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.

Keywords

Message Sequence Charts, semantics, implementation, validation, buffering, communication models, 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 (ITU-TS 1996). Two important reasons for

the popularity of MSCs 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 (Mauw and

Reniers 1994, Ladkin and Leue 1995, Grabowski, Graubmann and Rudolph 1993). The process algebra approach (Mauw and Reniers 1994) has been standardised as Annex B to ITU recommendation Z.120 (ITU-TS 1995). The partial order representation (Alur, Holzmann and Peled 1996) 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.

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 3. MSCs 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 (Mauw, Wijk and Winter 1993) every message is considered to be a single event. There is no buffering, and thus communication is synchronous.

In MSC (ITU-TS 1996), 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 implementation of communication is that an output precedes its corresponding input. This corresponds to the most general implementation model in which processes communicate via unbounded random-access buffers.

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 $\langle I, M, \mathit{from}, \mathit{to}, \{<_i\}_{i\in I}\rangle$, 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 $\{!m \mid \mathit{from}(m) = i\} \cup \{?m \mid \mathit{to}(m) = i\}$.

In the above definition, $\mathit{from}(m)$ denotes the instance which sends message m. Like-

wise, $to(m)$ 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 $<^{inst} := \bigcup_{i \in I} <_i$, and the second requirement is formalised by the *output-before-input* order $<^{oi} := \{(!m, ?m) \mid m \in M\}$.

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 $<_k^{msc}$ or by $<^{msc}$ if k is known from the context.

Definition 2 For a given MSC $k = \langle I, M, from, to, \{<_i\}_{i \in I}\rangle$, the relation $<_k^{msc}$ is defined by $<_k^{msc} := (<^{inst} \cup <^{oi})^+$.

We define similar notions for 3-traces. We define the *output-before-transmit-before-input* order by $<^{oti} := \{(!m, !!m), (!!m, ?m) \mid m \in M\}$, and the relation $<^{m3}$ by adding the instancewise ordering on the MSC.

Definition 3 For a given MSC $k = \langle I, M, from, to, \{<_i\}_{i \in I}\rangle$, the ordering $<^{m3}$ is defined by $<^{m3} := (<^{inst} \cup <^{oti})^+$.

It is easy to see that $<^{msc}$ is the restriction of $<^{m3}$ to output and input events. It may be the case that $<^{msc}$ 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 (ITU-TS 1996) inconsistent MSCs are considered illegal, and in (Ben-Abdallah and Leue 1997) 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 $<^{msc}$ and $<^{m3}$ are partial orders.

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 ($!m$ and $?m$), a 3-trace those of transmit events ($!!m$) as well.

Definition 4 (2-traces, 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 $\{!m, ?m \mid m \in M\}$. This ordering is denoted by $<_t^{trace}$. A trace $(e_1, e_2, \ldots, e_n)$ is denoted $e_1\, e_2\, \ldots\, e_n$. A *3-trace* is equal to a 2-trace, except for the fact that it contains transmit events as well.

Definition 5 (MSC-trace) A 2-trace t is said to be a trace of the MSC k iff it is defined over the messages M of k, and $<_k^{msc} \subseteq <_t^{trace}$. A 3-trace t is said to be a trace of the MSC k iff it is defined over the messages M of k, and $<_k^{m3} \subseteq <_t^{trace}$.

A 3-trace can be turned into a 2-trace by removing all transmit events ($!!m$). 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 iff the 2-trace of which it is an extension is a trace of the MSC and $<_k^{oti} \subseteq <_t^{trace}$.

For MSC 2a in Figure 3 the following orderings hold: $!a <^{msc} ?a$, $!b <^{msc} ?b$, and $?a <^{msc} ?b$. The first two are implied by the $<^{oi}$-order, the third by the $<^{inst}$-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 $!a\ !b\ !!b\ !!a\ ?a\ ?b$.

3 IMPLEMENTATION MODELS

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. 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 model of the MSC standard: the only assumption made about the implementation of communication is that output precedes input.

Finally, we consider the following possibility of using no buffers at all, denoted by *nobuf*. In this case 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. To denote the different architectures, we use the notation (X,Y), where X denotes the type of output buffer, and Y the type of input buffer.

3.2 Examples of communication models

In Figure 1 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 (*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 (*global*,*nobuf*) too. The second example shows the (*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 (*pair*,*pair*) model.

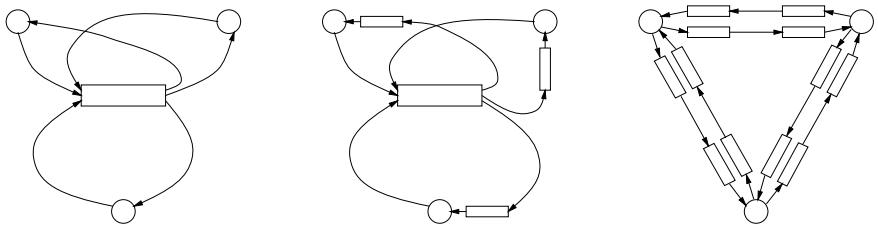


Figure 1 Some models: (*nobuf*,*global*), (*global*,*inst*) and (*pair*,*pair*).

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 (*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 (*pair*,*msg*), but if we leave out the *save* construct we obtain (*pair*,*inst*) and if we also do not consider the possibility of delayed channels, we have (*nobuf*,*inst*). Some examples of physical architectures are: an asynchronous complete mesh has a (*nobuf*,*pair*) architecture, and an Ethernet connection with locally buffered input and output behaves like (*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 6 (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 iff $\forall_{m\in M} \neg \exists_{e\in\{!m,!!m,?m|m\in M\}} !m <_t^{trace} e <_t^{trace} !!m$.

- *global-output*: The order of two output events is respected by the corresponding transmit events. A 3-trace t is *global*-output implementable iff $\forall_{m,m'\in M} !m <_t^{trace} !m' \Leftrightarrow !!m <_t^{trace} !!m'$.
- *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 iff $\forall_{m,m'\in M}\ from(m) = from(m') \Rightarrow (!m <_t^{trace} !m' \Leftrightarrow !!m <_t^{trace} !!m')$.
- *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 iff $\forall_{m,m'\in M}\ from(m) = from(m') \wedge to(m) = to(m') \Rightarrow (!m <_t^{trace} !m' \Leftrightarrow !!m <_t^{trace} !!m')$.
- *msg-output*: A 3-trace t is always *msg*-output implementable.

The input implementabilities are defined analogously.

Definition 7 (Input-implementability) A 3-trace t is

- *nobuf*-input implementable iff $\forall_{m\in M} \neg \exists_{e\in\{!m,!!m,?m|m\in M\}} !!m <_t^{trace} e <_t^{trace} ?m$;
- *global*-input implementable iff $\forall_{m,m'\in M} !!m <_t^{trace} !!m' \Leftrightarrow ?m <_t^{trace} ?m'$;
- *inst*-input implementable iff $\forall_{m,m'\in M}\ to(m) = to(m') \Rightarrow (!!m <_t^{trace} !!m' \Leftrightarrow ?m <_t^{trace} ?m')$;
- *pair*-input implementable iff $\forall_{m,m'\in M}\ from(m) = from(m') \wedge to(m) = to(m') \Rightarrow (!!m <_t^{trace} !!m' \Leftrightarrow ?m <_t^{trace} ?m')$;
- 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 8 A 3-trace is (*X*,*Y*)-*implementable* (for $X, Y \in \{$*nobuf, global, inst, pair, msg*$\}$) iff it is *X*-output implementable and *Y*-input implementable. A 2-trace is (*X*,*Y*)-*implementable* iff it can be extended 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 9 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 9 (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.

For the proof of this lemma, and of the other lemmas and theorems for which no proof is given in this paper, we refer to (Engels, Mauw and Reniers 1997).

The following lemmas give the orderings between the implementation models.

Lemma 10 Every (*global*, *global*)-implementable 2-trace is (*global*, *nobuf*)-implementable. Every (*inst*,*global*)-implementable 2-trace is (*inst*,*nobuf*)-implementable. Every (*pair*,*pair*)-implementable 2-trace is (*pair*,*nobuf*)-implementable. Every (*msg*, *msg*)-implementable 2-trace is (*msg*,*nobuf*)-implementable.

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 9 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. 10, together with the order provided by Lemma 9, gives us the equivalences as expressed in Figure 2 by means of the clustering of implementation models.

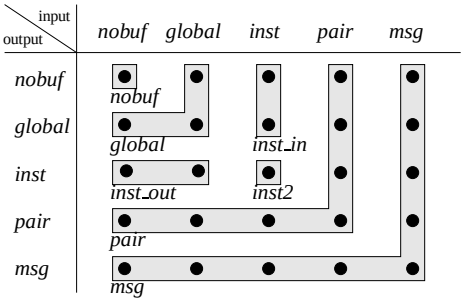


Figure 2 Equivalence of implementation models for traces.

For example, the models from the last column are equivalent. This can be seen as follows. Because of the analogue of Lemma 10, any (*msg*,*msg*)- implementable 2-trace is (*nobuf*,*msg*)-implementable, while Lemma 9 gives that any (*nobuf*,*msg*)-implementable 2-trace is (*X*,*msg*)-implementable, and every (*X*,*msg*)-implementable 2-trace is (*msg*,*msg*)-implementable.

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. That this is not the case, will be seen in Theorem 12 below. We name the equivalence classes as follows: *nobuf*, *global*, *inst_out*, *inst_in*, *inst2*, *pair*, *msg* (see Figure 2).

Note that of these seven cases only *inst2* is not of the form (*X*, *nobuf*) or (*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.

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 iff every 2-trace of the MSC can. The second is to define that an MSC can be implemented in a certain implementation model iff 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 11 An MSC k is said to be *strongly X-implementable*, notation X_s-implementable, iff all 2-traces t of k are X-implementable.

From this definition it follows immediately that the ordering of the implementation models for traces also holds for MSCs as far as strong implementability is concerned (see the left part of Figure 5). 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.

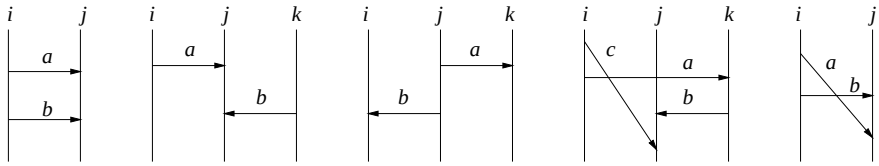


Figure 3 MSCs to distinguish the implementation models: strong case.

MSC 1 in Figure 3 shows an example that is $global_s$-implementable, but not $nobuf_s$-implementable. It is not $nobuf_s$-implementable, because the trace $!a\ !b\ ?a\ ?b$ is not. The inputs necessarily have to be ordered in the same way as the outputs, so it is $global_s$-implementable.

MSC 2a is $inst\_out_s$-implementable, but not $global_s$-implementable due to the trace $!b\ !a\ ?a\ ?b$. That MSC 2a is $inst\_out_s$-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 $inst\_in_s$-implementable, but not $global_s$-implementable due to the trace $!a\ !b\ ?b\ ?a$.

MSCs 2*a* and 2*b* show the difference between $inst\_out_s$ and $inst\_in_s$. MSC 2*a* is $inst\_out_s$-implementable, as mentioned before, but not $inst\_in_s$-implementable. The trace $!b\,!a\,?a\,?b$ is not *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 2*b* the reverse is the case: It is $inst\_in_s$-implementable, but not $inst\_out_s$-implementable. MSC 2*a* is $inst\_out_s$-implementable and therefore also $inst2_s$-implementable. We have already established that it is not $inst\_in_s$-implementable. Similarly, MSC 2*b* is $inst\_in_s$ and $inst2_s$-implementable, but not $inst\_out_s$-implementable. Together, these show that $inst\_out_s$, $inst\_in_s$ and $inst2_s$ are all different.

MSC 3 is an example of an MSC that is $pair_s$-implementable, but not $inst2_s$-implementable. It is easy to see that it is $pair_s$-implementable, because each message goes through a different buffer. Its only 2-trace is $!c\,!a\,?a\,!b\,?b\,?c$. If we try to extend this to an *inst2*-implementable 3-trace t', we need to have $!!c <_t^{trace} !!a <_t^{trace} !!b <_t^{trace} !!c$, which is impossible (the first $<_t^{trace}$ is because of the *inst*-output implementability and $!c <_t^{trace} !a$, the second is clearly true for every 3-trace of the MSC, and the third is because of the *inst*-input implementability together with $?b <_t^{trace} ?c$).

Finally, MSC 4 shows the difference between $pair_s$- and msg_s-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 5.

Together the examples 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 12 The classes *nobuf, global, inst_out, inst_in, inst2, pair,* and *msg* are different for implementability of traces and for strong implementability of MSCs, and for strong implementability they are ordered as shown in the left part of Figure 5.

5.2 Weak implementability

Definition 13 An MSC k is said to be *weakly X-implementable*, notation X_w-implementable, iff 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 for traces 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 14 Let k be an MSC over the set of messages M. Then we define the relations $<_k^{io}$ and $<_k^{ii}$ on $\{!m, ?m \mid m \in M\}$ and the relation $<_k^{i2}$ on $\{!m, !!m, ?m \mid m \in M\}$ as follows:

$$<_k^{io} := (<_k^{msc} \cup \{(?m, ?m') \mid m, m' \in M \wedge from(m) = from(m') \wedge !m <_k^{msc} !m'\})^+,$$

$$<_k^{ii} := (<_k^{msc} \cup \{(!m, !m') \mid m, m' \in M \wedge to(m) = to(m') \wedge ?m <_k^{msc} ?m'\})^+,$$

$$<_k^{i2} := (<_k^{m3} \cup \{(!!m, !!m') \mid m, m' \in M \wedge \mathit{from}(m) = \mathit{from}(m') \wedge !m <_k^{m3} !m'\} \\ \cup \{(!!m, !!m') \mid m, m' \in M \wedge \mathit{to}(m) = \mathit{to}(m') \wedge ?m <_k^{m3} ?m'\})^+.$$

We explain the definition of the ordering $<_k^{io}$ which is defined in order to check the *inst_out*-property. The ordering is obtained from $<_k^{msc}$ 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 *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 *inst_out*-implementable traces of the MSC are also traces of the ordering $<_k^{oi}$ as they respect the requirements for *inst_out*-implementability by definition, and vice versa. Basically this is what is expressed in Theorem 15.

Theorem 15 Let t be a 2-trace of an MSC k. Then, t is *inst_out*-implementable iff $<_k^{io} \subseteq <_t^{trace}$, t is *inst_in*-implementable iff $<_k^{ii} \subseteq <_t^{trace}$, and t is *inst2*-implementable iff there exists a 3-trace t' which is an extension of t such that $<_k^{i2} \subseteq <_{t'}^{trace}$.

Proof. We only give the proof for the last proposition. The proofs for the first two propositions follow the same line. Suppose that t is *inst2*-implementable. Then we must prove that $<_k^{i2} \subseteq <_{t'}^{trace}$ for some 3-trace t' which is an extension of t. For t' we choose any *inst2*-implementable 3-trace of t. It suffices to prove that $e <_{t'}^{trace} e'$ for an arbitrary pair of events $e, e' \in \{!m, !!m, ?m | m \in M\}$ with $e <_k^{i2} e'$. Since $e <_k^{i2} e'$ we have the existence of $e_1, \ldots, e_n$ such that $e \equiv e_1 < e_2 < \cdots < e_n \equiv e'$ where for all $1 \leq i < n$ we have one of the following:

- $e_i <_k^{m3} e_{i+1}$;
- $e_i \equiv !!m$, $e_{i+1} \equiv !!m'$, $\mathit{from}(m) = \mathit{from}(m')$ and $!m <_k^{m3} !m'$ for some $m, m' \in M$;
- $e_i \equiv !!m$, $e_{i+1} \equiv !!m'$, $\mathit{to}(m) = \mathit{to}(m')$ and $?m <_k^{m3} ?m'$ for some $m, m' \in M$.

For all of these cases we can conclude that $e_i <_{t'}^{trace} e_{i+1}$, and hence, $e <_{t'}^{trace} e'$.

Next, suppose that $<_k^{i2} \subseteq <_{t'}^{trace}$ for some extension t' of t. We must prove that t is (*inst*,*inst*)-implementable. Thereto it suffices to show that t' is (*inst*,*inst*)-implementable, i.e., that t' is *inst*-output implementable and *inst*-input implementable. We prove that t' is *inst*-output implementable, the proof that t' is *inst*-input implementable is analogous. Let $m, m' \in M$ such that $\mathit{from}(m) = \mathit{from}(m')$. Then it suffices to show that $!m <_{t'}^{trace} !m' \Leftrightarrow !!m <_{t'}^{trace} !!m'$. Thereto, suppose that $!m <_{t'}^{trace} !m'$. Since $\mathit{from}(m) = \mathit{from}(m')$, we have $!m <_k^{msc} !m'$. So $!!m <_k^{i2} !!m'$. Because $<_k^{i2} \subseteq <_{t'}^{trace}$ we therefore have $!!m <_{t'}^{trace} !!m'$. Suppose that $!m \not<_{t'}^{trace} !m'$. Then $!m' <_{t'}^{trace} !m$. With similar reasoning as before we obtain $!!m' <_{t'}^{trace} !!m$. Therefore, $!!m \not<_{t'}^{trace} !!m'$. □

Thus far, we have seen that the ordering $<_k^{io}$ contains all *inst_out*-implementable traces of MSC k. An MSC k is *inst_out*$_w$-implementable iff it has a trace t that is

inst_out-implementable. Clearly, such a trace exists iff there is a trace for the ordering $<_k^{io}$, in other words, iff $<_k^{io}$ is cycle-free. Therefore, we have the following characterisation theorem, which follows directly from Theorem 16.

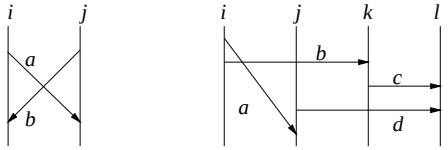
Theorem 16 An MSC k is $inst\_out_w$-implementable iff $<_k^{io}$ is cycle-free. An MSC k is $inst\_in_w$-implementable iff $<_k^{ii}$ is cycle-free. An MSC k is $inst2_w$-implementable iff $<_k^{i2}$ is cycle-free.

We use the alternative characterisations provided by the previous theorem in the proof of the equivalence of the classes $inst\_out_w$, $inst\_in_w$, and $inst2_w$.

Theorem 17 The implementation models $inst\_out_w$, $inst\_in_w$, and $inst2_w$ are equal.

Proof. We show that each $inst2_w$-implementable MSC is $inst\_out_w$-implementable. The reverse implication is trivial, and the proofs with $inst\_in_w$ are analogous. From Theorem 16 we see that it suffices to prove that $<^{io}$ is cycle-free if $<^{i2}$ is cycle-free. We prove this using contraposition, so we assume that $<^{io}$ has a cycle. Let $e_1 <^{io} e_2 <^{io} \cdots <^{io} e_n <^{io} e_1$ be an arbitrary largest cycle. For every ordering in the cycle, say $e_i <^{io} e_{i+1}$ either $e_i <^{msc} e_{i+1}$, and hence $e_i <^{i2} e_{i+1}$, or $e_i \equiv ?m$, $e_{i+1} \equiv ?m'$ for some $m, m' \in M$ such that $!m <^{msc} !m'$. If the first is always the case, then we have a cycle in $<^{msc}$, so certainly in $<^{i2}$. 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 $?m <^{io} ?m'$ and $?m' <^{io} ?m$. As is shown in (Engels et al. 1997), this implies that $!!m <^{i2} !!m'$ and $!!m' <^{i2} !!m$. Thus clearly $<^{i2}$ has a cycle. □

Theorem 17 establishes that the classes $inst\_out_w$, $inst\_in_w$, and $inst2_w$ are equivalent. In the remainder we denote this class by $inst_w$. The remaining models are all different. The MSCs 3 and 4 in Figure 3 show the difference between $inst_w$ and $pair_w$, and $pair_w$ and 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 4 is $global_w$-implementable, but not $nobuf_w$-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 5 MSC 6

Figure 4 MSCs to distinguish the implementation models: weak case.

MSC 6 is $inst_w$-implementable, but not $global_w$-implementable. It is not $global_w$-implementable, as can be seen thus: $!a <^{msc} !b$, so for every *global*-implementable

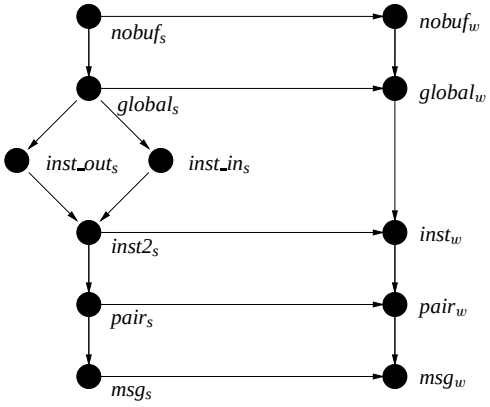


Figure 5 Incomplete hierarchy.

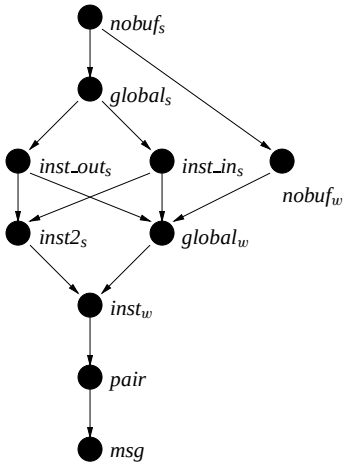


Figure 6 Final hierarchy.

trace t we must have $?a <_t^{trace} ?b$. Because $!d <^{msc} ?a$ and $?b <^{msc} !c$, we get $!d <_t^{trace} !c$. But we also have $?c <^{msc} ?d$, and thus $?c <_t^{trace} ?d$, from which it follows that t cannot be *global*-implementable. On the other hand, the trace $!a\ !b\ !d\ ?a\ ?b\ !c\ ?c\ ?d$ is *inst_out*-implementable, so the MSC is $inst_w$-implementable.

Theorem 18 The implementation models for weak implementability of the right part of Figure 5 are all different.

5.3 Combining the strong and weak hierarchy

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 5.

Theorem 19 establishes 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.

Theorem 19 An MSC is $pair_s$-implementable iff it is $pair_w$-implementable. An MSC is msg_s-implementable iff it is msg_w-implementable. Every $inst\_out_s$-implementable or $inst\_in_s$-implementable MSC is $global_w$-implementable.

In Figure 6 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 $inst\_in_s$ and $inst2_s$ to $nobuf_w$. We use the MSCs in Figure 7 to indicate that the first two arrows do not exist. MSC 7 is $global_s$-implementable, but not $nobuf_w$-implementable. On the other hand MSC 8 is $nobuf_w$-implementable, but not $inst2_s$-implementable. The trace $!a\ ?a\ !b\ ?b\ !c\ ?c$ is *nobuf*-implementable, while the trace $!b\ !c\ ?c\ !a\ ?a\ ?b$ is not *inst2*-implementable.

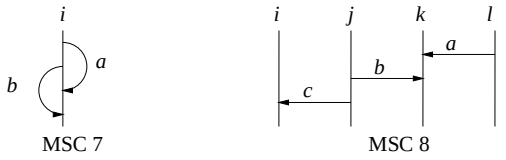


Figure 7 Distinguishing MSCs: comparing strong and weak.

The non-existence of an arrow from $inst2_s$ to $global_w$ is taken care of by MSC 6 in Figure 3. It has already been shown not to be $global_w$-implementable. It is $inst2_s$-implementable because every 2-trace of this MSC can be extended to an *inst2*-implementable 3-trace by adding $!!a$ and $!!b$ immediately after $!a$ and $!b$, and $!!c$ and $!!d$ immediately before $?c$ and $?d$.

Theorem 20 The implementation models from Figure 6 are all different, and they are ordered as expressed in Figure 6.

6 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 (Mauw et al. 1993). 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, as is shown in (Engels et al. 1997). An example of such a characterisation is given in Theorem 16. There it is stated that an MSC k is $inst\_out_w$-implementable iff the ordering $<_k^{io}$ (which is an extension of $<_k^{msc}$) is cycle-free. Thus checking if an MSC is $inst\_out_w$-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 liter-

ature for determining whether a given ordering is cycle-free. For the strong spectrum characterisations are given in (Engels et al. 1997) as well.

There are two papers in which a similar subject is discussed. In (Charron-Bost, Mattern and Tel 1996) four different implementations for MSC-like diagrams are discussed: *RSC* (Realizable with Synchronous Communication), *CO* (Causally Ordered), *FIFO* and *A* (Asynchronous). They find that there is a strict ordering $RSC \subset CO \subset FIFO \subset A$. As shown in (Engels et al. 1997), the implementations *RSC*, *FIFO* and *A* correspond to our implementations $nobuf_w$, *pair* and *msg*, while *CO* is positioned strictly between the implementations $inst_w$ and *pair*.

Another paper in which different communication models for MSC have been studied, is (Alur et al. 1996). 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.

REFERENCES

Alur, R., Holzmann, G. J. and Peled, D.: 1996, An analyzer for Message Sequence Charts, *Software - Concepts and Tools* **17**(2), 70–77.

Ben-Abdallah, H. and Leue, S.: 1997, Syntactic detection of process divergence and non-local choice in Message Sequence Charts, *in* E. Brinksma (ed.), *Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in *Lecture Notes on Computer Science*, Springer Verlag, pp. 259–274.

Charron-Bost, B., Mattern, F. and Tel, G.: 1996, Synchronous, asynchronous and causally ordered communication, *Distributed Computing* **9**(4), 173–191.

Engels, A., Mauw, S. and Reniers, M. A.: 1997, A hierarchy of communication models for Message Sequence Charts, *Technical Report CSR 97-11*, Eindhoven University of Technology, Department of Computing Science.

Grabowski, J., Graubmann, P. and Rudolph, E.: 1993, Towards a Petri net based semantics definition for Message Sequence Charts, *in* O. Færgemand and A. Sarma (eds), *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, North-Holland, pp. 179–190.

ITU-TS: 1995, *ITU-TS Recommendation Z.120 Annex B: Algebraic semantics of Message Sequence Charts*, ITU-TS, Geneva.

ITU-TS: 1996, *ITU-TS Draft Recommendation Z.120: Message Sequence Chart 1996 (MSC96)*, ITU-TS, Geneva.

Ladkin, P. and Leue, S.: 1995, Interpreting message flow graphs, *Formal Aspects of Computing* **7**(5), 473–509.

Mauw, S. and Reniers, M. A.: 1994, An algebraic semantics of Basic Message Sequence Charts, *The Computer Journal* **37**(4), 269–277.

Mauw, S., Wijk, M. v. and Winter, T.: 1993, A formal semantics of synchronous Interworkings, *in* O. Færgemand and A. Sarma (eds), *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, North-Holland, pp. 167–178.