Message Refinement: Describing Multi-Level Protocols in MSC

A. Engels
Eindhoven University of Technology,
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands.
engels@win.tue.nl

Abstract

We propose a mechanism by which single messages in a Message Sequence Chart can be used to describe a full communication protocol, which in turn is described by another MSC. In this way, an MSC can be described at one level of abstraction while neither having to worry about lower levels, nor losing them altogether. We show when such a replacement is allowed without causing deadlocks. For this purpose, it appears useful to introduce the concept of synchronous communication, which is also discussed. Together these subjects might provide a small, but useful extension of the MSC language.

Keywords
MSC, refinement, message refinement, protocols, synchronous communication, abstraction

1 Introduction

1.1 Motivation

One of the areas where Message Sequence Chart (MSC) [1] is most used, and the one for which the language was originally developped, is in the description of telecommunication protocols. Real life telecommunication protocols often have different levels of interpretation. Something that is regarded a single message at one level, can be a packet of messages at the next, while yet one level lower a number of regulation messages such as "are you ready to receive?" and "transmission successfully completed" might be added. At the lowest level, there are just a large number of bits being transferred in both directions.

As one single level is already quite complex by itself, one does not want to be concerned by what is going at at the lower levels when specifying a higher one. However, in MSC this can currently only be done by dropping those lower levels altogether, which might also be undesirable. One might be interested in possible interactions between the various levels, or the computer system that is used to test the implementation might only be able to interpret the communication at a lower level.

Thus, one would like to adapt the formalism in such a way that it is possible to switch between different levels. That way one can design the system or protocol at one level while still being able to see the result at a lower level. In this paper we will introduce the concept of *message refinement*, in which one message can be used to denote a collection of events, as a construct that can be used to make such switches.

1.2 Composition and refinement – a historical outline

Much discussion has been going on about the possibility of combining several MSCs to create one larger one. In many applications, MSCs tend to become unduly large, spanning several pages. One would like to break those up into smaller parts in order to gain a better overview.

In the oldest MSC-standard, MSC'92 [2], only one operation to break up or combine MSCs was defined, namely the so-called *instance refinement* [3]. Here one instance is used to show the behaviour of several instances. An example of instance refinement is given in Figure 1.

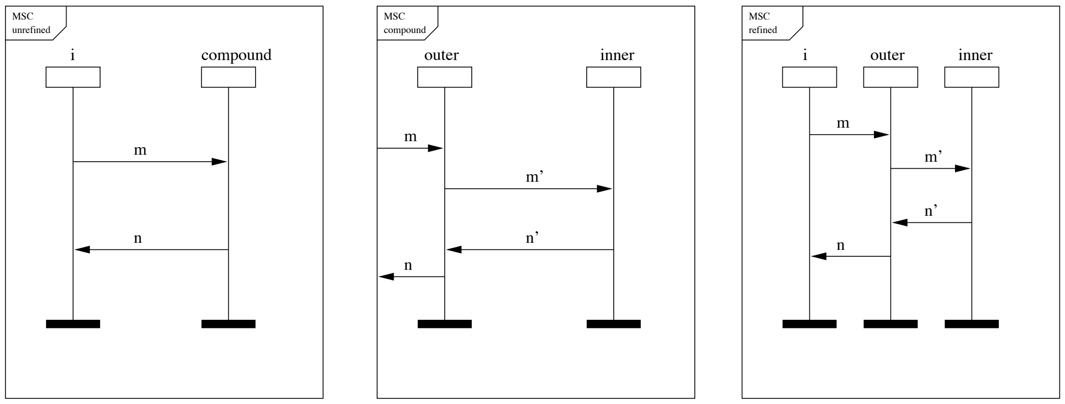

Figure 1: Instance Refinement

The instance 'compound' in the left MSC is *refined* by the middle MSC. That is, the middle MSC shows the internal behaviour of that instance, which appears to consist of two parts that communicate with each other as well as with their mutual environment. The external behaviour of the refining MSC should, of course, be equal to that of the instance to be refined - in this case, first receiving m, then sending n. Together the two MSCs shown here describe the same as the single MSC to the right.

In 1996, a new, extended version of the language was approved by the ITU [1, 4]. In this version, constructs were added for the explicit composition of MSCs. An example of such a construct is the use of reference MSCs.

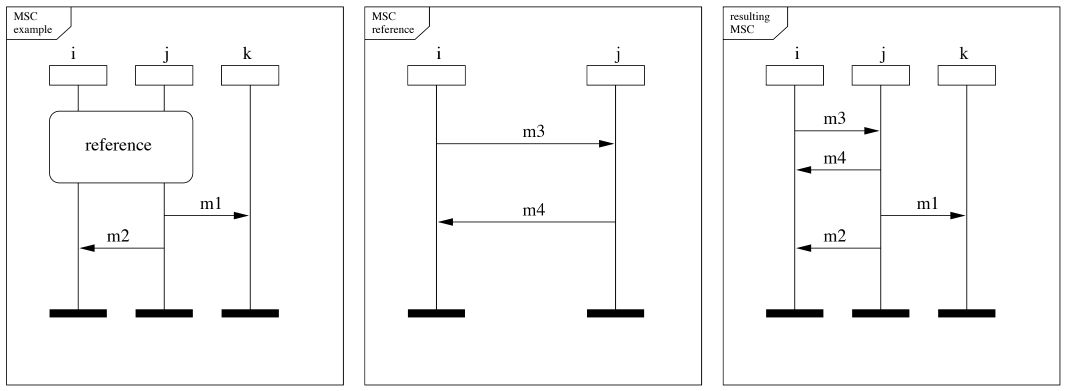

Figure 2: Reference MSC

In the left MSC, the block with the text 'reference' is a reference to the MSC called 'reference', which is shown at the middle. This MSC shows a part of the behaviour of the left MSC that is not shown in the original MSC, in this case the first part of the behaviour of the two leftmost instances. Together, these two MSCs show the same behaviour as the one on the right does.

Other added features include High-level MSCs [5], which can be used for parallel, sequential or alternative composition of simple MSCs, as well as for more complex constructions such as loops. We will not discuss those in this paper, and instead refer the reader to [5].

The idea of refinement (using one entity to stand for several of them) could be extended. Two logical ways of doing this are action refinement, in which a local action stands for a number of actions, and message refinement, in which one message stands for a larger protocol consisting of several messages and other events. With the appearance of MSC'96, action refinement adds little, as it can easily be modelled by replacing the action by a one-instance reference MSC. Message refinement will be addressed in this paper.

1.3 Acknowledgements

We would like to thank Michel Reniers for his ideas and support, Piet Bakker for his help on the editorial improvement of this document and Victor Bos for helping us out with latex. A special thank goes to Sjouke Mauw, who gave many pieces of useful advice on every part of the project.

2 Message Refinement

2.1 Protocol MSCs

The basic idea behind message refinement is to use a single message as the notation for some more complex behaviour. A separate MSC then defines this behaviour. Because this behaviour will generally be a protocol, showing how the information exchange, represented by the message, will occur,

The idea behind message refinement is to have one message stand for an MSC of its own. This MSC, as it shows the protocol used to send the original message, we will call a *Protocol MSC*. What are the properties of such an MSC?

First, there will be two instances, the *sender* and the *receiver*, that are to take the roles of the instances sending and receiving the message to be refined in the unrefined MSC (that is, the MSC in which only the high-level message is shown, the MSC in which the message is 'replaced' by the protocol MSC will be termed the *refined MSC*). However, there may be other instances as well. These describe (parts of) the medium between the communicating processes, or perhaps parts of the communicating processes themselves that specifically serve purposes in the input or output process only.

Furthermore, as there should be some sort of communication from the sender to the receiver, it is reasonable to assume there is some event at the sender that necessarily happens before some event at the receiver. When e_1 necessarily happens before e_2, we will write $e_1 << e_2$. That is, $e_1 << e_2$ iff e_1 is before e_2 in every possible trace (allowed sequence of events) of the MSC.

A third point is that we want our MSC to reach neither a deadlock (in which the system has not successfully terminated and yet is unable to perform any actions) nor a lifelock (in which the system keeps on running in loops without ever terminating). If any of these two would be the case, the protocol MSC could not really be regarded as just a refinement of the original message, as it would add some other behaviour as well. Deadlock is forbidden in the MSC standard [6], and an algorithm has been published to check for it [7].

Putting this all together we come to the definition set out below:

Definition 1 A protocol MSC is an MSC with the following added requirements:

1. There are two different special instances, which are termed the sender and the receiver. The other instances (if present) are termed *internal instances.*
2. There are events e_1 at the sender and e_2 at the receiver such that $e_1 << e_2$.
3. The MSC is free of deadlocks, and every finite beginning of a trace of the MSC can be extended to a finite trace.

2.2 Message Refinement

Having defined what a Protocol MSC is, we next define what Message Refinement means. Thus, given an MSC and a message in that MSC, what is the result when we replace the message by a given Protocol MSC? To define an MSC, we need to specify its instances and events, and the orderings between these events.

If an MSC k has a message m that is to be refined by a protocol MSC p, we expect not to fin $!m$ and $?m$ in the resulting MSC, as they have been replaced by p. All other events of k will be there, and are as much as possible undisturbed. Likewise, all events of p are present. They too are as much as possible undisturbed. All events of p that are on the sender taken together replace the event $!m$ of k. Thus, apart from their own orderings in p they also have to confirm to all orderings of $!m$ in k.

Definition 2 (Message Refinement) Let k be an MSC, let m be a message of k, that is, a message for which the sending $!m$ and the receipt $?m$ are events of k, and let p be a protocol MSC. Then the message refinement of m by p in k is the MSC with the following characteristics.

Its instances are all instances of k, and all internal instances of p.

Its events are all events of k with the exception of $!m$ and $?m$, and all events of p. Those events which in p are at the sender instead placed at the instance at which the event $!m$ takes place in k. Likeweise, the events at the sender are placed at the instance at which $?m$ takes place in k. The other events of p, and the remaining events on k are not changed.

There is an ordering of a given sort $e << e'$ between two events e and e' (for example, an instance order or a causal order) iff one of the following is the case:

* e and e' are both events of k and $e <<_k e'$.
* e and e' are both events of p and $e <<_p e'$.
* e is an event of k with $e <<_k !m$ and e' is an event at the sender of p.
* e' is an event of k with $?m <<_k e'$ and e is an event at the receiver of p.

We will denote the message refinement of m by p in k by $k[p/m]$
An example of message refinement we see in Figure 3.

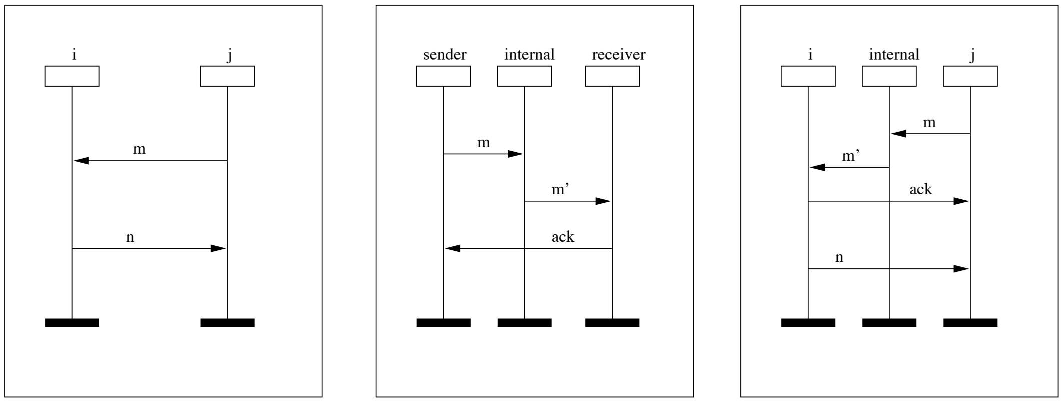

Figure 3: message refinement – an example

The left MSC is the original MSC, the centre one the protocol MSC, and the right one is the resulting MSC after message m has been refined by the protocol MSC. For example, because $!m$ is before $?n$ and at the same instance j in the original MSC, and $!m$ and $?ack$ are at the sender of the protocol MSC, they are also at instance j and before $?n$ in the resulting MSC.

3 Message Refinement and Synchronous Communication

3.1 When is Message Refinement allowed?

In Figure 4, a problem can be found: the left MSC and the protocol MSC in the centre are both perfectly valid MSCs. Yet, refining m by the given protocol MSC, will result in the MSC to the right, which contains a deadlock. After m has been sent, all three instances are waiting for a message that will never arrive.

Of course this is undesirable behaviour, so we would like to prevent it. However, to do so we need to know when such a situation might occur. We will see that for this purpose it is useful to distinguish between two types of protocol: *unidirectional* and *bidirectional* protocols. In a bidirectional protocol this is not the case:

Definition 3 A protocol MSC is bidirectional if in each trace of the MSC there is an event e at the receiver and an event e' at the sender such that e takes place before e', and is unidirectional otherwise.

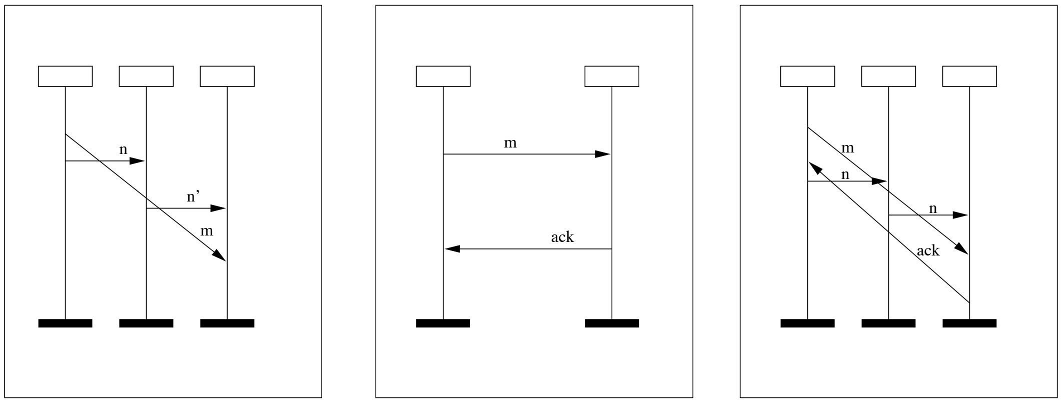

Figure 4: A problem with message refinement

We first look at unidirectional protocols. They are very close to the intuition of a single message. No deadlocks are created by the refinement of messages with unidirectional protocols, as the following theorem shows:

Theorem 4 Let k be an MSC, m a message of k, and p be a protocol MSC. Then, provided k and p are have no deadlocks themselves, $k[p/m]$ has no deadlocks either.

Proof Suppose $k[p/m]$ contains a deadlock. Then there should be events e and e' such that $e << e'$ and $e' << e$ simultaneously hold. If there were no such pair in which e is an event of k and e' one of p, then the pair would already have caused a deadlock in either k or p, so we may assume that e and e' are events of k and p, respectively.

$e << e'$ then implies that either $e <<_k !m$ ($<<_k$ of course being the $<<$-ordering of the original MSC k) and $e'' <<_p e'$ for some event e'' at the sender, or $e <<_k ?m$ and $e'' << e'$ for some event e'' of the receiver. Likewise, $e' << e$ implies that either $!m <<_k e$ and $e' <<_p e''$ for some event e'' of the sender, or $?m <<_k e$ and $e' <<_p e''$ for some event e'' at the receiver.

Because $!m <<_k ?m$, the only way in which $e <<_k !m$ or $e <<_k ?m$ can be combined with $!m <<_k e$ or $?m <<_k e$ without causing a deadlock in k is when $!m <<_k e <<_k ?m$. Then it must be the case that $e <<_p e'$ for some e'' at the receiver and $e' <<_p e'''$ for some e''' at the sender. However, in that case $e'' <<_p e'''$, which contradicts the unidirectionality of p. Thus we see there are no such e and e', so the refined MSC is free of deadlocks. ■

Bidirectional protocols are trickier. Here the anomaly shown in Figure 4 can occur. Luckily we can give the exact conditions under which it occurs. Intuitively one can say that the output and the input of the m must be able to happen arbitrarily close to eachother to avoid a deadlock.

Theorem 5 Let k be an MSC, p a protocol MSC (both containing no cycles), and m a message of k. Then $k[p/m]$ is free of cycles if and only if the following conditions hold:

1. $!m$ and $?m$ are not at the same instance in k
2. There is no event a such that $!m << a << ?m$

Proof if: If the conditions are met, there is a trace where $!m$ and $?m$ follow each other immediately. A valid trace of the refined MSC can now be found by taking such a trace, and replacing $!m \cdot ?m$ in this trace by any trace of p, renaming instances where needed.

only if: If $!m$ and $?m$ are at the same instance in k, then in the refined MSC each event coming from the sender will come before each event coming from the receiver. This will obviously lead to a deadlock if the protocol is bidirectional.

Now suppose there is an event $!m << a << ?m$. There are events e on the receiver and e' on the sender such that $e << e'$ in p. However, in $k[p/m]$ we now have $<< a << e << e' << a$, and thus a deadlock. ■

3.2 Synchronous Communication

In the present context it would be desirable to have an extra construct in the language to show *synchronous communication*, that is, a message being sent which does not take any time to go from the source to the destination directory. This looks like a useful extension in itself as well.

Such a synchronous communication can be implemented semantically in two ways: firstly as a single action that is shared by two instances, and secondly as two actions that have to be done without any other action between them. The first method of interpretation is probably preferable, because the second will is very hard to implement in process algebra – or in any of the other formalisms that have been used for proposed semantics for MSCs, for that matter. Anyway, any of the two representations can easily be translated into the other.

If the construct of synchronous communication would be present in the language, then avoiding deadlocks caused by message refinement can be done in the following way.

Requirement 6 A normal message may only be refined by a unidirectional protocol. A synchronous message may only be refined by a bidirectional protocol.

3.3 Semantics

Until now, MSC has been developed without much concern about the semantics. The semantics for MSC'92 were created only after the language itself was adopted. The same was done in the case of MSC'96. This way of working can easily lead to disasters, and actually has. Two of the features of MSC'96 – loops and gates – tend to give counter-intuitive results when combined [8]. To avoid this type of disaster in the future, it would be good if for every new feature that is proposed the semantics are proposed, or at least discussed, at the same time.

Although the original semantics for MSC was given in process algebra [9], the semantics for the new MSC'96 standard is only given operationally [10]. We will try to give an operational semantics for message refinement. Het $k[p/m]$ is the refined version of k, with p for m, while $k[p/m]^*$ is the same, but after $!m$, or in fact any of the events that replaces it, has already taken place. We will not explain these semantics any further, as we think there is a better option that will be given below.

These semantics assume that

1. $!m$ and $?m$ take place in k exactly once
2. The internal instances of p are different from any instances in k or the surrounding MSC
3. No event on the receiver of p can take place before any event on its sender has taken place. . It is possible to give rules that do not need these assumptions, but they would be more complicated. In these SOS-rules, $i(a)$ for an event a denotes the instance on which the event takes place.

$$\frac{k \xrightarrow{a} k', i(a) \notin \{!m, ?m\}}{k[p/m] \xrightarrow{a} k'[p/m]}$$

$$\frac{p \xrightarrow{a} p', i(a) \notin \{\text{sender}, \text{receiver}\}}{k[p/m] \xrightarrow{a} k[p'/m]}$$

$$\frac{k \xrightarrow{!m} k', p \xrightarrow{a} p', i(a) = \text{sender}}{k[p/m] \xrightarrow{a} k'[p'/m]^*}$$

$$\frac{k \xrightarrow{!m} k', k' \xrightarrow{?m} k'', p \xrightarrow{a} p', i(a) = i(?m)}{k[p/m] \xrightarrow{a} p' \circ k''}$$

$$\frac{k \xrightarrow{a} k', a \neq ?m, p \cdots\xrightarrow{a} p''}{k[p/m]^* \xrightarrow{a} k'[p''/m]^*}$$

$$\frac{p \xrightarrow{a} p', i(a) \neq i(?m)}{k[p/m]^* \xrightarrow{a} k[p'/m]^*}$$

$$\frac{k \xrightarrow{?m} k', p \xrightarrow{a} p', i(a) = i(?m)}{k[p/m]^* \xrightarrow{a} p' \circ k'}$$

$$\frac{k \xrightarrow{?m} k', k'{\downarrow}, p{\downarrow}}{k{\downarrow}\ [p/m]^*}$$

$$\frac{k{\downarrow}, p{\downarrow}}{k[p/m]{\downarrow}}$$

$$\frac{k{\downarrow}, p{\downarrow}}{k[p/m]^*{\downarrow}}$$

$$\frac{k \cdots\xrightarrow{a} k'', p \cdots\xrightarrow{a} p''}{k[p/m] \cdots\xrightarrow{a} k''[p''/m]}$$

$$\frac{k \cdots\xrightarrow{a} k'', p \cdots\xrightarrow{a} p''}{k[p/m]^* \cdots\xrightarrow{a} k''[p''/m]^*}$$

However, we prefer another way to include message refinement semantically. If we let it be not an operation *in* but an operation *on* the language, the problems become much less. With this I mean that message refinement is regarded as another way of writing down the MSC where the message has already been defined. That is, to get the semantics of an MSC with refinement, one performs an operation like the one in Definition 2 (but more precisely defined) to get the refined MSC. The semantics of the MSC with refinement is then defined to be equal to that of this refined MSC.

Synchronous communication can be semantically included rather easily. A synchronous communication can simply be implemented as a single event that has a place in the instance ordering of two different instances. Such a construct does not seem to cause any major problems.

4 Conclusions

An important issue in MSC is the addition of various ways of composition, that is, combining a number of smaller MSCs into one large MSC. A new way has been put forward in this paper, being message refinement in which a message can be replaced by a protocol consisting of a number of messages and possibly other events.

These protocols can be divided into two groups, namely unidirectional protocols and bidirectional protocols. Replacing a message by a unidirectional protocol causes no problems, but replacing it by a bidirectional protocol might cause deadlocks. One solution to this problem is the addition of synchronous communication, which might also be a useful addition to the language of itself. If we allow only synchronous messages to be replaced by bidirectional protocols, no deadlocks will occur.

To avoid problems in the semantics of MSC, it would be better to define protocol refinement, and other composition techniques also, not as an operator *in* the language, but as an operator *on* the language, describing a way in which MSCs can be changed into other MSCs. This way, no complicated semantic constructs are necessary to implement them.

5 *

5 REFERENCES

[1] ITU-TS. Message Sequence Chart (MSC). Recommendation Z.120, ITU-TS, Geneva, May 1996.

[2] ITU-TS. Message Sequence Chart (MSC). Recommendation Z.120, ITU-TS, Geneva, 1993.

[3] S. Mauw and M.A. Reniers. Refinement in interworkings. In U. Montanari and V. Sassone, editors, *CONCUR'96,roceedings of the Seventh Conference on Concurrency Theory*, volume 1119 of *Lecture Notes on Computer Science*, pages 671–686. Springer Verlag, 1996.

[4] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts (MSC'96). In *Tutorials of the First joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'96)*, October 1996.

[5] S. Mauw and M.A. Reniers. High-level Message Sequence Charts. In *SDL '97: Time for Testing - SDL, MSC and Trends. Proceedings of the Eighth SDL Forum*, pages 291–306. Elsevier Science Publishers, 1997.

[6] M.A. Reniers. Static semantics of message sequence charts. Technical Report CSR 96-19, Eindhoven University of Technology, September 1996.

[7] Hanêne Ben-Abdallah and Stefan Leue. Syntactic detection of process divergence and non-local choice in Message Sequence Charts. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in Lecture Notes on Computer Science, pages 259–274. Springer Verlag, 1997.

[8] M.A. Reniers. (title to be announced). Master's thesis, Eindhoven University of Technology, 1998? to appear.

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

[10] S. Mauw J.M.T. Cobben, A. Engels and M.A. Reniers. Formal semantics of message sequence charts. Technical Report CSR 97-17, Eindhoven University of Technology, 1998. to appear.