their local correctness can be established by strengthening the criteria of that selection:

```
⟨ e: S.v.e ⟩
```

$$\begin{array}{lll} \text{where } S.v.h \equiv & ((h \in out.v \ \wedge \ h \notin in.v) \ \vee \ (h \in in.v \cup loop.v \ \wedge \ root.h = h)) \\ \wedge & (\forall f : f \neq h \ \wedge \ f \in edge.v : \ root.h \leq root.f) \\ \wedge & (\forall f : f \neq h \ \wedge \ f \in in.v : \ (root.h, dist.h) < (root.f, dist.f)) \end{array}$$

For maintenance of invariant P_0 below we have already strengthened $h \in out.v$ with a conjunct $h \notin in.v$.

To guarantee that the selection of an edge e: $S.v.e$ causes no deadlock, we require as a system invariant that there exists such an edge e:

$$P_0 \ \ (\forall v : edge.v \neq \emptyset : \ (\exists h : h \in edge.v : \ S.v.h))$$

Maintenance of invariant P_0 under the mute statements that establish $loop.v \subseteq mut.v$ is guaranteed using the above strengthening. Thus invariant P_0 can only be endangered by the **if**-statement in the large parallel composition. In what follows, we focus on a node u with an edge $g : S.u.g$ in relation to a statement in node v. We use r and d for variables of node v.

We first consider the statements in nodes $v : v \neq u$ that affect an edge $f : v \sim_f u$ with $g \neq f$. We will ensure that in case $S.u.g$ is violated by such a statement, $S.u.f$ is established, and hence invariant P_0 is maintained. Using the direction property ($f \notin in.u$), $S.u.g$ can only be violated by an assignment $\{r < root.g\}$ $root.f := r$. Using $S.u.g$ this assignment establishes $(\forall h : f \neq h \wedge h \in edge.u : root.f < root.h)$. What remains to establish $S.u.f$ is to ensure that $f \in out.u$ holds. The only possibly-violating statements that do not guarantee this are muting an outgoing edge and maintaining a muted edge. Using $S.u.g$ and condition $r < root.g$ (and hence $r < root.f$), we can exclude them by strengthening their guards with a conjunct $root.f \leq r$ (unless $u = v$).

Then we consider the statements in nodes $v : v \neq u$, that affect an edge $f : v \sim_f u$ with $g = f$. We will ensure that $S.u.g$, or rather $S.u.f$, cannot be violated by such a statement. Since $S.u.f$ implies $f \notin mut.u$, we do not need to consider statements for muted edges. Fortunately the statements for $f \in in.v$ and turning $f \in out.v$ cannot violate $S.u.f$. So what remains is muting an outgoing edge. Since this statement can violate $S.u.f$, we want to strengthen its guard with a conjunct that implies $\neg S.u.f$. Using $f \in out.v$ (i.e. $f \in in.u$), a local way to do this is using a conjunct $root.f \neq f$.

Finally, we consider the statements in node $v : v = u$ that affect an edge $f : f \in edge.v \ \wedge \ f \neq e$. In case $g = e$, $S.v.e$ is maintained if we require pre-assertion $(root.e, dist.e) \leq (r, d)$. In case $g \neq e$, we will ensure that $S.v.g$ is maintained if $\neg S.v.e$ holds, viz. by requiring pre-assertion $(\forall h : S.v.h : \ root.h < r) \ \vee \ S.v.e$.

```
par f, u : u ∼_f v ∧ f ≠ e →
      {? (root.e, dist.e) ≤ (r, d)} {? (∀h : S.v.h : root.h < r) ∨ S.v.e}
      {sig.v_f ∨ r ≤ root.f} {f ∉ in.v ∨ (r, d) < (root.f, dist.f)}
  ⟨ if f ∈ out.v ∧ r = root.f ∧ root.f ≠ f →
        mute f
    [] f ∈ out.v ∧ r < root.f →
        root.f, dist.f, sig.u_f := r, d + 1, true ;
        turn f
    [] f ∈ in.v ∧ (r, d) < (root.f, dist.f) →
        root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r
    [] f ∈ mut.v ∧ r < root.f ∧ u ≠ v →
        root.f, dist.f, sig.u_f := r, d + 1, true ;
        unmute f as u →_f v
    [] f ∈ mut.v ∧ r ≤ root.f ∧ (r = root.f ∨ u = v) →
        root.f := r
    [] sig.v_f →
        skip
    fi ⟩
rap
```

Since we have strengthened some guards related to $f \in out.v$ and $f \in mut.v$, we must again ensure as a pre-assertion that at least one of the guards evaluates to *true*. As a basis we use the two corresponding pre-assertions from the beginning of this section. Thanks to including the option to unmute a muted edge (in Section 8.2.2), these assertions are strong enough for the guards related to $f \in mut.v$. For the guards related to $f \in out.v$ we require an extra pre-assertion, viz. $sig.v_f \ \vee \ f \notin out.v \ \vee \ r < root.f \ \vee \ root.f \neq f$:

```
par f, u : u ∼_f v ∧ f ≠ e →
      {? sig.v_f ∨ f ∉ out.v ∨ r < root.f ∨ root.f ≠ f}
      {? (root.e, dist.e) ≤ (r, d)} {? (∀h : S.v.h : root.h < r) ∨ S.v.e}
      {sig.v_f ∨ r ≤ root.f} {f ∉ in.v ∨ (r, d) < (root.f, dist.f)}
  ⟨ ... ⟩
rap
```

We first consider assertion $(\forall h : S.v.h : \ root.h < r) \ \vee \ S.v.e$. Although we could continue with it in its current shape, we will try to eliminate it. First we strengthen it into a more convenient condition as follows:

$$
\begin{array}{ll}
 & (\forall h : S.v.h : \ root.h < r) \ \vee \ S.v.e \\
\Leftarrow & \quad \{ \text{ use conjunct } (\forall f : f \in edge.v : \ root.h \leq root.f) \text{ of } S.v.h \ \} \\
 & (\exists f : f \in edge.v : \ root.f < r) \ \vee \ S.v.e \\
\equiv & \quad \{ \text{ calculus } \} \\
 & (\forall f : f \in edge.v : \ r \leq root.f) \ \Rightarrow \ S.v.e
\end{array}
$$

By definition of $S.v.e$, we can use required assertion $(root.e, dist.e) \leq (r, d)$ to reduce this condition to *true*, if we strengthen an invariant and an assertion by leaving out disjunct $sig.v_f$, and generalize that assertion into $(\forall f : f \neq e \ \wedge \ f \in in.v : \ (r, d) < (root.f, dist.f))$. Thus we obtain:

```
  {? inv e ∈ out.v ∨ (e ∈ in.v ∪ loop.v ∧ root.e = e)}
par f, u : u ∼_f v ∧ f ≠ e →
      {? (root.e, dist.e) ≤ (r, d)} {? sig.v_f ∨ f ∉ out.v ∨ r < root.f ∨ root.f ≠ f}
      {? (∀f : f ≠ e ∧ f ∈ in.v : (r, d) < (root.f, dist.f))}
  ⟨ if ... fi ⟩
rap
```

Note that the strengthened invariant is indeed maintained by the algorithm; its local correctness can be achieved by leaving out disjunct $sig.v_f$ from the related assertions. The strengthened assertion is globally correct under the other nodes using the direction property, and the node itself cannot violate it. Its local correctness is guaranteed by leaving out disjunct $sig.v_f$ in the related pre-assertion.

The global correctness of the remaining queried assertions is guaranteed using the descendence property and the signalling property respectively. Their local correctness is guaranteed by requiring the following pre-assertion of the preceding assignment:

$$\begin{array}{|l} \{? \ (\forall f : f \neq e \ \wedge \ f \in out.v \ \wedge \ root.f = f : \ sig.v_f \ \vee \ root.e < f)\} \\ r, d := root.e, \ dist.e \end{array}$$

Using the descendence and the signalling property, the global correctness of this queried assertion is guaranteed. For its local correctness, note that it is a post-assertion of the selection of an edge $e : S.v.e$. Therefore we strengthen predicate S into

$$\begin{array}{lll} S.v.h \ \equiv & & ((h \in out.v \ \wedge \ h \notin in.v) \ \vee \ (h \in in.v \cup loop.v \ \wedge \ root.h = h)) \\ & \wedge & (\forall f : f \neq h \ \wedge \ f \in edge.v : \ root.h \leq root.f) \\ & \wedge & (\forall f : f \neq h \ \wedge \ f \in in.v : \ (root.h, dist.h) < (root.f, dist.f)) \\ & \wedge & (\forall f : f \neq h \ \wedge \ f \in out.v \ \wedge \ root.f = f : \ root.h < f) \end{array}$$

By strengthening S we also strengthen invariant P_0; hence we have to reconsider its correctness. Thanks to the descendence property, the additional conjunct cannot lead to more violations of $S.v.h$ in invariant P_0 if we require the additional invariant

$$P_1 \ \ (\forall f :: \ root.f \leq f)$$

which itself is maintained by the descendence property. Whenever $S.v.f$ must be established, we must also ensure that the new conjunct is established. Recall that each time that $S.v.f$ must be established in a state where $S.v.g$ holds, it is accompanied with an assignment that establishes $root.f < root.g$. Using $S.v.g$ the new conjunct follows from invariant P_1.

What remains is the related condition $(\forall h : S.v.h : \ root.h < r) \ \vee \ S.v.e$. It still reduces to *true*, if we also strengthen assertion $sig.v_f \ \vee \ f \notin out.v \ \vee \ r < root.f \ \vee \ root.f \neq f$ into $(\forall f : f \neq e \ \wedge \ f \in out.v \ \wedge \ root.f = f : \ r < f)$. Its local correctness can be achieved by leaving out disjunct $sig.v_f$ from the related assertions. Its global correctness follows from invariant P_1.

Thus we obtain the following algorithm (in which we left out the annotation):

```
par f : f ∈ loop.v  →
   ⟨ if f ∉ mut.v → mute f   []   f ∈ mut.v → skip fi ⟩
rap ;

do true →
     par f : f ∈ edge.v  →  sig.v_f := false rap ;

     e: S.v.e ;
     r, d := root.e, dist.e ;

     par f, u : u ∼_f v ∧ f ≠ e  →
        ⟨ if f ∈ out.v ∧ r = root.f ∧ root.f ≠ f →
               mute f
          [] f ∈ out.v ∧ r < root.f →
               root.f, dist.f, sig.u_f := r, d + 1, true ;
               turn f
          [] f ∈ in.v ∧ (r, d) < (root.f, dist.f) →
               root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r
          [] f ∈ mut.v ∧ r < root.f ∧ u ≠ v →
               root.f, dist.f, sig.u_f := r, d + 1, true ;
               unmute f as u →_f v
          [] f ∈ mut.v ∧ r ≤ root.f ∧ (r = root.f ∨ u = v) →
               root.f := r
          [] sig.v_f →
               skip
          fi ⟩
     rap ;

     await( (∃f : f ∈ edge.v :  sig.v_f) )
od
```

Thus we obtained a partially correct, stabilizing and deadlock-free algorithm. What remains is initialization and topology changes.

8.2.5 Initialization and topology changes

In this section we deal with the following two related issues: initialization of the system invariants and dealing with the topology changes. Since our algorithm has to deal with dynamically changing topologies, for initialization we can simply assume that initially there are no nodes and no edges. All other (possibly more realistic) "initial" configurations can be obtained using the topology changes, which need to be considered anyhow.

There are two basic building blocks of the network topology, viz. a node without edges (i.e. a bus without bridge portals), and an edge that connects two nodes with only one incident edge (i.e. a bridge). To be able to build all possible network topologies, we consider the following topology changes: creating and removing such basic building blocks, and merging and splitting nodes.

Upon a topology change, all nodes that are involved in the topology change are notified[5]. Since the structure of the main computational unit of the algorithm (i.e. the bus) may be changed by topology changes, we adopt the net update proposal to abort and afterwards restart the algorithm in the nodes that are notified about

[5]Net update jargon: a *bus reset* occurs.

a topology change. Using that the algorithm for any node has no pre-assertion, we can guarantee the correctness of the algorithm and its annotation by only ensuring that the system invariants are maintained by the topology changes.

Both the creation and the removal of nodes without edges obviously maintain the invariants. The creation of an edge f that connects two nodes containing only that edge maintains the invariants if the edge is non-muted and $root.f$ has initial value f. The invariants are also maintained by removal of an edge such that the remaining nodes have no edges. Furthermore invariant P_1 is not affected by either merging or splitting nodes.

To ensure that invariant P_0 is maintained by merging nodes u and v into a node $w : edge.w \neq \emptyset$, we must ensure that an edge $h : S.w.h$ can be chosen. Typical candidates are edges h with the smallest $(root.h, dist.h)$ value such that $S.u.h$ or $S.v.h$ holds. For the last two conjuncts of $S.w.h$, we must require invariant P_2:

$$P_2 \;\; (\forall f : \neg mut.f : \; f \leq root.f \;\equiv\; dist.f \leq 0)$$

To maintain this invariant under creation of an edge f, we require initial value 0 for $dist.f$. This invariant is also maintained by the algorithm if we adopt the following invariant (which is also maintained):

$$P_3 \;\; (\forall f :: \; 0 \leq dist.f)$$

In general, invariant P_0 is not maintained under splitting nodes. As topology changes cannot be avoided, violations of the invariant should be detected and restored. Since the invariant is of the shape "for each node some locally checkable condition holds", its violation can locally be detected by at least one node. In general it cannot be locally restored by the nodes that can detect the violation. So a more global approach is needed, like a network reset[6], e.g. [AAG87] but using our more restricted start criterion, that re-initializes the network under maintenance of the invariants. This is possible since the invariants hold in an empty graph, and both adding edges and nodes, and merging nodes maintain the invariant. For the details of a possible network reset procedure we refer to [AAG87].

8.2.6 Full annotation

We have frequently left out parts of the algorithm and its annotation. In this section we provide the fully annotated program for each node $v : edge.v \neq \emptyset$. Its

[6] Net update jargon: *panic*, but in many cases net update first attempts to restore the invariant locally. Apart from some conditions that correspond to violations of our invariant P_0, the main start criterion for panic exploits that if a reset is necessary, then $dist.f$, for some edge f, would eventually exceed a certain large value. Like in [AG94], this depends on an upperbound on the size of the network. Although in practical applications a large upperbound is usually available, there is a risk for performance problems.

proof has been checked using the theorem prover PVS as discussed in Section 9.4.

```
par f : f ∈ loop.v  →
   ⟨ if f ∉ mut.v → mute f    []    f ∈ mut.v → skip fi ⟩
rap ;

 {inv loop.v ⊆ mut.v}
do true →
     par f : f ∈ edge.v  →  sig.v_f := false rap ;

      {loop.v ⊆ mut.v}
     e: S.v.e ;
      {loop.v ⊆ mut.v} {e ∈ out.v ∨ (e ∈ in.v ∪ loop.v ∧ root.e = e)}
      {(∀f : f ≠ e ∧ f ∈ edge.v : sig.v_f ∨ root.e ≤ root.f)}
      {(∀f : f ≠ e ∧ f ∈ in.v : (root.e, dist.e) < (root.f, dist.f))}
      {(∀f : f ≠ e ∧ f ∈ out.v ∧ root.f = f : root.e < f)}
     r, d := root.e, dist.e ;

      {inv sig.v_e ∨ root.e = r} {inv sig.v_e ∨ e ∉ out.v ∨ dist.e ≤ d}
      {inv loop.v ⊆ mut.v} {inv e ∈ out.v ∨ (e ∈ in.v ∪ loop.v ∧ root.e = e)}
     par f, u : u ∼_f v ∧ f ≠ e  →
           {sig.v_f ∨ r ≤ root.f} {(∀f : f ≠ e ∧ f ∈ in.v : (r, d) < (root.f, dist.f))}
           {(root.e, dist.e) ≤ (r, d)} {(∀f : f ≠ e ∧ f ∈ out.v ∧ root.f = f : r < f)}
       ⟨ if f ∈ out.v ∧ r = root.f ∧ root.f ≠ f →
              mute f
         [] f ∈ out.v ∧ r < root.f →
              root.f, dist.f, sig.u_f := r, d + 1, true ;
              turn f
         [] f ∈ in.v ∧ (r, d) < (root.f, dist.f) →
              root.f, dist.f, sig.u_f := r, d + 1, sig.u_f ∨ root.f ≠ r
         [] f ∈ mut.v ∧ r < root.f ∧ u ≠ v →
              root.f, dist.f, sig.u_f := r, d + 1, true ;
              unmute f as u →_f v
         [] f ∈ mut.v ∧ r ≤ root.f ∧ (r = root.f ∨ u = v) →
              root.f := r
         [] sig.v_f →
              skip
         fi ⟩
          {sig.v_f ∨ root.f = r} {sig.v_f ∨ f ∉ out.v} {sig.v_f ∨ f ∉ in.v ∨ d < dist.f}
     rap ;

      {R.v} {loop.v ⊆ mut.v}
     await( (∃f : f ∈ edge.v : sig.v_f) )
od
```

Invariants:

P_0: $(\forall v : edge.v \neq \emptyset : (\exists h : h \in edge.v : S.v.h))$
P_1: $(\forall f :: root.f \leq f)$
P_2: $(\forall f : \neg mut.f : f \leq root.f \equiv dist.f \leq 0)$
P_3: $(\forall f :: 0 \leq dist.f)$

Definitions:

$$
\begin{array}{lll}
S.v.h \equiv & & ((h \in out.v \wedge h \notin in.v) \vee (h \in in.v \cup loop.v \wedge root.h = h)) \\
& \wedge & (\forall f : f \neq h \wedge f \in edge.v : root.h \leq root.f) \\
& \wedge & (\forall f : f \neq h \wedge f \in in.v : (root.h, dist.h) < (root.f, dist.f)) \\
& \wedge & (\forall f : f \neq h \wedge f \in out.v \wedge root.f = f : root.h < f) \\
 & & \\
R.v \equiv & & (\forall f : f \in edge.v : sig.v_f \vee root.f = r) \\
& \wedge & (\forall f : f \in out.v : sig.v_f \vee f = e) \\
& \wedge & (\forall f : f \in out.v : sig.v_f \vee dist.f \leq d) \\
& \wedge & ((\forall f : f \in in.v : sig.v_f \vee d < dist.f) \vee (\forall f : f \in out.v : sig.v_f)) \\
& \wedge & sig.v_e \vee edge.v = \emptyset \vee e \in out.v \vee (e \in in.v \cup loop.v \wedge root.e = e)
\end{array}
$$

8.2.7 Performance improvement

Notice that in statement $e : S.v.e$, the variables of all incident edges are involved. This assumes a snapshot on all incident edges, which may yield a considerable performance penalty. In this section we show that the snapshot is not required to be consistent.

To formalize this snapshot, we introduce in each node the local variables $root'$, $dist'$ and $state'$. Then we precede the selection of an edge e with for each edge $f : f \in edge.v$ an assignment $root'.f, dist'.f, state'.f := root.f, dist.f, state.v.f$, where we use $state.v.f$ as a shorthand to denote whether edge f is an incoming edge, an outgoing edge or a muted edge with respect to node v. To show that an edge e can safely be selected using these copied values, we propose to replace the corresponding part of the algorithm by:

$$
\begin{array}{l}
edge' := \emptyset \ ; \\
\\
\ \{\textbf{inv}\ loop.v \subseteq mut.v\}\ \{\textbf{inv}\ (\exists e : e \in edge.v :\ T.v.e)\} \\
\textbf{par}\ f : f \in edge.v\ \rightarrow \\
\quad sig.v_f := false \ ; \\
\quad root'.f, dist'.f, state'.f, edge' := root.f, dist.f, state.v.f, edge' \cup \{f\} \\
\textbf{rap}\ ; \\
\\
\ \{loop.v \subseteq mut.v\}\ \{(\exists e : e \in edge.v :\ T.v.e)\}\ \{edge' = edge.v\} \\
\ \{(\forall f : f \in out.v\ \wedge\ root.f = f :\ f \in out'.v\ \wedge\ root'.f = f)\} \\
\ \{(\forall f : f \in edge.v :\ (root.f, dist.f) \leq (root'.f, dist'.f))\}\ \{(\forall f : f \in in.v :\ f \in in'.v)\} \\
\ \{(\forall f : f \in edge.v :\ (sig.v_f \wedge f \notin in.v)\ \vee\ (root.f, dist.f) = (root'.f, dist'.f))\} \\
\ \{(\forall f : f \in edge.v :\ (f \in out'.v\ \vee\ (f \in in'.v \cup loop.v\ \wedge\ root'.f = f)) \Rightarrow \\
\qquad\qquad (f \in out.v\ \vee\ (f \in in.v \cup loop.v\ \wedge\ root.f = f)))\} \\
e\colon T.v.e \ ; \\
\\
\ \{loop.v \subseteq mut.v\}\ \{e \in out.v\ \vee\ (e \in in.v \cup loop.v\ \wedge\ root.e = e)\} \\
\ \{(\forall f : f \neq e\ \wedge\ f \in edge.v :\ sig.v_f\ \vee\ root.e \leq root.f)\} \\
\ \{(\forall f : f \neq e\ \wedge\ f \in in.v :\ (root.e, dist.e) < (root.f, dist.f))\} \\
\ \{(\forall f : f \neq e\ \wedge\ f \in out.v\ \wedge\ root.f = f :\ root.e < f)\}
\end{array}
$$

where $T.v.e \equiv (root, dist, state.v := troot, tdist, tstate).(S.v.e)$
with for $f : f \in edge'$: $troot.f, tdist.f, tstate.f = root'.f, dist'.f, state'.f$
and for $f : f \notin edge'$: $troot.f, tdist.f, tstate.f = root.f, dist.f, state.v.f$

All previous assertions are trivially maintained by this modification. We first consider the local correctness of the last series of assertions, which are the post-assertions according to Section 8.2.6. It follows from two parts: the pre-assertions of the selection of an edge e and the definition of $T.v.e$ in case $edge' = edge.v$:

$$
\begin{array}{lll}
T.v.e \equiv & & (e \in out'.v\ \vee\ (e \in in'.v \cup loop.v\ \wedge\ root'.e = e)) \\
& \wedge & (\forall f : f \neq e\ \wedge\ f \in edge.v :\ root'.e < root'.f) \\
& \wedge & (\forall f : f \neq e\ \wedge\ f \in in'.v :\ (root'.e, dist'.e) < (root'.f, dist'.f)) \\
& \wedge & (\forall f : f \neq e\ \wedge\ f \in out'.v\ \wedge\ root'.f = f :\ root'.e < f)
\end{array}
$$

Then we consider the pre-assertions of the selection of an edge e, apart from the first three ones. For their global correctness, note that they reflect many global correctness arguments that we used before, especially the three properties mentioned in Section 8.2.3. Their local correctness is established conjunct-wise for each edge.

The global correctness of assertion $edge' = edge.v$ is guaranteed, and its local correctness is established conjunct-wise for each edge (we have left out the corresponding annotation). Note that we use local variable $edge'$ only for the annotation.

The remaining assertion and invariant ($\exists e : e \in edge.v : T.v.e$) deserve special attention. Their local correctness and maintenance under the statements in the new parallel composition is guaranteed by construction. Their proof of global correctness is analogous to the first part of the proof of maintenance of P_0 in Section 8.2.4.

8.2.8 Example

To get an operational idea of how the algorithm can behave, we have included in Figure 8.1 one possible behavior of the algorithm. The steps are coarse-grained in the sense that they represent executions of a full body of the repetition of a node. Furthermore, it does not contain any topology change.

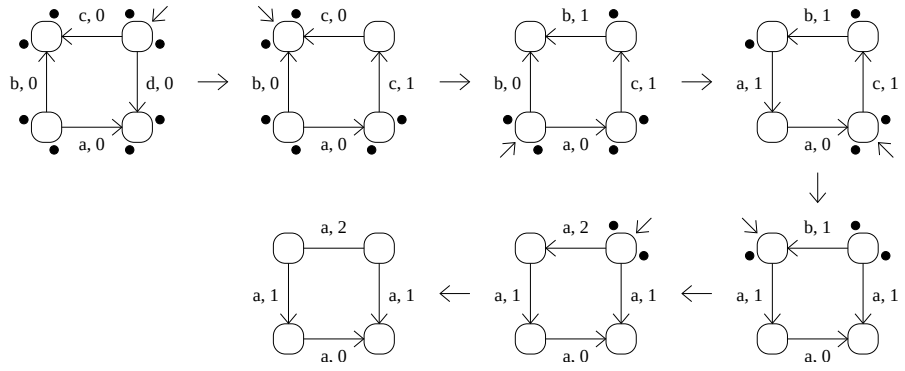


Figure 8.1 Example behavior

The figure contains a sequence of seven networks. The upper-left network is a just initialized network, and the lower-left network is a stabilized network. The networks contain four nodes that are interconnected by four edges with identities $a, b, c, d : a < b < c < d$. The nodes are represented by circles, the edges $\sim_f$ are represented by lines labeled with $root.f, dist.f$, and in particular the edges $\rightarrow$ in the spanning tree are represented by arrows between nodes. For clarity reasons,

we did not explicitly include the identities of the edges, but they can easily be derived from the initial network since initially for each edge f we have $root.f = f$. A filled dot near a node v and an edge f denotes that $sig.v_f$ holds. Diagonal arrows indicate the node that is going to perform the body of the repetition of the algorithm according to this scenario.

8.3 Implementation

In Section 8.2, we have developed an algorithm for a graph abstraction of an IEEE 1394.1 network. In this section we show that abstract algorithms can easily be implemented on the underlying portals, which are the computational units. We first transform the algorithm into a more convenient shape. Then we use a transformational technique, which does not depend on the previous annotation, to make the parallelism in the nodes and edges more explicit. Finally we assign these parallel components to the portals.

8.3.1 Convenient shape

We first transform the algorithm from Section 8.2, including the optimization of Section 8.2.7, into a more convenient shape for the transformation. Using the snapshot from Section 8.2.7, assignment $r, d := root.e, dist.e$ can be transformed into the local assignment $r, d := root'.e, dist'.e$ (see Section 6.2.6). We also make the ranges of the parallel compositions more homogeneous by extending them to all incident edges of the node and introducing additional selection statements within their bodies. Furthermore we abstract from some details by introducing functions F and G (which we will not explicitly specify). Thus in the rest of this section we consider the following algorithm for each node $v : edge.v \neq \emptyset$:

```
par f : f ∈ edge.v →
   ⟨ if ⟨ f ∈ loop.v\mut.v → mute f   []   f ∉ loop.v\mut.v → skip fi ⟩
rap ;

do true →
    par f : f ∈ edge.v →
        sig.v_f := false ;
        root'.f, dist'.f, state'.f := root.f, dist.f, state.v.f
    rap ;

    e, r, d := F(root', dist', state') ;

    par f, u : u ∼_f v →
        root.f, dist.f, state.v.f, state.u.f := G(e, r, d, f, root.f, dist.f, state.v.f)
    rap ;

    await( (∃f : f ∈ edge.v : sig.v_f) )
od
```

8.3.2 Explicit parallelism

In this section, we make the parallelism in the nodes more explicit. For each node $v : edge.v \neq \emptyset$ we introduce one component[7] $C.v$ and a component $B.f.v$ per edge $f : f \in edge.v$. Component $C.v$ is a version of the algorithm that delegates the statements in the parallel compositions with respect to edge f to component $B.f.v$.

The communication and synchronization between these components must be via the bus, so we have to introduce message communication. In what follows we briefly describe the implementations of the three constructions that need to be implemented:

- sequential compositions $S;T$ can be replaced by distributed sequential compositions, i.e. by sending a message upon completion of statement S, and only starting the execution of statement T after receiving the message;
- assignments $x := E$ can be replaced by distributed assignments, i.e. by sending a message with value E, and after receiving this value assigning it to local variable x;
- statements **await**(E) can be replaced by waiting for a message that guarantees condition E, and sufficiently often sending the message.

We introduce 6 types of messages, some of which have parameters, with names that weakly reflect their purpose: "ready", "request", "response" (r, d, s), "update" (e, r, d), "done" and "awake". Then we obtain for component $C.v$:

```
par f : f ∈ edge.v →
    receive "ready" from B.f.v
rap ;

do true →
    par f : f ∈ edge.v →
        send "request" to B.f.v ;
        receive "response" (root'.f, dist'.f, state'.f) from B.f.v
    rap ;

    e, r, d := F(root', dist', state') ;

    par f : f ∈ edge.v →
        send "update" (e, r, d) to B.f.v ;
        receive "done" from B.f.v
    rap ;

    receive at least one "awake" from B.f.v, for at least one edge f
od
```

And for component $B.f.v$, with $f, v, u : u \sim_f v$, we obtain:

[7] Net update jargon: the *coordinator* of the bus, but it delegates less, i.e. it is more centralized, and hence it does not optimally exploit the potential parallelism on the bus.

```
⟨ if ⟨ f ∈ loop.v\mut.v → mute f   []   f ∉ loop.v\mut.v → skip fi ⟩ ;
send "ready" to C.v ;

do true →
    if ∃ "request" from C.v →
        receive "request" from C.v ;
        sig.v_f := false ;
        send "response" (root.f, dist.f, state.v.f) to C.v ;

        receive "update" (e, r, d) from C.v ;
        root.f, dist.f, state.v.f, state.u.f := G(e, r, d, f, root.f, dist.f, state.v.f) ;
        send "done" to C.v

    [] sig.v_f →
        send "awake" to C.v
    fi
od
```

This implementation using messages is just a sketch that can still be improved, e.g. with respect to the amount of "awake" messages.

8.3.3 Deployment

Finally we map these components and the data to the portals, as depicted in Figure 8.2. For each edge $f : u \sim_f v$, we equip the corresponding bridge with components $B.f.u$ and $B.f.v$, and with the variables of the edge. Strictly speaking, for each self-loop only one component $B.f.v$ should be created, but it turns out that the more homogeneous option of two component, viz. $B.f.u$ and $B.f.v$, does no harm. The identity of the edge is just the identity of one of its portals.

Less straightforward is to ensure that on each bus (i.e. a node) with at least one portal there is exactly one component C. To ensure that there is at least one component C, we equip each portal with such a component; and to ensure that there is at most one component C, a leader election protocol can be used on the bus to activate only one of them. Such a leader[8] election protocol on a single bus is already part of the IEEE 1394 standard.

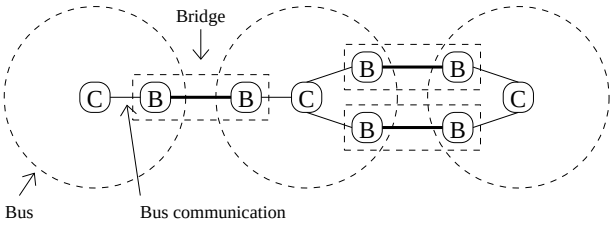


Figure 8.2 Deployment architecture

[8]Net update jargon: the portal with the *highest physical identity*, which is computed by each portal after the self-identification phase on the bus.

8.4 Comparison

In absence of the topology change of splitting a bus, our algorithm is effectively equal to net update. The issues caused by splitting a bus raised much discussion in the last stages of the development of net update. In case splitting a bus causes a node to lose all paths to the root of the spanning tree, our algorithm initiates a (global) reset, while net update contains some local mechanisms and a stronger reset criterion that try to avoid a reset. In general a global mechanism is necessary, but we have not compared the efficiency of the two approaches.

This non-trivial algorithm is distributed at two levels, namely at the bus-bridge level and at the portal level. Thanks to the abstractions we have used, we could deal with these two levels in isolation, and we did not need to address various minor low-level issues that play a role in the description of net update in the standard.

In comparison to net update, our final algorithm allows more non-determinism, which we have initially introduced as a means to delay design decisions. The result is that our algorithm does not guarantee shortest paths to the root (which was no part of our specification), and that our algorithm better exploits the potential parallelism within each bus.

8.5 Conclusions and further work

We have analyzed the spanning tree algorithm of net update by reconstructing a version of it, starting from its specification. There is a striking similarity between net update and the algorithm we have thus obtained, but they are not completely identical. Hence this analysis is not a correctness proof of net update, but it exposes the core ideas of net update.

In comparison with the model checking approach, our approach requires more human effort. On the other hand, there is no real method that describes how to use the results of a model checker to fix an algorithm that happens to be incorrect. Our approach is constructive in the sense that it provides clues on how to obtain a correct algorithm, and it reveals the essential ingredients of the algorithm.

Our derivation is based on the method of [FvG99], and it is mainly guided by the proof obligations for its correctness. In addition we have used a few hints about the way net update is supposed to behave. Thus the algorithm is developed hand-in-hand with its correctness proof. Applying these techniques to a case study like the present one is quite laborious, but the proposed style of reasoning remains effective. This illustrates that these techniques are valuable for the analysis of complex industrial algorithms.

Nevertheless there is a considerable danger that attempts to apply this method

end up in a complete mess. On one side this can be caused by not identifying the right concepts, but on the other side for large algorithms and for algorithms with many requirements, a large amount of the work consists of copying assertions, applying simple substitutions, and doing many similar proofs. Therefore we will experiment in Chapter 9 with some tool support for this method.

As sketched in Section 6.2 the essence of this method is not complicated, though a complaint about applications of it, is that it would not be intuitive. The requested intuition is demanded in terms of concrete operational behavior of the algorithm, while a virtue of this method is that it is based on an axiomatic semantics, which is more abstract than an operational semantics. Independent of any particular formal method used, it is extremely important to construct algorithms using proper abstractions, especially in case non-determinism plays an important role like in parallel systems.

This method also illustrates that large parts of the construction of algorithms follow directly from the proof obligations. Instead of using formalizations only as a means to precise definition, more attention should be payed to the opportunities of formula manipulation. Nevertheless, at some points real design decisions must be made, which cannot be motivated by formulae alone. In this chapter, these decisions are typically marked by short descriptions of net update.

An important direction for further work is to identify the right formal concepts and abstractions for the design of distributed algorithms, e.g. the way in which the message communication is introduced. In Chapter 7 we have described an algorithm directly in terms of messages, similar to the style proposed in [Gol03]. In the current chapter we have experimented with another approach, similar to the one advocated in [Hoo00]. In this way we design a distributed algorithm like any other parallel algorithm in terms of variables, while the messages are only introduced in the implementation. Although this approach seems not to be feasible for Chapter 7 as discussed in Section 7.1, it does introduce two nice levels of abstraction, which in turn can be developed using different techniques.

Chapter 9

Incremental verification of Owicki/Gries proof outlines

In this chapter that is based on [MW05], we develop some tool support for the programming method of Feijen/van Gasteren [FvG99]. This assertion-based method is based on the theory of Owicki/Gries [OG76], and supports the construction of parallel algorithms hand-in-hand with their correctness proof.

To enhance the applicability of assertion-based methods, the required amount of human effort needs to be reduced. In this chapter we focus on the proof efforts. Since current theorem provers offer much automatization, we address the integration of automated theorem provers with assertion-based methods. In this way we can also assess the practical usability of current automated theorem provers.

Applications of assertion-based methods, including [FvG99], are typically incremental: an initially incomplete annotation is extended repeatedly until all proof obligations can be proved. In contrast to much related work that focuses on the formalization of languages, we emphasize this incremental nature.

To experiment with our approach, we have implemented a tool that uses the PVS theorem prover [ORS92] as a back-end. Nevertheless this chapter is largely independent from any specific theorem prover.

Overview In Section 9.1 we evaluate related work, after which we describe the main ingredients of our approach in Section 9.2. Based on these ingredients, in Section 9.3 we discuss the tool that we have developed and the generated PVS input files. In Section 9.4 we present the practical results that we have obtained. Finally Section 9.5 contains the conclusions.

9.1 Related work

Tool support for formal methods is a very active research area. Aside from the enormous interest in techniques related to model checking, some (recent) work addresses axiomatic proofs based on Hoare logic and on the theory of Owicki/Gries. In this overview of related work, we discuss some work on using theorem provers for the verification of programs.

In [Hoo98] and related publications, there are experiments in modeling proof rules in PVS. The emphasis is on distributed real-time systems, and reusable theories about time have been developed. However, the models of case studies look ad-hoc, and exploiting automatization of PVS is not a key issue.

Other related work originates from formalizing the Java programming language, e.g. in [Ábr05, JP03]. Distracting complications in such a language are the many object-orientation issues. In [JP03] threads and concurrency are excluded, and in [Ábr05] the emphasis is on formalizations of the semantics of the language instead of exploiting theorem prover capabilities. Getting closer to our methodological goals, [Fra99] addresses the construction of sequential programs and their correctness proofs in the style of [Dij76].

9.1.1 Theory of Owicki/Gries in Isabelle

The closest related work is [PN02, NPN99], in which the Owicki/Gries theory is formalized in the Isabelle theorem prover [Pau94]. This work has some nice theoretical aspects. For example, the formalization does not refer explicitly to control points, which are also not contained in the Owicki/Gries theory itself. Moreover, using that the proof obligations are generated within the Isabelle theorem prover, soundness with respect to operational semantics has been proved.

For practical use, there is a dedicated Isabelle tactic. To prove an annotated program, a user submits a goal of the shape "this given annotated program is correct" to the theorem prover. Then the tactic is applied to conclude that this goal follows from a large proof obligation, consisting of the corresponding Owicki/Gries proof obligations. Afterwards, this large proof obligation needs to be proved using the theorem prover's usual techniques. For effective practical use, this approach has some disadvantages which we discuss in Section 9.2.

9.2 Design points

Although the approach of [PN02] is related to our goals, it does not effectively support incremental assertion-based methods. In what follows we discuss the problems and our proposals to overcome them. We will not compare the theorem

provers in detail, although in Section 9.4 we briefly address this issue.

9.2.1 Decomposing the proof obligation

To verify an annotated program, the tactic of [PN02] generates one big combined proof obligation. This is probably caused by the fact that tactics in the theorem prover need to transform a single proof into one other proof goal. Afterwards, the theorem prover is used to try to prove the generated proof obligation. Such an approach has two disadvantages.

First, dealing with large proof obligations heavily relies on the capabilities of theorem provers to reduce them into smaller chunks that can easily be proved. As mentioned by [JP03], this easily becomes a bottleneck, because when the size of proof obligations exceeds certain limits, theorem provers become very time consuming or run out of memory. It is advantageous to split proof obligations into smaller ones before employing a theorem prover.

The second problem originates from the incremental and iterative nature of the assertion-based methods. In typical applications, a program and its annotation are repeatedly modified as required for parts of their proof that cannot be completed yet. Since these frequent changes are usually small, many parts of a previous proof attempt can be reused, at least theoretically. So much theorem proving work can be saved by splitting proof obligations into *reusable* parts.

To maximize the reuse of unchanged parts of the proof obligation, we propose to split the proof obligation such that the typical steps in the methods only affect a small number of parts. In correspondence with the structure of the Owicki/Gries theory, this can be achieved by splitting according to the assertion being proved, to local or global correctness and to the particular statement being involved. Thus the individual parts of the proof obligation can be identified by a triple ("local/global", assertion, statement).

9.2.2 Stabilizing the proof scripts

A step that occurs very often in assertion-based methods is adding an assertion. Apart from the new proof obligations for correctness of the assertion itself, the assertion also pops up in many existing proof obligations. More specifically, the assertion pops up as a conjunct in the left argument of some Hoare triples. Since Hoare triples are anti-monotonic in their left argument, see Section 6.2.3, their correctness is maintained. In terms of the *wlp*-versions of the proof obligations, the assertion pops up as a conjunct in the antecedent of the implication, which indeed weakens the proof obligation. However, there is no guarantee that the old proof script of the theorem prover is also a proof script for the new proof obligation. In practice this hinders the effective use of theorem provers for incremental methods.

Instead of trying to correct the old proof scripts, we propose to ensure that there are no textual changes in these proof obligations, nor in the ingredients employed by the old proof scripts. To that end, we need to fully decouple the assertions from each other. Consider the typical example $[P \wedge Q \Rightarrow Z]$ (e.g. a proof obligation for local correctness) with predicates P, Q and Z. To decouple Z from P and Q, we use the principle of indirect inequality and obtain $(\forall X :: [X \Rightarrow P \wedge Q] \Rightarrow [X \Rightarrow Z])$, assuming that predicate X is fresh (i.e. not yet in use). Then P and Q can be decoupled using that implication is conjunctive in its consequent:

$$(\forall X :: [X \Rightarrow P] \wedge [X \Rightarrow Q] \Rightarrow [X \Rightarrow Z])$$

In a theorem prover this can be modeled as proof obligation $[X \Rightarrow Z]$, after declaring dummy predicate X as a logical variable and posing the two axioms $[X \Rightarrow P]$ and $[X \Rightarrow Q]$. If (later on) this proof obligation should be weakened into $[P \wedge Q \wedge R \Rightarrow Z]$, then only an axiom $[X \Rightarrow R]$ needs to be added. Hence correctness of the old proof script for $[X \Rightarrow Z]$ cannot be endangered as long as all used axioms are employed explicitly. Note that these introduced axioms cannot cause soundness problems, since dummy X was fresh.

Instead of introducing a fresh predicate X for each proof obligation, we can exploit the structure of the Owicki/Gries theory to reuse some of them. Namely, the antecedent of the implication in each proof obligation is the conjunction of all assertions at one or two control points. We introduce a fresh predicate per control point, relate it to the corresponding assertions using axioms, and use (combinations of) these predicates instead of the dummy predicate explained before. In case there are two control points involved, viz. for global correctness proofs, this can be justified by applying the technique of indirect inequality to $[P \wedge Q \Rightarrow Z]$ twice, yielding:

$$(\forall X, Y :: [X \Rightarrow P] \wedge [Y \Rightarrow Q] \Rightarrow [X \wedge Y \Rightarrow Z])$$

9.2.3 Exploiting invariants

Usually some assertions are located at several control points. Such assertions are typically invariants, and many of their proof obligations (and proofs) are almost identical. For effective practical use, the redundant part of this proof load must be reduced. To this end we propose not to treat invariants just as abbreviations, but to use the dedicated proof rules from Section 6.2.3. Apart from the well-known repetition invariants supported by [PN02], we also consider invariants of parallel compositions.

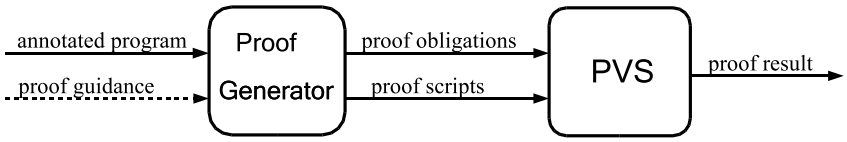


Figure 9.1 Tool architecture

9.3 Experimental environment

In this section we describe the experimental environment we have built, which is schematically depicted in Figure 9.1. Our proof generator reads the annotated program, and recursively decomposes it into atomic actions and assertions. Then internally the corresponding proof obligations are generated independent of the target theorem prover. Finally input files for the specific theorem prover (e.g. PVS) are generated that consist of proof obligations and corresponding proof scripts. Afterwards they are verified by the theorem prover in batch-mode.

In case some proof obligations are not successfully verified by the theorem prover, there are two options:

- the proof obligation does not hold, and hence the annotated program must be adapted (according to the assertion-based method);
- the proof obligation holds, but the generated proof script is not appropriate.

In the latter case, the user can influence the proof scripts by supplying proof guidance. In either case, afterwards our tool is used again to generate the new proof obligations and proof scripts.

In what follows, we first describe how to model the program in PVS, and then we show the generated proof obligations. Finally, we explain the generated proof scripts and the opportunities to influence them using proof guidance. As the modeling language of PVS is generally considered to be very readable, we will only briefly explain some aspects of it.

9.3.1 Running example: parallel linear search

As a running example we use a parallel linear search algorithm, which solves the following problem: Given a number of boolean functions on the naturals, find a value that is mapped by one of these functions to the value *true*. A collection of components ought to be used, such that for each function there is a component that is completely dedicated to the function.

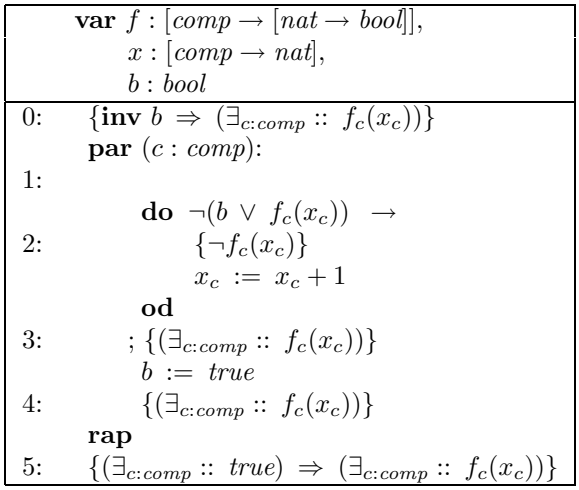


Figure 9.2 Parallel linear search example

A solution is the annotated program of Figure 9.2, which is a generalization of the two-component version in [FvG99]. The upper part consists of the declaration of the variables, in which the type *comp* denotes the set of component identifiers. For each component c, function f_c is the corresponding given function. Variable b is a shared program variable of type boolean, and for each component c, variable x_c is a local program variable of type natural.

The lower part contains the annotated program. The numbers that are followed by a colon are labels that identify the control points. If termination of the system needs to be guaranteed, we must assume that there exists an x that fulfills the post-assertion (at label 5), and that initially $(\forall_{c:comp} :: x_c = 0)$ holds.

9.3.2 Program model

In this section we discuss our model of atomic statements and assertions in PVS.

Identifiers

For the reuse as proposed in Section 9.2, we need identifiers for the assertions and statements. We only explicitly assign identifying labels to the control points, and identify the assertions and statements by the label of their control point and some serial number (or rather character). For example, the invariant at the control point with label 0, and the statement and assertion at the control point with label 2 in the running example are referred to as *inv_0a*, *stat_2* and *ass_2a*.

Data types

Elementary data types are part of the default library of PVS, so only the additional domain-specific data types need to be modeled. In the running example only a type *comp* for the identifiers of the components needs to be defined.

```
comp: type
```

Then a type *state* can be defined as a record that contains all variables. It will be used to denote states of the entire program.

```
state: type = [# f : [comp → [nat → bool]] ,
                x : [comp → nat] ,
                b : bool
             #]
```

Annotation

The two types of annotation, viz. assertions and invariants, are just predicates on the state.

```
inv_0a: pred[state] =
    lambda (s : state):    s`b  ⇒  exists (c : comp): s`f(c)(s`x(c))

ass_2a(c : comp): pred[state] =
    lambda (s : state):    not( s`f(c)(s`x(c)) )
```

Note that in the language of PVS, $s`b$ selects field b from record s.

Statements

Recall that we only need to address the atomic statements, since their composition into larger statements is part of the program's structure. The atomic actions are typically assignments and evaluations of guards. We model the assignments directly using their *wlp*. Notice that for the guards of the repetitions, evaluation to *true* and evaluation to *false* are two different atomic actions, and hence they could be modeled using two *wlp* predicate transformers. We prefer to model guards as a single predicate since the corresponding two *wlp* predicate transformers can easily be generated from it.

```
wlp_stat_2(c : comp)(p : pred[state]): pred[state] =
    lambda (s : state):    p(s with [`x(c) := s`x(c) + 1])

guard_1a(c : comp): pred[state] =
    lambda (s : state):    not( s`b  or  s`f(c)(s`x(c)) )
```

9.3.3 Proof obligations

Using these definitions, we can describe the generated proof obligations. We will distinguish between local correctness, global correctness and invariance.

Control points

Before presenting the proof obligations for correctness of the annotated program, we first address the implementation of the technique described in Section 9.2.2. For each control point, say with label i, we introduce a logical variable *lab_i*, and relate it with axioms to the assertions and invariants that hold at the control point.

```
lab_2: [comp → pred[state]]

lab_2_inv_0a: axiom
    forall (s : state): forall (c : comp):    lab_2(c)(s) ⇒ inv_0a(s)

lab_2_ass_2a(c : comp): axiom
    forall (s : state): forall (c : comp):    lab_2(c)(s) ⇒ ass_2a(c)(s)
```

To apply the technique from Section 9.2.2 to the proof obligations for invariants of parallel compositions, we also define a logical variable *scp_i* for each control point i in which a parallel composition starts. We relate this logical variable to the control points in the scope of the parallel composition using an axiom.

```
scp_0: pred[state]

def_scp_0: axiom
    forall (s : state):
        scp_0(s) = ( lab_0(s)
                     or (exists (c : comp): lab_1(c)(s))
                     ...
                     or (exists (c : comp): lab_4(c)(s)) )
```

In fact *scp* is just an abbreviation, which could be defined directly without a separate axiom. However, in that case the PVS theorem prover will often expand *scp* which leads to bad performance. Rather than working directly with the above definition of *scp*, we want to use properties of *scp* similar to the relations between control points and assertions above. The above axiom is only used in the generated proof script for the following relation between *scp* variables and invariants.

```
scp_0_inv_0a: lemma
    forall (s : state):    scp_0(s) ⇒ inv_0a(s)
```

Local correctness

Now we consider the proof obligations for the local correctness of the assertions and invariants. We assume that the initial assertion of the program is the precondition of the program. Recall that if an assertion $\{P\}$ is not an initial assertion, then it must be established by the preceding statement $\{Q\}\ S$, i.e. $\{Q\}\ S\ \{P\}$ must be a correct Hoare triple. We directly generate the proof obligations in their *wlp* version.

```
loc_ass_4a_stat_3: lemma
    forall (s : state):
        forall (c : comp): lab_3(c)(s) ⇒
            wlp_stat_3(c)(ass_4a(c))(s)
```

Such a lemma "loc_ass_4a_stat_3", for the local correctness of assertion ass_4a by preceding atomic statement stat_3, is structured as follows:

- For all states,
- and for each component that is about to execute the statement,
- the statement establishes the post-assertion.

For the local correctness of the post-assertion of a parallel composition we generate the following proof obligation.

```
loc_ass_5a_stat_4: lemma
    forall (s : state):
        (forall (c : comp): lab_4(c)(s)) ⇒
        ass_5a(s)
```

This proof rule is only complete in case the type *comp* is non-empty. In contrast to many tools, in PVS the types are allowed to be empty. This rule can easily be extended to the special case that *comp* is empty, but the resulting inhomogeneous rule turns out to have a bad influence on the performance of the prover.

To aid proving the above lemma, we also generate for each parallel composition the following additional lemma. We need to make this trivial lemma explicit to ensure that it is exploited, although its proof is simply generated.

```
parallel_statement_lemma_0: lemma
    forall (P : [comp → bool], Q : [comp → bool]):
        (forall (c : comp): P(c)) ⇒
        ( (forall (c : comp): Q(c)) = (forall (c : comp): P(c) ⇒ Q(c)) )
```

Global correctness

Then we continue with global correctness. Recall that each assertion $\{P\}$ of a component must be maintained under each statement $\{Q\}\ S$ that can be executed by another component, i.e. $\{P \wedge Q\}\ S\ \{P\}$ must be a correct Hoare triple. We directly generate the proof obligations in their *wlp* version.

```
glob_ass_4a_stat_2: lemma
    forall (s : state):
        forall (c : comp): lab_4(c)(s) ⇒
            forall (d : comp): lab_2(d)(s) ⇒
                not(c = d) ⇒
                    wlp_stat_2(d)(ass_4a(c))(s)
```

Such a lemma "glob_ass_4a_stat_2", for the global correctness of assertion ass_4a under statement stat_2, is structured as follows:

- For all states,
- if there is a component at the control point of the assertion,
- then for each component that is about to execute the statement,
- such that the two components are different (i.e. in the common enclosing parallel compositions, not all identifying variables are equal),
- the statement (re-)establishes the assertion.

Invariants

Finally we address invariants of parallel compositions. Local correctness as a pre-assertion of the parallel composition is the same as for assertions. Global correctness under the statements outside the parallel composition is almost the same, but with the first occurrence of *lab* replaced by *scp*. What remains is maintenance, or invariance, under each statement within the parallel composition.

```
inv_inv_0a_stat_2: lemma
    forall (s : state):
        scp_0(s) ⇒
            forall (c : comp): lab_2(c)(s) ⇒
                wlp_stat_2(c)(inv_0a)(s)
```

Notice that term scp_0(s) is not superfluous in case the invariant is placed within another parallel composition. Such a lemma "inv_inv_0a_stat_2", for invariance of invariant inv_0a under statement stat_2, is structured as follows:

- For all states,
- if there is a component within the scope of the parallel composition,
- then for each component that is about to execute the statement,
- the statement (re-)establishes the invariant.

9.3.4 Proof scripts

For these generated proof obligations, proof scripts are generated that rely on the automatization offered by PVS. Such an automated proof might be feasible since atomic actions are typically simple. Furthermore, proving these proof obligations might be easier than proving correctness of the algorithm, since an annotation can be exploited. However, we must be prepared that human intervention in the proof is required, so we also discuss some possibilities for human guidance.

Default script

The default proof script consists of the following three parts:

```
(skosimp* :preds? t)

(lemma "lab_2_ass_2a")
(inst -1 "s!1" "c!1")
... ...

(branch (grind :if-match nil)
  ((then (try (reduce) (fail) (skip))
    (then (inst? :if-match all) (then (reduce :if-match all) (fail) ) ) )) )
```

The first command decomposes the top-level structure of the proof obligation and introduces skolem constants (and type constraints) for the bound variables, viz. for the state and for the identifying variables of the components. Then the axioms that relate control points to assertions are *explicitly* employed (as required in Section 9.2.2), and the known constants are substituted.

From a logical point of view, the order in which the axioms are introduced is irrelevant. However, it turns out that PVS prioritizes the axioms introduced last, which can have serious consequences for the run-time performance. We have exploited the heuristic that for a global correctness proof of an assertion, it is usually very important to use that the assertion is also a conjunct at the left-hand side of the Hoare triple. Hence we have ensured that the corresponding axiom is the last axiom that is introduced.

What remains in the script is the real work, consisting of some strategies to automatically complete the proof. It is in fact an extension of the "lazy-grind"

strategy. First it applies "grind" without quantifier instantiation, and then it repeatedly tries the normal "reduce" with heuristic quantifier instantiation. If this "reduce" does not complete the proof, then repeatedly all instantiations of a bound variable are substituted and "reduce" is applied again. This proof script does not use induction, since we did not need it so far.

Proof hints

The strategies in the last line of the default proof script may become very time-consuming. Therefore it is often effective to interrupt the prover after a while and to restart after applying proof hints. Using proof hints, the generated proof script can be improved by reducing the employed collection of assertions, thereby exploiting that usually developers can easily indicate which assertions are not relevant for a proof. In [GGH05] a similar notion of dependency relations is used to provide an a-posteriori summary of a huge manual interactive proof.

Manual proof

Suppose a generated proof script is still very time consuming, or it cannot prove the proof obligation. If the proof obligation does hold, then the user can manually develop a PVS proof script and provide it to our tool. In this way completeness of our proof approach is established, but it would not be practical to use many manual proofs.

9.4 Experiments

In this section we summarize some experiments to investigate the strength of our method, and in particular of the developed proof scripts. To measure the run times, we have used a 3 GHz Intel Pentium 4 processor with hyper-threading.

9.4.1 Small algorithms

We have experimented first with some elementary annotated algorithms, based on e.g. [FvG99, Moo02, PN02]. These examples include a parallel linear search, a wait-free consensus protocol, monitored phase synchronization, and some mutual exclusion algorithms (like semaphores, ticket algorithm and Peterson's algorithm for two components). Using our tool environment, correctness of these examples has been proved automatically without using any proof guidance. Each example required less than a minute of system time.

The parallel linear search example has revealed a peculiarity of the automated strategies of PVS. The antecedent of the assertion at control point 5 (see Figure 9.2), viz. ($\exists_{c:comp} :: \mathit{true}$), is correctly skolemized to *true* after introducing a skolem constant of type *comp*. However, it turns out that the automated strategies do not use this skolem constant, and hence the antecedent has effectively been weakened to just *true*. The result is that the proof obligation is not automatically provable, although we could finally circumvent this problem. We have reported this issue to the developers of PVS, and in November 2005 we have been informed that it has been fixed and that the fix will be part of the forthcoming release of PVS.

Since [GH98] rates the automation in Isabelle and PVS as comparably good, we have initiated some experiments with Isabelle as a back-end. The study in [Kou06] extends our work to Isabelle, and it indicates that from our practical perspective, the automatization in Isabelle is less effective than the automatization in PVS, especially in treating quantifications. Since quantifications occur frequently, this seriously increases the required amount of manual interaction with the prover.

9.4.2 Larger algorithms

We have also verified two larger algorithms. First of all, we have verified the fully-annotated wait-free handshake register from [Hes98], see also [Moo02]. In [Hes98] it is mentioned that it "took only some eight hours" to construct his mechanical proof in the NQTHM prover. Using our default proof script, correctness of this annotated algorithm has been proved in about two minutes of system time, without using any proof hints or manual proofs.

The most complicated algorithm we have verified is the distributed spanning tree algorithm from Section 8.2.6. It needs to be mentioned that there is a huge gap in complexity between this algorithm and the other algorithms we have discussed. In particular, dynamic networks needed to be modeled and there are complicated assertions and statements. This verification effort has revealed one small error in an earlier version of the manually constructed annotation. After strengthening one invariant in a straightforward manner, the annotated algorithm has been proved. In addition we have separately verified the claims about maintenance of the annotation under topology changes from Section 8.2.5.

For this spanning tree algorithm, almost 90% of the generated proof obligations has been proved automatically, and 33 proof obligations have been proved after supplying proof hints. Finally manual proofs have been provided for 11 proof obligations, which is less than 4% of the total number of proof obligations. Some automated proofs have been interrupted after a while in order to save time, so these results might be improved by letting PVS run much longer using the default scripts. Running the whole proof again takes about two and half hour of system time.

9.5 Conclusions and further work

Using our tool that generates proof obligations and proof scripts, and feeds them to PVS, more than 95% of the proof obligations for the spanning tree algorithm from Chapter 8 could be handled automatically. By splitting proof obligations into small chunks, and by designing proof scripts that are robust against common program modifications, we have made our approach suitable for incremental methods.

The generated proof scripts rely on the automatization provided by the theorem prover. For further work, the proof scripts may be refined, e.g. by clever case analysis or by optimizations for typical patterns. Also additional theorem provers may be used, but special attention is required for dealing with quantifiers, which is currently a weak point of automated theorem provers.

The current tool deals with partial correctness only. A possible extension would be to check the deadlock freedom of selected statements based on their pre-assertion. Another extension would be to verify simple termination arguments based on a given variant function, which must be well-founded, and which must be decremented by each statement.

Based on these techniques, experiments need to be done with truly incremental developments, e.g. in the style of [FvG99]. Therefore our tool should be extended with better interaction with the user, and it would be nice to hand over the bookkeeping from the user to the tool. A point of concern in such an integrated tool environment is a lack of flexibility, while constructing an algorithm requires the use of suitable abstractions and notational devices for the specific problem under consideration.

An awkward disadvantage of using a theorem prover like PVS to try to prove a given proof obligation, is that a failing proof attempt usually provides no clue whatsoever. This is in sharp contrast with the manual calculational proofs in [FvG99], which are used constructively to guide the further program development. This is an important issue that needs to be addressed in order to obtain effective tool support of constructive formal methods.

Chapter 10

Conclusions

During the development of a distributed algorithm, the sting is often in the tail as it turns out to be complicated to get some details right. We have addressed such problems for IEEE 1394.1's distributed spanning tree algorithm, which is called net update. After presenting an alternative algorithm in Chapter 7, we have derived in Chapter 8 a distributed spanning tree algorithm that is strikingly similar to net update. For the latter approach, the need for effective theorem prover support became clear, which we have developed in Chapter 9.

Chapters 8 and 9 show that the programming method of [FvG99] can be used to analyze non-trivial industrial algorithms. Although the overall structure of the spanning tree algorithm was prescribed by the net update proposals, the method helped to fill in many details in a constructive way; typically these details were the ones that were changed in almost every consecutive net update proposal.

A final issue that needs to be addressed is the influence of our work on the final standard. The model checking work by [vLRG03] has had the biggest influence, probably because it is close to the descriptions of the net update proposals. The more abstract work in Chapters 7 and 8 has had very limited influence, mainly because the work was performed during a late stage of protocol development. It has increased our understanding of the intended algorithm, and it has helped to construct counter-examples for some of the many proposals. Unfortunately we must conclude that the algorithm in the final standard [IEE05] has not been completely verified before its approval.

Nevertheless, formal analysis does contribute to the quality of protocol standards, especially when formal methods experts are part of the protocol development team. In particular for applying constructive formal methods, it is important to make the specification of the protocols very explicit, and in an early development phase. Thus sound contributions can be delivered at the right moment, which is more

constructive than trying to correct existing (wrong) proposals, and which makes it easier to get these contributions into the final protocol standard.

Chapters 7 and 8 underline (once more) the importance of solving problems at the right level of abstraction. In particular we have used various levels of abstraction and distribution, and we have used a reset technique to abstract from a class of topology changes. Apart from using any particular formal method, recognizing in an early stage of algorithm development the need for convenient abstractions, would increase the quality of protocol standards.

Bibliography

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 358–370. IEEE, 1987. [66, 74, 75, 76, 80, 96]

[Ábr05] E. Ábrahám. *An assertional proof system for multithreaded Java - theory and tool support*. PhD thesis, Universiteit Leiden, January 2005. [106]

[AEY03] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. *IEEE Transactions on Software Engineering*, 29:623–633, 2003. [11]

[AEY05] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. *Theoretical Computer Science*, 331:97–114, 2005. [27]

[AG94] A. Arora and M.G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1039, September 1994. [74, 76, 96]

[BAL97] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proceedings of the 3rd workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *LNCS*, pages 259–274. Springer, 1997. [27, 34, 37, 44]

[BG92] D.P. Bertsekas and R.G. Gallager. *Data networks*. Prentice-Hall, Inc., 1992. [65]

[BM95] J.C.M. Baeten and S. Mauw. Delayed choice: an operator for joining message sequence charts. In *Proceedings of the 7th conference on Formal Description Techniques*, pages 340–354, 1995. [16, 25]

[Bri90] E. Brinksma. Constraint-oriented specification in a constructive specification technique. In *Proceedings of the REX Workshop on Stepwise*

Refinement of Distributed Systems, volume 430 of *LNCS*, pages 130–152, 1990. [17]

[CEN00] European Committee for Standardization (CEN). *Health informatics - Interoperability of patient connected medical devices*, 2000. European prestandard ENV 13735: 2000 E. [5]

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974. [63]

[Dij76] E.W. Dijkstra. *A discipline of programming.* Prentice-Hall, Englewood Cliffs, 1976. [70, 106]

[DM06] B. Dongol and A.J. Mooij. Progress in deriving concurrent programs: emphasizing the role of stable guards. In *Proceedings of the 8th conference on Mathematics of Program Construction*, volume 4014 of *LNCS*, pages 140–161. Springer, 2006. [71]

[EMR02] A.G. Engels, S. Mauw, and M.A. Reniers. A hierarchy of communication models for message sequence charts. *Science of Computer Programming*, 44:253–292, 2002. [17]

[Fra99] M.G.J. Franssen. Cocktail: a tool for deriving correct programs. In *Proceedings of the 6th Workshop on Automated Reasoning*, 1999. [106]

[FvG99] W.H.J. Feijen and A.J.M. van Gasteren. *On a method of multiprogramming.* Springer, 1999. [67, 69, 70, 71, 103, 105, 110, 116, 118, 119]

[Gär03] F.C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical Report IC/2003/38, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, June 2003. [63]

[Gen05] B. Genest. Compositional message sequence charts (CMSCs) are better to implement than MSCs. In *Proceedings of 11th conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 429–440. Springer, 2005. [23, 27]

[GGH05] H. Gao, J.F. Groote, and W.H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 17:21–42, 2005. [116]

[GH98] W.O.D. Griffioen and M. Huisman. A comparison of PVS and Isabelle/HOL. In *Proceedings of the 11th conference on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, pages 123–142. Springer, 1998. [117]

[GHS83] R.G. Gallager, P.A. Humblet, and P.M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983. [65]

[GMP03] E.L. Gunter, A. Muscholl, and D.A. Peled. Compositional message sequence charts. *International Journal on Software Tools for Technology Transfer*, 5(1):78–89, November 2003. [11, 21, 22]

[Gol03] D. Goldson. Extending the theory of Owicki and Gries with asynchronous message passing. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference*, pages 532–541. IEEE Computer Society, 2003. [104]

[GY84] M.G. Gouda and Y.T. Yu. Synthesis of communicating finite-state machines with guaranteed progress. *IEEE Transactions on Communications*, COM-32(7):779–788, July 1984. [46, 55]

[Hél01] L. Hélouët. Some pathological message sequence charts and how to detect them. In *Proceedings of the 10th SDL forum*, volume 2078 of *LNCS*, pages 348–364. Springer, 2001. [27]

[Hes98] W.H. Hesselink. Invariants for the construction of a handshake register. *Information Processing Letters*, 68:173–177, 1998. [117]

[Hey00] S. Heymer. A semantics for MSC based on Petri-Net components. In *Proceedings of the 2nd workshop on SDL And MSC*, 2000. [11]

[HJ00] L. Hélouët and C. Jard. Conditions for synthesis of communicating automata from HMSCs. In *Proceedings of the 5th workshop on Formal Methods for Industrial Critical Systems*, 2000. [13, 21, 27, 34, 40]

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. [68]

[Hoo98] J. Hooman. Developing proof rules for distributed real-time systems with PVS. In *Proceedings of the workshop on Tool Support for System Development and Verification*, volume 1 of *BISS Monographs*, pages 120–139. Shaker, 1998. [106]

[Hoo00] R.R. Hoogerwoord. A formal development of distributed summation. Computing Science Report 00-09, Technische Universiteit Eindhoven, April 2000. [104]

[Huo05] X. Huo. A concurrent wave panic protocol for dynamic 1394.1 networks. Master's thesis, Technische Universiteit Eindhoven, July 2005. [67]

[IEE96] Institute of Electrical and Electronics Engineers. *IEEE standard for a high performance serial bus*, August 1996. IEEE Std 1394-1995. [61]

[IEE05] Institute of Electrical and Electronics Engineers. *IEEE standard for high performance serial bus bridges*, July 2005. IEEE Std 1394.1-2004. [61, 119]

[ISO89] International Standards Organization. *Information processing systems – Open Systems Interconnection – LOTOS - a formal description technique based on the temporal ordering of observational behaviour*, 1989. ISO 8807:1989. [17]

[ITU00] International Telecommunication Union - Telecom Standardization. *Message Sequence Chart*, 2000. ITU-T Recommendation Z.120. [6]

[JP03] B.P.F. Jacobs and E. Poll. Java program verification at Nijmegen: developments and perspective. Report NIII-R0318, University of Nijmegen, 2003. [106, 107]

[KL98] J.-P. Katoen and L. Lambert. Pomsets for message sequence charts. In *Proceedings of the 1st workshop on SDL and MSC*, 1998. [8, 11, 13, 15, 20, 21, 29]

[KM00] E. Kindler and A. Martens. Cross-talk revisited: what's the problem? *Petri Net Newsletter*, 58:4–10, April 2000. [34]

[Kou06] J.C. Koudijs. Automated verification of Owicki/Gries proof outlines: comparing PVS and Isabelle. Master's thesis, Technische Universiteit Eindhoven, January 2006. [117]

[LL97] S. Leue and P.B. Ladkin. Implementing and verifying MSC specifications using Promela/XSpin. In *Proceedings of the 2nd SPIN workshop*, volume 32 of *DIMACS Series*, 1997. [44]

[MG05] A.J. Mooij and N. Goga. Dealing with non-local choice in IEEE 1073.2's standard for remote control. In *Proceedings of the 4th SDL and MSC workshop on System Analysis and Modeling*, volume 3319 of *LNCS*, pages 257–270. Springer, 2005. [5, 43]

[MGR05] A.J. Mooij, N. Goga, and J.M.T. Romijn. Non-local choice and beyond: intricacies of MSC choice nodes. In *Proceedings of the 8th conference on Fundamental Approaches to Software Engineering*, volume 3442 of *LNCS*, pages 273–288. Springer, 2005. [27, 43]

[MGW04] A.J. Mooij, N. Goga, and J.W. Wesselink. A distributed spanning tree algorithm for topology-aware networks. In *Proceedings of the 2nd conference on Design, Analysis, and Simulation of Distributed systems*, pages 169–178. The Society for Modeling and Simulation International, 2004. [73]

[MGWB03] A.J. Mooij, N. Goga, J.W. Wesselink, and D. Bošnački. An analysis of medical device communication standard IEEE 1073.2. In *Proceedings of the 2nd conference on Communication Systems and Networks*, pages 74–79. IASTED, ACTA Press, 2003. [5, 6, 53]

[MM01] P. Madhusudan and B. Meenakshi. Beyond message sequence graphs. In *Proceedings of the 4th conference on Fundamental Approaches to Software Engineering*, volume 2245 of *LNCS*, pages 256–267. Springer, 2001. [11, 22, 52]

[Moo02] A.J. Mooij. Formal derivations of non-blocking multiprograms. Master's thesis, Technische Universiteit Eindhoven, August 2002. Also appeared as Computer Science Report 02-13, Technische Universiteit Eindhoven, 2002. [116, 117]

[MR94] S. Mauw and M.A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4):269–277, 1994. [11]

[MRW06] A.J. Mooij, J.M.T. Romijn, and J.W. Wesselink. Realizability criteria for compositional MSC. In *Proceedings of the 11th conference on Algebraic Methodology And Software Technology*, volume 4019 of *LNCS*, pages 248–262. Springer, 2006. An earlier extended version appeared as Computer Science Report 06-11, Technische Universiteit Eindhoven, 2006. [11, 27]

[Muc02] H. Muccini. An approach for detecting implied scenarios. In *Proceedings of the 2nd workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, 2002. [38]

[Muc03] H. Muccini. Detecting implied scenarios analyzing non-local branching choices. In *Proceedings of the 6th conference on Fundamental Approaches to Software Engineering*, volume 2621 of *LNCS*, pages 372–386. Springer, 2003. [37, 38, 44]

[MW03] A.J. Mooij and J.W. Wesselink. A formal analysis of a dynamic distributed spanning tree algorithm. Computer Science Report 03-16, Technische Universiteit Eindhoven, December 2003. [81]

[MW05] A.J. Mooij and J.W. Wesselink. Incremental verification of Owicki/Gries proof outlines using PVS. In *Proceedings of the 7th International Conference on Formal Engineering Methods*, volume 3785 of *LNCS*, pages 390–404. Springer, 2005. [105]

[NPN99] T. Nipkow and L. Prensa Nieto. Owicki/Gries in Isabelle/HOL. In *Proceedings of the 2nd conference on Fundamental Approaches to Software Engineering*, volume 1577 of *LNCS*, pages 188–203. Springer, 1999. [106]

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976. [69, 71, 105]

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: a prototype verification system. In *Proceedings of the 11th Conference on Automated Deduction*, volume 607 of *LNAI*, pages 748–752. Springer, 1992. [105]

[Pau94] L.C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, 1994. [106]

[Per85] R. Perlman. An algorithm for distributed computation of a spanning tree in an extended LAN. In *Proceedings of the 9th Symposium on Data Communications*, pages 44–53. ACM, 1985. [63, 66, 76]

[Per00] R. Perlman. *Interconnections: bridges, routers, switches, and internetworking protocols*. Addison-Wesley, Amsterdam, 2000. [66]

[PN02] L. Prensa Nieto. *Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002. [106, 107, 108, 116]

[Pra86] V. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986. [8, 11, 12, 16]

[Ren99] M.A. Reniers. *Message sequence chart: syntax and semantics*. PhD thesis, Technische Universiteit Eindhoven, June 1999. [6, 21, 22]

[Rom99] J.M.T. Romijn. *Analyzing industrial protocols with formal methods*. PhD thesis, Universiteit Twente, October 1999. [61]

[Uch03] S. Uchitel. *Incremental elaboration of scenario-based specifications and behaviour models using implied scenarios*. PhD thesis, Faculty of Engineering of the University of London, February 2003. [37, 38]

[UKM01] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *Proceedings of the 8th European Software Engineering Conference*, pages 74–82. ACM Press, 2001. [38]

[UKM03] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, February 2003. [11, 44, 47]

[vLRG03] I. van Langevelde, J.M.T. Romijn, and N. Goga. Founding FireWire bridges through Promela prototyping. In *Proceedings of the workshop on Formal Methods for Parallel Programming*. IEEE Computer Society Press, 2003. [61, 66, 67, 81, 119]

[Vor04] S. Vorstenbosch. A global reset-protocol for a dynamic 1394.1 network. Internship report, Technische Universiteit Eindhoven, May 2004. [67]

[WGMS05] J.W. Wesselink, N. Goga, A.J. Mooij, and R. Spronk. Formal methods impact on ANSI standard HL7/IM: filling gaps in MSC theory. In *Proceedings of the 18th Canadian Conference on Electrical and Computer Engineering*, pages 1656–1659. IEEE, 2005. [54]

[XdRH97] Q. Xu, W.-P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, pages 149–174, 1997. [70]

Index

Summary

This research is part of the NWO project "Improving the Quality of Protocol Standards". In this project we have cooperated with industrial standardization committees that are developing protocol standards. Thus we have contributed to these international standards, and we have generated relevant research questions in the field of formal methods.

The first part of this thesis is related to the ISO/IEEE 1073.2 standard, which addresses medical device communication. The protocols in this standard were developed from a couple of MSC scenarios that describe typical intended behavior. Upon synthesizing a protocol from such scenarios, interference between these scenarios may be introduced, which leads to undesired behaviors. This is called the realizability problem.

To address the realizability problem, we have introduced a formal framework that is based on partial orders. In this way the problem that causes the interference can be clearly pointed out. We have provided a complete characterization of realizability criteria that can be used to determine whether interference problems are to be expected. Moreover, we have provided a new constructive approach to solve the undesired interference in practical situations. These techniques have been used to improve the protocol standard under consideration.

The second part of this thesis is related to the IEEE 1394.1-2004 standard, which addresses High Performance Serial Bus Bridges. This is an extension of the IEEE 1394-1995 standard, also known as FireWire. The development of the distributed spanning tree algorithm turned out to be a serious problem.

To address this problem, we have first developed and proposed a much simpler algorithm. We have also studied the algorithm proposed by the developers of the standard, namely by formally reconstructing a version of it, starting from the specification. Such a constructive approach to verification and analysis uses mathematical techniques, or formal methods, to reveal the essential mechanisms that play a role in the algorithm. We have shown the need for different levels of abstraction, and we have illustrated that the algorithm is in fact distributed at two levels. These techniques are usually applied manually, but we have also developed

an approach to automate parts of it using state-of-the-art theorem provers.

Samenvatting

Dit onderzoek maakt deel uit van het NWO project “Improving the Quality of Protocol Standards”. In dit project hebben we samengewerkt met industriële standaardisatie commissies die protocol standaarden ontwerpen. Zodoende hebben we bijgedragen aan deze internationale standaarden, en hebben we relevante onderzoeksvragen opgeleverd op het gebied van formele methoden.

Het eerst deel van dit proefschrift is gerelateerd aan de ISO/IEEE 1073.2 standaard, die gaat over de communicatie tussen medische apparaten. De protocollen in deze standaard werden ontwikkeld op basis van een aantal MSC scenario's die typisch gewenst gedrag beschrijven. Bij het samenstellen van een protocol uit zulke scenario's, kunnen verstoringen tussen deze scenario's worden geïntroduceerd die tot ongewenst gedrag leiden. Dit wordt wel het realiseerbaarheids-probleem genoemd.

Om het realiseerbaarheids-probleem te behandelen, hebben we een formeel raamwerk geïntroduceerd dat is gebaseerd op partiële ordeningen. Op deze manier kan het probleem dat de verstoringen veroorzaakt helder naar voren worden gebracht. Wij hebben een volledige karakterisering van realiseerbaarheids-criteria vastgesteld die gebruikt kan worden om te bepalen of ongewenste verstoring te verwachten is. Daarnaast hebben we een nieuwe constructieve benadering opgeleverd om in praktische situaties de ongewenste verstoring te verhelpen. Deze technieken zijn gebruikt om de door ons bestudeerde protocol standaard te verbeteren.

Het tweede deel van dit proefschrift is gerelateerd aan de IEEE 1394.1-2004 standaard, die gaat over het efficiënt verbinden van seriële communicatie-bussen door middel van bruggen. Dit is een uitbreiding van de IEEE 1394-1995 standaard, welke ook bekend staat als FireWire. De ontwikkeling van het gedistribueerde opspannende boom algoritme bleek een lastig probleem te zijn.

Om dit probleem aan te pakken, hebben we eerst een veel eenvoudiger algoritme ontworpen en voorgesteld. We hebben ook het door de ontwerpers voorgestelde algoritme bestudeerd, namelijk door een versie ervan formeel te reconstruëren, te beginnen bij de specificatie. Zo'n constructieve benadering van verificatie en analyse maakt gebruik van wiskundige technieken, oftewel formele methoden, om

de essentiële mechanismen te onthullen die een rol spelen in het algoritme. Wij hebben de behoefte aan verschillende lagen van abstractie aangetoond, en we hebben laten zien dat het algoritme eigenlijk gedistribueerd is op twee niveaus. Deze technieken worden doorgaans handmatig toegepast, maar wij hebben ook een methode ontwikkeld om delen te automatiseren met behulp van geavanceerde stelling bewijzers.

Curriculum Vitae

Arjan Mooij was born in Rotterdam on the 3rd of June 1979. He completed the pre-university education at the Carolus Borromeus College in Helmond in 1997. Afterwards he studied computer science at the Technische Universiteit Eindhoven. In 2002 he graduated *cum laude* on his master's thesis entitled "Formal derivations of non-blocking multiprograms". His master's thesis was written under supervision of Wim Feijen and it addresses the calculational design of non-blocking parallel algorithms. Subsequently he has worked on the NWO project "Improving the Quality of Protocol Standards", which has led to the present thesis.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of

Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model.* Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing.* Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes.* Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search.* Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection.* Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases.* Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics.* Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems.* Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods.* Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems.* Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems.* Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules.* Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System.* Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars.* Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic.* Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms.* Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes.* Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols.* Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor.* Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant.* Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs.* Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language.* Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Fa-

culty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14