Design Decisions on Data and Guards in MSC2000

A. Engels, engels@win.tue.nl
Eindhoven University of Technology

Abstract

Among the most important extensions in MSC in the new MSC2000 Standard are data and guards. In this paper some of the design choices that were faced when extending the language with these concepts are presented, and reasons are given why these have been made in the way they were made, and why certain static requirements are present in the language.

Keywords
MSC, data, conditions, guards, MSC2000, standardization, semantics, language design

1 Introduction

Among the new concepts introduced in MSC in the new MSC2000 Standard [9] is the incorporation of data languages in MSC. This incorporation has not been easy. Several choices had to be made, and problems had to be solved. In this article we will describe the problems that were encountered and the solutions that were found.

These solutions have resulted in decisions about how the standard looks and what restrictions it contains. The standard does not give the motivations for these decisions and restrictions. Still, it would be good to have them documented. Otherwise, what are valid and thought-out design decisions might look like merely ad hoc choices. Furthermore, only if the actual decision points and decision criteria are known a further discussion on these decisions is possible.

1.1 Principles

During the discussions on the new standards, the work has been guided by the following principles. It is where these principles clash that problems arise.

1. Correspondence to Existing Practice.

   Several of the subjects that are being introduced, and this certainly holds for data, are already being used in existing practice. The standardization committee wanted to link with this existing practice - the new MSC standard should include as much as possible of this existing practice. Where possible, what is already done unofficially should be made official.

2. Semantic Simplicity.

   When something is introduced, its semantics should be clear. That does not mean that the formal semantics should immediately be added (at the moment there are no plans known to us for a complete formal semantics of MSC2000), but it does mean that there should be a good and complete *intuition* of what those semantics should look like.

   The semantics should not only be clear, they should also be simple. Creating the semantics should not be unduly hard, and it should be relatively straightforward, once the semantics have been defined, to decide what the semantics of a given MSC are. This is of importance both to people working with the language on a theoretical level and to tool builders, who want to include functionality such as the simulation of the behaviour of an MSC, which can only be done if the semantics are not too complex.

3. Intuitive Semantics.

   In line with the first principle, the semantics should be intuitive for the user. Many MSCs, even though they include not yet officially implemented features, have a meaning that is intuitively clear. The semantics should conform to this intuitive meaning.

4. Backward Compatibility.

   Old MSCs, made according to the MSC'96 or MSC'92 standard, should still be usable under the new MSC2000 standard. Within the standardization committee a 'loose' definition of backward compatibility has been followed: it might be allowed to have the necessity to change a Message Sequence Chart made according to the MSC'96 standard, as long as there were only small, syntactic changes to be made. For example, having to write '**novars**' in each action box would have been an acceptable change, having to change each action box into two would not.

5. Compositionality.

   We want MSC to be a compositional language, that is, the meaning of a whole can be extracted from nothing more than the meaning of the parts and the way those parts are connected. Compositionality was regarded an important principle by the standardization committee; however, it did not play an important role in the choices mentioned here because none of the alternatives that were proposed seriously threatened compositionality.

1.2 History

MSC-like diagrams have been used for a long time [10]. Their use as auxiliary diagrams for SDL specification led the ITU to decide to standardize the language. The first standard of the MSC language was published by the ITU in 1992 [6]. In 1996 a new standard [7] was presented, which had additions in particular regarding composition. 1996 to 2000 would be a new study period, and new concepts would be added to the language, among which the incorporation of data was high on the agenda. The first attempt to formally combine MSC with a data language was made by Grabowski, Hogrefe, Nussbauer and Spichiger, who combined MSC and ASN.1 [5]. Baker and Jervis next presented a more general usable data language [1].

Starting from the work of Baker and Jervis, Feijs and Mauw decided to go even more general. Instead of defining one standard language to go along with MSC, they introduced a framework in which a large range of languages could be described, thus for each of these giving the interface of the language with MSC [4]. This method of incorporating data has several advantages. In the first place, its generality enables large groups of users to use the constructs by adding their own preferred data language. In the second place, because the MSC language and the data language both get their own, well-described, domains, the data language will not influence the rest of MSC, so different data languages will not cause differences elsewhere in the language. In the third place, because all MSC is concerned about in the data language is what is in the interface, the risk of semantical problems in the data language 'spilling over' into MSC is minimized. In the fourth place, because MSC does not specify which language is to be used, the MSC community also does not burden itself with the task of maintenance of whichever data language would otherwise be chosen.

Although the MSC standardization community agreed with this principle, there was a strong wish to extend the work of Feijs and Mauw, more specifically to introduce the possibility of dynamic data. In the original framework, variables could only be used as placeholders for an unknown value. The semantics would then be that any of the possible values for the variable could occur. There was a wish to have the possibility to assign values to a variable which then could be used later in the MSC. Feijs and Mauw, together with the current author, investigated the possibilities and problems of incorporating this possibility in a second paper [3]. Partially based on this paper, in an ITU experts meeting at the Eindhoven University of Technology, the subject of data was then incorporated in the MSC2000 standard.

Another extension of the MSC language that is closely related to data, is the subject of guards. In the MSC'92 and MSC'96 standards, conditions were included, but did not have any formal semantical meaning [8, 12], except for some static restrictions on HMSCs [11]. In practice they were often used to combine MSCs - a final condition was given to some MSCs, an initial condition to others, and two MSCs could be combined (only) if the final condition of the first MSC corresponded to the initial condition of the second [13]. It was felt that it would be a good idea to standardize and extend this feature. Some conditions would then act as guards, meaning that one can only pass the condition if their text is in some sense 'true'. Others would define a kind of state for the MSC, which could then be used by the guarding conditions. Obviously, the text in a guarding condition would often be some boolean expression in the data language, and thus this subject was seen as closely related to that of data.

In this paper we will first give an overview of the problems and choices indicated in [3], together with the choices that have been made on these points in the final MSC2000 standard, sometimes with an indication of the reasons for these choices. We will also show some of the problems that occurred especially on the subject of conditions and guards, and how these led to the actual static requirements on conditions that can be found in the standard. We will not be going into the syntactic choices, but only in the larger conceptual choices and the semantics connected to data and guards.

2 Data

In [3] several choices regarding the introduction of variables are mentioned. We will repeat them here, with some discussion about which choice was actually made, and why.

2.1 Static vs. dynamic nature of a variable

The first choice was whether the data would be static, dynamic, or somewhere in between. In [3] four possibilities were distinguished, which, from most static to most dynamic, were called:

1. fully static variables
2. parameter variables
3. single time assignable variables
4. multiple times assignable variables

Fully static variables: In a fully static environment, the values of variables are either completely pre-determined, or not defined at all. In the latter case the semantics is taken to consist of *all* possible behaviours for *any* valuation of the various variables. This is the situation that was already covered by the earlier paper [4]. This way of including data causes the least problems. However, the price that has to be paid is that it also provides the lowest amount of possibilities for the user.

Parameter variables: Parameter variables play a role within HMSC or MSC reference expressions, the idea being that one provides a value for one or more variables while calling the reference MSC. In calling the MSC, the value of some variables are given as parameters to the reference. An example of how this works is shown in Figure 1.

The MSC *sending* calls the MSC *sendmessage* twice, but the first time with *sendmessage(1)* and the second time with *sendmessage(42)*. The effect of this, is that the MSC *sendmessage* is used with the value of 1 for p the first time, and the value of 42 for p the second time. Thus, first the message $s(1)$ is sent and received, then $s(42)$. Compared to the next two options with

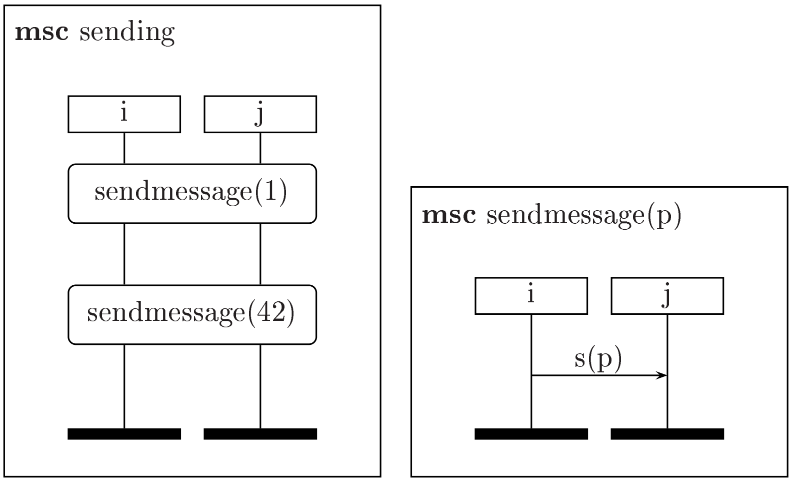

Figure 1: Parametric Data

assignable variables, Parametric data is semantically less complicated. On the other hand, it is also less powerful. Of course introducing both options results in an even greater power of expression.

Single time assignable variables: Here a variable can be assigned at any place in the MSC, but once it is assigned, it cannot get a new value, at least not within its current scope. So each time a variable is accessed, it will still have the same value. This will solve some of the problems with variables that are shown below, because these typically occur when the value of a variable is changed before it is used, or between several uses of the same variable. However, to actually make this solution work, one would also have to take steps to make the usage of a variable before its assignment impossible, otherwise one still has to deal with two different values - an indeterminate one before it is assigned, and a determinate one afterwards.

Multiple times assignable variables: Here, a variable can at any time be given a value, and this value can be changed later. This choice offers most possibilities for the user, but at the same time also does most to complicate the language. Thus, introducing this option would also give the most problems to think about. A first example is in Figure 2

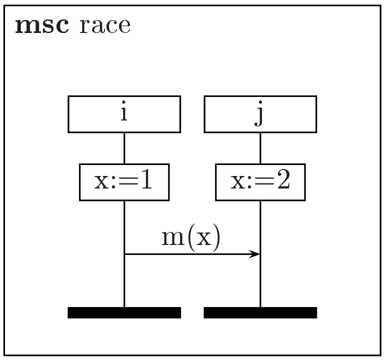

Figure 2: Problematic MSC (simple case)

What message is to be sent her? $m(1)$ or $m(2)$? Or does it depend on the order in which the assignments happen? And what if the change of the value of x from 1 to 2 happens between sending and receiving of the message? Will we have to remember the old value of x?

Despite these problems, which were mentioned in [3], it was felt that multiple-times assignable variables were needed, because users would like to have them. To keep this type of 'race conditions' manageable, it was decided to have one specific instance be the 'owner' of a variable. The variable could only be changed on this instant.

The first idea was that the usage of such a variable would also be restricted to that same instance. No other instance could use the value of the variable. However, this proved too restrictive. From a usage point of view, it seemed necessary that an MSC like the one in Figure 3 could be drawn.

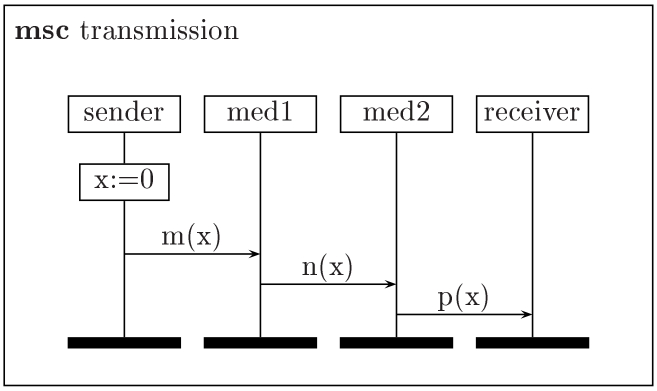

Figure 3: Transmission of variables

The idea here, is that at one point the value of x is decided, which then through a series of messages is sent to another part of the system described by the MSC. To enable this, it was decided that the value of variables of one instance could be used at another instance, provided that the value had been sent to that instant through messages. Thus, the value of a variable on an instance other than the owner is changed only when this instance receives the value through some message. Not only does this make the semantics more intuitive than one in which a change in value of a variable is 'magically' copied to all instances of the MSC, it also is more in line with the basic semantical ideas behind MSC: any transfer of information from one instance to another should be explicitly shown.

Because their added value in expressionality of the language seemed useful, while they did not add more problems than the ones that already had to be solved for the assignable variables, parametric data have also been included in the MSC language.

2.2 Binding of Variables

Another matter was, how these variables could change their values. For parametric variables this is obvious: They receive their value because the MSC is called with a certain actual parameter list. For dynamic variables, we have to distinguish between owning and not-owning instances.

An instance that is not the owning instance of the variable receives a new value for the variable when a message arrives that contains that variable in its parameter list. The new value of the variable, as seen by that instance, will be the value the variable had on the sending instance at the time of sending.

For the owning instance, two major ways of giving values to variables were discussed in [3]. Both have been included in the MSC language. The most obvious is through a direct binding. This occurs when a text like $x := 2$ is found inside an action box. A second way is through the propagation of a message through a gate. This is best explained with an example (see Figure 4:

The incoming message here gives x the value of 3. Of course this can only be done if x is a variable of the receiver. In all of these cases, the outgoing message (the message in the MSC

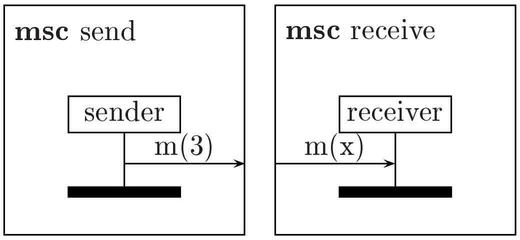

Figure 4: Assignment through gated messages

'`send`') should contain some expression as one of its parameters, while the incoming message (the same message in the MSC '`receive`') should contain a variable to be bound. This can also be done with MSC reference expressions, see Figure 5.

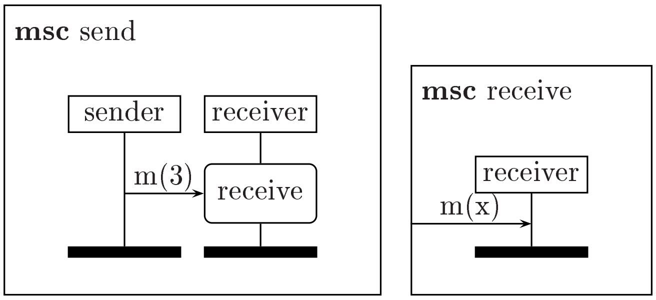

Figure 5: Variable assignment for MSC reference expressions

An instance that does not own the variable, changes its (local) value when it receives a message that has that variable in one of its parameter expressions. It is then changed to the value that this variable has on the sending instance. Although this has been designed to correspond to existing practice, it can still lead to some counter-intuitive results, as in Figure 6.

Assume that x is a variable owned by instance i.

Intuitively, one might expect that k always sends back ack(2), because 2 is the latest value of x, and k has already received that value. However, k does not 'know' which is the latest value, so it will always assume that x has the last value that it has received. Thus, if $n(1)$ arrives at k after $m'(2)$, ack(1) rather than ack(2) will be sent.

2.3 Undefined variables

Another problem raised in [3] was what one had to do with variables being used before they have any value assigned to them. The main choices possible were:

1. Disallow this by static rules
2. Regard it as a semantical deadlock
3. Regard all undefined variables as universally quantified
4. Use default values for each variable

All of these options have their downsides (the first may be hard, the second leads to unintuitive semantics, the third leads to an explosion of the number of possible traces and the fourth extends the interface with the data language by a default value for each domain, see [3] for a more extensive

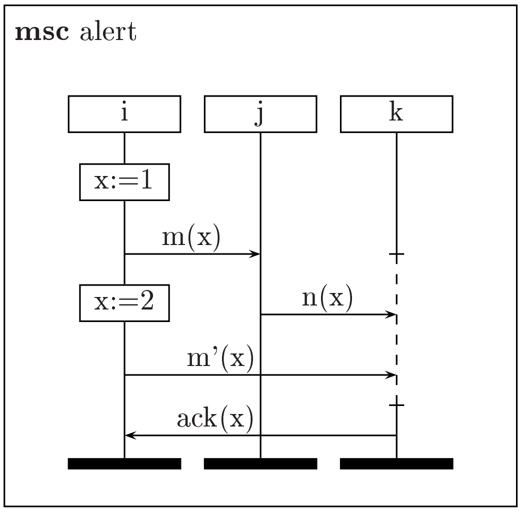

Figure 6: A counter-intuitive case

discussion). In MSC, the first option has been chosen (see [9], section 5.4: *In a defining MSC there must be no trace through an MSC in which a variable is referenced without being defined.*)

3 Guards

The largest problems with data were met when trying to introduce guards. In the older versions of the MSC standard, MSC'92 [6] and MSC'96 [7], conditions had very little function. Semantically, they had no meaning at all, except for statically forbidding some HMSCs, and thus they formally were no more than comments. However, it was found that in practice conditions were being used in a more functional manner, namely to create MSCs like those in Figure 7.

The conditions here function as a method to decide in which order the MSCs are gone through. More precisely, one may go from one MSC to another if, and only if, the second MSC starts with the same condition as the first ends with. Thus, after finishing MSC *try, one can go to* retry or *success, and likewise after MSC* retry, while after MSC *success one can only go to MSC* try.

3.1 Including Guards in the Language

An attempt was made to formalize this method of using conditions. Certain conditions would act as guards, others as defining conditions. One can only pass through a guard if it is equal to the last defining condition that was encountered. The above MSCs would then become a correct MSC when included in an HMSC like the one in Figure 8.

The conditions at the bottom of the MSCs should be defining conditions, and the conditions at the top should be guards. To avoid any confusion about which conditions are defining conditions and which are guards, all guards have the keyword '**when**' added to them. Thus, in the above case, apart from adding the HMSC one should also change '*ready*' and '*sending*' into '**when** *ready*' and '**when** *sending*' in the conditions at the tops of the MSCs, while leaving the conditions at the bottoms as they are.

When combining these guarding conditions with data, the possibility arises to use data expressions as guards. More precisely, if a data expression can have boolean values (true or false), it could be used as a guard, which then can only be passed if it is true. In this way, behaviour that depends on the value of variables can be described. An example is in Figure 9, where it is checked whether a certain variable is zero:

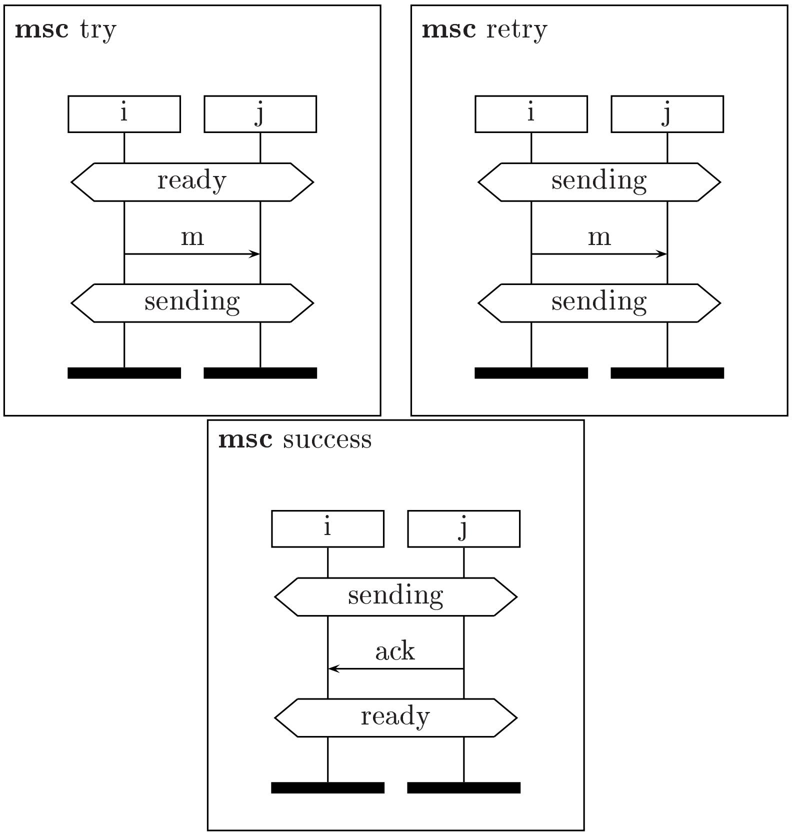

Figure 7: conditions as a coupling mechanism

3.2 Semantical Proposals

Unfortunately, this option also resulted in some semantical problems. Central question was, at which time the guard would be evaluated.

Figure 10 shows these problem. Suppose that this MSC starts its trace with $x := 1$, and after that the message m is sent and received. Which of the messages a and b (if any) can then be sent? There existed two schools of thought, neither of which in the end prevailed.

The first argued that at that time the value of x would be 1. Thus, message a could be sent, but message b could not. The second preferred to look at the MSC as a whole, rather than just one point in time. Because the action $x := 2$ is above, and thus 'before' the alt-expression, it is logical that it will affect the choice. Thus, the second option (sending b) can be chosen, but not the first.

The disagreement can be described as a disagreement on the time and place where the guard is evaluated. In the first proposal, the guard is evaluated by the first instance to do an action after the guard, the principle could be called 'first past the post'. In the second proposal, the guard is evaluated by the instance that owns the variables in the guard.

The arguments for the two proposals came from two of the principles mentioned before.

The first proposal was based on semantical simplicity. Adopting the second proposal would require that one first calculates the full semantics of an MSC without looking at the guard, then removes the traces that do not adhere correctly to the guards present. This would make it impossible to find out the next event in an MSC trace without first calculating the complete semantics of the MSC. This makes any semantic calculation extremely hard. It is also different from the ex-

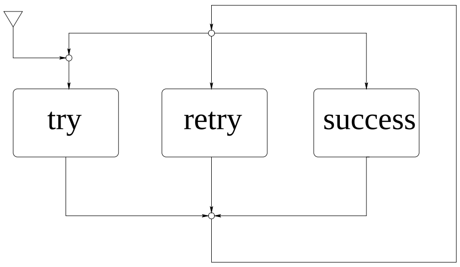

Figure 8: An HMSC for Figure 7

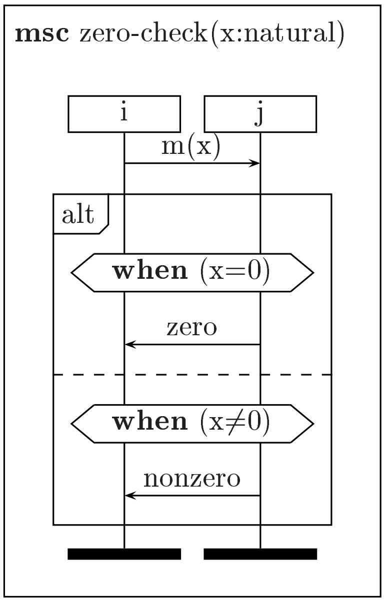

Figure 9: Usage of data in guards

isting semantics of MSC, in which a deadlock does not perpetrate back in time. In one term, this objection could be called 'backward causality': What happens now is dependent on what happens or can happen at a later time.

The second proposal was based on intuitivety. This view, which is more whole-system-based, is closer to the intuition of the users. Thus, having a different semantics will lead to MSCs that mean something different from what they are thought to mean.

3.3 Semantical Problems

Apart from the lack of intuitivety (for the first proposal), and the complicated semantics (for the second proposal), there are also some more profound problems, certainly with the second proposal. For some MSCs this proposal provides no semantics at all, because guards have to be evaluated by instances that never pass them. An example of these problems is in the left MSC in Figure 11

If we first look at the semantics of this MSC without guards, we see that one possible trace is to start with $x := 1$, then send and receive m, then go through the a-loop indefinitely. Is this trace still a valid trace in the situation where guards are present? Under the second proposal we do not know: The guard should be evaluated when the second instance looks at it - but in this trace the

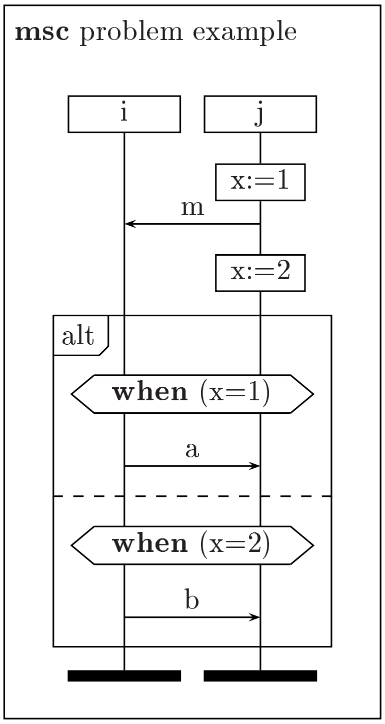

Figure 10: Problems with data in guards

second instance never does.

A similar problem occurs in the right MSC in Figure 11

This MSC will deadlock at some time, because of the inline expression with its only choice guarded by false. However, can the action a happen before the deadlock? Again, the guard $x = 2$ has to be evaluated when instance j passes it - which is never.

Both proposed semantics also lead to some very strange behaviours for some MSCs. The first semantics gives some unexpected ordering requirements in Figure 12

Under the first proposed semantics, instance i must defer sending message a until j has done the action $x := 2$. This thus leads to a synchronization, or at least an extra ordering requirement, which is not clear in the syntax of the language, and defies intuition.

For the second proposal, causality can go strange ways, for example in Figure 13

In this MSC instance i can send message a if and only if j changes the value of x to 1. If i sends out a message before j sets the value of x, i decides what choice j is going to make. Again, this is a connection (causal rather than ordering in this case) that is not obvious from the syntax of the MSC.

3.4 Static Requirements as a Solution

Neither option seemed very attractive. Instead, one would like a semantics that is intuitive from both the usage and the semantic point of view, rather than just one. Instead of choosing one of the alternatives, it was preferred to restrict the MSC language. Only those MSCs where both provided the same result would be allowed.

To see which MSCs these are, we have to go back to where we defined the two proposed semantics. There it was said that their difference was in the time and place where the guard is evaluated. The first semantics evaluates it when the first instance goes through it (by performing some action

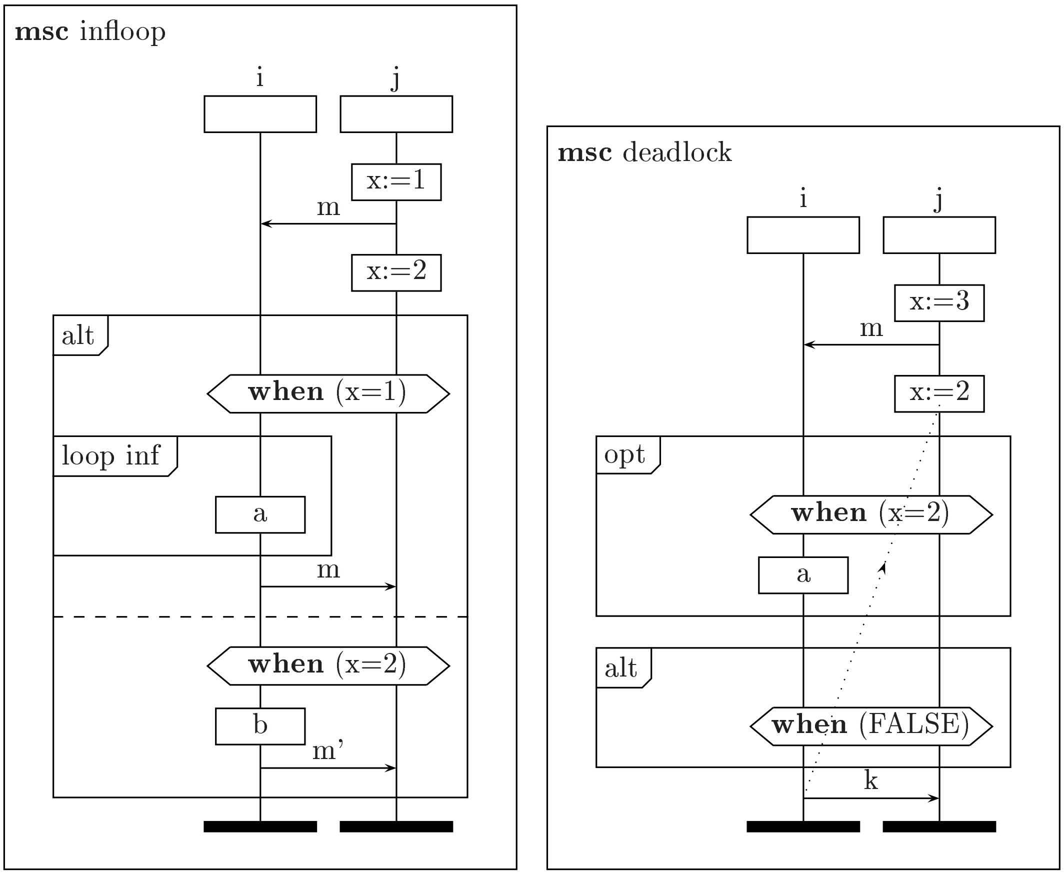

Figure 11: Two MSCs with semantic unclarity

coming after the guard), the second semantics when the owning instance of the variables goes through it.

The two semantics will provide the same result if the value of the guard does not change, or if they both evaluate it at the same time and place. The evaluation of a guard does not change if it does not contain any dynamic variables. Thus, any such guard is unproblematic. If there are dynamic variables involved, the evaluation may be different at different points in time. Thus, in this case both semantics should evaluate the guard at the same time. This is only the case if the owning instance of the variables is the first to do an action after the guard. Such an instance is called *ready*, and thus the following static requirement was added to the language:

> If a guard contains a data expression, then this expression must be of type Boolean. If this expression furthermore contains dynamic variables, it may only cover a single instance, which thus must be the only ready instance of the scope.[1]

3.5 Non-Data Guards

Similar problems, where the truth value of a guard becomes indeterminate, can also occur for guards without data, which get their truth value from defining guards. The problem occurs in

[1]The text in the published version [9] is slightly different. The quoted text is taken from a soon-to-be-official correction to the published version. In this correction it is also made clear that the restriction that all ready instances must be covered by the guard holds for all guards, not just for those with data.

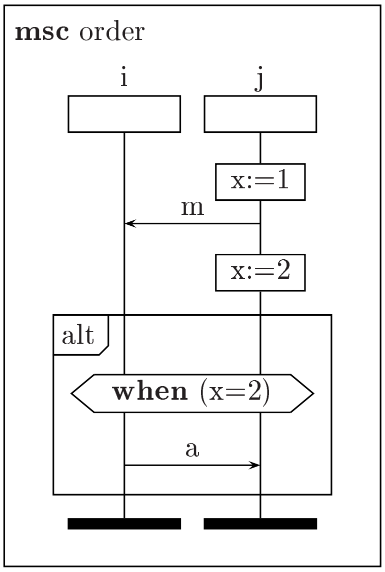

Figure 12: Unexpected ordering requirements

MSCs like the one in Figure 14.

If one simply would have one state for each instance, instance i would enter the alt-expression in state A, while instance j would do so in state B, so they disagree on which of the alternatives has to be chosen. This has been solved by allowing *only* conditions on exactly the same instances to be looked at by the guard. The setting of B is thus not of importance, and both instances here agree that the first alternative has to be chosen.

4 Conclusions

The incorporation of data in the MSC language is an important extension, but in doing so a number of choices had to be made, and a number of problems had to be overcome. In this paper some of the main choices have been highlighted, and an overview of the actual decisions and the reasons for those has been given.

One of the largest problems came with the inclusion of conditions as guards. Two semantics were proposed. One of them was very useful from a semantic point of view, but in some cases would lead to possibly counterintuitive results. The other agreed with the ideas of users, but proved semantically cumbersome, and sometimes did not provide any outcome at all. Because neither proposed semantics was appealing, the decision was taken instead to only allow the non-problematic cases. This led to a static requirement, stating that whenever dynamic variables occur in a gate, the instance on which these variables are designed must be the only 'ready' instance.

It is useful to keep track of these design decisions, and the reasons for which they are made. Not only does it make clear to users and toolmakers why the language is as it is, it also shows the type of problems that might again occur in later attempts to extend the language - whether this be by the introduction of new, perhaps somewhat similar concepts, or by the easening of the existing syntactic constraints.

4.1 Acknowledgments

I would like to thank Sjouke Mauw who gave valuable advise in several stages of the creation of this paper. I would also like to thank Øysten Haugen, Clive Jervis, Jan Docekal and Michel Reniers

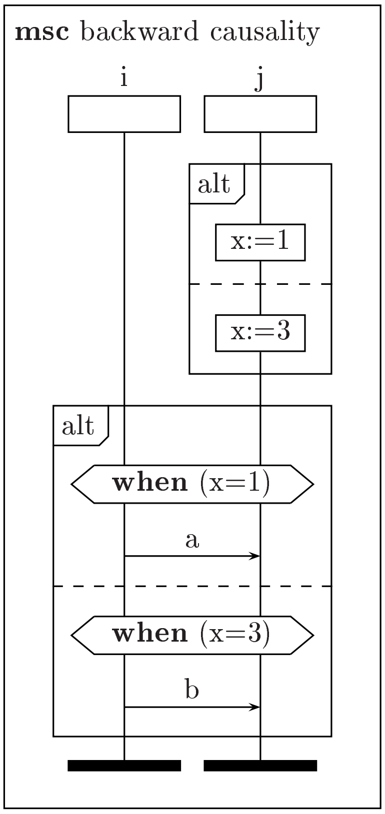

Figure 13: MSC with backward causality

for their participation in the original discussions on which this paper is based, and Loe Feijs for his involvement in getting the idea for this paper. The MSCs in this paper were created using the MSC LaTeX package[2].

References

[1] P. Baker and C. Jervis. Formal description of data. Temporary Document TD-L16, Experts Meeting ITU-T SG 10, Lutterworth, October 1997.

[2] V. Bos and S. Mauw. A LaTeX macro package for Message Sequence Charts. http://-www.win.tue.nl/∼sjouke/mscpackage.html, April 1999.

[3] A. Engels, L.M.G. Feijs, and S. Mauw. MSC and data: Dynamic variables. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99, Proceedings of the Ninth SDL Forum*, pages 105–120, Montreal, June 1999. Elsevier Science Publishers.

[4] L.M.G. Feijs and S. Mauw. MSC and data. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC*, number 104 in Informatik-Berichte, pages 85–94, Berlin, Germany, June 1998. Humboldt-Universität zu Berlin.

[5] J. Grabowski, D. Hogrefe, I. Nussbaumer, and A. Spichiger. Test case specification based on MSCs and ASN.1. In R. Bræk and A. Sarma, editors, *SDL'95 with MSC in CASE, Proceedings of the 7th SDL Forum*, pages 307–322, Oslo, Norway, September 1995. Elsevier Science Publishers.

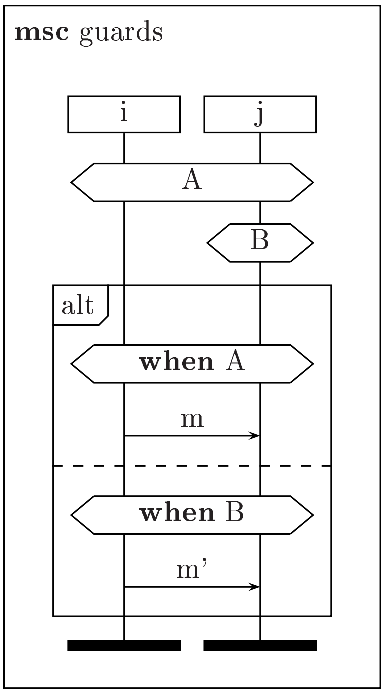

Figure 14: Why not all defining conditions affect a guard

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

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

[8] ITU-TS. Algebraic semantic of MSC'96. Recommendation Z.120 Annex B, ITU-TS, Geneva, 1998.

[9] ITU-TS. Message Sequence Chart (MSC). Recommendation Z.120, ITU-TS, Geneva, 2000.

[10] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[11] M.A. Reniers. Static semantics of Message Sequence Charts. Technical Report CSR 96-19, Eindhoven University of Technology, Department of Computer Science, 1996.

[12] M.A. Reniers. *Message Sequence Chart: Syntax and Semantics.* PhD thesis, Eindhoven University of Technology, June 1999.

[13] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996. Special issue on SDL and MSC, guest editor Ø. Haugen.