MSC and data: dynamic variables

A.G. Engels, L.M.G. Feijs, S. Mauw

Department of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands.
engels@win.tue.nl, feijs@win.tue.nl, sjouke@win.tue.nl

The extension of the MSC language with more advanced data concepts is one of the current topics of discussion in the MSC standardization community. A recent paper at the SAM98 workshop by two of the current authors [2] treated the extension of MSC with static variables. Feasibility of an approach to parameterize the MSC language with a data language was shown.

We have extended this research by studying the combination of MSC with a data language containing dynamic variables. Rather than giving a precise proposal of the way in which an actual data language must be added to the MSC language, we discuss options and problems. Choices have to be made, for example with respect to scope, use of variables, and the way of assigning variables. For some particular combination of the options mentioned above, we give a formal operational semantics of the combined MSC/Data language. It is argued that the interface between the data language definition and the MSC language definition should be explicit.

List of Keywords: Extension of existing language, Formal semantic models, MSC, Data, Variables.

1. INTRODUCTION

Quite high on the list of possible extensions for the language MSC [4] (Message Sequence Charts) is data. Currently, the language has hardly any data concept. At best, data can be expressed as a parameter of a message which is simply considered as a syntactical extension of the message name. Operations can be defined informally by means of actions. Again, this is considered as a purely syntactical concept.

There is clearly a need for a more extensive treatment of data. This is in line with the trend that MSC is becoming a language that is more and more useful for the complete description of system behaviour, rather than for displaying single traces. But also when using MSC for the visualization of traces, actual data values may be observed.

Since MSC is closely related to SDL [3], some things can be learned from the way in which SDL deals with data. The first formal data language integrated with SDL was based on algebraic specifications. These are known for having a very simple syntax and a clear semantical foundation. In practice, however, the functional style of an algebraic specification showed to be too difficult for people used to an imperative language. Therefore, an alternative data language, ASN.1 [5], was adopted. This enforced the development of

a second recommendation, which exists next to the first one. Currently, the development of SDL2000 involves a redesign of the SDL data language.

This situation has several drawbacks. Both recommendations have a large overlap, and thus there is a maintenance problem. Furthermore it requires a new semantics definition. In which sense is the semantics dependent on the actual data language? And finally it is not clear what will happen if a new paradigm (such as Java) gets into the picture. Will a third recommendation be developed?

We clearly do not want these problems to occur when extending MSC with data. Therefore, we have initiated research on the extension of MSC with data. In a previous paper [2] we have discussed several issues related to the extension of MSC with a data language containing static variables. We have argued in favour of developing an explicit interface between the MSC behavioural part and the MSC data part. This would overcome some of the drawbacks mentioned above. Furthermore, we have designed such an interface, appropriate for two classes of languages, namely, algebraic specifications and constraint syntax languages such as ASN.1 [5]. In Section 2 we will summarize the results of this research.

In this paper, we study the extension of MSC with a language containing dynamic variables. These are variables whose value may change during "execution" of an MSC. This situation is clearly more complicated than the case of static variables. We mention some questions that arise: Which MSC constructs can be used for changing the value of a variable? What should be the scope of a variable? How often may a variable have a value be assigned to it? How to determine in which state an expression should be evaluated? How to handle references to undefined variables?

We will discuss all these questions, formulate possible answers and discuss their respective merits. As in [2] we aim at defining an interface between the MSC language and some data language with dynamic variables. However, we expect that a uniform interface cannot be defined. That is an interface which is suited regardless of the answers to the above mentioned questions. Nevertheless, we have studied such an interface for one particular combination of answers and we have roughly defined the operational semantics of this particular MSC/Data language.

This paper is structured as follows. In Section 2 we summarize the findings of [2]. Section 3 contains the description of a simple example showing the combination of Basic Message Sequence Charts with a basic data language. In Section 4 we discuss some questions concerning the extension of MSC with dynamic variables. The operational semantics of a combined MSC/Data language is sketched in Section 5. We will end with some concluding remarks.

Acknowledgments We thank Jan Friso Groote, Frans Meijs, Jaco van de Pol, Michel Reniers and all members of the MSC standardization group for their fruitful discussions on this topic.

2. STATIC DATA

In this section we summarize the findings of [2], which addresses the question how the MSC language could be extended with a data type formalism. It is argued that it may be better not to choose a particular data type formalism for standardization as a

part of MSC, but instead set up the recommendation in such a way that the actual data language can be considered as a parameter. This leads to the research question of how to parameterize the MSC language with a data language. Starting from the idea that MSC is parameterized over some grammar with some semantical information, the latter research question is investigated in [2] by noting those properties of the data formalism that are required in order to formally define syntax, well-formedness and semantics of an MSC with data. As demonstrated in [2] it is indeed possible to define such an interface between MSC and data. As usual, the interface is a two-way contract; it describes both the required assumptions concerning MSC behaviour and the sets and functions to be provided by the data formalism.

In [2] it is found that this interface is sufficient for connecting quite distinct data formalisms to MSC. This is demonstrated by two case studies. The first case study is an algebraic data language. The second case study is a constraint syntax language which takes ASN.1 as a starting point and is based on the proposal from Baker and Jervis [1].

It is interesting to remark that [2] exploits a notion of variables too. In the setting of the present paper, they are a kind of static variables: the semantics is taken to consist of *all* possible behaviours of giving values to the variables.

3. EXAMPLE LANGUAGE

In this paper we will use a simplified data language to explain the various features of dynamic data in MSC. Our language will consist of:

- The data types of Naturals and booleans
- Variables x, y, z and $x_1, x_2, \ldots$ signifying naturals, and p, q, r and $p_1, p_2, \ldots$ signifying booleans
- The operators $+$, $\cdot$, $\wedge$, $\vee$, and $=$

It also contains variable declarations in the form *var x: Natural* or *var p: Boolean*, and assignments of the form $x := e$, with x a variable and e an expression. Variable declarations are placed in the MSC, near the MSC name. Assignments are made in local actions.

As said, we will use this example to show various proposals and choices. To get a first idea of how MSC with variables might look like, we show an example in Figure 1.

Intuitively what happens here is that x gets the value 1, $m(x)$, which thus should be $m(1)$ is sent and received, y gets the value 4, and $k(5)$ is sent and received.

For this simple example, clear and unambiguous semantics can be given. In more complex cases, with for example multiple assignments or restricted scopes, this may not be the case, and explicit choices have to be made. This is what will be discussed in the next section.

4. CHOICES

When designing a combined MSC/Data language, there are several choices with respect to the precise interaction of the two languages. We will first list the major issues and discuss possible answers in the next sections.

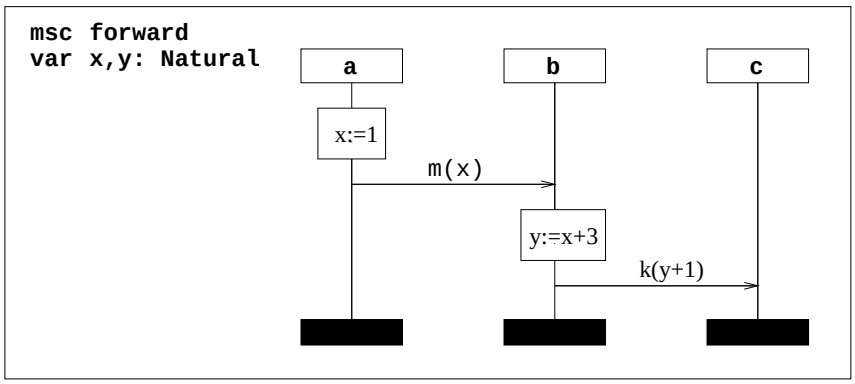

Figure 1. Example MSC with data

1. The extent to which the variables are behaving dynamically or statically.
2. The place where assignment of variables may take place.
3. The places where and the ways in which a variable is used.
4. The way undefined variables are handled.
5. What scope variables have, both regarding their extension over one, some or all instances and regarding their extension in time.

4.1. Static vs. dynamic nature of a variable

When talking about the dynamic nature of a variable, we are dealing with the liberty which we have in manipulating the data – the more dynamically variables are handled, the easier data can be manipulated. We distinguish four gradations here:

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 addition is the situation described in [2] and chapter 2 of the present paper.

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. It can be used especially when the same behaviour needs to be described for different values for some of the actions, see for example Figure 2.

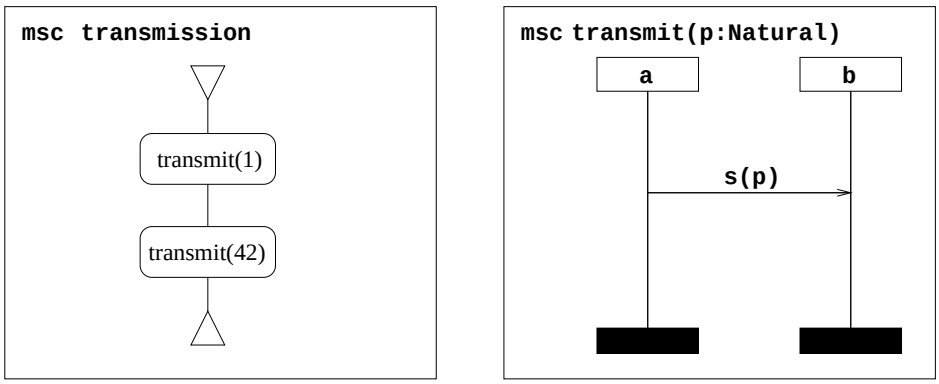

Figure 2. Example MSC with parameter variables

Intuitively, this means that the reference MSC *transmit* is called twice, but the first time with p equal to 1, and the second time with p equal to 42. Thus, first the message $s(1)$ is sent and received, then $s(42)$. Parametric data is semantically less complicated than the next two options of assignable variables. On the other hand, it is also less powerful. Of course, it would also be possible to include both options, resulting 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. A problem here is what we should do with attempts to access a variable before it is given a value. We will go further into this question below. An example of an MSC with single time assigned variables can be found in Figure 1.

Multiple times assignable variables: This choice offers most expressiveness to the user. No restrictions apply; the variables can have their value changed at any time (provided they have been declared), and as often as is wanted. On the other hand, it is also the most complex one, thus possibly causing problems to those theorizing about the language and the tool makers. One problem is, that one sometimes would like to use an *old* value of a variable in the interpretation of an expression. See for example Figure 3 (assuming global variables). Intuitively it is clear that the sending of the message $m(x)$ uses the last assignment to x, and thus its current value. But what about its receipt? If that too would use the current value of x, we could have the trace $x := 1; s(m(1)); x := 2; r(m(2))$ (here s and r are used to denote the sending and reception of a message). But this would mean that the received message is unequal to the sent message. It would be more natural to let the final receipt be $m(1)$. This would imply that it refers to the value of x at some time in the past (namely, when the message was sent). Although it is not impossible to formulate a semantics that describes this, it is cumbersome, and the resulting semantics might become non-transparent.

Note that this same problem can also arise with single time assignable variables, if we allow access to the variable before it is used – what is important, is that there is an assignment between the two usages of the variable.

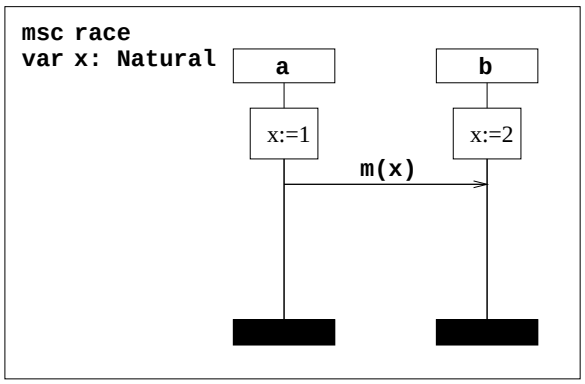

Figure 3. MSC with assignable variables and message

An even more complicated situation occurs when we consider multi-instance events. These are events that work on more than one instance but do not represent a point in time where all instances involved synchronize. One might think of conditions in this regard. There are proposals to use conditions as guards. This creates a problem when the truth value of a condition changes between the different times it is checked by the various instances. This problem will be described more extensively below, when we discuss the use of data in conditions.

4.2. Place of assignment

When having dynamic variables, one needs a construct to assign them a value. In the examples up to now, we have used local action for this purpose. Of course a new construct could be introduced to do so, but it seems both possible and preferable to use an existing construct for this, so that the language is not extended more than is necessary.

If we have static variables, no assignment takes place at all. Instead, all variables are quantified universally, that is, they can have *any* valid value, and a behaviour that corresponds to any value is a valid behaviour of the MSC. This is described in more detail in [2].

With parametric variables, we do have assignments, but they are necessarily part of the call itself, so again we have no options to choose from.

Thus, the only place where this question really comes up is with the (single or multiple times) assignable variables.

Apart from local actions, we could also use message inputs for assigning values to a variable. The idea is that a message which has received a value when sent, and has only a variable as its value when received, in that way sets the variable. This is shown in Figure 4. The variable x in the reference MSC is set by the fact that the actual value of the message $m(x)$ received from the environment is $m(3)$. One problem with this way of working could be that it may not be clear when a message receipt is a variable assignment and when it is not. For example, $m(3)$ could be unified with $m(x)$, but can $m(3+y)$ be unified with $m(4)$, or even $m(x+4)$?

Of course, some other MSC constructs could be used as an assignment, but we will not

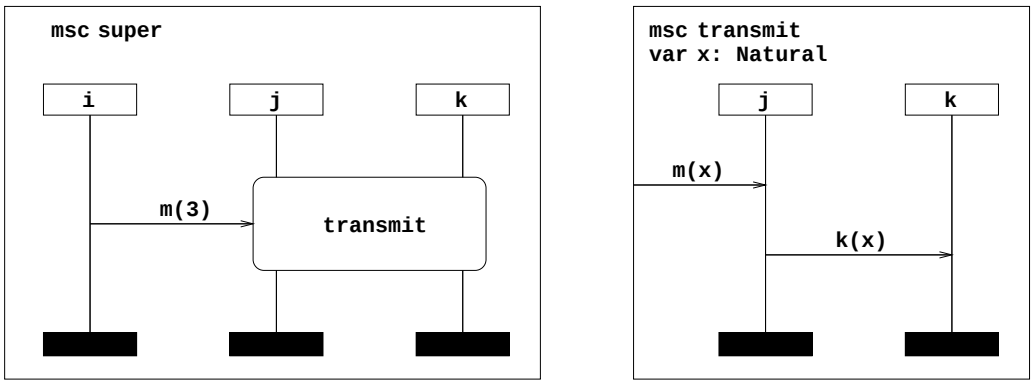

Figure 4. Message input as an assigning action

go into them here. Below when talking about data and conditions we will see one option in which conditions are used as assignments.

4.3. Place of referencing a variable

The next question we will discuss is the places and ways that a variable can be used. Basically, any place where now some string text appears in MSC which is not further specified, we could replace it with an expression in the data language. And such an expression could be, or could include, a variable.

This usage only requires evaluation of the expression. Such unspecified texts exist in local actions, messages, timers, and several other constructs, even instance names.

A more involved usage of expressions is in conditions. Currently, conditions do not have any dynamic meaning in the semantics of MSC. When data are added, they might be used as guarding conditions. To do so, one would put a boolean expression in the condition. The condition could then be passed only if the expression in it were true. For an example of this, see the left MSC in Figure 5. Instance a sends a message, containing the value of x, to instance b. If x equals zero, then the second alternative cannot be chosen, so it has to be the first, and instance b replies with message *zero*. If x does not equal zero, the first alternative cannot be chosen, and the second one will be, resulting in the message *nonzero*.

Unfortunately, we run into problems in cases like the one in the right MSC in Figure 5 Here, instance a gives x the value 0, then sends k, and arrives at the choice. It may now select the second alternative, change x to 1, and wait for $m(2)$ to arrive. However, it is possible that the right instance arrives at the choice of the alt-expression after x has been changed to 1, and then cannot pass the condition, at least not to the lower of the two choices where the left instance has gone. The question is: What should we do with such a case?

There are several options. We found at least the following, but this list is possibly not complete:

1. If we have only static and parametric data, or only single time assignable variables that cannot be used before their assignment, there is no problem.

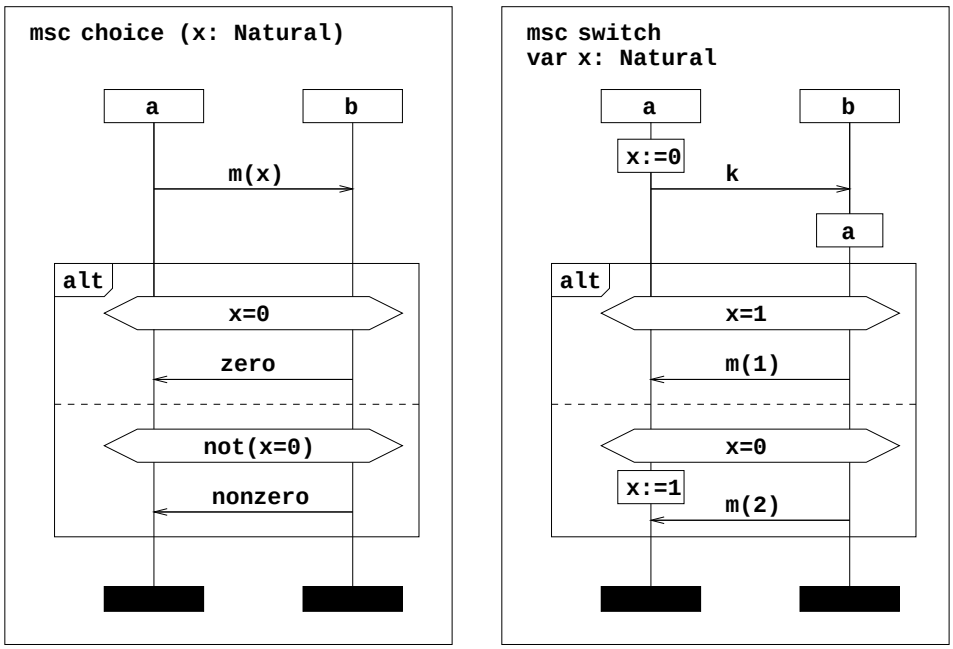

Figure 5. Using conditions as guards

2. Simply ignore this problem, and just go ahead with the semantics. This will cause the right MSC in Figure 5 to deadlock in the situation above, as the right instance will try to enter the second alternative, but cannot do so.

3. 'First one to pass the condition decides'. That is, the first instance going through the condition checks its truth value, and if it finds the condition to be true, then all instances can go through, no matter what the actual truth value at the time when they do so is.

4. Make a condition into a synchronization point. That is, all instances have to pass the condition at the same time. And thus, they have to evaluate the guarding expression at the same time, resulting in the same outcome.

5. Let each instance separately choose between the alternatives. In our example this would mean that whereas the left instance chose the lower alternative, the right one may still choose the upper one. We would like to advise against this alternative, though. It goes straight against the current MSC semantics, in which all instances do choose the same alternative. Another objection is that it solves the problem only when the condition guards a choice, not in other cases.

6. Make the assignments of variables in global conditions themselves, and let each instance remember its own copy of the variable, updating it when it goes through the condition. Although this does solve the problem, the working of variables may well differ much from people's intuitive ideas about them.

4.4. Handling of an undefined variable

In many cases it will be possible to refer to a variable while it does not have a value. It is not a priori clear how this should be managed. We will list several possible options:

1. Forbid this, and use static requirements to enforce this disallowance. This can be done if one uses a simple data language, and does not use complicated MSC structures (such as guarded loops), but if one uses a data language that is strong enough to function as a complete programming language, or if more complicated MSC functionalities are included, this may become difficult, or even impossible.

2. Detect the problem during dynamic evaluation. In this case we check runtime whether a variable is initialized. If not, we get a dynamic error (that is, semantically, a deadlock).

3. All undefined variables are regarded as universally quantified. That is, they can have any possible initial value. The semantics of the MSC is then the delayed choice of all possible behaviours for any initial value (or set of initial values). This is basically the same treatment as given to static variables in [2]. Thus, a dynamic variable is treated as a static one until its first assignment. A problem with this approach is that the number of alternatives could be infinite, and it is hard to give semantics for an infinite delayed choice, the resulting semantics without doubt being both ugly and hard to work with.

4. Each (type of) variable has a default value. Until its first assignment it has this default value. This way there are no undefined variables. We need to have a default value for each data domain, though.

The choice between these options will also be dependent on other choices. For example, the last two options, where the variable has some value before the first assignment, do not fit very well with the 'single assignment' dynamic variables. The reason that single assignment is simpler than multiple assignment, is that the variable will have the same value each time it is used. The last two solutions will remove this advantage, so we see no reason why, if they are used, one would prefer single assignments to multiple assignments.

4.5. Scope of a variable

A further point on which we can make different choices is in the definition of the scope of a variable. That is, once a variable has been declared, on which part of the MSC can it be used? This is the scope of that variable. Scopes might be nested, in which case the variables in the outer scope can also be used in the inner scope, unless a new declaration of the same variable has taken place. If a variable is used in two different scopes, then the two uses of the variable have nothing in common, and they should be regarded as two different variables that happen to share the same name.

We can distinguish two different dimensions to the scope: Block scope and architectural scope.

The block scope of a variable is a separated (framed) part of an MSC where the variable is defined. Such a separated part could be a complete MSC document, a single MSC, an MSC reference expression or an Inline expression. There might be more choices, but these seem to be the most logical ones.

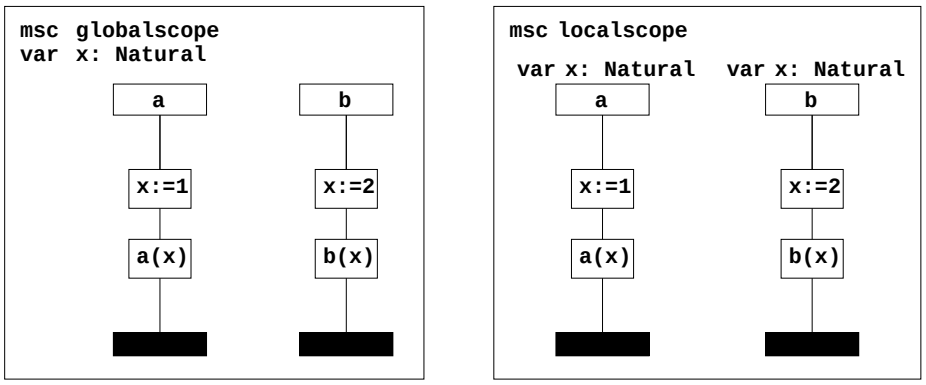

Figure 6. The difference between local and global architectural scope

Apart from this there is also the architectural scope. This gives the locality with respect to the instances in an MSC. We could define a variable to be defined on only one instance, or on all instances of the MSC. Possibilities in between, where a variable is defined on a number of instances (for example, the instances that reside on one processor), could also be considered, although it might be harder to find a syntax for that option.

The difference between (architecturally) local and global variables can be seen in Figure 6. On the left we see an MSC with a global variable x, on the right one with two local variables, both called x. The difference is that on the right, instance a will always do $a(1)$, as for this instance the value of x is 1, while the right instance will do $b(2)$, as for this instance the value of x is 2. On the left, where the variable x is global, it does not matter where the value of x has been changed, and thus both instances will use the last value of x, wherever it came from. Here $x := 1; x := 2; a(2); b(2)$ is a possible trace – at the time instance a executes $a(x)$ the value of x is 2, so that is the value that is used.

Of course one could decide to make all variables local, in that case the MSC on the left would act just like the one on the right, the creation of the variable x being shorthand for creating such a variable on all instances.

It might be argued that MSC is a language in which all communication is displayed explicitly, which implies that the introduction of a shared variable paradigm goes against the spirit of MSC. This would make the (architecturally) local character of variables the more logical choice. If one uses local variables, one does however also need a way to transport the value of a variable from one instance to another. A logical way to do so would be through messages - see however the problems that are mentioned about this in the 'place of assignment' subsection.

However, if this is chosen, one should also decide whether and how the value of a variable can be communicated from one instance to another.

5. SEMANTICS

As stated before, our aim is to parameterize the MSC language with a data language. To do so also requires to parameterize the MSC semantics. Included in such a parame-

terization needs to be an interface that specifies the information that is needed from the data language.

What exactly this interface looks like, depends on the choices made with respect to the issues raised above. For example, if one chooses to use default values to solve the problem of undefined variables, one needs these default values in the interface.

We will give example semantics for two choices, namely parametric variables with global architectural scope, and single-assignment dynamic variables with static check of undefined variables and global architectural scope. Several other choices will also be mentioned in short, giving an idea of how they can be elaborated as well as the problems this might cause.

5.1. General Interface

We first give a general idea about the interface that is needed, although some things will be added or removed depending on the exact choices that are made.

We would, in general, need 3 kinds of expressions:

- A set of declarations D, being the strings that represent variable declarations
- A set of assignments A, being the strings that represent (or can represent) assignments to variables
- A set of expressions E, being the strings that represent (or can represent) expressions

Apart from this we also need a set of possible variables Var, which in most cases will be a subset of E, and a semantic domain $\mathcal{S}$, in which the expressions will be interpreted.

To define the semantics of the parameterized MSC language, we need the notion of a state. A state gives a snapshot of the values of all variables involved.

A state consists of:

- A set of defined variables $V \subseteq Var$
- A valuation function $\varphi : V \to \mathcal{S}$, giving the values of the variables. The set of all valuation functions is called Φ.

Then, there need to be functions to interpret the various texts.

- For declarations: $d : D \to \mathcal{P}(\text{Var})$ giving the variables that are declared by D
- For assignments: A set $A_V \subseteq A$ for each set of variables V, giving the set of assignments that may actually be used, given that only variables in V are defined, and a state transition function $\tau : \Phi \times A \to \Phi$. $\tau(\varphi, a)$ denotes the new state the MSC turns into when assignment a is executed in state (V, φ). Note that $\tau(\varphi, a)$ needs only be defined when $a \in A_V$, where V is the set of variables on which φ is defined.
- For expressions: A set $E_V \subseteq E$ for each set of variables V, giving the set of expressions that may actually be used, given that only variables in V are defined, and an interpretation function $I_\varphi : E_V \to \mathcal{S}$, where V is the set of variables on which φ is defined. $I_\varphi(x)$ gives the value that x is interpreted to.

5.2. The Interface for our example

For our example language, the various elements of the Interface are described below.

- Var consists of all natural and boolean variables. Natural variables are x, y, z, $x_1, x_2, \ldots$; boolean variables are p, q, r, $p_1, p_2, \ldots$
- Natural expressions are formed from integers, integer variables, and the operators $+$ and $\cdot$, boolean expressions are formed from booleans, boolean variables, and the operators $\neg$, $\vee$, $\wedge$ and $=$ (the latter acting on natural expressions)
- D consists of all expressions of the form 'Var x: Nat', with x a natural variable or 'Var p: Bool', with p a boolean variable, and of lists of such expressions
- A consists of all expressions of the form $x := e$, with either x a boolean variable and e a boolean expression, or x a natural variable and e a natural expression
- E consists of all expressions of the form $a(x)$ with a a string of characters, and x a natural or boolean expression.
- $\mathcal{S}$ consists of the naturals, the booleans, as well as all expressions $a(x)$, with a some string and x a natural or boolean.
- $d(x)$ consists of all variables in x
- A_V and E_V consist of those expressions that contain only variables in V.
- The value of a natural or boolean expression e, given a valuation function φ is its numerical or truth value when each variable x in that expression is replaced by its value $\varphi(x)$. This value will be called $\varphi(e)$.
- $\tau(\varphi, x := e)$ is equal to φ with the value of x changed into $\varphi(e)$.
- $I_\varphi(a(x))$, with a some string and x some expression, equals $a(I_\varphi(x))$.

5.3. General Semantics

In a state (V, φ), all expressions must be in E_V and all assignments in A_V. Provided one uses variables with a well-defined scope, this should not be hard to check statically.

If the MSC is in a state (V, φ), all events such as $action(a, i)$ and $send(m, i, j)$, have as their "name" parts (i.e. a for an action and m for a message) an expression, which thus must be in E_V. The semantics for such an action are then equal to the semantics of $action(I_\varphi(a), i)$ and $send(I_\varphi(m), i, j)$, respectively in 'normal' MSC, apart from the two exception below:

- If we use assignable variables, using local actions as assignments, then a local action $action(a, i)$ may also have $a \in A_V$. In this case it causes a state change from (V, φ) to $(V, \tau(\varphi, a))$.
- If we use multiple times assignable variables, or single time assignable variables that can be used before their assignment, then to give the semantics of a message receipt, we need to use the state at the time the message was sent. We will discuss the problems this might result in, and how they might be solved, when discussing the extension of the semantics to other options.

Using the semantics of existing MSC, the semantics of MSC with data can now be described by (we will include the first but not the second exception):

$$\frac{x \overset{action(a,i)}{\longrightarrow} x', a \in E}{(x, V, \varphi) \overset{action(I_\varphi(a),i)}{\longrightarrow} (x', V, \varphi)}$$

$$\frac{x \overset{action(a,i)}{\longrightarrow} x', a \in A}{(x, V, \varphi) \overset{action(a,i)}{\longrightarrow} (x, V, \tau(\varphi, a))}$$

An operational rule like $x \xrightarrow{a} x'$ means that in an expression x one can do an action a, to end up in the expression x'. In the above rules, the interpretation of MSC with data (x, V, φ), is derived from that of MSC without data x – the step below the line can be taken if the step above the line can.

5.4. Parametric Variables

The interpretation to parametric variables as given below complies with the *call-by-value* principle used in many programming languages for instantiating the values of procedural parameters.

In parametric data, we do not have explicit assignments, so we can do away with A, and declarations will probably also not be necessary, as they are implicit as well. What remains are E, Var, E_V, and I_φ.

In parametric variables, we could have an MSC reference expression (most important example) refer to an MSC $msc\_name(e_1, e_2, \ldots e_n)$, with $e_1, \ldots, e_n \in E_V$, provided that MSC msc_name is defined as msc $msc\_name(x_1, \ldots, x_n)$, with $x_i \in Var$, $x_i \neq x_j$.

In this case the semantics of the MSC reference expression $msc\_name(e_1, \ldots, e_n)$ in a state (V, φ) can be found by pre-processing the MSC msc_name by changing expressions e into $I_{\varphi^*}(e)$, using the state $(V \cup \{x_1, \ldots, x_n\}, \varphi^*)$, where:

- $\varphi^*(x) = e_i$ if $x = x_i$
- $\varphi^*(x) = \varphi(x)$ if $x \notin \{x_1, \ldots, x_n\}$

Parametric variables are relatively simple, semantically speaking, which can be seen from the fact that pre-processing, as described above, is enough to describe the semantics. When we get to dynamical data, this will not be the case – we will have to use the state within the operational calculations

5.5. Single Time Assignable Variables

Our second example will use the following setting: single time assignable variables, static checks that unassigned variables are not used, architecturally global variables, and assignments in local actions.

Here it is important to use unique names for variables. To do this, each time we meet a declaration of an already used variable, we use a new name. Each action in the scope gets a list of which variable is actually used when some variable (like x) is meant.

That is, if we get a variable x in a scope, we define a fresh variable x_i, and everywhere within the scope, all actions get a pointer, saying that instead of the value of x, the value

of x_i has to be checked. To be able to do so, the declaration of variables should be checked by the semantics at a time when it is clear what is and what is not the scope.

To do so, we need to 'rename' the variable within the scope, if it also has a meaning outside the scope. Thus, we will need to have renaming operators $\rho_{(x\mapsto y)} : E \mapsto E$ and $\rho_{x\mapsto y} : A \mapsto A$ for each pair of variables x and y, defining what is the result when x is replaced by y. Furthermore, to make everything work, Var should be (countably) infinite, and ρ should have certain nice properties, for example $I_{\varphi^*}(\rho_{x\mapsto y}(e)) = I_\varphi(e)$, when φ is not defined on y, and φ^* is given by adding $\varphi(y) = \varphi(x)$ to φ.

Suppose an MSC reference expression is called, which consists of a variable declaration D, and an MSC-part k. Then the semantics of MSC-reference expression (D, k) in a state (V, φ) is that of $\rho_{x_1\mapsto y_1}(\rho_{x_2\mapsto y_2}(\ldots(k)\ldots))$ in a state $(V \cup \{y_1, y_2, \ldots y_n\}, \varphi)$ (lifting ρ to MSC actions and expressions, then to a list of them, in the obvious way). Because the value of the new variables cannot be used until they have been assigned, how φ is extended to the new variables does not matter. $x_1, \ldots, x_n$ are the variables defined by D, that is, $v(D)$, and $y_1, \ldots, y_n$ are n unused variables.

Having done all this, we can then use the semantics as given in section 5.3.

5.6. Types

In the example language, there are variables in two types, natural and boolean. It is probably useful to add this concept to the MSC semantic interface, so we know that the expressions $x \wedge y$ is only legal if x and y are boolean expressions. Above we have used different sets of variables (p, q, r, $p_1, \ldots p_n$ for booleans and x, y, z, $x_1, \ldots x_n$ for naturals) to distinguish the two types, but in more realistic languages, one variable could be of more than one type, depending on the preceding declaration. Of course, which types exist, what their semantical interpretation is, and how the variables are given types, should be inherited from the data language. It has no actual relevance for the MSC language and its semantics. An exception could be the type of booleans, which would be the only allowable type if guarding expressions in expressions are added (see section 4.2); a similar treatment could also be given to naturals, which are used to give the number of times a loop must be passed, this could also be extended to include general expressions of the appropriate type.

Semantically, a semantic domain is asked for each possible type; their interaction is completely left to the data language. Using types would give a somewhat more complicated interface, which we will not give here.

5.7. Other choices

Above for two possible choices, the semantics have been discussed. We will now look at the other options, but in much less detail, giving only the main differences with our example settings in both the interface and the semantics.

Multiply assignable variables: Compared to singly assignable variables, the extra problem is in the situation that the value of a variable may change between the sending and the receipt of a message. Yet, the receipt should use the values as they were when the message was sent. To make this possible, one should add either to the process algebra expression or to the state, a list of messages that have been sent, and how they were interpreted. Then, when a message must be received, instead of looking up the current value, one should use the value given by this list. This method only works if one strictly

keeps the uniqueness of message names. The semantics currently assume this uniqueness, but in the presence of loops, maintaining it is tricky.

Apart from this problem, multiply assignable variables can be given the same semantics that have been given to singly assignable variables.

Assignment in messages: One option here would be to make an incoming message expression correspond to an outgoing message expression, provided there is some assignment to variables that would allow it. The value of the variable after that would be any value for which the 'fit' would work. Still, there is the problem of when an incoming message is defining a value, and when it is merely using it. We do not see an immediately obvious answer to this question.

Guarding conditions: First, we need to define a set of Boolean expressions B, which have B_V and I_φ like normal expressions have E_V and I_φ, but necessarily have $\{\text{true}, \text{false}\}$ as their semantic domain. All conditions need to have a text which is a boolean expression in B_V. Where until now each condition could be gone through without changing anything, this now will only be true for conditions that evaluate to 'true'. If a condition evaluates to 'false', it acts as a deadlock.

The more detailed semantics for such a guarding condition, depends on what choice is made to deal with the 'changing value' problem, whether it be one of the five options we provide or yet another one.

Dynamical check of unused variables: The state now contains two sets of variables instead of one. One of these is the existing set V of defined variables, the other a set $V' \subseteq V$ of variables that actually have received a value. An action containing an expression may only be done when its expression is not just in E_V, but in $E_{V'}$. An action being disallowed is equivalent to making it a deadlock. For assignments, whether an assignment is allowed is dependent both on V and on V', resulting in $A_{V,V'}$ instead of A_V. If a variable gets assigned a value, it is added to V'.

Default values for unused variables: It is necessary to add the default variables for each variable (or, more likely, one default value for each variable type) to the interface. Apart from that, the problem of variable value changes, found in multiply assignable variables, also needs to be dealt with again; we propose the same solution.

Unused variables are universally quantified: We need the semantics to be the delayed choice of all possible 'default value' semantics. A problem here is that this may be a delayed choice of infinitely many options. Such an infinite operator makes things very complicated. Rules can probably be found for it, but most likely will be both ugly and unworkable.

Local variables: Instead of one set of variables V and one valuation function φ, we now have one set and one function for each instance in the MSC. The set and the function of the instance on which an action takes place are used to determine the semantics of that action. Of course message receipt is an exception, checking the instance from which the message was sent rather than the instance on which the action itself takes place.

When we have globally defined variables with local values, it seems logical to allow messages to make the value of a variable known to other instances. To allow this, we would need an extra interface function $c : E \to \text{Var}$, giving the variables whose values are sent when the expression is sent.

For each message, a 'snapshot' of the sender's state when the message was sent is

remembered, and not only is this snapshot used to interpret the receipt of the message, but also for the variables in $c(e)$, the value on the receiving instance is set to the value they have in this snapshot.

6. CONCLUSIONS

In this paper we have argued that the extension of MSC with a data language shall be accomplished in such a way that the data language definition is a parameter of the MSC language definition. This will overcome maintenance problems of the recommendation and will make it possible to anticipate at the variety of data languages already used in conjunction with MSC.

In [2] we have shown feasibility of this approach for the case of a data language with static variables, using as our examples an algebraic specification language and a constraint syntax language. In the current paper we have extended this research to dynamic variables. Rather than giving a precise description of how to incorporate dynamic data variables in MSC, we have listed a range of questions and possible answers concerning the connection between MSC and a data language. Since most of the questions are orthogonal, in the sense that possible answers to one question do not restrict the answers to other questions, this gives raise to a large variety of options.

One such option is the following: variables can be assigned only once, static checks on the MSC guarantee that no references to uninitialized variables are made, variables are known to all instances, and only in local actions variables can be assigned a value.

For a number of such options we have experimented with defining the interface between the MSC language and the data language, and we have sketched a semantics of this combined language, based on such an interface. These experiments indicate that the parameterization approach is also feasible for dynamic variables. Of course, a more detailed treatment of the semantics of the combined MSC/Data language is dependent upon the design choices made by the MSC standardization group. However, the present paper gives an overview of both the possibilities and the problems from a semantic point of view.

REFERENCES

1. P. Baker and C. Jervis. Formal description of data. Experts meeting SG10, Lutterworth TDL16, ITU-TS, 1997.
2. L.M.G. Feijs and S. Mauw. MSC and data. In Yair Lahav, Adam Wolisz, Joachim Fischer, and Eckhardt Holz, editors, *SAM98 - 1st Workshop on SDL and MSC. Proceedings of the SDL Forum Society on SDL and MSC*, number 104 in Informatikberichte, pages 85–96. Humboldt-Universität Berlin, 1998.
3. ITU-TS. *ITU-TS Recommendation Z.100: Specification and Description Language (SDL)*. ITU-TS, Geneva, 1988.
4. ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, 1997.
5. D. Steedman. *Abstract syntax notation one (ASN.1): the tutorial and reference.* Technology Appraisals Ltd., 1990.