A hierarchy of communication models for Message Sequence Charts
Engels, A.G.; Mauw, S.; Reniers, M.A.

Published: 01/01/1997

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
A Hierarchy of Communication Models for Message Sequence Charts

by

A. Engels, S. Mauw and M.A. Reniers

97–11
Eindhoven University of Technology
Department of Mathematics and Computing Science

A Hierarchy of Communication Models for Message Sequence Charts

by

A. Engels, S. Mauw and M.A. Reniers

ISSN 0926-4515

All rights reserved
editors: prof.dr. R.C. Backhouse
         prof.dr. J.C.M. Baeten

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Reports 97/11
Eindhoven, August 1997
A Hierarchy of Communication Models for Message Sequence Charts

A. Engels, S. Mauw, M.A. Reniers,
Department of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{engels|sjouke|michelr}@win.tue.nl

Abstract

In a Message Sequence Chart (MSC) the dynamical behaviour of a number of cooperating processes is depicted. An MSC defines a partial order on the communication events between these processes. This order determines the physical architecture needed for implementing the specified behaviour, such as a FIFO buffer between each of the processes. In a systematic way, we define 50 communication models for MSC and we define what it means for an MSC to be implementable by such a model. Some of these models turn out to be equivalent, in the sense that they implement the same class of MSCs. After analysing the notion of implementability, only ten models remain, for which we develop a hierarchy.

1 Introduction

In recent years much attention has been paid to graphical languages for the visualisation of communication traces in distributed systems. One of the most popular classes of formalisms for this purpose is the class of sequence charts. Of those, Message Sequence Chart (MSC) has been standardised by the International Telecommunication Union (ITU) as Recommendation Z.120 [IT96]. Two important reasons for the popularity of MSC are that they provide a clear intuition to both engineers and designers and at the same time posses a well-defined semantics.

Although MSC is primarily concerned with presenting the asynchronous communication between processes in a distributed system, no information is given as to the way in which these communications are supposed to be realized in an implementation. The only assumption about the implementation of communication is that an output precedes its corresponding input.

This impossibility to specify the communication model becomes a problem when a specific communication model is presupposed, for example due to hardware requirements. Whenever MSC is used to specify the communication behaviour, the question arises whether the behaviour defined by an MSC is feasible with respect to the desired communication model. It may be the case that all traces defined by the MSC are feasible, that at least one trace is, or that none of the traces is feasible. For example, an MSC with two inherently crossing messages cannot be implemented with an architecture containing one single global FIFO buffer for message exchange.

There are two approaches to deal with this under-specification in MSC. The first is to select a single preferred model and revise the semantics of MSC accordingly. Keeping in mind the broad context in which MSC is used in practice, this option is not realistic. The only acceptable choice would be the
most general random-access buffer model that has been chosen in the current standardised semantics of MSC.

The alternative would be to allow the user of MSC to indicate the desired communication model explicitly. This can be done by extending the syntax of MSC with a means to specify the intended model and by developing dedicated tools for the analysis of MSC with respect to certain implementation models. We propose to study this second alternative and it is our aim in this paper to provide a solid and formal basis for defining the relation between a communication model and an MSC.

For a given MSC we define the notions of strong and weak implementability. Strong implementability of an MSC in a given communication model means that all traces of the MSC can be realized with the given communication model and weak implementability means that there is a trace that can be realized.

In this way, we attach to each implementation model the class of MSCs that are strongly or weakly implementable with respect to that model. A natural question to ask is whether there are communication models that define the same class of MSCs. This means that for a given MSC one has a choice of communication model for implementation. It turns out that the initial number of fifty MSC classes can be reduced to a hierarchy of ten different models.

Acknowledgements We would like to thank Thijs Cobben, Loe Feijs, Herman Geuvers and Bart Knaack for their valuable input.

2 Message Sequence Charts

In this section we explain the semantical foundations of Message Sequence Chart (MSC). We use a partial order on the events of an MSC to express the semantics. In literature several ways to define the semantics of MSC are proposed [MR94a, LL95, GGR93]. The process algebra approach [MR94b] has been standardised as Annex B to ITU recommendation Z.120 [IT95]. The partial order representation [AHP96] used in this paper coincides with most of these proposals for the class of Basic Message Sequence Charts. We also define the traces expressed by an MSC.

2.1 Basics

The MSCs studied here consist of a collection of instances (or processes) with a number of messages attached to them. These are known as Basic Message Sequence Charts, but in this paper we use the term MSCs to denote them.

Some examples of MSCs can be seen in Figure 1. They consist of vertical lines, denoting the various communicating processes, which we call 'instances' and arrows between these instances, denoting exchanged messages.

We allow messages from an instance to itself, but we only consider closed systems, that is, we do not consider messages to the environment. Neither do we consider any other specific features such as local actions and recursion. We assume that the names of the instances and messages are unique. Therefore, the instances to which a message is attached are determined uniquely by the message name.

The easiest way to express the semantics of such a simple MSC is by using a partial order on the events that are comprised in an MSC. Depending on the particular dialect of the MSC language, one can assign different classes of events to an MSC. For example, in Interworkings [MWW93] every message is considered to be a single event. There is no buffering, and thus communication is synchronous.

In MSC [IT96], messages are divided into two events, the output and the input of the message. The output of message $m$ is denoted by $!m$ and the input by $?m$. The only assumption about the implemen-
In this paper we go one step further, and add a third event, denoted by $!!m$, that we call transmit $m$. The basic idea is that a message passes two buffers before arriving at its destination. The intuition here is that $!m$ denotes the putting of a message into an output buffer, $!!m$ is the transmission of the message from the output buffer to the appropriate input buffer, and $?m$ is the removal of the message from the input buffer. We assume these events to be instantaneous. Furthermore, we concentrate on FIFO-buffers only.

Although the intermediate transmit events $!!m$ play a crucial role in our description of the communication models, we do not encounter them in the definition of an MSC, nor in the partial order describing the formal semantics of an MSC. An MSC still describes a partial order on output and input events only.

**Definition 1 (MSC)** An MSC is a quintuple $(I, M, from, to, \{<_i\}_{i \in I})$, where $I$ is a finite set of instances, $M$ is a finite set of messages, from and to are functions from $M$ to $I$, and $\{<_i\}_{i \in I}$ is a family of orders. For each $i \in I$ it is required that $<_i$ is a total order on $\{!m \mid from(m) = i\} \cup \{?m \mid to(m) = i\}$.

In the above definition, $from(m)$ denotes the instance which sends message $m$. Likewise, $to(m)$ denotes the instance which receives message $m$. Given an instance $i$, the order $<_i$ denotes in which order the events attached to instance $i$ occur. The order $<_i$ is lifted in the trivial way to the set $\{!m, ?m, !!m \mid m \in M\}$.

The partial order denoting the semantics of an MSC is derived from two requirements. First, the order of the events per instance is respected, and second, a message can only be received after it has been sent. The first requirement is formalised by defining the partial order $<_\text{inst}$:

$$<_\text{inst} := \bigcup_{i \in I} <_i,$$

and the second requirement is formalised by the output-before-input order $<_\text{oi}$:

$$<_\text{oi} := \{(!m, ?m) \mid m \in M\}.$$

Now, we define the partial order induced by the MSC as the transitive closure (denoted by $^+$) of the instancewise order and the output-before-input order. For an MSC $k$, we denote this order by $<_k^{\text{msc}}$ or by $<_k^{\text{msc}}$ if $k$ is known from the context.
Definition 2 For a given MSC $k = \langle I, M, \text{from}, \text{to}, \{<_i\}_{i \in I} \rangle$, the relation $<_k^{\text{msc}}$ is defined by $<_k^{\text{msc}} := (<_i^{\text{inst}} \cup <_i^{\text{ot}})^+$. We define similar notions for 3-traces. We define the output-before-transmit-before-input order by

\[<_i^{\text{ot}} := \{(\text{!}m, \text{!}m), (\text{!!}m, ?m) \mid m \in M\},\]

and the relation $<_k^{\text{m3}}$ by adding the instancewise ordering on the MSC.

Definition 3 For a given MSC $k = \langle I, M, \text{from}, \text{to}, \{<_i\}_{i \in I} \rangle$, the ordering $<_k^{\text{m3}}$ is defined by $<_k^{\text{m3}} := (<_i^{\text{inst}} \cup <_i^{\text{ot}})^+$. It is easy to see that $<_k^{\text{msc}}$ is the restriction of $<_k^{\text{m3}}$ to output and input events. It may be the case that $<_k^{\text{msc}}$ does not define a partial order, due to cyclic dependencies of the events. Such an MSC is said to contain a deadlock, or is called inconsistent. In Z.120 [IT96] inconsistent MSCs are considered illegal, and in [BAL97] an algorithm is described for determining whether a given MSC is consistent. In the remainder of this paper we consider consistent MSCs only, which implies that both $<_k^{\text{msc}}$ and $<_k^{\text{m3}}$ are partial orders.

2.2 Traces

From an operational point of view, one can say that an MSC describes a set of traces. We distinguish 2-traces and 3-traces. A 2-trace denotes the ordering of output and input events ($\text{!}m$ and $?m$), a 3-trace those of transmit events ($\text{!!}m$) as well.

Definition 4 (2- and 3-traces) A 2-trace $t$ over a set of messages $M$ is a total ordering $(e_1, e_2, \ldots, e_n)$ of the set $\{\text{!}m, ?m \mid m \in M\}$. From now on we use the shorthand $\{\text{!}/?\} M$ to denote this set. A trace $(e_1, e_2, \ldots, e_n)$ is denoted $e_1 e_2 \ldots e_n$.

A 3-trace is the same as a 2-trace, except for the fact that it contains transmit events as well. We denote the $i$th element of a trace $t$ by $t_i$, and its length by $|t|$.

Definition 5 (Trace order) For a trace $t$ over a set of messages $M$ we define an order $<_t^{\text{trace}}$ on $\{\text{!}/?\} M$ (or $\{\text{!}!/?\} M$ if $t$ is a 3-trace), for all $1 \leq i \leq |t|$ and $1 \leq j \leq |t|$ by $t_i <_t^{\text{trace}} t_j \iff i < j$. Thus, an event $e_i$ is smaller, according to $<_t^{\text{trace}}$, than an event $e_j$ if and only if it occurs earlier in the trace.

Definition 6 (MSC-trace) A 2-trace $t$ is said to be a trace of the MSC $k$ if and only if it is defined over the messages $M$ of $k$, and $<_k^{\text{msc}} \subseteq <_t^{\text{trace}}$. A 3-trace $t$ is said to be a trace of the MSC $k$ if and only if it is defined over the messages $M$ of $k$, and $<_k^{\text{m3}} \subseteq <_t^{\text{trace}}$. A 3-trace can be turned into a 2-trace by removing all transmit events ($\text{!}m$). If, for a 3-trace $t$ this results in a 2-trace $t'$, then $t$ is said to be an extension of $t'$. It is not hard to see that a 3-trace $t$ is a trace of an MSC $k$ if and only if the 2-trace of which it is an extension is a trace of the MSC and additionally the output-before-transmit-before-input order is respected: $<_t^{\text{ot}} \subseteq <_t^{\text{trace}}$.

Lemma 7 For an MSC $k$ over $M$, and events $e, e' \in \{\text{!}/?\} M$, we have $e <_t^{\text{trace}} e'$ for all 2-traces $t$ of $k$ if and only if $e <_k^{\text{msc}} e'$.

Proof The 'if'-part is trivial. For the 'only if'-part we use contraposition. Suppose that $e \not<_k^{\text{msc}} e'$. Then the relation $<_k^{\text{msc}} \cup \{(e', e)\}$ does not contain a cycle. Thus it can be extended to a total order $\langle$. Because $<_k^{\text{msc}} \subseteq \langle$, $\langle$ will be the trace-order $<_t^{\text{trace}}$ of some trace $t$ of $k$. In this trace we will have $e' <_t^{\text{trace}} e$, and thus $e \not<_t^{\text{trace}} e'$. □
2.3 Examples

Figure 1 shows two examples of a Message Sequence Chart. The first example is an inconsistent MSC. Instance \( i \) cannot send message \( b \) before it receives message \( a \). Likewise, instance \( j \) starts with receiving message \( b \) before sending \( a \). Therefore, this MSC has a cyclic dependency and there is no trace that it can perform. In the partial order semantics that we defined above, this is shown by the fact that \( <^{\text{msc}} \) has a cycle, and thus is not a partial order.

The second example implies the following orderings: \( !a <^{\text{msc}} ?a, !b <^{\text{msc}} ?b \), and \( ?a <^{\text{msc}} ?b \). The first two are implied by the \( <^{\text{oi}} \)-order, the third by the \( <^{\text{ins}} \)-order. The MSC has exactly three 2-traces: \( !a ?a !b ?b, !a !b ?a ?b \), and \( !b !a ?a ?b \). These 2-traces can be extended to ten 3-traces, such as \( !a !a ?a !b !b ?b \) and \( !a !b !b !b !a ?a ?b \).

3 Implementation models

In this section we discuss possible architectures for realizing an MSC. We consider only implementation models consisting of FIFO buffers for the output and input of messages. For MSC traces, we define what it means to be implementable on some architecture.

3.1 Locality of buffers

The particular implementation models which we are interested in are constructed of processes that communicate with each other via FIFO buffers. We assume that the buffers have an unbounded capacity. We discern two uses of buffers, namely for the output and for the input of messages.

A second distinction can be made based on the locality of the buffer. From most global to most local we distinguish the following types:

- **global**: A global FIFO buffer: All messages from all instances pass this buffer.
- **inst**: A FIFO buffer, local to an instance: All messages sent (or received) by one single instance go through the same buffer.
- **pair**: A FIFO buffer, local to two instances: All messages that are sent from one specific instance to another specific instance go through this buffer.
- **msg**: A FIFO buffer, local to a message: There is one buffer for every message.

This last model, a buffer per message, is a specific architecture to catch up the cases in which the buffers do not behave like FIFO queues, but as random-access buffers. Taking into account the assumption that messages are unique, it can easily be seen that it is equivalent to a global random-access buffer. A communication model with only a random-access buffer represents the implied model of the MSC standard: the only assumption made about the implementation of communication is that output precedes input, no more, and no less.

Finally, we consider the following possibility:

- **nobuf**: There are no buffers; communication is synchronous.

We assume that the transmission from an instance to its output buffer, from one buffer to another buffer, or from an input buffer to the instance it belongs to, is synchronous. We also assume that all output buffers are of the same type, and similarly that all input buffers are of the same type. This results in four possibilities for the output as well as for the input. Adding the possibility of using no buffer at
all, we have a total of 25 possible architectures, as shown in Figure 2. To denote the elements of this scheme, we use the notation \((X,Y)\), where \(X\) denotes the type of output buffer, and \(Y\) the type of input buffer.

<table>
<thead>
<tr>
<th></th>
<th>nobuf</th>
<th>global</th>
<th>inst</th>
<th>pair</th>
<th>msg</th>
</tr>
</thead>
<tbody>
<tr>
<td>nobuf</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>global</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>inst</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>pair</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>msg</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Figure 2: Implementation models.

3.2 Examples of communication models

In Figure 3 we give examples of a physical architecture of three communication models. A circle denotes an instance, a rectangle denotes a buffer, and an arrow denotes a communication channel. Each example contains three instances. The first example illustrates the \((\text{nobuf,global})\) model. There is no output buffer, and one universal input buffer. As there is no output buffer, the messages go straight into the input buffer. This single buffer could be regarded as an output buffer as well, so this example is an illustration of \((\text{global,nobuf})\) too. The second example shows the \((\text{global,inst})\) model. There is one general output buffer and every instance has a local input buffer. The third architecture is an example of the \((\text{pair,pair})\) model.

Figure 3: Some models: \((\text{nobuf,global}), (\text{global,inst})\) and \((\text{pair,pair})\).

Please note that not all models described in Figure 2 make equally sense. For example, the model \((\text{global,inst})\) (i.e., a shared medium for transmitting messages and an input buffer for each process) is more natural than the exotic \((\text{global,pair})\) model.

Many of these architectures occur in practice as either the underlying communication architecture of a programming language or as a physical architecture. We give some examples of languages. The model \((\text{nobuf,nobuf})\) is typical for process algebraic formalisms based on synchronous communication, such as LOTOS and ACP. The specification language SDL, which is closely related to MSC, has as a general communication model \((\text{pair,msg})\), but if we leave out the \textit{save} construct we obtain \((\text{pair,inst})\) and if we also do not consider the possibility of delayed channels, we have \((\text{nobuf,inst})\). Some examples of physical architectures are: an asynchronous complete mesh has a \((\text{nobuf,pair})\) architecture, and an Ethernet connection with locally buffered input and output behaves like \((\text{inst,inst})\).
3.3 Implementability

The main question of this paper is, whether a given MSC can be the behaviour of a given implementation model. To answer this question, we first give a formal definition of what it means for a trace to have a certain implementability property. The definitions below can be seen as a formalisation of the notions introduced in Section 3.1.

Definition 8 (Output-implementability)

- **nobuf-output**: Every output event is directly followed by the corresponding transmit event. Thus, output and transmit event may be combined into one new event. A 3-trace $t$ is *nobuf-output* implementable if and only if

$$\forall m \in M \; \exists e \in \{!/?\} \; !m <_t e <_t !!m.$$ 

- **global-output**: The order of two output events is respected by the corresponding transmit events. A 3-trace $t$ is *global-output* implementable if and only if

$$\forall m, m' \in M \; !m <_t m' \iff !!m <_t !!m'.$$

- **inst-output**: The order of any two output events from the same instance is respected by the corresponding transmit events. A 3-trace $t$ is *inst-output* implementable if and only if

$$\forall m, m' \in M \; \text{from}(m) = \text{from}(m') \Rightarrow (!m <_t m' \iff !!m <_t !!m').$$

- **pair-output**: The order of two outputs with the same source and the same destination, is respected by the corresponding transmit events. A 3-trace $t$ is *pair-output* implementable if and only if

$$\forall m, m' \in M \; \text{from}(m) = \text{from}(m') \land \text{to}(m) = \text{to}(m') \Rightarrow (!m <_t m' \iff !!m <_t !!m').$$

- **msg-output**: A 3-trace $t$ is always *msg-output* implementable.

For *msg-output* implementability we can remark that it can be put in line with the three definitions preceding it, by restating it as $\forall m, m' \in M \; m = m' \Rightarrow (!m <_t m' \iff !!m <_t !!m').$ For *nobuf-output* implementability such a translation is not possible; this is qualitatively another definition.

The input implementabilities are defined analogously.

Definition 9 (Input-implementability)

- **nobuf-input**: A 3-trace $t$ is *nobuf-input* implementable if and only if

$$\forall m \in M \; \exists e \in \{!/?\} \; e <_t m <_t !m.$$ 

- **global-input**: A 3-trace $t$ is *global-input* implementable if and only if

$$\forall m, m' \in M \; !m <_t m' \iff ?m <_t ?m'.$$
• **inst-input:** A 3-trace \( t \) is *inst-input* implementable if and only if
\[
\forall m, m' \in M \quad \text{to}(m) = \text{to}(m') \Rightarrow (!m \xleftarrow{\text{trace}} !m' \leftrightarrow ?m \xleftarrow{\text{trace}} ?m').
\]

• **pair-input:** A 3-trace \( t \) is *pair-input* implementable if and only if
\[
\forall m, m' \in M \quad \text{from}(m) = \text{from}(m') \wedge \text{to}(m) = \text{to}(m') \Rightarrow (!m \xleftarrow{\text{trace}} !m' \leftrightarrow ?m \xleftarrow{\text{trace}} ?m').
\]

• **msg-input:** A 3-trace \( t \) is always *msg-input* implementable.

Having defined formally the notions of output- and input-implementability, we now combine them and obtain our notion of communication model.

**Definition 10** A 3-trace is said to be *(X,Y)*-implementable (for \( X, Y \in \{ \text{nobuf, global, inst, pair, msg} \} \)) if and only if it is \( X \)-output implementable and \( Y \)-input implementable. A 2-trace is said to be *(X,Y)*-implementable if and only if it can be extended (by adding \(!m\)’s) to a 3-trace that is *(X,Y)*-implementable.

### 4 Classification of implementability of traces

To each of the implementation models defined in the previous section we can associate the set of all traces that are implementable in the model. Based on the subset relation on these sets of traces, we can order implementation models. We consider two models equivalent if they have the same set of implementable traces.

In Lemma 11 we give a classification of the notions of output-implementability. It states that a trace that is implementable on a certain architecture is also implementable on an architecture where these buffers are partitioned into buffers with a more restricted locality. For example, if a trace can be implemented on an architecture with one output buffer per instance, it can also be implemented on an architecture with an output buffer per pair of instances (provided the input buffers remain the same).

**Lemma 11 (Classification of output-implementability)**

• Every *nobuf*-output implementable trace is *global*-output implementable.

• Every *global*-output implementable trace is *inst*-output implementable.

• Every *inst*-output implementable trace is *pair*-output implementable.

• Every *pair*-output implementable trace is *msg*-output implementable.

**Proof** For 3-traces this follows directly from the definitions. For 2-traces this follows from the definition plus the fact that it holds for 3-traces.

The following lemmas give the orderings between the implementation models.

**Lemma 12**

• Every *(global,global)*-implementable 2-trace is *(global,nobuf)*-implementable.

• Every *(pair,pair)*-implementable 2-trace is *(pair,nobuf)*-implementable.
From (m) have m.
The 3-trace implementable, !m = ?m = ?m'.

Equally true. Next, we describe how the above lemmas are useful in ordering the models. Lemma 12 and Lemma 13, together with the order provided by Lemma 11, give us the equivalences as expressed in Figure 4 by means of the clustering of implementation models.

For example, the models from the last column are equivalent. This can be seen as follows. Because of the analogue of Lemma 12, any (msg, msg)-implementable 2-trace is (msg, msg) implementable, while Lemma 11 gives that any (nobuf, msg)-implementable 2-trace is (X, msg)-implementable, and every (X, msg)-implementable 2-trace is (msg, msg)-implementable.

Lemma 13 Every (inst, global)-implementable 2-trace is (msg, nobuf)-implementable.

Proof Let t be a 2-trace over the set of messages M, and let t' be a 3-trace that is an (inst, global)-implementable extension of t. If suffices to construct an (inst, nobuf)-implementable extension t'' of t. The 3-trace t'' is obtained from t by adding the transmit event !m immediately before ?m for each message m ∈ M. This t'' is nobuf-input implementable by definition, so it suffices to prove that t'' is inst-output implementable. Thereto, let m, m' ∈ M such that from(m) = from(m'). We have to prove that !m t''(m) m' m. Then, since t'' is pair-output implementable, we have m t''(m) m'. Because t'' is pair-input implementable, we have ?m t''(m) m''. So ?m t''(m) m'. Then, since t'' is nobuf-input implementable, we have m t''(m) m'. Second, suppose !m t''(m). Since t'' is nobuf-implementable, we have ?m t''(m). Therefore, ?m t''(m). Since t'' is pair-output and pair-input implementable we have !m t''(m). Then, !m t''(m), and since t'' is an extension of t also !m t''(m).

For the previous lemmas the analogue obtained by switching output buffers and input buffers is equally true. Next, we describe how the above lemmas are useful in ordering the models. Lemma 11 provides us with a partial ordering on the various implementations: Any (X, Y)-implementable trace is implementable by all implementation models located to the right of or below (X, Y) in Figure 2. Lemma 12 and Lemma 13, together with the order provided by Lemma 11, give us the equivalences as expressed in Figure 4 by means of the clustering of implementation models.
Now we have brought down the number of implementation models to only seven different classes. Of course some of these could still be equivalent for other reasons than the above lemmas. That this is not the case, will be seen in Corollary 18 below. We name the equivalence classes as follows: nobuf, global, inst.out, inst.in, inst2, pair, msg (see Figure 4). Of these the first two and last two will be clear immediately, inst.out means there is an instancewise output buffer and a global or no input buffer, inst.in means there is an instancewise input buffer and a global or no output buffer, and inst2 means there is both an instancewise output buffer and an instancewise input buffer.

Figure 4: Equivalence of implementation models for traces.

Theorem 14 For traces, the seven implementation models are ordered as shown in Figure 5.

Proof This follows from the Lemmas 11 to 13 as explained above.

Note that of these seven cases only inst2 is not of the form \((X, \text{nobuf})\) or \((\text{nobuf}, X)\). As these forms imply that there is respectively no input buffer or no output buffer, of these seven cases only the case inst2 needs two buffers, all other cases can be modelled such that each message goes through at most one buffer. It will prove useful to have a characterisation of these implementabilities (except for inst2 of course) that does not use transmits.
Lemma 15 Let $t$ be a 2-trace over a set of messages $M$. Then

- $t$ is nobuf-implementable if and only if
  \[ \forall m \in M \exists e \in \{!,?\}^M \; \text{trace} \; e \; \text{trace} \; m; \]

- $t$ is global-implementable if and only if
  \[ \forall m, m' \in M \; \text{trace} \; m \; \Rightarrow \; \text{trace} \; m'; \]

- $t$ is inst.out-implementable if and only if
  \[ \forall m, m' \in M \; \text{from}(m) = \text{from}(m') \Rightarrow (\text{trace} \; m \; \Rightarrow \; \text{trace} \; m'); \]

- $t$ is inst.in-implementable if and only if
  \[ \forall m, m' \in M \; \text{to}(m) = \text{to}(m') \Rightarrow (\text{trace} \; m \; \Rightarrow \; \text{trace} \; m'); \]

- $t$ is pair-implementable if and only if
  \[ \forall m, m' \in M \; \text{from}(m) = \text{from}(m') \land \text{to}(m) = \text{to}(m') \Rightarrow (\text{trace} \; m \; \Rightarrow \; \text{trace} \; m'); \]

- $t$ is always msg-implementable.

Proof The proofs for this are easily found by realizing that a 2-trace is ($X$,nobuf)-implementable exactly if the conditions for $X$-output implementability hold with $!!m$ everywhere replaced by $?m$. 

5 Classification of MSCs

There are two principal ways to lift the definition of implementability from the level of traces to the level of MSCs. The first is to define that an MSC can be implemented in a certain communication model if and only if every 2-trace of the MSC can. The second is to define that an MSC can be implemented in a certain implementation model if and only if some 2-trace can. We call these notions strong and weak implementability. We first focus on the strong implementability, then on weak implementability. After this we consider the relation between classes from the strong and the weak spectrum.

5.1 Strong implementability

Definition 16 An MSC $k$ is said to be strongly $X$-implementable, notation $X_i$-implementable, if and only if all 2-traces $t$ of $k$ are $X$-implementable.

From this definition it follows immediately that the ordering of the implementation models for traces as given in Figure 5 also holds for MSCs as far as strong implementability is concerned (see Figure 7). Next, we demonstrate that the implementation models, obtained by lifting them from the trace level to MSCs in the strong way, are indeed different. This is achieved by finding examples of MSCs that are in one class but not in another.
MSC 1 in Figure 6 shows an example that is \textit{global}-implementable, but not \textit{nobuf}-implementable. It is not \textit{nobuf}-implementable, because the trace \(!a \ldots b \ ?a \ ?b\) is not. The inputs necessarily have to be ordered in the same way as the outputs, so it is \textit{global}-implementable.

MSC 2a is \textit{inst.out}-implementable, but not \textit{global}-implementable due to the trace \(!b \ !a \ ?a \ ?b\). That MSC 2a is \textit{inst.out}-implementable can be seen as follows: All messages go through a different output buffer, so there is no problem with the output buffers at all. Similarly, MSC 2b is \textit{inst.in}-implementable, but not \textit{global}-implementable due to the trace \(!a \ !b \ ?b \ ?a\).

MSCs 2a and 2b show the difference between \textit{inst.out}, and \textit{inst.in}. MSC 2a is \textit{inst.out}-implementable, as mentioned before, but not \textit{inst.in}-implementable. The trace \(!b \ !a \ ?a \ ?b\) is not \textit{inst.in}-implementable, because the inputs of instance \(j\) do not reach the input buffer in the order in which they are to be manipulated. For MSC 2b the reverse is the case: It is \textit{inst.in}-implementable, but not \textit{inst.out}-implementable. MSC 2a is \textit{inst.out}-implementable and therefore also \textit{inst2}-implementable. We have already established that it is not \textit{inst.in}-implementable. Similarly, MSC 2b is \textit{inst.in} and \textit{inst2}-implementable, but not \textit{inst.out}-implementable. Together, these show that \textit{inst.out}, \textit{inst.in}, and \textit{inst2} are all different. One might suspect that the class of \textit{inst2}-implementable MSCs is simply equal to the intersection of the classes of \textit{inst.out}-implementable and \textit{inst.in}-implementable MSCs. This is not the case, as can easily be shown by combining the MSCs 2a and 2b into one MSC (see MSC 8 in Figure 12).

MSC 3 is an example of an MSC that is \textit{pair}-implementable, but not \textit{inst2}-implementable. It is easy to see that it is \textit{pair}-implementable, because each message goes through a different buffer. Its only 2-trace is \(!c \ !a \ ?a \ ?b \ ?c\). If we try to extend this to an \textit{inst2}-implementable 3-trace \(t\)', we need to have \(!c <_{t}^{\text{trace}} !a <_{t}^{\text{trace}} !b <_{t}^{\text{trace}} !c\), which is impossible (the first \(<_{t}^{\text{trace}}\) is because of the \textit{inst.out} output implementability and \(!c <_{t}^{\text{trace}} !a\), the second is clearly true for every 3-trace of the MSC, and the third is because of the \textit{inst.input} implementability together with \(!b <_{t}^{\text{trace}} !c\)).

Finally, MSC 4 shows the difference between \textit{pair}- and \textit{msg}-implementability. All other implementation models are also pairwise different. This result is obtained due to the transitive closure of the ordering as presented in Figure 7.

Together the examples used in the above proof show that if we look at strong implementability, the seven remaining implementation models are indeed different for MSCs, and thus that they are also different for 2-traces.

**Theorem 17** The implementation models for strong implementability of Figure 7 are different and these are ordered as expressed in Figure 7.

**Proof** In the above text we have demonstrated by means of counterexamples that the implementation models must be different. Also the ordering has been explained above.

**Corollary 18** The classes \textit{nobuf}, \textit{global}, \textit{inst.out}, \textit{inst.in}, \textit{inst2}, \textit{pair}, and \textit{msg} are different for traces.
5.2 Weak implementability

Definition 19 An MSC $k$ is said to be weakly $X$-implementable, notation $X_w$-implementable, if and only if there is an $X$-implementable 2-trace $t$ of $k$.

As was the case for strong implementability, for weak implementability we also have the ordering as expressed in Figure 5 as a starting point. However, using weak implementability, we do not have anymore that all implementation models differ. To see this, we first give an alternative way to characterise some of the implementations and prove that these are equivalent to the original definition.

Definition 20 Let $k$ be an MSC over the set of messages $M$. Then we define the relations $<^i_{k}$ and $<^i_{k}$ on $\{!/?\}M$ and $<^2_{k}$ on $\{!/?\}M$ as follows:

$<^i_{k}:= (<^m_{k} \cup \{(m, m') \mid m, m' \in M \land \text{from}(m) = \text{from}(m') \land m <^m_{k} m'\})^+$,

$<^j_{k}:= (<^m_{k} \cup \{(m, m') \mid m, m' \in M \land \text{to}(m) = \text{to}(m') \land m <^m_{k} m'\})^+$,

$<^2_{k}:= (<^m_{k} \cup \{(!m, !m') \mid m, m' \in M \land \text{from}(m) = \text{from}(m') \land m <^m_{k} m'\}
\cup \{(!m, !m') \mid m, m' \in M \land \text{to}(m) = \text{to}(m') \land m <^m_{k} m'\})^+$.

We explain the definition of the ordering $<^i_{k}$ which is defined in order to check the $\text{inst.out}$-property. The ordering is obtained from $<^m_{k}$ by adding pairs of input events to it. More specifically, if two outputs are defined on the same instance of the MSC, and thus are ordered in some way, then we add their corresponding input events in the same order. This is motivated as follows. For a trace to be $\text{inst.out}$-implementable it is required that the input events are ordered in this way anyway. Thus by adding this pair explicitly we construct an ordering representing the MSC given that it has to be implemented on an architecture with one output buffer per instance.

The $\text{inst.out}$-implementable traces of the MSC are also traces of the ordering $<^i_{k}$ as they respect the requirements for $\text{inst.out}$-implementability by definition, and vice versa. Basically this is what is expressed in Lemma 21.

Lemma 21 Let $t$ be a 2-trace of an MSC $k$. Then,
• \( t \) is \( \text{inst\.out}\)-implementable if and only if \( \ll k \subseteq \ll \text{trace} \).

• \( t \) is \( \text{inst.in}\)-implementable if and only if \( \ll i_k \subseteq \ll \text{trace} \).

• \( t \) is \( \text{inst2}\)-implementable if and only if there exists an extension \( t' \) of \( t \) such that \( \ll k \subseteq \ll \text{trace} \).

**Proof**

We only give the proof for the last proposition. The proofs for the first two propositions follow the same line.

First, suppose that \( t \) is \( \text{inst2}\)-implementable. Then we must prove that \( \ll k \subseteq \ll \text{trace} \) for some 3-trace \( t' \) which is an extension of \( t \). Let the arbitrary 3-trace \( t' \) be an extension of \( t \) that is \( \text{inst2}\)-implementable. Suppose that \( e \ll k e' \) for arbitrary events \( e, e' \in \{!/!/?\} M \). Now it suffices to prove \( e \ll k e' \). Since \( e \ll k e' \) we have the existence of events \( e_1, \ldots, e_n \) such that \( e \equiv e_1, e' \equiv e_n \) and for all \( 1 \leq i < n \) we have one of the following:

- \( e_i <^m \) \( e_{i+1} \);
- \( e_i \equiv !m \) and \( e_{i+1} \equiv !m' \) for some \( m, m' \in M \) such that \( \text{from}(m) = \text{from}(m') \) and \( !m <^m \) \( !m' \);
- \( e_i \equiv !m \) and \( e_{i+1} \equiv !m' \) for some \( m, m' \in M \) such that \( \text{to}(m) = \text{to}(m') \) and \( !m <^m \) \( !m' \).

In the first case we immediately have \( e_i \ll k e_{i+1} \). Due to the fact \( t' \) is an \( \text{inst2}\)-implementable 3-trace, and thus both \( \text{inst-output} \) and \( \text{inst-input} \) implementable, we can conclude that \( e_i \ll k e_{i+1} \) for the second and third case as well. Since \( \ll \text{trace} \) is transitively closed we have \( e \ll k e' \), which completes this part of the proof.

Second, suppose that \( \ll k \subseteq \ll \text{trace} \) for some 3-trace \( t' \) which is an extension of \( t \). We must prove that \( t \) is \( (\text{inst,inst}) \)-implementable. Thereto it suffices to show that \( t' \) is \( (\text{inst,inst}) \)-implementable, i.e., that \( t' \) is \( \text{inst-output} \) implementable and \( \text{inst-input} \) implementable. We prove that \( t' \) is \( \text{inst-output} \) implementable, the proof that \( t' \) is \( \text{inst-input} \) implementable is analogous. Let \( m, m' \in M \) such that \( \text{from}(m) = \text{from}(m') \). Then it suffices to show that \( !m \ll l_t \) \( !m' \). Thereto, suppose that \( !m \ll l_t !m' \). Since \( \text{from}(m) = \text{from}(m') \), we have \( !m \ll k !m' \). Because \( \ll k \subseteq \ll \text{trace} \) we therefore have \( !m \ll \text{trace} !m' \). Suppose that \( !m \ll k !m' \). Then \( !m' \ll \text{trace} !m \). With similar reasoning as before we obtain \( !m' \ll \text{trace} !m \). Therefore, \( !m \ll k !m' \).

Thus far, we have seen that the ordering \( \ll k \) contains all \( \text{inst.out} \)-implementable traces of MSC \( k \). An MSC \( k \) is \( \text{inst.out} \)-implementable if and only if it has a trace \( t \) that is \( \text{inst.out} \)-implementable. Clearly, such a trace exists if and only if there is a trace for the ordering \( \ll k \), in other words, if and only if \( \ll k \) is cycle-free.

**Lemma 22**

Let \( k \) be an MSC. Then,

- \( k \) is \( \text{inst.out} \)-implementable if and only if \( \ll k \) is cycle-free;
- \( k \) is \( \text{inst.in} \)-implementable if and only if \( \ll k \) is cycle-free;
- \( k \) is \( \text{inst2} \)-implementable if and only if \( \ll k \) is cycle-free.

**Proof**

This lemma follows immediately from the previous lemma.

We use the alternative characterisations provided by the previous theorem in the proof of the equivalence of the classes \( \text{inst.out} \), \( \text{inst.in} \), and \( \text{inst2} \).
Lemma 23 Let $k$ be an MSC over the set of messages $M$ and let $m, m' \in M$. If $?m <_k ?m'$, then $!!m <^2_k !!m'$

**Proof** Suppose that $?m <_k ?m'$. Then by the definition of $<_k$ we have the existence of events $e_1, \ldots , e_n$ such that $e_1 ?m, e_n ?m'$, and for $1 \leq i < n$ we have one of the following:

- $e_i <^\text{msg} e_{i+1}$
- $e_i ?p, e_{i+1} ?p'$ for some $p, p' \in M$ such that from$(p) = from(p')$ and $!p <^\text{msg} !p'$.

In the second case we immediately have $!!p <^2_k !!p'$ as $!!p <^\text{msg} !!p'$ and from$(p) = from(p')$ by the definition of $<_k$. In the first case we have a sequence of events where the smallest steps are due to $<_\text{inst}$ or due to $<_o$. In this sequence any subsequence of events which are defined on the same instance can be replaced by one single step as the instancewise order is total. As a result we have the existence of messages $m_1, \ldots , m_n$ such that $e_i ?^{\text{inst}} m_1 <^{o} ?^{\text{inst}} m_2 <^{o} ?^{\text{inst}} m_3 \ldots <^{o} ?^{\text{inst}} m_{n-1} <^{o} ?^{\text{inst}} m_n <^{o} ?^{\text{inst}} m'$ $<^{\text{inst}} e_{i+1}$, where $f ?^{\text{inst}} f'$ if and only if $f <^{\text{inst}} f'$ or $f \equiv f'$. Now we observe that we only have the following three possibilities for $<_\text{inst}$:

- $!q ?^{\text{inst}} !q'$ for some $q, q' \in M$ such that from$(q) = from(q')$. Then also $!!q <^2_k !!q'$.
- $?q ?^{\text{inst}} !q'$ for some $q, q' \in M$ such that to$(q) = from(q')$. As $?q <^{\text{inst}} !q'$ and to$(q) = from(q')$ we have $?q <^\text{msg} !q'$ and hence $?q <^2_k !q'$. Together with $!!q <^2_k ?q$ we obtain $!!q <^2_k !q'$.
- $?q ?^{\text{inst}} ?q'$ for some $q, q' \in M$ such that to$(q) = to(q')$. Then also $!!q <^2_k !!q'$.

Thus we obtain $!!e_i <^2_k !!e_{i+1}$ for all $1 \leq i < n$. Therefore $!!m <^2_k !!m'$.

Lemma 24 The implementation models $\text{inst.out}_w, \text{inst.in}_w$, and $\text{inst2}_w$ are equal.

**Proof** We show that each $\text{inst2}_w$-implementable MSC is $\text{inst.out}_w$-implementable. The reverse implication is trivial, and the proofs with $\text{inst.in}_w$ are analogous. From Lemma 22 we see that it suffices to prove that $<_w$ is cycle-free. Now we assume that $<_w$ is cycle-free. We prove this using contraposition, so we assume that $<_w$ has a cycle. Let $e_1 <^w e_2 <^w \ldots <^w e_n <^w e_1$ be an arbitrary largest cycle. For every ordering in the cycle, say $e_i <^w e_{i+1}$, either $e_i <^\text{msg} e_{i+1}$, and hence $e_i <^2 e_{i+1}$, or $e_i ?m, e_{i+1} ?m'$ for some $m, m' \in M$ such that $!m <^\text{msg} !m'$ and from$(m) = from(m')$.

If the first is always the case, then we have a cycle in $<_\text{msg}$, so certainly in $<_2$. Now assume we have the second at least once in the cycle. In that case we have at least two inputs in the cycle, say $?m$ and $?m'$. Then $?m <^w ?m'$ and $?m' <^w ?m$. Lemma 23 gives that this implies that $!!m <^2_k !!m'$ and $!!m' <^2_k !!m$. Thus clearly $<_2$ has a cycle.

Lemma 24 establishes that the classes $\text{inst.out}_w, \text{inst.in}_w$, and $\text{inst2}_w$ are equivalent. In the remainder we denote this class by $\text{inst}_w$. The remaining models are all different. MSC 3 and MSC 4 in Figure 6 show the difference between $\text{inst}_w$ and $\text{pair}_w$, and $\text{pair}_w$ and $\text{msg}_w$ respectively in the weak case too (these MSCs have only one 2-trace, so their weak implementability equals their strong implementability). MSC 5 in Figure 8 is $\text{global}_w$-implementable, but not $\text{nobuf}_w$-implementable. The
trace $!a !b ?a ?b$ is global-implementable, but because both outputs must have been executed before any input can be processed, there is no nobuf-implementable trace.

MSC 6 is $\text{inst}_w$-implementable, but not $\text{global}_w$-implementable. It is not $\text{global}_w$-implementable, as can be seen thus: $!a <_{\text{msc}} !b$, so for every global-implementable trace $t$ we must have $?a <_{t} ?b$. Because $!d <_{\text{msc}} ?a$ and $?b <_{\text{msc}} ?c$, we get $!d <_{t} ?c$. But we also have $?c <_{t} ?d$, and thus $?c <_{t} ?d$, from which it follows that $t$ cannot be global-implementable. On the other hand, the trace $!a !b !d ?a ?b !e?e?d$ is $\text{inst}.\text{out}$-implementable, so the MSC is $\text{inst}_w$-implementable.

Theorem 25  The implementation models for weak implementability of Figure 9 are all different and they are order as expressed in Figure 9.

Proof  The counterexamples that imply that the implementation models are different are given above. The ordering of the models is inherited from the ordering of the implementation models with respect to traces. Lemma 24 provides that the implementation models $\text{inst}.\text{out}_w$, $\text{inst}.\text{in}_w$, and $\text{inst}2_w$ are equal.

5.3 Combining the strong and weak hierarchy

We now have 12 possible implementations left: nobuf$_s$, global$_s$, $\text{inst}.\text{out}_s$, $\text{inst}.\text{in}_s$, $\text{inst}2_s$, pair$_s$ and msg$_s$ in the strong case, and nobuf$_w$, global$_w$, $\text{inst}_w$, pair$_w$, and msg$_w$ in the weak case. These are ordered as shown in Figure 10. An arrow pointing from one of the classes to the other means that all MSCs that are implementable in the type corresponding to the first class are also implementable in the type corresponding to the second. Any superfluous arrows (those that can be inferred from the transitivity of the relation) have been removed. However, there could be (and it will be shown that there are)
implications present that are not shown in this diagram. Some of these arrows could, for example, be non-strict.

The relations between classes in the strong implementability hierarchy and the relations between classes in the weak hierarchy have been studied extensively in the previous sections. In this section we focus on the relations between implementation models from the different hierarchies. From the definitions of strong and weak implementability it is clear that any $X_s$-implementable MSC is also $X_w$-implementable. These orderings are also depicted in Figure 10.

First, we prove that some classes can be identified.

**Lemma 26** An MSC $k$ is $pair_s$-implementable if and only if it is $pair_w$-implementable.

**Proof** Clearly any $pair_s$-implementable MSC is also $pair_w$-implementable. This is proven as follows. Suppose that MSC $k$ is $pair_s$-implementable. This means that all its 2-traces are $pair$-implementable. Since every MSC has at least one trace this implies that there is a $pair$-implementable 2-trace of $k$. Therefore $k$ is $pair_w$-implementable.

It remains to prove that any $pair_w$-implementable MSC is also $pair_s$-implementable. Let $k$ be a $pair_w$-implementable MSC. Let $t$ be an arbitrary 2-trace of $k$. Let $m, m' \in M$ such that $\text{from}(m) = \text{from}(m')$ and $\text{to}(m) = \text{to}(m')$.

First, suppose that $!m <^t \text{trace} !m'$. Then, because $\text{from}(m) = \text{from}(m')$ and $!m <^t \text{trace} !m'$, we have $!m <^k \text{msg} !m'$. Since $k$ is $pair_w$-implementable there exists a trace $t'$ that is $pair$-implementable. Since $!m <^k \text{msg} !m'$ we have $!m <^{t'} \text{trace} !m'$. Since $t'$ is $pair$-implementable we have by Lemma 15 that $?m <^{t'} \text{trace} ?m'$. Because $\text{to}(m) = \text{to}(m')$ we then have $?m <^k \text{msg} ?m'$. Therefore we have $?m <^{t'} \text{trace} ?m'$, which completes this part the proof.

Second, suppose that $?m <^{t'} \text{trace} ?m'$. With a similar reasoning as in the previous case we easily obtain $!m <^t \text{trace} !m'$.

**Lemma 27** An MSC $k$ is $msg_s$-implementable if and only if it is $msg_w$-implementable.

**Proof** Trivial, because every 3-trace is $msg$-implementable, and thus each 2-trace is as well.

Lemmas 26 and 27 establish that the classes $pair_s$ and $pair_w$, and $msg_s$ and $msg_w$ are equivalent. In the remainder we denote these by $pair$ and $msg$ respectively.
Next we are going to proof that any \textit{inst.out}-implementable MSC is \textit{globalw}-implementable. To do this we first give some alternative characterisations for these implementations.

\textbf{Lemma 28} An MSC $k$ is \textit{inst.out}-implementable if and only if $<^{\text{msc}}_k \subseteq <^{\text{io}}_k$. An MSC $k$ is \textit{inst.in}-implementable if and only if $<^{\text{io}}_k = <^{\text{msc}}_k$.

\textbf{Proof} We only give the proof for the first proposition. The proof of the second proposition follows the same lines.

First, suppose that MSC $k$ is \textit{inst.out}-implementable. By definition $<^{\text{msc}}_k \subseteq <^{\text{io}}_k$, so it only remains to be proven that $<^{\text{io}}_k \subseteq <^{\text{msc}}_k$. Suppose that $e <^{\text{io}}_k e'$ for arbitrary $e, e' \in \{|/\}M$. Then we have the existence of $e_1, \ldots, e_n$ such that \( e = e_1, e' = e_n \) and for all $1 \leq i < n$ we have one of the following:

- $e_i <^{\text{msc}}_k e_{i+1}$;
- $e_i \equiv \equiv m$ and $e_{i+1} \equiv \equiv m'$ for some $m, m' \in M$ such that $\text{from}(m) = \text{from}(m')$ and $!m <^{\text{msc}}_k m'$.

In the second case we have, by Lemma 7, $!m <^{\text{trace}}_t m'$ for every 2-trace $t$ of $k$. Since $k$ is \textit{inst.out}-implementable we have that every 2-trace of $k$ is \textit{inst.out}-implementable. Thus, by Lemma 15 and the assumption that $\text{from}(m) = \text{from}(m')$ we have $!m <^{\text{trace}}_t m'$ for every 2-trace $t$ of $k$. Then, again by Lemma 7, we have $?m <^{\text{msc}}_k m'$. Thus in any case we have $e_i <^{\text{msc}}_k e_{i+1}$ and therefore also $e <^{\text{msc}}_k e'$ which was to be proven.

Second, suppose that $<^{\text{io}}_k = <^{\text{msc}}_k$. Then we must prove that MSC $k$ is \textit{inst.out}-implementable. Let $t$ be a 2-trace of $k$, and let $m, m' \in M$ such that $\text{from}(m) = \text{from}(m')$. Suppose $!m \not<^{\text{trace}}_t m'$. Then because of $\text{from}(m) = \text{from}(m')$, we have $!m <^{\text{msc}}_k m'$. By the definition of $<^{\text{io}}_k$ we then have $?m <^{\text{msc}}_k m'$. By the assumption that $<^{\text{io}}_k = <^{\text{msc}}_k$, this implies $!m <^{\text{msc}}_k m'$, and thus $?m <^{\text{trace}}_t m'$. Reversely, suppose that $?m \not<^{\text{trace}}_t m'$. Then $!m <^{\text{trace}}_t m'$ as $<^{\text{trace}}_t$ is a total order. With a similar reasoning as in the previous case we obtain $?m <^{\text{trace}}_t m'$, and therefore $?m \not<^{\text{trace}}_t m'$. $\blacksquare$

For a similar characterisation of \textit{globalw}-implementability we define a relation $<^{g}_k$.

\textbf{Definition 29} Let $k$ be an MSC. The relation $<^{g}_k$ on $\{|/\}M$ is defined as the smallest relation that satisfies:

1. $<^{\text{msc}}_k \subseteq <^{g}_k$;
2. $<^{g}_k$ is transitive;
3. $!m <^{g}_k m' \iff ?m <^{g}_k m'$ for all $m, m' \in M$.

\textbf{Lemma 30} An MSC $k$ is \textit{globalw}-implementable if and only if the relation $<^{g}_k$ is cycle-free.

\textbf{Proof} First, suppose that $k$ is \textit{globalw}-implementable. Let $t$ be a \textit{global}-implementable trace of $k$. Then $<^{\text{trace}}_t$ adheres to the restrictions in Definition 29, and thus $<^{g}_k \subseteq <^{\text{trace}}_t$, and $<^{g}_k$ is cycle-free.

Second, suppose that the relation $<^{g}_k$ is cycle-free. The idea of the proof is that we extend this relation until it is a total order. Then, if we can prove that the trace corresponding with this total order is \textit{global}-implementable, we are done.

We extend the relation $<^{g}_k$ to form an order $<$ through the following algorithm:

1. $S := \{|/\}M$, $< := <^{g}_k$
2. Let \( e \) be any smallest element of \( S \) with respect to \( < \), that is, any element of \( S \) for which there is no \( e' \in S \) with \( e' < e \).

3. \( S := S \setminus \{ e \} \)

4. \(<' := (e' \cup (\langle e, e' \rangle | e' \in S))^+ \)

5. If \( e \equiv m \) for some \( m \in M \), then \( := (\cup \{ \langle m, m' \rangle | m' \in S \})^+ \)

6. Repeat steps 2 to 5 until \( S = \emptyset \)

We first remark that the following invariant holds: \( m < m' \Rightarrow \forall m \in S \) for all \( m, m' \in M \). This clearly holds at the beginning, and only pairs \( \langle m, m' \rangle \) are added for which \( m \notin S \) since otherwise \( m \) is not a smallest element of \( S \).

For step 2 of the above algorithm to be well-defined it is necessary that \( < \) is cycle-free. After step 1 \( < \) is cycle-free because by the assumption \( \langle k \rangle \) is cycle-free. There are two places where the relation \( < \) is extended, namely step 4 and step 5. Step 4 maintains cycle-freeness of \( < \). This can be seen as follows. Let \( e \) be an arbitrary smallest element of \( S \) with respect to \( < \). Suppose that by adding the pairs \( \langle e, e' \rangle \) for \( e' \in S \setminus \{ e \} \) to \( < \) a cycle appears. Then \( e' < e \) for some \( e' \in S \setminus \{ e \} \) which contradicts the assumption that \( e \) is a smallest element of \( S \) with respect to \( < \).

Also step 5 maintains cycle-freeness. Suppose that \( l \) is a smallest element of \( S \) with respect to \( < \) with respect to \( S \). Suppose that a cycle is introduced by step 5. This can only be the case if a pair \( \langle m, m' \rangle \) is added to \( < \) for which we already had \( m < m' \in S \) and \( m' \in S \). By the previously mentioned invariant we have \( m' < k m \). By the definition of \( \langle k \rangle \) then also \( \forall m' \in S \). As \( l \in S \) this contradicts the assumption that \( l \) was a smallest element of \( S \) with respect to \( < \). Now we have established that step 2 of the algorithm is well-defined.

The algorithm is guaranteed to terminate as the number of elements of the finite set \( S \) is decreased by one every time the body of the repetition is executed. Furthermore, observe that \( < \) is a total ordering on the events not contained in \( S \) after every execution of the body of the repetition (i.e., after step 5). Thus, upon termination of the algorithm, \( < \) is total order on \( \{ l \} M \). This total order corresponds to a trace of the MSC as \( \langle k \rangle \subseteq \langle k \rangle \subseteq < \).

All that remains to be proven is that \( < \) corresponds to a \( \text{global}\)-implementable trace of \( k \). Note that after step 1, for all \( m, m' \in M \) we have \( m \triangleleft m' \Rightarrow \forall m < m' \). If in step 4, an ordering \( \forall m < m' \) is added then in step 5 \( \forall m < m' \) is added. If in step 4, an ordering \( m < m' \) is added, then \( m \notin S \) and therefore \( \forall m \triangleleft m' \). Thus at the end of step 5 again: for all \( m, m' \in M \) we have \( \forall m \triangleleft m' \Rightarrow \forall m < m' \). Thus the trace corresponding with \( < \) is \( \text{global}\)-implementable.

**Lemma 31** Every \( \text{inst.out}_s \)- or \( \text{inst.in}_r \)-implementable MSC is \( \text{global}_w \)-implementable.

**Proof** We prove this for an \( \text{inst.out}_s \)-implementable MSC. For an \( \text{inst.in}_r \)-implementable MSC the proof is completely analogous.

We proof this by contradiction, so we assume that \( k \) is an \( \text{inst.out}_s \)-implementable MSC that is not \( \text{global}_w \)-implementable. By Lemma 28 we have \( \langle k \rangle = \langle msc \rangle \), and by Lemma 30 we have that \( \langle k \rangle \) has a cycle.

First we note that \( \langle k \rangle \) can be constructed by the following algorithm:

1. \( \langle k \rangle := \langle msc \rangle \)

2. \( \langle k \rangle := \langle k \rangle \cup \{ \langle m, m' \rangle \mid m \triangleleft k m \} \)
3. \( <^k_\delta := <^k_\delta \cup \{ (m, m') \mid ?m <^k_\delta m' \} \)

4. \( <^k_\delta := <^k_\delta^+ \)

5. Repeat steps 2 to 4 until no change occurs.

In the proof we need to have information as to the way in which \( <^k_\delta \) was constructed. Therefore we extend the above algorithm to incorporate the additional information. In the next algorithm a relation \( < \) on \( \{!m, ?m\} M \times \{m, r_1, r_2, t\} \) is constructed such that \( e <^k_\delta e' \) if and only if \( (e, e', r) \in < \) where \( r \in \{m, r_1, r_2, t\} \). The letters \( m, r_1, r_2 \) and \( t \) encode a reason why \( e <^k_\delta e' \). If \( (e, e', m) \in < \), then \( e <^k_\delta e' \). If \( (e, e', r_1) \in < \) or \( (e, e', r_2) \in < \), then this is due to step 2 or step 3 from the above algorithm respectively. If \( (e, e, t) \in < \), then \( e <^k_\delta e' \) is due to transitivity of \( <^k_\delta \). We simply write \( e < e' \) if we are not interested in the third part of the triple, i.e., the encoding of the reason for the events to be ordered.

1. \( ::= \{ (e, e', m) \mid e <^m_\delta e' \} \)

2. \( ::=< \cup \{ (m, m', r_1) \mid !m <^l_\delta m' \} \)

3. \( ::=< \cup \{ (m, m', r_2) \mid ?m <^l_\delta m' \} \)

4. Compute the transitive closure of \( < \) with respect to the first and second entry of the triple. In all cases the third entry of the triple will be \( t \).

5. Repeat steps 2 to 4 until no change occurs.

Let \( e_1, e_2, \ldots, e_n \equiv e_1 \) be a cycle of \( <^k_\delta \). Then it is also a cycle of \( < \). For all \( 1 \leq i < n \) we have one of the following:

- \( (e_i, e_{i+1}, m) \in < \);
- \( (e_i, e_{i+1}, r_1) \in < \) for some \( m, m' \in M \) such that \( e_i \equiv !m, e_{i+1} \equiv !m' \) and \( !m <^l_\delta !m' \);
- \( (e_i, e_{i+1}, r_2) \in < \) for some \( m, m' \in M \) such that \( e_i \equiv ?m, e_{i+1} \equiv ?m' \) and \( ?m <^l_\delta ?m' \);
- \( (e_i, e_{i+1}, t) \in < \) such that there exists an event \( e \) such that \( e_i < e \) and \( e < e_{i+1} \).

We call the steps \( e_i < e_{i+1}, g \)-steps. We will change them into other kinds of steps in the following way. Let there be a \( g \)-step from \( e_i \) to \( e_{i+1} \)

1. if \( (e_i, e_{i+1}, m) \in < \) then the \( g \)-step becomes an \( m \)-step: \( e_i \mapsto e_{i+1} \);

2. if \( e_i \equiv !m \) and \( e_{i+1} \equiv !m' \) for some \( m, m' \in M \) such that \( !m <^l_\delta !m' \) and \( (e_i, e_{i+1}, r_1) \in < \), then we replace this \( g \)-step by the steps \( ?m \not< !m, !m <^l_\delta !m' \), and \( !m' \not< ?m' \);

3. if \( e_i \equiv ?m \) and \( e_{i+1} \equiv ?m' \) for some \( m, m' \in M \) such that \( ?m <^l_\delta ?m' \) and \( (e_i, e_{i+1}, r_2) \in < \), then we replace this \( g \)-step by the steps \( ?m \not< !m, ?m <^l_\delta ?m' \), and \( ?m' \not< !m' \);

4. if \( (e_i, e_{i+1}, t) \in < \), then for an arbitrary event \( e \) such that \( e_i < e \) and \( e < e_{i+1} \), the \( g \)-step is replaced by \( g \)-steps \( e_i < e \) and \( e < e_{i+1} \).

The above steps are repeated until no \( g \)-steps are left.

The following will hold in this algorithm:
1. Because of the way in which $<$ was constructed before, the algorithm ends in a finite number of steps. Then the cycle of $<$-steps is changed into a cycle of $\Rightarrow$-steps, $\Rightarrow$-steps and $\supset$-steps only.

2. By the way in which $<$ has been constructed and then changed we find the following. If $e < e'$, then $e <_k e'$. If $e \Rightarrow e'$, then $e <_{msc} e'$. If $e \supset e'$, then $e \equiv m$ and $e' \equiv m$ for some $m \in M$. If $e \supset e'$, then $e \equiv m$ and $e' \equiv m$ for some $m \in M$.

3. The numbers of $\Rightarrow$-steps and $\supset$-steps are equal after each step of the algorithm, and in particular when the algorithm has terminated.

Now clearly the result of the algorithm applied to all $g$-steps from the cycle $e_1, e_2, \ldots, e_n \equiv e_1$ is a cycle of $\Rightarrow$, $\Rightarrow$, and $\supset$ steps with as many $\Rightarrow$-steps as $\supset$-steps. We call such a cycle a quasi-cycle of order $N$, where $N$ is the number of $\Rightarrow$-steps.

We prove that this cycle can be changed into a quasi-cycle of order 0. Let the order be greater than 0. Because the quasi-cycle is a cycle, and contains at least one $\supset$-step and at least one $\Rightarrow$-step, there will be at least one $\Rightarrow$-step, such that after that $\Rightarrow$-step a $\supset$-step will take place before the next $\Rightarrow$-step. Thus the quasi-cycle contains a subsequence $?m \Rightarrow m \Rightarrow \cdots \Rightarrow ?m' \supset m'$ for some $m, m' \in M$. As there are only $\Rightarrow$-steps from $m$ to $m'$, we have $?m <_{msc} ?m'$. However, then, by definition, $?m <_{io} ?m'$, from which we get $?m <_{msc} ?m'$ from the assumption that $<_{msc} <_{io}$. Thus by removing all steps between $?m$ and $?m'$, we still have a cycle which is a quasi-cycle, but of order $N - 1$. Repeating this, we will finally obtain a quasi-cycle of order 0. However, a quasi-cycle of order 0 is a cycle of only $\Rightarrow$-steps, that is, a cycle of $<_{msc}$.

Thus we see that, given the assumption, $<_{msc}$ must have a cycle. This is impossible, so the assertions cannot simultaneously hold, so each $inst.out$-implementable MSC is $global_w$-implementable. 

In Figure 11 we give all communication models that remain after the identifications obtained until now. The arrows between these models follow also from the previous theorems and lemmas. Finally, we have to prove that the arrows between models from the strong and weak hierarchy are strict and that there are no additional arrows necessary. It suffices to show that the following arrows do not exist: $global_s$ to $nobuf_w$, $nobuf_w$ to $inst2_s$, and $inst2_s$ to $global_w$. The rest then follows because of transitivity. For example, the nonexistence of an arrow from $global_s$ to $nobuf_w$ implies the nonexistence of an arrow from $inst.out_s$ to $nobuf_w$, because if the second arrow exists then, by transitivity, also the first must
exist. Similarly we obtain the nonexistence of arrows from $\text{inst}_1s$ and $\text{inst}_2s$ to $\text{nobuf}_w$. We use the MSCs in Figure 12 to indicate that the first two arrows do not exist. MSC 7 is $\text{global}_1$-implementable, but not $\text{nobuf}_w$-implementable. It has one trace, $!a ?a !b ?b$, which is $\text{global}$-implementable, but not $\text{nobuf}$-implementable. We see that MSC 7 contains only one instance, so all messages are messages to the same instance that sent them. This is no coincidence, as will be seen from Corollary 34 below. On the other hand MSC 8 is $\text{nobuf}_w$-implementable, but not $\text{inst}_2$s-implementable. That it is $\text{nobuf}_w$-implementable can be seen from the picture, which shows that there is the trace $!a ?a !b ?b !e ?e$, which is $\text{nobuf}$-implementable. However, the trace $!b !e ?e !a ?a ?b$ is not $\text{inst}_2$-implementable: Because $?b$ is after $?a$ in the trace, $!!b$ must be after $!!a$ to make the trace $\text{inst}$-input implementable, while because $!b$ is before $!c$, $!!b$ must be before $!!c$ to make the trace $\text{inst}$-output implementable. However, $!!b$ cannot be both before $!!c$ and after $!!a$.

**Theorem 32** The implementation models from Figure 11 are all different, and they are ordered as expressed in Figure 11.

**Proof** This has been explained in the above text.

Next, we show why we needed an MSC with messages from an instance to that same instance for the counterexample MSC 7 in Figure 12. First, we provide an alternative characterisation of $\text{global}_1$-implementability.

**Lemma 33** An MSC $k$ is $\text{global}_1$-implementable if and only if for all $m, m' \in M$, we either have both $!m <^\text{MSC} !m'$ and $?m <^\text{MSC} ?m'$, or we have both $?m' <^\text{MSC} !m$ and $!m' <^\text{MSC} ?m$.  

**Proof** First, suppose that MSC $k$ is $\text{global}_1$-implementable. Let $m, m' \in M$. Without loss of generality we may assume $!m' <^\text{MSC} m$. Then it suffices to prove that $!m <^\text{MSC} !m'$ and $?m <^\text{MSC} ?m'$. Now we can distinguish two cases: $!m <^\text{MSC} !m'$ and $!m <^\text{MSC} !m'$.

Suppose that $!m <^\text{MSC} m'$. Then, by Lemma 7, $!m <^\text{trace} !m'$ for every 2-trace $t$ of $k$. Since every 2-trace of $k$ is $\text{global}$-implementable, we have by Lemma 15 that $?m <^\text{trace} ?m'$ for every 2-trace $t$ of $k$. Then, again by Lemma 7, we have $?m <^\text{MSC} ?m'$, which completes this part of the proof.

Suppose that $!m <^\text{MSC} !m'$. Now three cases can be distinguished: (1) $?m <^\text{MSC} ?m'$, (2) $?m' <^\text{MSC} ?m$, and (3) $?m' <^\text{MSC} ?m'$ and $?m' <^\text{MSC} ?m$. In the first case we have $!m <^\text{MSC} !m'$ with a similar reasoning as above. This contradicts the assumption that $!m <^\text{MSC} !m'$ with a similar reasoning as above. This contradicts the assumption that $!m' <^\text{MSC} !m'$, so this case cannot occur. In the second case we have $!m' <^\text{MSC} !m'$ with a similar reasoning as above. This contradicts the assumption that $!m' <^\text{MSC} !m'$, so this case cannot occur. With respect to the third case we show that there exists a

![Figure 12: Distinguishing MSCs: comparing strong and weak.](image-url)
2-trace $t$ of $k$ such that $!m <^\text{trace}!m'$ and $?m <^\text{trace}?m$, thereby contradicting the assumption that $k$ is \textit{MSCs}-implementable. We define the ordering $<$ as follows: $< := (?m <^\text{MSC}!m, (?m', ?m))$. We prove that $<$ is cycle-free and thus that there exists a trace $t$ of $k$ such that $!m <^\text{trace}!m'$ and $?m <^\text{trace}?m$. Note that the ordering $<^\text{MSC}$ is cycle-free by assumption. Suppose that the ordering $<$ contains an element. Then this cycle must contain at least one of $!m <$ or $?m'$. If the cycle contains both $!m <$ and $?m' <$, then there are also cycles which contain exactly one of $!m <$ and $?m' <$ if only $!m <$ is in the cycle, then necessarily $!m' <^\text{MSC}?m$. This contradicts the assumption that $!m' \not<^\text{MSC}?m$. If $?m' <$ is in the cycle, then necessarily $?m <^\text{MSC}?m'$. This contradicts the assumption that $?m' \not<^\text{MSC}?m'$. Thus the third case cannot occur as well.

Second, suppose that for all $m, m' \in M$ we have $!m <^\text{MSC}!m'$ and $?m <^\text{MSC}?m'$ or $!m' <^\text{MSC}!m$ and $?m' <^\text{MSC}?m$. We must prove that MSC $k$ is \textit{globals}-implementable. Let $t$ be an arbitrary 2-trace of MSC $k$. Let $m, m' \in M$. By Lemma 15, it suffices to prove that $!m <^\text{trace}?m' \Rightarrow ?m <^\text{trace}?m'$. First, suppose that $!m <^\text{trace}?m'$. Then, by Lemma 7, $!m' <^\text{MSC}!m$. Therefore, by the assumption, $!m <^\text{MSC}!m$ and $?m <^\text{MSC}?m'$. So, by Lemma 7, we have $?m <^\text{trace}?m'$. Second, suppose that $?m <^\text{trace}?m'$. By similar reasoning we obtain $!m <^\text{trace}?m'$.

The following corollary explains that for MSCs without messages from an instance to the same instance, \textit{globals}-implementability implies \textit{nobuff} implementability. Thus any counterexample for this implication should have at least one message from an instance to the same instance.

\textbf{Corollary 34} Let $k$ be an MSC without any messages that are sent and received by the same instance. If $k$ is \textit{globals}-implementable, then $k$ is \textit{nobuff}-implementable.

\textbf{Proof} Suppose that $k$ is \textit{globals}-implementable. Now it suffices to prove that there is a 2-trace $t$ of $k$ that is \textit{nobuff}-implementable. From the lemma above we conclude that $<^\text{MSC}$ constitutes a total order when restricted to the output events only, say $!m_1 <^\text{MSC}!m_2 <^\text{MSC} \cdots <^\text{MSC}!m_n$. Then clearly the trace $t = !(m_1, m_2, \ldots, m_n)$ is \textit{nobuff}-implementable. The total ordering associated with this trace can be described as follows: $<^\text{trace} = \{!(m, m'), (!(m, m'), (?m, ??m') |!m <^\text{MSC}!m' \cup ^\text{alt} \}$. All that remains to be proven is that this trace $t$ is indeed a 2-trace of $k$, i.e., for all $e, e' \in \{!/?\}M$ we have $e <^k e'$ implies $e <^\text{trace} e'$. Suppose that $e <^k e'$. Then we have the existence of events $e_1, \ldots, e_p$ such that $e \equiv e_1, e' \equiv e_p$ and for all $1 \leq i < p$ we have one of the following:

- $e_i <^{\text{inst}} e_{i+1}$;
- $e_i <^{\text{alt}} e_{i+1}$.

In the second case we have $e_i <^\text{trace} e_{i+1}$ as $<^{\text{alt}} <^\text{trace}$. In the first case the following four cases can be distinguished:

- $e_i \equiv !(m$ and $e_{i+1} \equiv !(m'$ for some $m, m' \in M$ such that $\text{from}(m) = \text{from}(m')$. Then clearly $!m <^\text{MSC}!m'$ and therefore $!m <^\text{trace}!m'$.
- $e_i \equiv !(m$ and $e_{i+1} \equiv !(m'$ for some $m, m' \in M$ such that $\text{from}(m) = \text{to}(m')$. By the previous lemma we can distinguish two cases:
  - $!m <^\text{MSC}!m'$ and $?m <^\text{MSC}?m'$. Clearly, $!m <^\text{trace}!m'$ and $!m <^\text{trace}?m'$, so $!m <^\text{trace}?m'$. $!m' <^\text{MSC}!m$ and $?m' <^\text{MSC}?m$. As there are no messages that are sent and received by the same instance we have $\text{from}(m') \neq \text{from}(m)$. Then $!m' <^\text{MSC}!m'$ must be due to the
existence of \( m'' \in M \) such that \( !m' \not<_k \ MSC \ m' \not<_k MSC \ m'' \not<_k \ MSC \ m \). Therefore by the previous lemma we have \( !m' \not<_k MSC \ m'' \not<_k MSC \ m \). Then we also have \( !m' \not<_k MSC \ m \) which contradicts the assumption that \( !m \not<_t trace \ m' \). This case can therefore not occur.

- \( e_i \equiv \equiv m \) and \( e_{i+1} \equiv \equiv m' \) for some \( m, m' \in M \) such that \( to(m) = from(m') \). In this case we have \( ?m' <_k MSC \ m' \), and therefore \( ?m <_k MSC \ m' \). By the previous lemma then also \( !m <_k MSC \ m' \). So, by the definition of \( <_t trace \) also \( ?m <_t trace \ m' \).

- \( e_i \equiv \equiv m \) and \( e_{i+1} \equiv \equiv m' \) for some \( m, m' \in M \) such that \( to(m) = to(m') \). Since \( ?m <_k MSC \ m' \) we have by the previous lemma that \( !m <_k MSC \ m' \). Therefore, by the definition of \( <_t trace \) \( ?m <_t trace \ m' \). Thus, \( ?m <_t trace \ m' \). Thus in all cases we have \( e_i <_t trace \ e_{i+1} \). Therefore, \( e <_t trace \ e' \) which completes the proof. 

\[
\]

\section{Characterisations}

In this section we provide alternative ways to describe the various implementations which, for example, can be used in algorithms. These definitions are easier to check automatically than the ones we used before, and therefore could be used in tools to determine the implementation models that a given MSC satisfies. A number of these characterisations have already been presented elsewhere in this paper, see for example Lemma 22 and Lemma 30.

In fact, we only need new characterisations for \( nobuf_w \), \( nobuf_s \) and \( inst_2 \). For \( msg \) and \( pair \) the fact that weak and strong implementability specify the same classes of MSCs leads directly to an easy characterisation, while characterisations for \( global_w \), \( global_s \), \( inst_w \), \( inst_ou_t \) and \( inst_in \) already been given earlier in this paper.

\begin{definition}
Let \( k \) be an MSC over the set of messages \( M \). The relation \( <_w \) on \( M \) is for all \( m, m' \in M \) such that \( m \not= m' \) defined by \( m <_w m' \) if and only if \( !m <_k MSC \ m' \).
\end{definition}

\begin{lemma}
An MSC \( k \) is \( nobuf_w \)-implementable if and only if the relation \( <_w \) is cycle-free.
\end{lemma}

\begin{proof}
Let \( k \) be an MSC over the set of messages \( M \). First, suppose that \( k \) is \( nobuf_w \)-implementable. Suppose furthermore that \( <_w \) has a cycle, say \( m_1 <_w m_2 <_w \cdots <_w m_n <_w m_1 \). Then, from the definition of \( <_w \) and \( <_k MSC \), we obtain for all \( 1 \leq i < n \) that \( !m_i <_k MSC \ m_{i+1} \) and \( !m_{i+1} <_k MSC \ m_i \). Then, for every trace \( t \) of \( k \), we must have \( !m_i <_t trace \ m_{i+1} \) and \( !m_{i+1} <_t trace \ m_i \) for all \( 1 \leq i < n \). Since \( k \) is \( nobuf_w \)-implementable, there is a \( nobuf_w \)-implementable trace \( t' \). For this trace \( t' \) we also have \( !m_i <_t trace \ m_{i+1} \) for all \( 1 \leq i < n \). But then we have \( !m_1 <_t trace \ m_2 <_t trace \cdots <_t trace \ m_n \) and since \( !m_1 \equiv !m_n \) we thus have a cycle. Thus such a \( nobuf_w \)-implementable trace \( t' \) does not exist. This contradicts the assumption that \( k \) is \( nobuf_w \)-implementable. Therefore, \( <_w \) is cycle-free.

Second, suppose that \( <_w \) is cycle-free. We extend \( <_w \) to a total order \( < \), say \( m_1 < m_2 < \cdots < m_n \) where \( M = \{ m_1, m_2, \ldots, m_n \} \). Then the trace \( t = \{ m_1, m_2, m_2, \ldots, m_n \} \) is clearly \( nobuf_w \)-implementable. Thus, if suffices to prove that the trace \( t \) is a trace of MSC \( k \). Thereto, suppose that \( e <_w e' \) for some \( e, e' \in \{1/2\}* \). \( M \). We distinguish four cases:

- \( e \equiv !m \) and \( e' \equiv !m' \) for some \( m, m' \in M \). As \( !m <_w MSC \ m' \) and \( !m' <_w MSC \ m' \), we also have \( !m <_w MSC \ m' \). Then, by the definition of \( <_w \), we have \( m <_w m' \), and therefore \( !m <_t trace \ m' \).
\[ e = !m \text{ and } e' = ?m' \text{ for some } m, m' \in M. \] If \( m = m' \), then trivially \( !m <_t \text{trace} ?m' \). Otherwise, by the definition of \( <_k \), we have \( m < _k m' \), and therefore \( !m <_t \text{trace} ?m' \).

\[ e = ?m \text{ and } e' = !m' \text{ for some } m, m' \in M. \text{ As } !m <_k \text{msg} ?m \text{ and } ?m <_k \text{msg} ?m', \text{ we have } \!m <_k \text{msg} ?m'. \text{ Then, by the definition of } <_k, \text{ we have } m <_k m' \text{, and therefore } \!m <_t \text{trace} ?m'. \]

In each of the four cases we have \( e <_t \text{trace} e' \), which completes the proof.

**Lemma 37** If an MSC \( k \) is nobufs-implementable, then \( <_k \text{msg} \) is a total order.

**Proof** Let \( k \) be an MSC over the set of messages \( M \). We use contraposition, so assuming that \( <_k \text{msg} \) is not a total order, we prove that \( k \) is not nobufs-implementable. Let \( t \) be an arbitrary 2-trace of the MSC. Because \( <_k \text{msg} \) is not a total order, there are events \( e, e' \in \{!/!!/?\}M \) such that \( e <_t \text{trace} e' \), but not \( e <_k \text{msg} e' \). For any event \( e'' \in \{!/!!/?\}M \) with \( e <_t \text{trace} e'' <_t \text{trace} e' \) we have either \( e <_k \text{msg} e'' \) or \( e'' <_k \text{msg} e' \) as otherwise \( e <_k \text{msg} e' \). So there also is a such a pair of events that are immediately after one another in the trace \( t \). Then, interchanging these events would result in another trace \( t' \) of the MSC. It cannot be the case that both \( t \) and \( t' \) are nobufs-implementable.

**Lemma 38** An MSC \( k \) is inst2-implantable if and only if \( <_k \text{inst} = <_k \text{msg} \).

**Proof** Let \( k \) be an MSC over the set of messages \( M \). First, suppose that \( k \) is inst2-implantable. By definition, \( <_k \text{msg} \subseteq <_k \text{inst} \), so it only remains to be proven that \( <_k \text{inst} \subseteq <_k \text{msg} \). Suppose that \( e <_k \text{inst} e' \) for some \( e, e' \in \{!/!!/?\}M \). Then we have the existence of \( e_1, \ldots, e_n \) such that \( e \equiv e_1, e' \equiv e_n \) and for all \( 1 \leq i < n \) we have one of the following:

- \( e_i <_k e_{i+1} \);
- \( e_i = !m \text{ and } e_{i+1} = !m' \) for some \( m, m' \in M \) such that \( \text{from}(m) = \text{from}(m') \) and \( !m <_k \text{msg} !m' \);
- \( e_i = ?m \text{ and } e_{i+1} = !m' \) for some \( m, m' \in M \) such that \( \text{to}(m) = \text{to}(m') \) and \( ?m <_k \text{msg} !m' \).

In the second case we use induction on the number of output events \( !m'' \) that can be in between \( !m \) and \( !m' \) to prove that \( !m <_k \text{msg} !m' \).

- If there is no output event \( !m'' \) such that \( !m <_k \text{msg} !m'' <_k \text{msg} !m' \), then either \( !m <_k \text{inst} !m' \) or \( ?m <_k \text{msg} !m' \). In the first case, if \( !m <_k \text{msg} !m' \) did not hold, \( <_k \text{msg} \cup (\{!/!!/?\}!m) \) would be cycle-free. Any extension of this relation to a partial order would be \( <_t \text{trace} \) for a trace \( t \) that is not inst output-implementable, and thus not inst2-implementable. In the second case we have \( !m <_k \text{msg} ?m <_k \text{msg} !m' \).

- If there is at least one output event \( !m'' \) such that \( !m <_k \text{msg} !m'' <_k \text{msg} !m' \), then, using the induction hypothesis, we have \( !m <_k \text{msg} !m'' <_k \text{msg} !m' \).

For the third case a similar reasoning gives \( e_i <_k e_{i+1} \). Thus, in all cases we obtain \( e_i <_k e_{i+1} \) and therefore also \( e <_k e' \) which was to be proven.

Second, suppose that \( <_k \text{inst} \subseteq <_k \text{msg} \). Let \( t \) be a 3-trace of \( k \), and let \( m, m' \in M \). Suppose that \( \text{from}(m) = \text{from}(m') \) and \( !m <_t \text{trace} !m' \). Then we have that \( !m <_t \text{trace} !m' \) implies \( !m <_k \text{msg} !m' \). Then, by the
definition of \(<_{12}\), we have \(!m <_{k}^{12}!m'\). Since we assumed that \(<_{k}^{12} = <_{k}^{m3}\), we also have \(!m <_{k}^{m3}!m'\), and therefore \(!m <_{\text{trace}}!m'\).

Reversely, suppose that \(!m \not<_{\text{trace}}!m'\). With a similar reasoning as in the previous case we obtain \(!m \not<_{\text{trace}}!m'\). The proof that \(\tau_0(m) = \tau_0(m') \Rightarrow (!m <_{k}^{m3}!m' \Leftrightarrow !m <_{m3}^{m3}！m')\) is analogous.

In the following theorem we list the characterisations for implementability we have given in this paper and we add characterisations for the implementabilities not yet characterised.

**Theorem 39**

1. An MSC \(k\) is \(\text{nobuf}_w\)-implementable if and only if \(<_{k}^w\) is cycle-free.

2. An MSC \(k\) is \(\text{nobuf}_t\)-implementable if and only if it has exactly one trace, and that trace is \(\text{nobuf}\)-implementable.

3. An MSC \(k\) is \(\text{globals}\)-implementable if and only if for each pair of messages \(m\) and \(m'\) either both \(\text{lm} <_{k}^{\text{m3}}!m'\) and \(?m <_{k}^{\text{m3}}!m'\), or both \(\text{lm'} <_{k}^{\text{m3}}!m\) and \(?m' <_{k}^{\text{m3}}!m\) hold.

4. An MSC \(k\) is \(\text{global}_w\)-implementable if and only if \(<_{k}^g\) is cycle-free.

5. An MSC \(k\) is \(\text{inst} \_\text{outs}\)-implementable if and only if \(<_{k}^{\text{inst} \_\text{outs}} = <_{k}^{\text{m3}}\).

6. An MSC \(k\) is \(\text{inst} \_\text{ins}\)-implementable if and only if \(<_{k}^{\text{inst} \_\text{ins}} = <_{k}^{\text{m3}}\).

7. An MSC \(k\) is \(\text{inst}2_s\)-implementable if and only if \(<_{2}^{\text{inst}2_s} = <_{k}^{m3}\).

8. An MSC \(k\) is \(\text{inst}_w\)-implementable if and only if \(<_{k}^{\text{inst}_w}\) is cycle-free.

9. An MSC \(k\) is \(\text{pair}\)-implementable if and only if a randomly chosen \(t\) \(\text{pair}\)-implementable is.

10. An MSC \(k\) is always \(\text{msg}\)-implementable.

**Proof**

1. See Lemma 36.

2. If the MSC \(k\) is \(\text{nobuf}_t\)-implementable it has one trace because \(<_{k}^{\text{m3}}\) is a total order (Lemma 37).

3. See Lemma 33.

4. See Lemma 30.

5. See Lemma 28.


7. See Lemma 38.

8. See Lemma 22.

9. First, if \(k\) is \(\text{pair}\)-implementable, it is \(\text{pair}_t\)-implementable and thus every trace \(t\) of \(k\) is \(\text{pair}\)-implementable. Second, if a randomly chosen trace \(t\) is \(\text{pair}\)-implementable, then \(k\) is \(\text{pair}_w\)-implementable, and thus also \(\text{pair}_t\)-implementable.

10. See Lemma 27.
7 Comparison

In this section we will compare our conclusions with those found in related literature.

In [CBMT96] Charron-Bost et al. discuss three different implementations for MSC-like diagrams: RSC (Realizable with Synchronous Communication), CO (Causally Ordered) and FIFO. They also define A (asynchronous), but this is (just like msg in our hierarchy) used to denote the set of all allowable diagrams, not some subset. They find that there is a strict ordering RSC \subset CO \subset FIFO \subset A.

**Theorem 40** The implementations that in [CBMT96] are named RSC and FIFO are equal to the implementations nobuf\(_w\), and pair. The implementation CO is strictly between the implementations inst\(_w\) and pair.

**Proof**

- **RSC-nobuf\(_w\)**: Definition 3.6 in [CBMT96] states, after translating it into our terminology, that a computation is RSC if and only if there is a trace \(t\) for which for each \(m \in M\) we have that the set \(\{x \in C \mid \exists m <_{trace} x <_{trace} m\}\) is empty, which is equal to the definition that is obtained by combining Lemma 15 and Definition 19.

- **FIFO-pair**: The definition for FIFO in [CBMT96] (Definition 3.3) translates to (by rewriting the terminology of Charron-Bost et al. in ours): \(\exists m <_{msc} m'\land \text{from}(m) = \text{from}(m') \land \text{to}(m) = \text{to}(m') \Rightarrow \exists m <_{msc} m'\), or \(\exists m = \text{from}(m') \land \text{to}(m) = \text{to}(m') \Rightarrow \exists m <_{msc} m'\), which is seen to be equivalent to the definition in Lemma 15 once it is realized that (for the basic MSCs considered here) \(\text{to}(m) = \text{to}(m') \Rightarrow (\exists m <_{msc} m'\lor m' <_{msc} m)\) and \(\text{from}(m) = \text{from}(m') \Rightarrow (\exists m <_{msc} m'\lor m' <_{msc} m)\).

- **CO**: That the class of pair-implementable MSCs is strictly greater than that of CO-implementable MSCs is shown in [CBMT96]. Remains to be shown that the class of CO-implementable MSCs is strictly greater than that of inst\(_w\)-implementable MSCs. The definition of CO as given in [CBMT96] (definition 3.4) can be translated to \(\text{to}(m) = \text{to}(m') \land \exists m <_{msc} m'\), where \(m <_{msc} m'\). An example of an MSC that is CO-implementable, but not inst\(_w\)-implementable, is the MSC 'lobster' in Figure 13. It is CO-implementable, because there is no pair of messages with \(\text{to}(m) = \text{to}(m')\) where \(m <_{msc} m'\). It remains to be proven that each inst\(_w\)-implementable MSC is CO-implementable. We do this using contraposition, so let \(k\) be an MSC that is not CO-implementable. We then have that there are messages with \(m <_{msc} m'\), \(m <_{msc} m'\) and \(\exists m = \text{to}(m) = \text{to}(m')\). From the last two we can derive that \(m <_{msc} m', \text{and thus we have both } m <_{msc} m' <_{msc} m', \text{so the MSC } k \text{ is not inst}_w\)-implementable.

Another paper in which different communication models for MSC have been studied, is [AHP96]. The models from our hierarchy are incomparable with their models, because the ordering of certain combinations of events on an instance is subject to a chosen communication model, thereby relaxing our fundamental total ordering of events on an instance.
8 Concluding remarks and future research

We have considered implementation models for asynchronous communication in Message Sequence Chart. These models contain of FIFO buffers for the sending and reception of messages. By varying the locality of the buffers we have arrived, in a systematic way, at 25 models for communication. With respect to traces, consisting of putting a message into a buffer and removing a message from a buffer, there are seven different models.

By lifting this implementability notion from traces to Message Sequence Charts in two ways, strong and weak, we obtain fourteen models. After identification, ten essentially different models on the level of Message Sequence Charts remain.

For defining the models we have used the notion of 3-traces; these are a natural extension of normal MSC-traces if a message can pass two buffers on its way from source to destination.

In this paper, we have only considered Basic Message Sequence Charts. An interesting question is how to transfer the notions and properties defined for this simple language to the complete language MSC. As many of our theorems rely on the fact that the events on an instance are totally ordered, an extension to MSC with more sophisticated ordering mechanisms (e.g., coregion and causal ordering) will imply a revision of the hierarchy. Another interesting question is whether the implementation properties are preserved under composition by means of the operators of MSC.

Furthermore, we have restricted ourselves to the treatment of architectures in which each message has exactly one possible communication path and where each such path contains at most two buffers. The extension to more flexible architectures is non-trivial and is expected to lead to an extension of the hierarchy.

Finally, our assumption of infinite FIFO buffers may be relaxed, allowing other types of buffers and buffers with finite capacity.

The results obtained in this paper form a solid base for several applications. First, they allow us to discuss the relation between different variants of MSC, such as Interworkings [MWW93]. Interworkings presuppose a synchronous communication mechanism. An Interworking can be considered as the restriction of the semantics of an MSC to only the nobuf-implementable traces. Thus, an MSC can be interpreted as an Interworking if and only if there is at least one such trace, i.e., the MSC is nobuf$_{w}$-implementable. We also envisage more practical applications. Consider a tool in which a user can select a communication model, draw an MSC and invoke an algorithm to check if the MSC is implementable with respect to the selected model. Alternatively, the user can provide an MSC and use a tool to determine the minimal architecture, according to our hierarchy, which is needed for implementation.

Often a user is interested in the question whether all traces of his MSC are implementable with respect to a certain architecture. We can also envisage two possible uses relying on the implementability of a single trace. First, MSCs are often used to display one single trace, for example if it is the
result of a simulation run. In this case, the question is not whether the MSC is strongly or weakly implementable, but whether the implied trace is implementable (as defined in Section 4). Second, given an MSC, a user may want to know if at least one trace is implementable and if so, which trace that is. He is interested in a witness. Both applications can easily be derived from the results on weak implementability. The algorithms (see below) can easily be modified to check implementability of a given trace and to produce a witness.

A more involved application would be to use a selected communication model to reduce the set of traces defined by a given MSC to only those traces that are implementable on the given model. In this way, the semantics of an MSC would be relative to some selected model.

For most of these applications computer support would be useful. Based upon the definitions presented in this paper, it is feasible to derive efficient algorithms. All models in the weak-spectrum can be characterised in terms of the cycle-freeness of an extended ordering relation. An example of such a characterisation is given in Theorem 22. There it is stated that an MSC $k$ is $\text{inst.out}_w$-implementable iff the ordering $\prec^o_k$ (which is an extension of $\prec^m_{\text{MSC}}$) is cycle-free. Thus checking if an MSC is $\text{inst.out}_w$-implementable boils down to checking cycle-freeness of this relation. This immediately gives a wide range of efficient implementations for checking class-membership as many algorithms are known in literature for determining whether a given ordering is cycle-free. For the strong spectrum characterisations are given as well.

Note that the MSCs that distinguish between the different models are surprisingly simple. This indicates that the differences between the classes will appear not only in theory, but also in practice. Besides that, for these distinguishing MSCs, it is not easy to indicate at a glance to which class they do or do not belong. This also supports our view that mechanical support for determining whether a given Message Sequence Chart belongs to a given class is necessary.
References


### Computing Science Reports

**In this series appeared:**

<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>96/01</td>
<td>M. Voorhoeve and T. Basten</td>
<td>Process Algebra with Autonomous Actions, p. 12.</td>
</tr>
<tr>
<td>96/02</td>
<td>P. de Bra and A. Aerts</td>
<td>Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12</td>
</tr>
<tr>
<td>96/04</td>
<td>S. Mauw</td>
<td>Example specifications in phi-SDL.</td>
</tr>
<tr>
<td>96/06</td>
<td>W.M.P. van der Aalst and T. Basten</td>
<td>Life-Cycle Inheritance A Petri-Net Based Approach, p. 18.</td>
</tr>
<tr>
<td>96/07</td>
<td>M. Voorhoeve</td>
<td>Structural Petri Net Equivalence, p. 16.</td>
</tr>
<tr>
<td>96/11</td>
<td>T. Laan</td>
<td>AUTOMATH and Pure Type Systems, p. 30.</td>
</tr>
<tr>
<td>96/12</td>
<td>F. Kamareddine and T. Laan</td>
<td>A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.</td>
</tr>
<tr>
<td>96/13</td>
<td>T. Borghuis</td>
<td>Priorian Tense Logics in Modal Pure Type Systems, p. 61</td>
</tr>
<tr>
<td>96/14</td>
<td>S.H.J. Bos and M.A. Reniers</td>
<td>The $I^4$-C-bus in Discrete-Time Process Algebra, p. 25.</td>
</tr>
<tr>
<td>96/15</td>
<td>M.A. Reniers and J.J. Vereijken</td>
<td>Completeness in Discrete-Time Process Algebra, p. 139.</td>
</tr>
<tr>
<td>96/17</td>
<td>E. Boiten and P. Hoogendijk</td>
<td>Nested collections and polytypism, p. 11.</td>
</tr>
<tr>
<td>96/18</td>
<td>P.D.V. van der Stok</td>
<td>Real-Time Distributed Concurrency Control Algorithms with mixed time constraints, p. 71.</td>
</tr>
<tr>
<td>96/19</td>
<td>M.A. Reniers</td>
<td>Static Semantics of Message Sequence Charts, p. 71</td>
</tr>
<tr>
<td>96/20</td>
<td>L. Feijs</td>
<td>Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.</td>
</tr>
<tr>
<td>96/23</td>
<td>W.M.P. van der Aalst</td>
<td>Structural Characterizations of sound workflow nets, p. 22.</td>
</tr>
<tr>
<td>96/24</td>
<td>M. Voorhoeve and W. van der Aalst</td>
<td>Conservative Adaption of Workflow, p.22</td>
</tr>
<tr>
<td>96/25</td>
<td>M. Vaccari and R.C. Backhouse</td>
<td>Deriving a systolic regular language recognizer, p. 28</td>
</tr>
<tr>
<td>97/06</td>
<td>M. Franssen</td>
<td>Tools for the Construction of Correct Programs: an Overview, p. 33.</td>
</tr>
<tr>
<td>97/07</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Bounded Stacks, Bags and Queues, p. 15.</td>
</tr>
<tr>
<td>Year</td>
<td>Authors/Submitted to</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>-------</td>
</tr>
<tr>
<td>97/08</td>
<td>P. Hoogendijk and R. C. Backhouse</td>
<td>When do datatypes commute?</td>
</tr>
<tr>
<td>97/10</td>
<td>P.C.N. v. Gorp, E.J. Luit, D.K. Hammer and E.H.L. Aarts</td>
<td>Distributed real-time systems: a survey of applications and a general design model</td>
</tr>
</tbody>
</table>