MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Realizability of Message Sequence Graphs

DIPLOMA THESIS

Martin Chmelík

Brno, Spring 2011

Declaration

Hereby I declare, that this thesis is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

In Brno, May 26, 2011
Martin Chmelík

Advisor: RNDr. Vojtěch Řehák, Ph.D.

Acknowledgement

I would like to thank my advisor RNDr. Vojtěch Řehák, Ph.D. for help and guidance throughout the work on this thesis. Also I thank all members of the ITI laboratory, for a superior working environment. I would like to thank my family for supporting me during my studies.

This thesis was created within a joint project of ANF DATA spol. s r.o. and the research centre Institute for Theoretical Computer Science (ITI).

Abstract

In this work we focus on the realizability problem of Message Sequence Graphs, i.e., the problem whether a given specification is correctly distributable among parallel components. In the general setting, the problem is proved to be undecidable. Therefore, we search for a restricted class of Message Sequence Graphs that admits a realization, but moreover it is decidable whether a given specification is a member of the class.

We design a new class of Message Sequence Graph specifications that admits a deadlock–free realization by overloading the existing messages with additional bounded control data. We show that the presented class is the largest known class of deadlock–free realizable specifications.

Keywords

Message Sequence Charts, MSC, basic Message Sequence Chart, bMSC, Message Sequence Graph, MSG, realization, realizability, Communicating Finite State Machines, CFM.

Contents

Introduction

The Message Sequence Charts and the extended variant Message Sequence Graphs are formalisms widely used to specify message passing systems such as communication protocols and interprocess communication. As it offers both a graphical and a textual language, it is intuitive and enables the use of formal methods at the same time.

The possibility to use formal methods is in a great interest of software designers as it enables to discover potential problems and design flaws in an early development stage.

Many properties of Message Sequence Graphs can lead to difficulties when trying to implement the specification. The presence of a race condition in the specification can lead to a system deadlock. This is studied in [3, 5, 7]. Another interesting properties are boundedness of the message channels [4], the possibility to reach a non–local branching node [13], deadlocks, livelocks and many more.

There are tools supporting not only drawing Message Sequence Graphs, but are also capable of performing the verification of various properties — uBet [17], MSCan [14] and the Sequence Chart Studio [16] that is developed at Faculty of Informatics, Masaryk University.

In this work we focus on the realizability property, i.e., when the system is correctly distributable among independent parallel components. As the target architecture we use the Communicating Finite State Machines. A Message Sequence Graph specification is realized by a Communicating Finite State Machine implementation if every behavior described in the specification can be executed by the implementation and no additional behavior is added in the implementation.

It turns out, that not all the systems specified using Message Sequence Graphs are realizable. To determine whether a given system is realizable is proved to be undecidable, even if the resulting implementation is not required to be deadlock–free.

Many classes of Message Sequence Graphs that admit a realization are proposed in the literature — the bounded specifications [2], the regular specifications [15] etc.

Overloading the message content with additional control data allows us to realize rich classes of specifications — the local choice and locally–cooperative specifications [9].

In this work we define a new class of Message Sequence Graphs and provide an algorithm for deadlock–free realization of the class. Moreover, we show that the class is the largest known deadlock–free realizable class.

Chapter 1

Preliminaries

In this chapter we introduce the formalisms we are going to use throughout this work.

1.1 Message Sequence Charts

Message Sequence Charts are a textual and graphical formalism that is standardized by the International Telecommunications Union (ITU–T) as Recommendation Z.120 [10]. It is used to model interaction among parallel components in a distributed environment.

The formalism consists of two parts. The first formalism — basic Message Sequence Chart (*bMSC*) is suitable for designing finite communication patterns. The second formalism — High–level Message Sequence Chart deals with combining those patterns into more complex designs using techniques like alternation, iteration and restricted nesting. In this work we adopt the approach used in [2, 4] and use an equally expressive formalism Message Sequence Graphs (MSG).

1.1.1 basic Message Sequence Charts

A *bMSC* identifies a single finite execution of a message passing system. Processes are denoted as vertical lines — instances. Message exchange is represented by an arrow from the sending process to the receiving process.

Every process identifies a sequence of actions — sends and receives — that are to be executed in the order from the top of the diagram. The communication among the instances is not synchronous and can take arbitrarily long. Let us define the previous concept more formally.

Definition 1. a *bMSC* M is defined as a tuple $(E, <, \mathcal{P}, \mathcal{T}, P, \mathcal{C}, \mathcal{M})$ where

- E is a finite set of events.
- $<$ is a partial ordering on E called visual order.

- $\mathcal{P}$ is a finite set of processes.
- $\mathcal{T} : E \to \{\text{send}, \text{receive}\}$ is a labeling function dividing events into two types — send and receive.
- $P : E \to \mathcal{P}$ is a mapping that associates each event with a process.
- $\mathcal{C}$ is a finite set of message contents.
- $\mathcal{M} \subseteq (\mathcal{T}^{-1}(\text{send}) \times \mathcal{T}^{-1}(\text{receive}))$ is a bijective mapping, relating every send with a unique receive, such that for any $(e, f) \in \mathcal{M}$ we have $P(e) \neq P(f)$ — a process cannot send messages to itself.

Visual order $<$ is defined as the reflexive and transitive closure of

$$\mathcal{M} \cup \bigcup_{p \in \mathcal{P}} <_p$$

where $<_p$ is a total order on $P^{-1}(p)$.

A *bMSC* is first–in–first–out (*FIFO*) iff the visual order satisfies for all events e, f, e', f' and some processes p, p' the following condition

$$((e, f), (e', f') \in \mathcal{M} \wedge e <_p e' \wedge P(f) = P(f')) \Rightarrow f <_{p'} f'.$$

We will consider only FIFO *bMSCs* in this work. Examples of *bMSCs* can be seen in Figure 1.1.

Once the partial ordering of the events is defined, we can define a linearization — a total order that is an extension of the partial order $<$. We will represent linearization as a string over alphabet Σ.

$$\Sigma = \{p!q(m) \mid p, q \in \mathcal{P}, m \in \mathcal{C}\} \cup \{q?p(m) \mid p, q \in \mathcal{P}, m \in \mathcal{C}\}$$

Where $p!q(m)$ denotes the send event of message m from process p to process q. Likewise, $q?p(m)$ represents a receive event of message m by q from process p. The set of all linearizations for a given *bMSC* M is denoted as $\mathcal{L}(M)$.

Example 1. Let us consider the *bMSC* M_3 in Figure 1.1, then the linearization set for *bMSC* M_3 is equal to:

$$\begin{aligned}\mathcal{L}(M_3) = \{ \quad & p!q(a)\ q?p(a)\ p!q(b)\ q?p(b), \\ & p!q(a)\ p!q(b)\ q?p(a)\ q?p(b) \quad \}\end{aligned}$$

It turns out that specifying finite communication patterns is not sufficient for modelling complex systems. Therefore, an extension is introduced in the following section.

1.1.2 Message Sequence Graphs

Message Sequence Graphs allow us to combine *bMSCs* into more complex systems using alternation and iteration. An *MSG* is a directed graph that contains beside other nodes also two special nodes — the initial and the terminal node. The nodes of the graph are labelled with *bMSCs*. Therefore, every finite path from the initial node to the terminal node identifies a sequence of *bMSCs*. Every finite sequence of *bMSCs* can be composed into a single *bMSC* as we will show later. Hence, an *MSG* is a finite representation of an infinite set of *bMSCs*.

Definition 2. An *MSG* is defined as a tuple $G = (\mathcal{S}, \tau, s_0, s_f, L)$ where

- $\mathcal{S}$ is a finite set of states,
- $\tau \subseteq \mathcal{S} \times \mathcal{S}$ is an edge relation,
- $s_0 \in \mathcal{S}$ is an initial state,
- $s_f \in \mathcal{S}$ is a terminal state, and
- $L : \mathcal{S} \to$ MSC is a labeling function.

We assume that there is no incoming edge to s_0 and no outgoing edge from s_f. Moreover, we assume that the terminal node is reachable from every node in the graph.

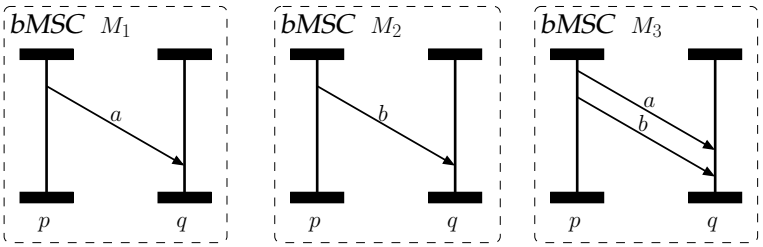

Figure 1.1: *bMSC* sequential composition

Next we formally define runs and paths in an *MSG*.

Definition 3. Given an *MSG* $G = (\mathcal{S}, \tau, s_0, s_f, L)$ a finite sequence of states $s_1, s_2, \ldots, s_k$, where $\forall\ 1 \leq i < k : (s_i, s_{i+1}) \in \tau$, is a path. a path, where $s_1 = s_0$ and $s_k = s_f$, is a run.

Given two *bMSCs* their sequential composition is a single *bMSC* intuitively created by appending actions of every process from the latter *bMSC* at the end of the process from the precedent *bMSC*. An example can be seen in Figure 1.1, where *bMSC* M_3 is a sequential composition of *bMSCs* M_1 and M_2.

Definition 4. Given two *bMSCs* $M_1 = (E_1, <_1, \mathcal{P}, \mathcal{T}_1, P_1, \mathcal{C}_1, \mathcal{M}_1)$ and $M_2 = (E_2, <_2, \mathcal{P}, \mathcal{T}_2, P_2, \mathcal{C}_2, \mathcal{M}_2)$, then their sequential composition is the *bMSC* $M_1 \cdot M_2 = ((E_1, 1) \cup (E_2, 2), <, \mathcal{P}, \mathcal{T}_1 \cup \mathcal{T}_2, P_1 \cup P_2, \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{M}_1 \cup \mathcal{M}_2)$, where $<$ is a transitive closure of $<_1 \cup <_2 \cup \bigcup_{p \in \mathcal{P}} (P_1^{-1}(p) \times P_2^{-1}(p))$.

Note, that we consider the weak concatenation, i.e., the events from the latter *bMSC* may be executed even before some events from the precedent *bMSC*.

Once we have defined paths in an *MSG* and how to concatenate *bMSCs*, we can extend the *MSG* labeling function L to paths. Let $\sigma = s_1 s_2 \ldots s_n$ be a path in *MSG* G, then

$$L(\sigma) = L(s_1) \cdot L(s_2) \cdot \ldots \cdot L(s_n)$$

Similarly we can extend the linearization set. We define $\mathcal{L}(G)$ for a given *MSG* G:

$$\mathcal{L}(G) = \bigcup_{\sigma \text{ is a run in } G} \mathcal{L}(L(\sigma))$$

Two *MSGs* are said to be language–equivalent iff they have the same linearization set.

1.2 Communicating Finite–State Machines

A natural formalism for implementing *bMSCs* are Communicating Finite–State Machines (*CFM*) that are used for example in [1, 3, 9]. The *CFM* consists of a finite number of finite–state automatons that communicate with each other by passing messages via unbounded FIFO channels.

Definition 5. For a given finite set $\mathcal{P}$, the Communicating Finite–State Machine (*CFM*) $\mathcal{A}$ consists of Finite–State Machines (*FSMs*)

$$(\mathcal{A}_p)_{p \in \mathcal{P}}.$$

Every finite–state machine $\mathcal{A}_p$ is a tuple $(\mathcal{S}_p, A_p, \rightarrow_p, s_p, F_p)$, where:

- $\mathcal{S}_p$ is a finite set of states,
- A_p is a set of actions,
- $\rightarrow_p \subseteq \mathcal{S}_p \times A_p \times \mathcal{S}_p$ is a transition relation,
- $s_p \in \mathcal{S}_p$ is the initial state, and
- $F_p \subseteq \mathcal{S}_p$ is a set of local final states.

With each pair of *FSMs* $\mathcal{A}_p, \mathcal{A}_q$ we associate an unbounded FIFO error–free channel $\mathcal{B}_{p,q}$. In every configuration the content of the channel is a finite word over a finite alphabet $\mathcal{C}$.

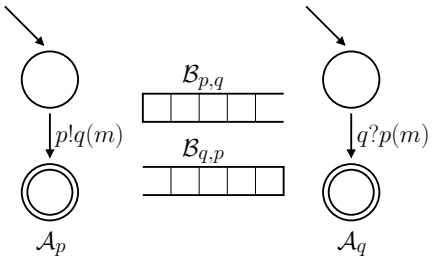

Figure 1.2: *CFM example*

Whenever an *FSM* $\mathcal{A}_p$ wants to send a message $m \in \mathcal{C}$ to $\mathcal{A}_q$, it inserts m into channel $\mathcal{B}_{p,q}$. We denote this action by $p!q(m)$. Provided there is a message m in the head of channel $\mathcal{B}_{p,q}$, the *FSM* $\mathcal{A}_q$ can receive message m from $\mathcal{A}_p$. This action is represented by $q?p(m)$. An example of a *CFM* is shown in Figure 1.2.

A configuration of a *CFM* $\mathcal{A} = (\mathcal{A}_p)_{p\in\mathcal{P}}$ is a tuple $C = (q, B)$, where $q \in \prod_{p\in\mathcal{P}}(\mathcal{S}_p)$ and $B \in (\mathcal{C}^*)^{\mathcal{P}\times\mathcal{P}}$ — local states of the *FSMs* together with the contents of the buffers. The *CFM* execution starts in an initial configuration

$$s_0 = \prod_{p\in\mathcal{P}}(s_p)$$

with all the channels empty. The *CFM* is in an accepting configuration, if every *FSM* is in some of its final states and all the channels are empty. We will say that a configuration is a deadlock, if no accepting configuration is reachable. A *CFM* is deadlock–free if no deadlock configuration is reachable from the initial configuration.

A *CFM* execution is a finite sequence of configurations $C_1, C_2, \ldots C_n$ such that C_1 is the initial configuration, C_n is an accepting configuration

and for all $1 \leq i < n$ the configuration C_{i+1} is reached from C_i by performing an action from the following alphabet

$$\Sigma = \{p!q(m) \mid p, q \in \mathcal{P}, m \in \mathcal{C}\} \cup \{q?p(m) \mid p, q \in \mathcal{P}, m \in \mathcal{C}\}.$$

We denote this fact by $C_i \xrightarrow{w} C_{i+1}$, where $w \in \Sigma$. We can label every execution $\sigma = C_1, C_2, \ldots, C_n$ with a string over alphabet Σ — linearization. The i–th letter of the linearization is the action labelling the $C_i \to C_{i+1}$ transition. For a given *CFM* $\mathcal{A}$ the set of all linearizations is denoted by $\mathcal{L}(\mathcal{A})$.

Chapter 2

Realizability of Message Sequence Graphs

In this chapter we introduce the notion of realizability. Once the design of the system specified using an *MSG* is finished, the next step is to produce a distributed implementation of the system — realize the system. For a given *MSG* we construct a *CFM* such that every execution specified in the *MSG* specification can be executed by the *CFM* and the *CFM* does not introduce any additional unspecified execution.

We will say that an *MSG* G is realized by a *CFM* $\mathcal{A}$ if $\mathcal{L}(G) = \mathcal{L}(\mathcal{A})$. However, it turns out that not all the *MSG* specifications can be realized. A natural question arises whether it is possible to algorithmically detect non–realizable specifications.

The question in our setting is introduced in [1]. The authors define two notions for realizability. The weak–realizability where the resulting *CFM* may not be deadlock–free, and the safe–realizability where the *CFM* is guaranteed to be deadlock–free. The result does not cover *MSGs* in general, but considers arbitrary finite *bMSC* set. Checking the safe–realizability can be done using a polynomial time algorithm. Surprisingly, the complexity of determining weak–realizability is co–NP–complete.

This work is extended in [2], where the authors consider a restricted class of bounded[1] *MSGs* and show that weak–realizability is undecidable. However, safe–realizability is in EXPSPACE. The author in [11] improves the result of [2] and proves that the complexity of safe–realizability is in fact EXPSPACE–complete for bounded *MSGs*. Moreover, the author shows that safe–realizability is undecidable for general *MSGs*.

As the results are quite unsatisfactory, many authors change the original setting of the problem. In [15], the *CFM* accepting condition is modified by introducing global final states. It results into deadlock–free realization class of regular *MSGs*.

Another interesting approach is to allow the *FSMs* to add some control data into outgoing messages. In [9] the *FSMs* are allowed to guess a predic-

1. For a given *MSG* there exists a bound for the number of messages in transit.

	Weak–realizability	**Safe–realizability**
Finite *bMSC* set	co–NP–complete [1]	Polynomial [1]
Bounded *MSG*	Undecidable [2]	EXPSPACE–complete [11]
MSG	Undecidable	Undecidable [11]

Table 2.1: Realizability overview

tion of its future behavior and send it within its outgoing messages. This allows a class of locally–cooperative *MSGs* to be realized, however, the *CFM* is not deadlock–free.

Definition 6. [1] An *MSG* G is weakly–realizable iff there exists a *CFM* $\mathcal{A}$, such that $\mathcal{L}(G) = \mathcal{L}(\mathcal{A})$.

Definition 7. [1] An *MSG* G is safely–realizable iff there exists a deadlock–free *CFM* $\mathcal{A}$, such that $\mathcal{L}(G) = \mathcal{L}(\mathcal{A})$.

In this chapter we describe the projection construction, which is a standard realization technique. We show that it works surprisingly well for *bMSCs*, but is not suitable for *MSGs*. We formalize the concept of adding control data into messages and show that it allows us to realize more specifications.

2.1 Projection construction

Projection construction is one of the most natural realizations. We firstly consider only realization of a single *bMSC* M.

A projection on a process $p \in \mathcal{P}$ is a sequence of events that are to be executed by process p in *bMSC* M – denoted as $M|_p$. For every $p \in \mathcal{P}$ we construct a *FSM* $\mathcal{A}_p$, that accepts a single word — $M|_p$. This construction is surprisingly powerful, and models the *bMSC* linearizations as the following proposition shows.

Proposition 1. Let M be a *bMSC*, then *CFM* $\mathcal{A} = (M|_p)_{p\in\mathcal{P}}$ is a deadlock–free realization, i.e., $\mathcal{L}(M) = \mathcal{L}(\mathcal{A})$.

The approach can be naturally extended to the *MSGs*. For every process $p \in \mathcal{P}$ and for each node $s \in \mathcal{S}$, we construct a projection *FSM* $\mathcal{A}_{p,s}$ that is accepting a single word $L(s)|_p$.

These *FSMs* are combined to get the resulting *FSM* $\mathcal{A}_p$. The initial state of $\mathcal{A}_p$ is the initial state of $\mathcal{A}_{p,s_0}$, where s_0 is the *MSG* initial node. We map the initial state of automaton $\mathcal{A}_{p,s'}$ to the final state of $\mathcal{A}_{p,s}$ whenever there

is an edge from s to s' in the *MSG* graph. The final states are final states of $\mathcal{A}_{p,s}$, where node s_f is reachable from s, without process p executing any action.

However, as can be seen in Figure 2.1, this is not necessarily a deadlock–free realization. The example can easily reach a deadlock configuration — *FSM* P chooses to execute branch M_2 and *FSM* Q executes branch M_3.

It is easy to extend the current example, in such a way, that the resulting *CFM* is even not a realization. To show this, suppose there are four processes in the *MSG*. Two of the processes exchange a message in the left branch and the remaining two processes exchange a message in the right branch. Then the realization obtained by projection construction necessarily contains an execution, where both of the messages are exchanged.

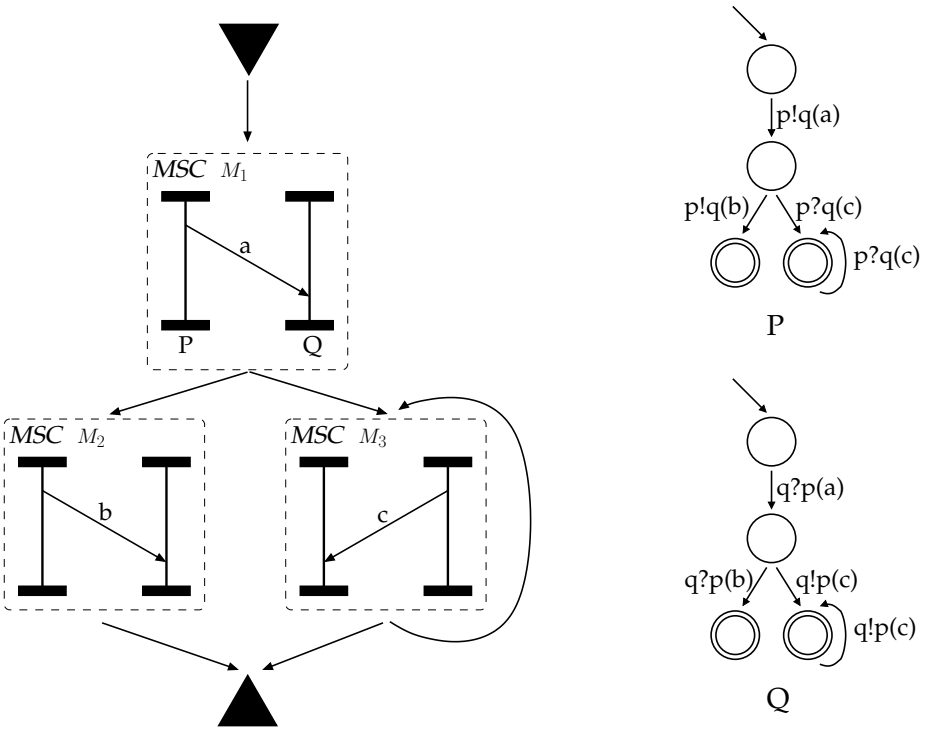

Figure 2.1: *MSG* projections

2.1.1 *bMSC* vs. *CFM*

In this section, we show that the linearizations of both formalisms have some properties in common. The alphabet we are considering in both formalisms is:

$$\Sigma = \{p!q(m) \mid p, q \in \mathcal{P}, m \in \mathcal{C}\} \cup \{q?p(m) \mid p, q \in \mathcal{P}, m \in \mathcal{C}\}$$

Definition 8. [1] a word $w \in \Sigma^*$ is well–formed iff for every prefix v of w, every receive event in v has a matching send in v.

A word $w \in \Sigma^*$ is complete iff every send event in w has a matching receive event in w.

The properties from the previous definition should be satisfied by linearizations from both our formalisms, as it should not be possible to receive a message that has not been sent and terminate with messages still in transit.

The following two lemmas show that the two formalisms can produce in some sense the same linearizations.

Lemma 1. Let $w \in \mathcal{L}(M)$ for some *bMSC* M, then $w \in \mathcal{L}(\mathcal{A})$, where $\mathcal{A} = (M|_p)_{p \in \mathcal{P}}$.

Proof. Follows directly from Proposition 1. □

Lemma 2. Let $\mathcal{A}$ be a *CFM* and $w \in \mathcal{L}(\mathcal{A})$, then there exists a *bMSC* M such that $w \in \mathcal{L}(M)$.

Proof. Every $w \in \mathcal{L}(\mathcal{A})$ is a well–formed and complete word. Using results from [1] a word w is a *bMSC* (potentially non–FIFO) linearization iff it is well–formed and complete.

So there exists a potentially non–FIFO *bMSC* M, such that $w \in \mathcal{L}(M)$. It remains to show, that the *bMSC* M satisfies the FIFO condition to fulfill our *bMSC* definition, but that follows directly from using FIFO buffers in the *CFM*. □

2.2 Additional message content

Another possible approach when realizing *MSG* specifications is to add some additional control data into outgoing messages. Using this mechanism the *FSM* can inform others about its local configuration, decision etc. This approach is used for example in [9, 15].

Consider again Figure 2.1. The deadlock is reached when both *FSMs* choose to send a message after branching. However, *FSM* P could add a prediction to its outgoing message in *bMSC* M_1, determining for both *FSMs* which branch to choose. Notice, that such construction would lead to a deadlock–free realization.

To formalize the concept we introduce a new finite alphabet $\mathcal{C}'$. The message m is no longer an element from $\mathcal{C}$, but $m \in \mathcal{C} \times \mathcal{C}'$. It is also necessary to redefine the realizability condition, as the alphabets are no longer compatible.

To do so, we define a function π that modifies the elements of the alphabet — makes a projection on the original message content. Let $p, q \in \mathcal{P}$ and $(m_1, m_2) \in \mathcal{C} \times \mathcal{C}'$, then

- $\pi(p!q(m_1, m_2)) = p!q(m_1)$
- $\pi(p?q(m_1, m_2)) = p?q(m_1)$

We extend the π function to linearizations $w \in \Sigma^*$. Let $w = w_1, w_2, \ldots, w_n$.

$$\pi(w) = \pi(w_1)\pi(w_2)\ldots\pi(w_n)$$

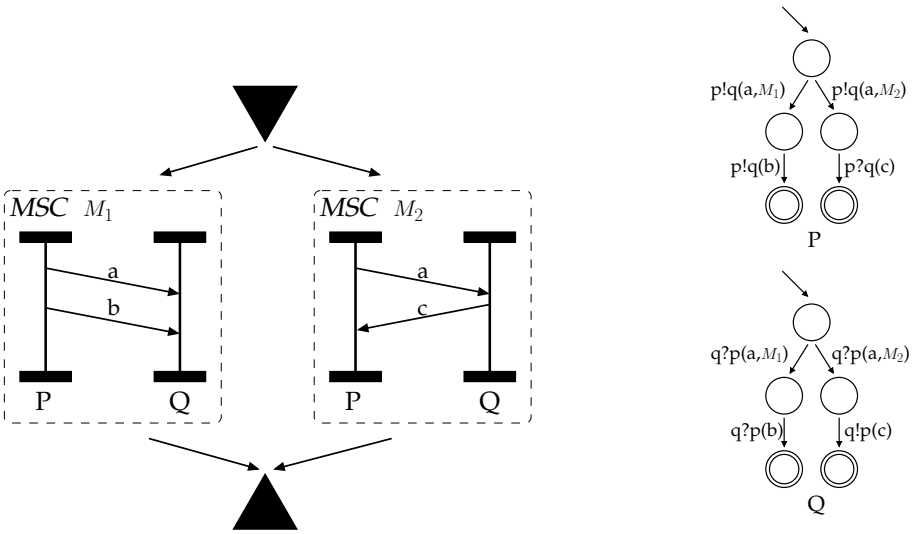

Figure 2.2: Realization, with additional data.

Finally, for a given *CFM* $\mathcal{A}$ with additional data in messages, we define the linearization set

$$\mathcal{L}'(\mathcal{A}) = \{\pi(w) \mid w \in \mathcal{L}(\mathcal{A})\}.$$

An *MSG* G is said to be weakly–realizable with additional data, if there exists a *CFM* $\mathcal{A}$, such that $\mathcal{L}(G) = \mathcal{L}'(\mathcal{A})$. It is safely–realizable if the *CFM* is deadlock–free.

A natural question is whether this construction allows us to realize more *MSG* specifications. The *MSG* in Figure 2.2 violates a necessary condition for it to be safely–realizable (see [1] for details). Therefore, the *MSG* is not deadlock–free realizable. On the other hand, a deadlock–free realization with additional data in the messages is presented in Figure 2.2.

Chapter 3

Decidable Message Sequence Graphs

It turns out that the main obstacle when realizing *MSGs* are the choice nodes — nodes with multiple outgoing edges. It is necessary to ensure that all *FSMs* choose the same run through the *MSG* graph — choose the same branch in every choice node. This can be hard to achieve as the system is distributed.

In this chapter we present a known class of local–choice *MSG* specifications that admits a deadlock–free realization with additional data. We also define a new class of decidable *MSGs* and compare the expressivity of the presented classes.

3.1 Local–choice specifications

Local–choice *MSG* specifications are a well–known class studied by many authors [8,9,12].

Firstly we introduce a few necessary terms. Let M be a *bMSC*, we say that a process $p \in \mathcal{P}$ initiates the *bMSC* M if there exists an event e in M, such that $P(e) = p$ and there is no other event e' in *bMSC* M such that $e' < e$.

For a given *MSG*, every node $s \in \mathcal{S}$ identifies a subset of processes that initiate the communication after this node — *triggers*(s). Note that it may not be sufficient to check only the direct successor nodes in the *MSG*.

Definition 9. Let $G = (\mathcal{S}, \tau, s_0, s_f, L)$ be an *MSG*. For node $s \in \mathcal{S}$ the set *triggers*(s) contains process p if and only if there exists a path $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$ in G, such that $(s, \sigma_1) \in \tau$ and p initiates *bMSC* $L(\sigma)$.

Definition 10. A choice node u is local–choice iff $|\textit{triggers}(u)| = 1$. An *MSG* specification G is local–choice iff every choice node in G is local–choice.

Local–choice specifications have the communication after every choice node initiated by a single process — the choice leader. An example of a

local–choice specification can be seen in Figure 2.2. In [9] a deadlock–free realization with additional data in messages is proposed.

Whenever a local–choice node is reached, the leader *FSM* of this choice node attaches to all its outgoing messages the information about the node it is currently executing. The rest of the *FSMs* is forwarding the information. This construction is sufficient to obtain a deadlock–free realization, for details see [9].

It is easy to see that every *MSG* G specification such that $\mathcal{L}(G) = \mathcal{L}(G')$ for some local–choice *MSG* G' is also deadlock–free realizable. Authors in [8] show that it is algorithmically decidable to determine whether a given *MSG* is language–equivalent to some local–choice *MSG*. If this is the case, an effective algorithm for constructing the local–choice *MSG* is provided.

The class of *MSG* specifications that are language–equivalent to some local–choice specifications is to the best of our knowledge the largest class of deadlock–free realizable specifications in the standard setting, i.e., FIFO channels, local final states etc.

3.2 Decidable specifications

We present a new class of *MSG* specifications — the decidable *MSGs*. The concepts are based on work presented in [6].

The difficulties when realizing *MSGs* are introduced by choice nodes. In local–choice *MSGs*, the additional message content is used to ensure a single run through the graph is executed by all *FSMs*. In case of decidable *MSGs* the additional content serves the same purpose but besides informing about the node the *FSMs* are currently executing the *FSMs* also attach a prediction about its future execution.

This allows us to relax the restriction on choice nodes and allows certain non–local choice nodes to be present in the specification. However, it is necessary to be able to resolve every occurrence of the choice node — make the decision earlier and inform all relevant processes about the prediction.

Definition 11. Let $M = (E, <, \mathcal{P}, \mathcal{T}, P, \mathcal{C}, \mathcal{M})$ be a *bMSC* and $P \subseteq \mathcal{P}$ a subset of processes. A send event $e \in E$ is a resolving event for P iff the following condition holds

$$\forall p \in P \, \exists e_p \in P^{-1}(p) : e < e_p.$$

Resolving events in M for P can distribute information to all processes from P while executing the rest of M, provided that other processes are forwarding the information.

Definition 12. Let $G = (\mathcal{S}, \tau, s_0, s_f, L)$ be an *MSG*. a choice node u is said to be decidable iff:

- For every path σ from s_0 to u there exists a resolving event in *bMSC* $L(\sigma)$ for *triggers*(u).
- For every path $\sigma = s_1 s_2 \ldots u$ such that $(u, s_1) \in \tau$, there exists a resolving event in *bMSC* $L(\sigma)$ for *triggers*(u).

Intuitively, a choice node is decidable, if every path from the initial node is labelled by a *bMSC* with a resolving event for all events initiating the communication after branching. As it is necessary to attach only bounded information the same restriction is required to hold for all cycles containing the decidable node.

In [6] we propose an algorithm that determines whether a given choice node is decidable.

Definition 13. An *MSG* specification G is decidable iff every choice node is either local–choice or decidable.

Note that there is no bound on the distance between the resolving event and the choice node it is resolving.

3.2.1 Local–choice vs. Decidable *MSGs*

In the following we show that the decidable *MSG* specifications are more expressive than local–choice specifications.

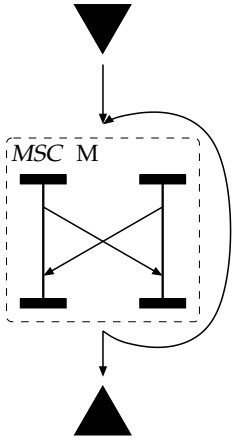

Figure 3.1: Decidable *MSG*

It is easy to see that every local–choice specification is also a decidable specification and that not every decidable *MSG* is local–choice. Moreover, it turns out that the class of *MSGs* that are language equivalent to some decidable *MSG* is more expressive than the class of *MSGs* that are language–equivalent to some local–choice specification.

Theorem 1. *The class of* MSGs *that are language–equivalent to some local–choice* MSG, *forms a proper subset of* MSGs *that are language–equivalent to some decidable* MSG.

Proof. The *MSG* specification in Figure 3.1 violates a necessary condition to be language–equivalent to some local–choice *MSG* (for details see [8]). On the other hand, the only choice node is decidable and so is the *MSG* specification. □

Chapter 4

Realizability of Decidable MSGs

In this chapter we provide an algorithm for deadlock–free realization of decidable *MSGs*.

The class of local–choice specifications admits a natural deadlock–free realization because every branching is controlled by a single process. When a choice node is reached, a unique process — the leader — chooses the continuation and informs the other processes within the existing communication.

As the *triggers* set for decidable–choice nodes can contain multiple processes, we need to ensure that all of them reach a consensus about which branch to choose.

To achieve this goal we allow the *FSMs* in certain situations add a behavior prediction to its outgoing messages. Those predictions are stored in the finite–state control units and are forwarded within the existing communication to other *FSMs*. A natural choice for a behavior prediction is a path in the *MSG* graph.

The length of the prediction should be bounded, as we can attach only bounded information to the messages and we need to store it in the finite–state control unit. Therefore, it may be necessary to generate the behavior predictions multiple times. As the realization should be deadlock–free, we must ensure that the predictions are not conflicting — generated concurrently by different *FSMs*. To solve this we sometimes send together with the prediction of the future behavior also a point where the next prediction should be generated.

Let $\mathcal{U} \subseteq \mathcal{S}$ denote the set of all decidable choice nodes for a given *MSG* and $\mathcal{L} \subseteq \mathcal{S}$ the set of all local–choice nodes.

Definition 14. A *prediction* for an *MSG* $G = (\mathcal{S}, \tau, s_0, s_f, L)$ is a pair $(\sigma, e) \in \mathcal{S}^* \times (E \cup \bot)$, where E are events from the *bMSC* $L(\sigma)$. The path σ is called a prediction path, event e is a deciding event. A prediction must satisfy one of the following conditions:

- The prediction path σ is the longest common prefix of all *MSG* runs.

This prediction path is named $initialPath$.

- The prediction path σ is the shortest path $\sigma = \sigma_1\sigma_2 \dots \sigma_n$ in G satisfying one of the following conditions

  1. $\sigma_n \in \mathcal{L}$.
  2. $\sigma_n \in \mathcal{U} \ \wedge \ \exists\, 1 \leq i < n : \sigma_i = \sigma_n$.
  3. $\sigma_n = s_f$.

To be able to easily access the first and the last *MSG* node of the prediction path $\sigma = \sigma_1\sigma_2 \dots \sigma_n$ we define functions:

$$\mathit{firstNode}(\sigma) = \sigma_1$$
$$\mathit{lastNode}(\sigma) = \sigma_n$$

Lemma 3. If the prediction path σ ends with a decidable choice node u, the *bMSC* $L(\sigma)$ contains a resolving event for $\mathit{triggers}(u)$ on $L(\sigma)$.

Proof. There are two options to consider

- In case $\sigma = \mathit{initialPath}$, then $\mathit{firstNode}(\sigma) = s_0$ and as node u is decidable, the path σ contains a resolving event for $\mathit{triggers}(u)$.
- Otherwise, the decidable node u occurs twice in the path σ. As every cycle containing a decidable node has to contain a resolving event for the node, it follows that there is a resolving event for $\mathit{triggers}(u)$ on path σ.

As there are no outgoing edges allowed in s_f, the terminal node $s_f \notin \mathcal{U}$. □

Next we show that for a given *MSG*, the size of a prediction is bounded. The number of events in a given *MSG* is finite. One can easily see that the length of the prediction path is bounded.

Proposition 2. For a given *MSG* $G = (\mathcal{S}, \tau, s_0, s_f, L)$ the length of a prediction path is bounded by $2 \cdot |\mathcal{S}|$.

When the *CFM* execution starts every *FSM* is initialized with an initial prediction — $(\mathit{initialPath}, e_i)$ — and starts to execute the appropriate projection of $L(\mathit{initialPath})$. The value of e_i depends on the *initialPath*. Let $\mathit{lastNode}(\mathit{initialPath}) = \sigma_n$. In case $\sigma_n \in \mathcal{U}$ the event e_i is an arbitrary resolving event from $L(\mathit{initialPath})$ for $\mathit{triggers}(\sigma_n)$. It follows from Lemma 3 that there exists such an event. If $\sigma_n \in \mathcal{L} \cup \{s_f\}$, we set $e_i = \perp$.

Every *FSM* stores two predictions, one that is being currently executed and a future prediction that is to be executed after the current one. Depending on the *lastNode* of the current prediction, there are multiple possibilities where to generate the future prediction.

- In case *lastNode* of the current prediction is in $\mathcal{L}$, the future prediction is generated by the local–choice leader, while executing the first event after branching.
- In case *lastNode* of the current prediction is in $\mathcal{U}$, the future prediction is generated by an *FSM* that executes the deciding event of the current prediction, while executing the deciding event.
- When the *lastNode* of the currentPrediction is s_f, no further execution is possible. Therefore, no new prediction is generated.

When a new prediction is generated we require the following condition to hold:

$$(\textit{lastNode}(\text{current prediction}), \textit{firstNode}(\text{future prediction})) \in \tau$$

Appending the future prediction at the end of the current prediction results in a path in the *MSG*.

In case an *FSM* generates a future prediction ending with a decidable choice node u, it chooses an arbitrary resolving event for $\textit{triggers}(u)$ to be the deciding event in the prediction. The existence of such an event follows from Lemma 3.

To ensure that other *FSMs* are informed about the decisions, both predictions are attached to every outgoing message. The computation ends when no *FSM* is allowed to generate any future behavior.

4.1 Algorithm

In this section we describe the realization algorithm. All the *FSMs* execute the same algorithm, an implementation of the *FSM* $\mathcal{A}_p$ is described in Algorithm 1.

We use an auxiliary function **path** that returns a prediction path for a given prediction. Every *FSM* stores a queue of events that it should execute — $eventQueue$. The queue is filled with projections of *bMSCs* labelling projection paths — $L(\text{prediction path})|_p$ for *FSM* $\mathcal{A}_p$. The execution starts with filling the queue with the projection of the $initialPath$.

Algorithm 1 Process p implementation

```
1: Variables: currentPrediction, nextPrediction, eventQueue;
2:
3: currentPrediction ← (initialPath, e_i);
4: nextPrediction ← ⊥;
5: eventQueue ←push(L(initialPath)|_p);
6:
7: while true do
8:   if eventQueue is empty then
9:     getNextNode();          {See Function 2}
10:  e ← pop(eventQueue);
11:  if e is a send event then
12:    if e is the deciding event in currentPrediction then
13:      node ← lastNode(path(currentPrediction));
14:      nextPrediction ← guessPrediction(node);
15:    send(e, currentPrediction, nextPrediction);
16:  if e is a receive event then
17:    receive(e, cP, nP);
18:    if nextPrediction = ⊥ then
19:      nextPrediction ← nP;
```

The *FSM* executes a sequence of events according to its $eventQueue$. In order to exchange information with other *FSMs*, it adds its knowledge of predictions to every outgoing message, and improves its own predictions by receiving messages from other *FSMs*.

In case the *FSM* executes a deciding event of the current prediction, it is responsible for generating the next prediction. The function **guessPrediction**(u) behaves as described in the previous section. It chooses a prediction (σ, e), such that $(u, \text{firstNode}(\sigma)) \in \tau$. If *lastNode*$(\sigma) \in \mathcal{U}$, then e is a chosen resolving event in *bMSC* $L(\sigma)$ for the *triggers* set of the *lastNode*(σ). Otherwise, we leave $e = \perp$.

If the $eventQueue$ is empty, the *FSM* runs the **getNextNode** function to determine the continuation of the execution. In case the *lastNode* of the current prediction is a decidable choice node and p is in the *triggers* set of this node, it uses the prediction from its variable $nextPrediction$ as its $currentPrediction$. The variable $nextPrediction$ is set to $\perp$.

If the *lastNode* of the currentPrediction is a local–choice node and p is the leader of the choice, it guesses the prediction and assigns it to the appropriate variables.

Function 2 getNextNode function for process p

```
1:  Function getNextNode()
2:    node ← lastNode(path(currentPrediction));
3:    if node ∈ 𝒰 ∧ p ∈ triggers(node) then
4:      currentPrediction ← nextPrediction;
5:      nextPrediction ← ⊥;
6:      eventQueue ←push(L(path(currentPrediction))|_p);
7:    else if node ∈ ℒ ∧ p ∈ triggers(node) then
8:      currentPrediction ← guessPrediction(node);
9:      nextPrediction ← ⊥
10:     eventQueue ←push(L(path(currentPrediction))|_p);
11:   else
12:     currentPrediction ← ⊥;
13:     nextPrediction ← ⊥;
14:     polling();          {See Function 3}
15: end function
```

Function 3 Polling function for process p

```
1:  Function polling()
2:    while true do
3:      if p has a message in some of its input buffers then
4:        receive(e, cP, nP);
5:        currentPrediction ← cP;
6:        nextPrediction ← nP;
7:        eventQueue ←push(L(path(currentPrediction))|_p);
8:        pop(eventQueue);
9:        return;
10: end function
```

Otherwise, the *FSM* forgets its predictions and enters a special polling state. This state is represented by the **Polling** function. Whenever the *FSM* receives a message it sets its predictions according to the message. The pop function on line 8 ensures the consistency of the $eventQueue$.

The **Polling** function is also a terminal state of the *FSM*. Hence, a successful execution is when all the *FSMs* are in the polling state having all the buffers are empty.

Chapter 5

Correctness

In this chapter we show that the class of decidable *MSGs* is deadlock–free realizable and that Algorithm 1 produces the realization. We first make a few observations of the algorithm execution.

For a given decidable *MSG* G we construct a *CFM* $\mathcal{A} = (\mathcal{A}_p)_{p\in\mathcal{P}}$ according to Algorithm 1.

Lemma 4. Let (σ, e_i) be a prediction. *FSM* $\mathcal{A}_p$ enters the **polling** function after executing $L(\sigma)$ iff

$$p \notin \mathit{triggers}(\mathit{lastNode}(\sigma)).$$

Proof. It holds for every prediction path σ that $\mathit{lastNode}(\sigma) \in \mathcal{U} \cup \mathcal{L} \cup \{s_f\}$. Note that $\mathit{triggers}(s_f) = \emptyset$ because no outgoing edge is allowed in the terminal state of an *MSG*. In case $p \in \mathit{triggers}$, then $\mathit{lastNode}(\sigma) \in \mathcal{U} \cup \mathcal{L}$ and one of the two branches in Function 2 **getNextNode** is evaluated to true and **polling** function is skipped.

□

It is not necessarily true that every *FSM* executes an event in every prediction. In fact multiple predictions can be executed by the *CFM*, while a particular *FSM* $\mathcal{A}_p$ executes the polling function and is not aware of predictions executed by other *FSMs*.

However, when a prediction path ends with a decidable choice node, all the processes in the *triggers* set are active in the prediction.

Lemma 5. Let (σ, e_i) be a prediction, such that $\mathit{lastNode}(\sigma) \in \mathcal{U}$, then

$$p \in \mathit{triggers}(\mathit{lastNode}(\sigma)) \Rightarrow L(\sigma)|_p \neq \emptyset$$

Proof. Let $lastNode(\sigma) = u$. According to Lemma 3, there exists a resolving event for $\mathit{triggers}(u)$ in the *bMSC* $L(\sigma)$. Hence, there exists an event on process p that is dependent on the resolving event, therefore $L(\sigma)|_p \neq \emptyset$. □

Another interesting observation is that it is possible to uniquely partition every *MSG* run into a sequence of prediction paths:

Proposition 3. Every run σ in G can be uniquely partitioned into a sequence of prediction paths such that $\sigma = initialPath\ w_2 \dots w_n$.

The following theorem shows that in fact it is not possible to execute simultaneously different predictions by different *FSMs*.

Theorem 2. *Let $\sigma = initialPath\ w_2\ \dots\ w_n$ such that every w_i is a prediction path. Then every* FSM $\mathcal{A}_p$ *for* $p \in$ triggers$(lastNode(w_n))$ *possesses the same future prediction (w_{n+1}, e_{n+1}), after executing the last event from $L(\sigma)$.*

Proof. We will prove the theorem by induction with respect to the length of path σ (measured by the number of prediction paths):

Base case Let the length of σ be 1, then $\sigma = initialPath$. We have to consider three options, depending on the type of the $lastNode(initialPath)$:

- Let $lastNode(initialPath) = s_f$, then *triggers*$(initialPath) = \emptyset$ and there is nothing to prove.

- Let $lastNode(initialPath) \in \mathcal{L}$, then there exists a single leader process in the *triggers* set. The *FSM* representing the leader process may choose prediction (w_2, e_2).

- The last option is that $lastNode(initialPath) \in \mathcal{U}$. Then the deciding event e_i in the initial prediction is not equal to $\perp$. The *FSM* executing the event guesses the next prediction (w_2, e_2).

  Let $p \in$ *triggers*$(initialPath))$. In case the *FSM* $\mathcal{A}_p$ is not guessing the prediction, we need to show that it receives the prediction in some of its incoming messages. As e_i is a resolving event, there exists an dependent event on process p. Let us denote the minimal of such events e_p. Then e_p is a receive event and it is easy to see that the prediction (w_2, e_2) is attached to the incoming message. Hence, every *FSM* $\mathcal{A}_p$ for $p \in$ *triggers*$(initialPath))$ has its variable $nextPrediction$ set to (w_2, e_2).

It follows from Lemma 4 that for every p not in the *triggers* set, the *FSM* $\mathcal{A}_p$ is in the polling state having its variable $nextPrediction$ set to $\perp$.

Induction step Let the length of σ be n. As in the base case, we have to consider multiple options:

- Let $lastNode(w_n) \in \{s_f\} \cup \mathcal{L}$, then the argument is the same as in the base case.
- So let $lastNode(w_n) \in \mathcal{U}$. From induction hypothesis follows that all *FSMs* $\mathcal{A}_p$ for $p \in$ *triggers*(w_{n-1}), start to execute prediction path w_n and all the others are in the polling state.

  Let $p \in$ *triggers*(w_n). We show that *FSM* $\mathcal{A}_p$ executes the projection $L(w_n)|_p$. It follows from Lemma 5 that this projection is non–empty. We have already shown that this is true for *FSMs* $\mathcal{A}_p$, such that $p \in$ *triggers*(w_{n-1}). In case $p \notin$ *triggers*(w_{n-1}), the *FSM* $\mathcal{A}_p$ is in the polling state. As it is not in the *triggers* set, its first action is a receive event. It is easy to see that the incoming message already contains the current prediction (w_n, e_n) and *FSM* $\mathcal{A}_p$ starts to execute $L(w_n)|_p$.

  The rest of the proof is similar to the base case. During the execution of the deciding event e_n a new prediction (w_{n+1}, e_{n+1}) is guessed and distributed to all *FSMs* $\mathcal{A}_p$ for $p \in$ *triggers*(w_n).

□

To show that Algorithm 1 is a deadlock–free realization of the class of decidable *MSGs* we need to show that $\mathcal{L}(G) = \mathcal{L}'(\mathcal{A})$. We will divide the proof into two parts, first showing that $\mathcal{L}(G) \subseteq \mathcal{L}'(\mathcal{A})$ and finishing with $\mathcal{L}'(\mathcal{A}) \subseteq \mathcal{L}(G)$.

5.1 $\mathcal{L}(G) \subseteq \mathcal{L}'(\mathcal{A})$

We show that for all $w \in \mathcal{L}(G)$ also holds that $w \in \mathcal{L}'(\mathcal{A})$. For every $w \in \mathcal{L}(G)$ there exists a run σ in G such that $w \in \mathcal{L}(L(\sigma))$.

We need to find a *CFM* execution, such that every *FSM* $\mathcal{A}_p$ executes the projection $L(\sigma)|_p$ and ends in a polling state with the *CFM* having all the channels empty. Then using Proposition 1, follows $\mathcal{L}(M) \subseteq \mathcal{L}'(\mathcal{A})$ and especially $w \in \mathcal{L}'(\mathcal{A})$.

According to Proposition 3 we can partition every run σ uniquely into a sequence of prediction paths — $initialPath\ w_2 \dots w_n$. This sequence is a natural candidate for prediction paths that should be guessed during the *CFM* execution.

Every *CFM* execution starts with an initial prediction $(initialPath, e_i)$. The guessed future prediction paths are $w_2, w_3 \ldots$. The guessing continues until the last prediction path w_n is executed. As σ is a run in *MSG* G, $lastNode(w_n) = s_f$. Therefore, *triggers*$(lastNode(w_n)) = \emptyset$. It follows from Lemma 4 that all the *FSMs* are in the polling state. All the channels are empty because of the well–formedness and the completeness of the *bMSCs* linearizations.

5.2 $\mathcal{L}'(\mathcal{A}) \subseteq \mathcal{L}(G)$

We show that for every $w \in \mathcal{L}'(\mathcal{A})$ also $w \in \mathcal{L}(G)$. According to Lemma 2, every $w \in \mathcal{L}'(\mathcal{A})$ identifies a *bMSC* M. To conclude this part of the proof, we find a run σ in G, such that $M = L(\sigma)$. As $\mathcal{L}(M) \subseteq \mathcal{L}(G)$ we get $w \in \mathcal{L}(G)$.

The σ run in G is defined inductively. Every *FSM* starts with executing the $initialPath$ prediction path. So it is safe to start the run σ with this prediction path.

According to Theorem 2 whenever some prediction w_i is executed, all *FSMs* $\mathcal{A}_p$ for $p \in$ *triggers*(lastNode(w_i)) agree on some future prediction w_{i+1} and all $\mathcal{A}_p$ such that p executes an event in *bMSC* $L(w_{i+1})$, execute the projections $L(w_{i+1})|_p$. All the other *FSMs* are in the polling state and are awakened only if needed.

The predictions are guessed in such a way that the following condition holds:

$$(\text{lastNode}(w_i), \text{firstNode}(w_{i+1})) \in \tau$$

So it is safe to append w_{i+1} at the end of σ. Next we show that σ ends with a terminal node. The *CFM* accepts when all the channels are empty and all the *FSMs* are in the polling state. Hence, the last prediction that was executed ended with a node with an empty triggers set. In general it is possible that this is may not be the terminal node, but every path from this node reaches s_f without executing any event. So we can safely extend σ with a path to a terminal node.

Corollary 1. Let G be a decidable *MSG*. Then the *CFM* $\mathcal{A}$ constructed by Algorithm 1 is a deadlock–free realization i.e. $\mathcal{L}(G) = \mathcal{L}'(\mathcal{A})$.

Chapter 6

Conclusion

In this work we studied the possibility of Message Sequence Graphs realizability, i.e., the possibility to make an efficient and correct distributed implementation of the specified system.

In general the problem of determining, whether a given specification is realizable is undecidable. Therefore, restricted classes of realizable specifications are in a great interest of software designers.

By allowing to attach bounded control data into existing messages, it turns out to be possible to realize specifications, that are not realizable in the standard setting.

We designed a new class of Message Sequence Graphs specifications, that admits a deadlock–free realization with additional control data in messages. Moreover, we showed that the designed class, is the largest known class of deadlock–free realizable Message Sequence Graph specifications.

Bibliography

[1] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *International Conference on Software Engineering*, pages 304–313, 2000.

[2] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.

[3] R. Alur, G.J. Holzmann, and D. Peled. An Analyzer for Message Sequence Charts. In *TACAS'96*, Lecture Notes in Computer Science, pages 35–48. Springer-Verlag, 1996.

[4] R. Alur and M. Yannakakis. Model Checking of Message Sequence Charts. *CONCUR'99: Concurrency Theory: 10th International Conference, Eindhoven, the Netherlands, August 24-27, 1999: Proceedings*, 1999.

[5] C.A. Chen, S. Kalvala, and J. Sinclair. Race Conditions in Message Sequence Charts. In *APLAS 2005: Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 195–211. Springer-Verlag, 2005.

[6] M. Chmelík. Deciding Non–local Choice in High–level Message Sequence Charts *Bachelor thesis*, Faculty of Informatics, Masaryk University, Brno, 2009.

[7] E. Elkind, B. Genest, and D. Peled. Detecting Races in Ensembles of Message Sequence Charts. In *TACAS'07*, volume 4424 of *Lecture Notes in Computer Science*, pages 420–434. Springer-Verlag, 2007.

[8] B. Genest and A. Muscholl. The Structure of Local-Choice in High-Level Message Sequence Charts (HMSC). 2002.

[9] B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-State High-Level MSCs: Model-Checking and Realizability. *Journal of Computer and System Sciences*, 72(4):617–647, 2006.

[10] ITU Telecommunication Standardization Sector Study group 17. ITU recommandation Z.120, Message Sequence Charts (MSC), 2004.

[11] M. Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theoretical Computer Science*, 309(1-3):529–554, 2003.

[12] A.J. Mooij, N. Goga, and J. Romijn. Non-local choice and beyond: Intricacies of MSC choice nodes. *Fundamental Approaches to Soft. Eng.(FASE 05)*, 2005.

[13] A. Mousavi, B. Far, and A. Eberlein. The Problematic Property of Choice Nodes in high-level Message Sequence Charts. Technical report, Technical report, Laboratory for Agent-Based Software Engineering, University of Calgary, 2006.

[14] MSCan – Message Sequence Charts analyzer. `http://aprove.informatik.rwth-aachen.de/~kern`. [Online; accessed 12-May-2011].

[15] M. Mukund, K.N. Kumar, and M. Sohoni. Synthesizing distributed finite-state systems from MSCs. *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000)*, 2000.

[16] SCStudio – Sequence Chart Studio. `http://scstudio.sourceforge.net/`, 2009. [Online; accessed 12-May-2011].

[17] UBET (formaly named MSC/POGA.). `http://cm.bell-labs.com/cm/cs/what/ubet/`. [Online; accessed 12-May-2011].