MSC and data

L.M.G. Feijs[1,2], *S. Mauw*[2]

[1] *Philips Research Laboratories Eindhoven.*
Prof. Holstlaan 4, 5656 AA, Eindhoven, The Netherlands.
[2] *Department of Mathematics and Computing Science,*
Eindhoven University of Technology,
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands.
feijs@win.tue.nl, sjouke@win.tue.nl

Abstract

The extension of the MSC language with more advanced data concepts is one of the current topics of discussion in the MSC standardization community. We discuss some problems and possibilities. By means of two case studies we study the practical consequences of our proposed approach.

Keywords

MSC, data, algebraic specifications, ASN.1, semantics

1 INTRODUCTION

Quite high on the list of possible extensions for MSC [4] 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 [6], 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 initiate research on the extension of MSC with data. The following three questions will guide us.

1. At which places in the language might data be useful?
2. Which extensions of MSC are natural after introducing data in MSC?
3. Is it possible to set up the recommendation in such a way that the actual data language can be considered as a parameter, which may be easily instantiated with any formal data language.

In this paper we will discuss these questions. In other words, the question we shall investigate is: "Is it possible to use an abstract data type *data type* for adaptation of data models? This would allow a combined use of data coming from different formalisms." We require that MSC/data, the future extension of MSC with data, is such that there is a clear and minimal coupling between the MSC behavioural description language and the data language.

We will illustrate this approach with two case studies. The first case study concerns the extension of MSC with a data language based on algebraic specifications. The second case study takes ASN.1 as a starting point and is based on the proposal from Baker and Jervis [2].

Acknowledgments We thank André Engels and Michel Reniers for proof reading previous versions of this document. We appreciated the discussions with Paul Baker, Jan Friso Groote, Clive Jervis, Frans Meijs and Jaco van de Pol on the extension of MSC with data.

2 THE ROLE OF DATA IN MSC

2.1 Where to add data?

In this section we make an inventory of places in an MSC where data might play a role. We do not advocate that data should be allowed at all these places; we merely list options. The most obvious place where data will play a role is as an argument to a message (e.g. one can send the message $m(len(buf)+1)$). Thus parameters of a message may become expressions in some data language. In general, every variant part of the language which is not clearly meant as an identifier might be considered a data expression. A quick scan through Z.120 gives the following list.

- instance parameter, instance kind
- action text
- message instance name, message parameter
- condition name
- timer instance name, timer duration
- msc name (or rather, its parameter)
- msc document name (or rather, its parameter)
- sdl reference
- loop boundaries
- create parameter

This is probably not a complete list. A nice idea would be to treat all object declarations (i.e. msg, inst, msc, mscdoc, timer, create) in the same way. That means that they have a name and a parameter list. This is a list of expressions over some data language. This imposes a requirement on the data language, namely that these expressions can be evaluated in order to understand the meaning of the MSC.

The interpretation of an action in a data language is somewhat different. It is not simply an expression, but a program(fragment) from the data language which might have side effects. This imposes the requirement that it must be possible to execute such a program in the data language and determine its side effect.

For conditions, there is a very simple interpretation if we can interpret them as predicates (i.e. a boolean function) over the data language. The canonical interpretation is that a condition blocks all its attached instances whenever the predicate evaluates to false. Otherwise, it simply allows continuation. This imposes the requirement that the data language must have predicates that can be evaluated.

2.2 Possible extensions by introducing data

A number of additional extensions come into the picture after having extended MSC with data. We only mention the following.

- An if-then-else construct

- If an instance sends some message to some other instance, the sender may parameterize the message with some concrete data value, while the receiver may bind this value to some (local) variable. Graphically this can be expressed by associating two message names to a message arrow. One near the sending instance (e.g. $m(4)$) and one near the receiving instance (e.g. $m(v)$). The interpretation is that after reception of the message the value of the variable v has become 4.

2.3 Parameterization of the standard with a data language

The question of how to parameterize the MSC language with a data language seems to be the most important one, because a good solution anticipates at a large number of problems.

Let us first have a look at possible ways to integrate data into MSC.

- private (identifier-based, as is now)
- fixed external (select one of Act-One, ASN.1, C, ...)
- parameterized over a fixed enumerated set of external languages (allow e.g. Act-One, ASN.1 and C)
- parameterized over a lexicon (we only use the lexical syntax of the data language)
- parameterized over a signature.
- parameterized over a context-free grammar
- parameterized over an attribute grammar
- parameterized over some grammar with some semantical information

The most appealing and general solution is the last one. It implies that the recommendation does not explicitly define the data language. It merely describes which syntactical and semantical properties a data language must have, in order to be considered as a valid instantiation. Syntactical requirements could be a restriction on the character set and the requirement that parenthesis are always properly nested (in order to be able to detect the end of the data expression in $m(\ldots)$). Another syntactical requirement could be a function to detect type-correctness of an expression.

Some semantical requirements have already been stated in the discussion on the first question, namely, we must have a function to evaluate a data expression and we must be able to evaluate a predicate. In case that we have an imperative language (i.e. a data language with the notion of a state or variables), we must also be able to determine the result of an action (i.e. a program fragment with side effects). Therefore, we must be able to keep track of the state of the system.

It may be assumed that the inclusion of a functional data language is much easier than of an imperative language. In a functional language, programs have no side effects, and thus an action simply occurs and variables are just place holders. In an imperative language, we must be able to determine the effect of an action on the state space, and furthermore we must be able to determine the scope of variables. Are variables local to an instance or global to all instances (shared variables seem to contradict the spirit of MSC)?

To allow parameterization of the MSC language, we must determine which properties of the data language are required in order to formally define syntax, well-formedness and semantics of an MSC with data.

Please note that we can reach this situation without ever having to select a particular data language. For every data language that meets the criteria, the integration of MSC with this language is a simple step. One only has to define the required functions for evaluation, etc.

Every user or every tool builder may introduce his own favored data language. In order to prevent MSC from becoming a moving target the MSC standardization group may select several possibilities and define for these languages the required functions.

3 CASE STUDIES

3.1 Approach chosen

In this section we choose to replace the present syntax for message parameters by a more general expression language with variables and subexpressions (the third • from Section 2.1). Then the MSC language

depends on the data language and we assume that this is to be done by letting the MSC language being parameterized over some grammar with some semantical information (the last • from our list in Section 2.3).

We learn from the work of Baker and Jervis [2] that it may be desirable to work with constraints which apply to the messages. In this way it is possible to completely fix a message, but it is also possible to give constraints to one or more fields, leaving the value of the other fields unspecified. This seems a good idea, which will be at the heart of the case study. Also from Baker and Jervis we take the idea to consider only languages whose syntax fits in a natural way in the m or $m(p_1, \ldots p_n)$ format of MSC96.

We want to experiment with two kinds of data languages, aimed at two different ways of using MSC and coming from two different backgrounds:

- an algebraic data language,
- a constrained syntax language.

The former is likely to reflect the needs when using MSC as a specification and design formalism. The idea is that variables may be used; such an 'MSC with variables' represents a large collection of normal MSCs, namely those which can be obtained by instantiating the variables. Algebraic data languages have a strong tradition in academic research.

The latter is more likely to reflect the needs when using MSC as test specification language. The idea is that certain messages or certain fields are deliberately left unspecified, for example because they are not relevant (yet) or because a value has to be provided later. Constrained syntax languages have a long tradition in industrial applications, notably the languages ASN.1 [6] and TTCN [7].

3.2 MSC and an algebraic specification language

Example
We start this section with a simple example that shows the combination of MSC with a data language based on algebraic specifications (see Figure 1).

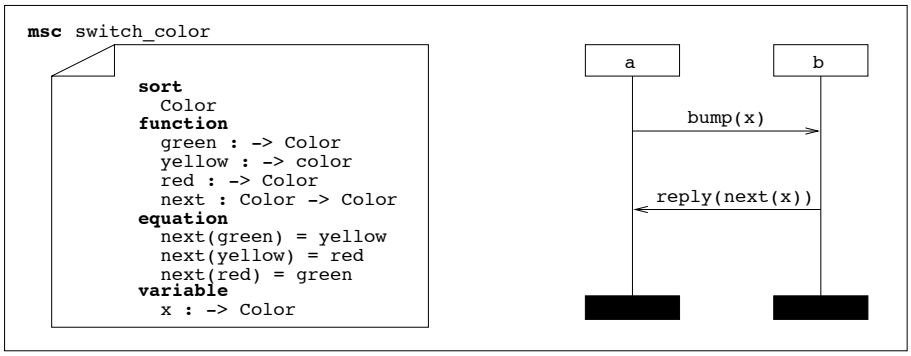

Figure 1: Combining MSC and algebraic specifications.

This MSC consists of two parts, namely, the data declaration and the behaviour specification. The data declaration specifies a signature: a sort named `color`, three constants of this sort, named `green`, `yellow` and `red`, and a function `next` from `Color` to `Color`. The meaning of this function is fully specified by the three equations. Finally a variable `x` of sort `Color` is declared.

The behavioural specification is a *standard* MSC. It describes two messages, a `bump` message, parameterized with the term `x` and a `reply` message, parameterized with the term `next(x)`. Clearly, it is the intention that these parameters refer to the algebraic specification.

The intended meaning of this MSC is that for any given value of `x`, a `bump` message is sent from instance `a` to instance `b`, followed by the `reply` from `b` with the next color. In a more formal way, we can say that the meaning of this MSC is the choice between three basic MSCs that are derived by substituting the three possible values `green`, `yellow` and `red` for `x`. In Figure 2 we show the intended meaning of the MSC from Figure 1, denoting the choice with the symbol $\mp$ which is known as the *delayed choice* (see [1]).

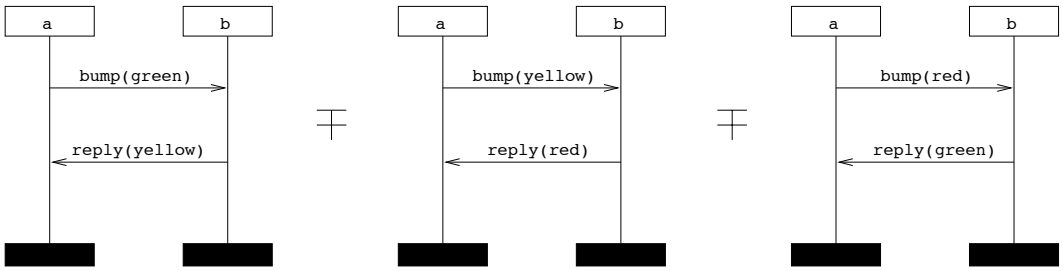

Figure 2: Intended semantics of the example.

Formalization

Next, we formalize the construction from the previous section. As explained before, our aim is to study what the *interface* is between the behavioural MSC language and the data language. We try to abstract as much as possible from the actual data language being used.

We consider Message Sequence Charts consisting of instances and messages only.

First, we concentrate on the behavioural (data-free) Message Sequence Charts. MSC_P denotes the class of MSCs with parameters of messages taken from some set P. This set will be instantiated later.

We presuppose existence of a semantical function S_P that maps MSCs into some semantical domain $\mathcal{A}_P$.

$$S_P : MSC_P \rightarrow \mathcal{A}_P$$

Furthermore, we assume that an operator $\mp$ for alternative composition is defined on the semantical domain:

$$\mp : \mathcal{A}_P \times \mathcal{A}_P \rightarrow \mathcal{A}_P$$

We assume that this operator is independent of the set P. In order to be able to generalize this operator to an operator $\sum$, we require that $\mp$ is associative and commutative and that there is a unit element for this operator. This unit element is the outcome when taking an empty summation. It is also needed to require that $\sum$ is well-defined for infinite sequences of arguments (unless only finite data domains are used).

Next, we consider the data language. Let D consist of all strings that represent a well-formed data declaration. In order to check whether some string is in D we need the predicate

$$wf_{decl} : \mathsf{String} \rightarrow \mathsf{Bool}$$

This predicate is necessary to check syntactical correctness of the data part of an MSC/data specification. So D is the set of strings s such that $wf_{decl}(s) = \mathsf{true}$. Let $T(decl)$ (or T for short) represent the set of well-formed terms over $decl \in D$, then we need the predicate

$$wf_{term} : D \times \mathsf{String} \rightarrow \mathsf{Bool}$$

This predicate is needed to check whether parameters of messages are syntactically well-formed.

Now we can define an MSC/data specification as a tuple $< decl, msc >$ such that $decl \in D$ and $msc \in MSC_T$.

In order to define the semantics of such an MSC/data specification, we need some additional information.

First, we must be able to make a distinction between open and closed terms. An open term (in case of an algebraic specification) is a term which contains variables. In general it is a term which can be made concrete (closed) in several ways by means of a substitution. Therefore, we need the predicate

$$closed : D \times T \rightarrow \mathsf{Bool}$$

This predicate determines whether a given term from T, given a data declaration from D has no variables. We denote the set of closed terms by T_{closed}. From the function *closed* we can derive a similar function which checks if all terms occurring in an MSC are closed.

$$closed : D \times MSC_T \rightarrow \mathsf{Bool}$$

From an open term, we may derive a number of closed terms by substituting concrete values for the variables. Thus we need a set of substitutions, which we call $SUBST$, and a function which applies a given substitution to some term:

$$apply : SUBST \times T \rightarrow T$$

We require that the function *apply* is the identity on the set of closed terms. We generalize this function in the obvious way to MSCs. The application of a substitution to an MSC consists of the application of the substitution to all terms occurring in the MSC.

Before we can finally present the semantics of the MSC/data specification, we presuppose that the closed terms are interpreted in some semantical data domain $\mathcal{D}$. Then we need a function

$$S_{data} : D \times T_{closed} \rightarrow \mathcal{D}$$

which defines the semantics of a given (closed) term with respect to a given data declaration. This function can be extended to closed MSCs in a straightforward way: replace all closed terms that occur as the parameter of a message by their interpretation in $\mathcal{D}$ (applying S_{data}). This gives a function

$$[\![.]\!]. : MSC_{T_{closed}} \times D \rightarrow MSC_{\mathcal{D}}$$

Finally, we can define the semantics of an MSC/data specification by means of the function

$$S : D \times MSC_T \rightarrow \mathcal{A}_{\mathcal{D}}$$

which is defined as

$$S(< decl, msc >) = \sum_{k \in V} S_{\mathcal{D}}(k)$$

where the set V is defined by

$$V = \{\, [\![apply(\sigma, msc)]\!]_{decl} \mid \sigma \in SUBST, closed(decl, apply(\sigma, msc)) \,\}$$

This means that, for a given $msc \in MSC_T$ with parameters from T, possibly containing variables, we first determine the set of MSCs that result from all possible closed substitutions. Next, we replace every closed term obtained in this way by its semantical interpretation. For all the resulting MSCs we determine the semantics and, finally, we consider the resulting semantical expressions as alternatives. This describes the transformation from Figure 1 to Figure 2. However, we should then read the MSCs in Figure 2 as the graphical representations of their semantical meaning in $\mathcal{A}_{\mathcal{D}}$.

3.3 MSC and a constrained syntax language

Example

In this section we study the extension of MSC with a constrained syntax language as proposed by Baker and Jervis [2]. The proposal is aimed at allowing a more flexible syntax for the specification of message parameters. Its main virtues are the following.

- Not all parameters of a message have to be provided. An unspecified parameter may have any value.
- Parameter values may be specified in an unordered way by using a reference syntax.
- Constraints can be used to reduce the number of allowed values of a parameter.

In Figure 3 we show an example of an MSC specification combined with a simple ASN.1 specification. Please note that this example deviates slightly from the syntax proposed in [2]. Namely, we have added a message name m, rather than considering the type `Frame` as the name of the message. The reason for this is that for a minimal interface between the MSC language and the data language, we do not want that the names of the messages have a semantical interpretation in both parts.

The ASN.1 part of the MSC/data specification declares a nested structure, named `Frame`. The first component of a `Frame` has name `P1` and ranges over the bitstrings of length 2. The second component, `P2`, is of type `Pair`. A `Pair` consists of two fields `X` and `Y`, which both contain a bit. Finally, a constraint

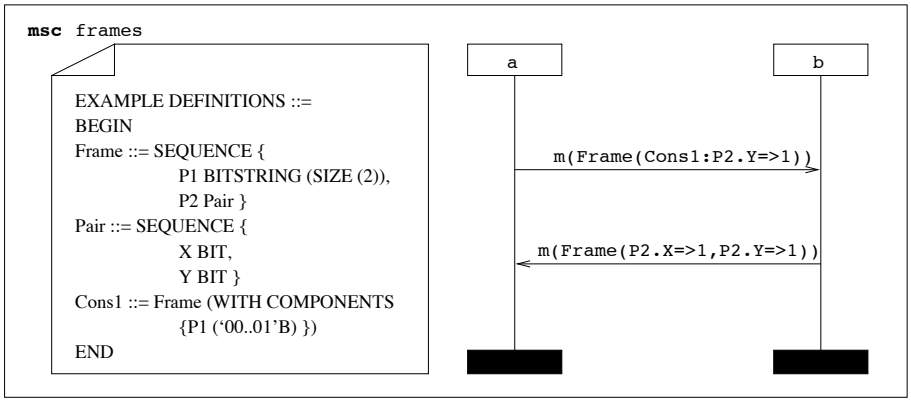

Figure 3: Combining MSC and constrained syntax notation.

`Cons1` is defined. When applied, it reduces the possible range of values for a `Frame` in the sense that only frames are allowed of which component `P2` ranges between the bitstring 00 and 01.

The meaning of the MSC is that first a message m is sent from instance a to instance b. Only messages are allowed of which the parameters satisfy constraint `Cons1` and the `Y` field equals 1. Next, a message m is sent back. For this message we know that both `X` and `Y` are equal to 1.

Just like the example in Figure 1, this MSC denotes a choice between a number of *standard* MSCs, namely all MSCs which satisfy the above mentioned requirements on the parameters. Since there are four possible instantiations of the first message and also four instantiations of the second message, there is a choice between 16 different MSCs.

Formalization
The formal treatment of the combination of MSC with a constrained syntax notation very much resembles the formalization of MSC with an algebraic specification language as in Section 3.2.

The only difference is that we should not consider a single substitution for a complete MSC, but a list of substitutions; one for each message. The reason for this is that the same constraint may occur within different messages, while the unspecified fields may have different values. A single substitution function would always give the same closed term for all identical constraints.

Define $SUBST^*$ as the set of all sequences of substitutions and let $\vec{\sigma} \in SUBST^*$. Then we generalize the function *apply* to lists of substitutions. The first substitution is applied to the first message (assuming some ordering on the messages), the second substitution to the second message, and so on.

Now, we define the semantics of an MSC/data specification as follows.

$$S(< decl, msc >) = \sum_{k \in V} S_{\mathcal{D}}(k)$$

where V is now defined as

$$V = \{\, [\![apply(\vec{\sigma}, msc)]\!]_{decl} \mid \vec{\sigma} \in SUBST^*, |\vec{\sigma}| = |msc|, closed(decl, apply(\sigma, msc)) \,\}$$

We use $|\vec{\sigma}|$ and $|msc|$ to denote the length of the sequence $\vec{\sigma}$ and the number of messages in msc, respectively. The only difference with the previous definition of the semantics is the use of these lists of substitutions.

3.4 Interface between MSC and data

From the above definitions, we can derive the *interface* which is needed between the two parts of the MSC/data language. In order to be able to define well-formedness and the meaning of such an MSC/data specification, we need that the predicates, sets and functions from Table 1 are defined.

In order to have one interface that works for both algebraic specification languages and constrained syntax languages we need an auxiliary function

$$is\_comp : SUBST \times SUBST \rightarrow \mathsf{Bool}$$

The reason for this is that we cannot simply adopt the semantics of the constrained syntax extension for the algebraic specification extension. This would mean that for every message we could select a different (incompatible) substitution, allowing the value of a variable to vary through the different messages. Therefore, we need the requirement that the substitutions in the list are all compatible. In the case of the algebraic specification extension this means that they agree upon the value of all variables occurring in the MSC. So the semantical expression for MSC/data specifications from Section 3.3 also holds for algebraic specifications, provided that we add the condition that all elements of $\vec{\sigma}$ are compatible.

Please note that in order for the constructions to be effective, all instantiations of the given functions must be computable.

Concerning behavioural MSC:	
semantical process domain	$\mathcal{A}_P$
semantical interpretation of MSCs	$S_P : MSC_P \to \mathcal{A}_P$
operator for alternative composition	$\mp : \mathcal{A}_P \times \mathcal{A}_P \to \mathcal{A}_P$
Concerning data:	
syntax check on data declarations	$wf_{decl} : \mathsf{String} \to \mathsf{Bool}$
syntax check on parameters	$wf_{term} : D \times \mathsf{String} \to \mathsf{Bool}$
closedness of terms	$closed : D \times T \to \mathsf{Bool}$
set of substitutions	$SUBST$
application of substitution	$apply : SUBST \times T \to R$
semantical data domain	$\mathcal{D}$
semantical interpretation of data	$S_{data} : D \times T_{closed} \to \mathcal{D}$
compatibility of substitutions	$is\_comp : SUBST \times SUBST \to \mathsf{Bool}$

Table 1: Interface between MSC and data

The functions needed in the previous paragraphs to define the semantics of an MSC/data specification which are not in Table 1, can be derived from the functions in this table.

3.5 Instantiating the interface

For every particular data language we only need to instantiate the interface from Table 1 in order to define syntax and semantics of a combined MSC/data specification. First, we discuss the instantiation of the interface for an extension of MSC with a simple algebraic specification language (e.g. from [5]).

The instantiation is sketched in Table 2. It shows that for most entries references to literature or standard solutions are provided.

Concerning behavioural MSC:	
$\mathcal{A}_P$	the process algebra as defined in Z.120 Annex B
$S_P : MSC_P \to \mathcal{A}_P$	the semantical function as defined in Z.120 Annex B
$\mp : \mathcal{A}_P \times \mathcal{A}_P \to \mathcal{A}_P$	the delayed choice operator from Z.120 Annex B
Concerning data:	
$wf_{decl} : \mathsf{String} \to \mathsf{Bool}$	see the BNF grammar and typing rules from [5]
$wf_{term} : D \times \mathsf{String} \to \mathsf{Bool}$	see the BNF grammar and typing rules from [5]
$closed : D \times T \to \mathsf{Bool}$	all terms constructed from function symbols only
$SUBST$	all functions from variables to (closed) terms
$apply : SUBST \times T \to R$	application of the (extended) substitution function
$\mathcal{D}$	the term model
$S_{data} : D \times T_{closed} \to \mathcal{D}$	the equivalence class of a term w.r.t. derivable equality
$is\_comp : SUBST \times SUBST \to \mathsf{Bool}$	substitutions agree on values for variables

Table 2: Instantiation of the interface for an algebraic specification language

In Table 3 we show the instantiation of the interface for the constrained syntax notation. For most components we refer to [2]. Since we allow all possible substitutions for all terms, we set the function

is_comp to true.

Concerning behavioural MSC:	
$\mathcal{A}_P$	the process algebra as defined in Z.120 Annex B
$S_P : MSC_P \rightarrow \mathcal{A}_P$	the semantical function as defined in Z.120 Annex B
$\mp : \mathcal{A}_P \times \mathcal{A}_P \rightarrow \mathcal{A}_P$	the delayed choice operator from Z.120 Annex B
Concerning data:	
$wf_{decl} : \mathsf{String} \rightarrow \mathsf{Bool}$	see the BNF grammar and typing rules from [2]
$wf_{term} : D \times \mathsf{String} \rightarrow \mathsf{Bool}$	see the BNF grammar and typing rules from [2]
$closed : D \times T \rightarrow \mathsf{Bool}$	terms of which all components are fully determined
$SUBST$	all functions that yield fully determined terms which satisfy the constraints
$apply : SUBST \times T \rightarrow R$	application of the substitution function
$\mathcal{D}$	the semantical model from [2]
$S_{data} : D \times T_{closed} \rightarrow \mathcal{D}$	the interpretation of a term in the model from [2]
$is\_comp : SUBST \times SUBST \rightarrow \mathsf{Bool}$	the function that always yields true

Table 3: Instantiation of the interface for a constrained syntax notation

4 CONCLUSIONS

This paper can be seen as a feasibility study on the incorporation of data in MSC. The wide range of different application areas of MSC does not allow for a single fixed data language. We have argued that this problem is addressed as a research topic; the present paper is meant as a contribution to this research, rather than as the *final solution*. We have shown that it is possible to parameterize over the data language actually being used. We have defined an interface between the behavioural MSC language and the data language, which enables us to give a complete syntactical and semantical definition of MSC/data. Although the interface is derived from only two data languages, this does not mean that it is only suited for these two languages. We expect that this interface is suitable for a large class of data languages.

Please note that the main question under investigation was defining the interface such that one language can take another language as a parameter. A priori, it was not clear that this would be possible at all, but our work demonstrates that we could factor-out the commonalities of two quite remote data languages. The topic of language engineering has some resemblance with the engineering of large computer programs: the major concern is in finding the right interfaces, creating freedom of implementation on one hand, and a range of future applications on the other.

We have started by listing many places in an MSC where data can play a role, but we have only exercized with message parameters. Within the MSC standardization group a selection of these places should be made. In this paper we have merely listed options, some of which are more useful than others. We expect that, e.g. the possibility to write predicates within conditions, thus guarding continuation of an MSC, will strengthen the use of conditions. Obviously, the interface then has to be extended with predicates, a well-formedness function for predicates and an evaluation function of predicates. So both the class of data languages and the choice where to allow data in an MSC will influence the interface.

The combination of named variables, acting as place holders, from an algebraic specification language with the unnamed variables implicit to the constrained syntax language, enforced the extension of the interface. A compatibility predicate on substitutions was needed to combine both approaches. When allowing other programming paradigms, the interface will have to be revised again.

We propose to conduct further case studies regarding more (and other types of) data languages, such as imperative languages containing state variables. At least the data languages supported by SDL should be studied in detail.

5 REFERENCES

[1] J.C.M. Baeten and S. Mauw. Delayed choice: an operator for joining Message Sequence Charts. In

D. Hogrefe and S. Leue, editors, *Formal Description Techniques, VII*, pages 340–354. Chapman & Hall, 1995.

[2] P. Baker and C. Jervis. Formal description of data. SG10 meeting Lutterworth TDL16, ITU-TS, October 1997.

[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] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[6] D. Steedman. *Abstract syntax notation one (ASN.1): the tutorial and reference*. Technology Appraisals Ltd., 1990.

[7] ISO/IEC Information technology. *OSI conformance testing methodology and framework, part 3: The Tree and Tabular Notation (TTCN)*. ISO document ISO9646. ISO/IEC, 1990.