Generating tools for Message Sequence Charts

S. Mauw[a] and E.A. van der Meulen[b]

[a] Dept. of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

[b] Dept. of Mathematics and Computing Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

The recent formalization of the semantics of Message Sequence Charts enables the derivation of tools for MSCs directly from this formal definition. We use the ASF+SDF Meta-environment to make a straightforward implementation of tools for transformation, simulation and requirements testing.

1. INTRODUCTION

Message Sequence Charts (MSCs) are a graphical method for the description of the interaction between system components [11]. Due to the recent formalization [7, 8, 12] of the semantics of Message Sequence Charts, we can consider MSC as a formal description technique.

Currently, this formalization has already influenced the development of the language (in particular with respect to composition of MSCs) and it is expected to also influence the use of MSCs.

Formalization will also have impact on the work of tool builders. The behavior of tools can be validated against the formal semantics, but even more valuable is the possibility to generate tools, or prototypes, directly from the formal definitions. This paper can be considered as a case study in the formal development of computer tools for programming languages.

In practice, tools for an informally defined language are developed mainly based on the intuition of the program designer. Unless all people have a common understanding of the language, this leeds to inconsistent tools. If a formal definition of the language is available, tools can also be based on the understanding of these formal semantics. This may lead to more consistent tools, but in practice this only works if the semantics is well accessible. A better approach would be to automatically implement the formal semantics of the language. This leeds to correct and consistent tools. A possible problem with this approach is that necessarily a formal semantics has a high level of abstraction and is not directed towards possible tools. Thus, automatic implementation of the formal semantics is not always feasible. An operational semantics and decisions on implementation details may be needed.

Our aim is to demonstrate how the abstract definitions of the formal semantics of Basic Message Sequence Charts (BMSCs) can be implemented. BMSCs are Message Sequence Charts with only the main features: communication and local actions. The techniques described in this paper transfer straightforward to the complete MSC language. As described in [7] the semantics of BMSCs is defined by a translation into process algebra. Because this translation is defined by means of equations and because the axioms defining process algebra are also equations, the obvious way of implementing the semantics of MSCs is by using algebraic specifications [1].

We used the ASF+SDF Meta-environment [3] for the implementation. With this system algebraic specifications can be implemented by means of term rewriting systems. Furthermore, a complete programming environment for BMSCs can be generated, including a syntax directed editor, a parser and a pretty printer.

In this paper we will only describe the structure of the tool set and highlight parts of the formal specification. The complete specification can be found in [9]. The implementation consists of three parts. The first part is the translation of BMSCs into process algebra expressions. This is based on the definition of the semantic functions in [7]. The second part is the definition of a simulator for BMSCs. Although a simulator is not part of the formal semantics, it can easily be derived from the operational semantics given in [7]. In fact the description of the simulator in [9] can be regarded as a formal specification of a simulation tool. The third part consists of an implementation of the static requirements for BMSCs expressed informally in [11] and formalized in [10].

This paper is structured in the following way. Section 2 contains a description of the ASF+SDF Meta-environment. In Section 3 we give a short overview of the BMSC language. Section 4 contains a description of the tool set and highlights parts of the specification.

Although this paper covers the complete semantics of BMSCs, it is not intended as a self-contained explanation of these semantics. Refer to [7] for a comprehensive treatment.

Acknowledgements

Thanks are due to Arie van Deursen, Wilco Koorn, Michel Reniers and Eelco Visser for their assistance during several phases of this project.

2. THE ASF+SDF META-ENVIRONMENT

The ASF+SDF Meta-environment [3] is a programming environment generator based on algebraic specifications. From a specification of the syntax and semantics of a language an environment is generated, in its simplest form consisting of a syntax directed editor and a term rewrite system. The generated environment can be customized further by means of the language SEAL [5, 6].

An algebraic specification [1] in ASF (Algebraic Specification Formalism) consists of two parts. A signature for describing sorts and functions, and a set of equations, which give an algebraic definition of the functions.

When specifying programming languages in an algebraic manner, the syntax for function definitions is found to be too restrictive. The formalism ASF+SDF therefore combines the algebraic specification formalism ASF with a formalism for defining syntax: SDF. SDF allows for the combined specification of concrete syntax (like in BNF) and abstract syntax. Hence, ASF+SDF is a formalism for writing algebraic specifications with user defined syntax. A modularization concept is part of ASF+SDF to support *design in the large*.

The most common strategy for implementing algebraic specifications is via term rewrite systems (TRSs, see [4]). An algebraic specification can be transformed into a TRS by interpreting the equations as rewrite rules from left to right. This TRS can be used to compute the value of a function application.

3. MESSAGE SEQUENCE CHARTS

Message Sequence Charts provide a graphical method for the description of the communication behaviour of system components. The ITU-TS (the Telecommunication Standardization Section of the International Telecommunication Union, the former CCITT) maintains recommendation Z.120 [11] which contains the syntax and an informal explanation of the semantics of Message Sequence Charts. A formal semantics based on process algebra has been proposed in [8]. This proposal is currently subject to standardization by the ITU ([12]).

In this paper we restrict ourselves to the core language of Message Sequence Charts, which we call Basic Message Sequence Charts (BMSCs). A Basic Message Sequence Chart concentrates on communications and local actions only. These are the features encountered in most languages comparable to Message Sequence Charts. Their semantics is described in [7].

A Basic Message Sequence Chart contains a (partial) description of the communication behavior of a number of instances. An instance is an abstract entity of which one can observe (part of) the interaction with other instances or with the environment. An instance is denoted by a vertical axis. The time along each axis runs from top to bottom.

A communication between two instances is represented by an arrow which starts at the sending instance and ends at the receiving instance. Although the activities along one

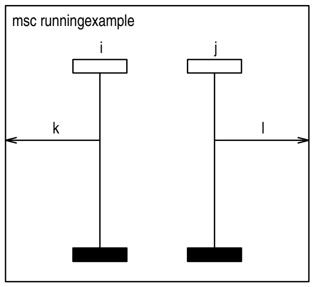


Figure 1. Example Basic Message Sequence Chart

single instance axis are completely ordered, we will not assume a notion of global time. The only dependencies between the timing of the instances come from the restriction that a message must have been sent before it is received.

Figure 1 shows a simple BMSC in which only communications with the environment are specified. This will be the running example in the remainder of this paper. Note that there is no ordering imposed on the events k and l.

The Basic Message Sequence Chart of Figure 1 has the following textual representation.

```
msc example1;
  instance i;
    out k to env;
  endinstance;
  instance j;
    out l to env;
  endinstance;
endmsc;
```

4. DESCRIPTION OF THE TOOLS

For each of the tools generated by the ASF+SDF Meta-environment we give a short description and highlight small parts of the algebraic specification. The complete specification can be found in [9]. First we give an overview of the relation between the tools.

4.1. Overview

Figure 2 describes the structure of the generated tool set. Boxes denote expressions in the given language and arrows represent transformations from one language to the other. Apart from the *INPUT* language which is plain ASCII, we consider the following languages. *MESSAGES* is the language of output messages generated by the requirements checker and the simulator, *BMSC* is the language of (parsed) Basic Message Sequence Charts, PA_{BMSC} is the process algebra theory used for describing the semantics of BMSCs (see [7]) and *BPA* is the sub-language of PA_{BMSC} that only contains the *normalized* PA_{BMSC} expressions. The generated tools are considered as transformation

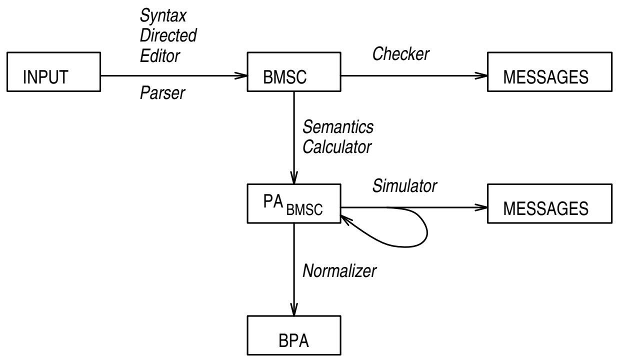


Figure 2. Structure of the tools

tools, described by algebraic specifications. The implementation of these specifications is internal to the ASF+SDF Meta-environment. We specified the following tools.

Syntax directed editor and parser The parser converts plain ASCII text into BMSC.

Checker The additional syntax requirements (static semantics) for BMSCs can be checked with this tool.

Semantics Calculator The semantics of a BMSC is described by a translation into the process algebra PA_{BMSC}. The Semantics Calculator computes the semantics of a BMSC.

Normalizer The normalizer reduces the expression resulting from the previous step to normal form. This tool makes it possible to inspect the complete behavior of the given BMSC.

Simulator Test runs of the BMSC can be generated interactively with the simulator. It offers the user a choice between all possible continuations. After selecting one event, it calculates the PA_{BMSC} expression that results after execution of the event.

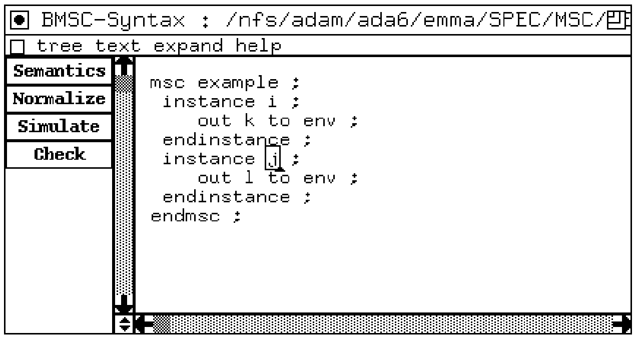
4.2. Syntax directed editor

A syntax directed editor for BMSC is automatically derived from the description of the syntax of BMSC in ASF+SDF. Part of the module BMSC-Syntax in which the syntax is specified is presented below. The module uses the module Identifiers which introduces MSCID (MSC names), IID (Instance names), MID (Message names) and AID (Action names).

The sorts MSC, MSC-BODY, INST-DEF, INST-BODY and EVENT are the non-terminals of the grammar. The production rules of the grammar are expressed in the context-free syntax section of the specification. For example, an MSC-BODY is either empty or consists of an INST-DEF followed by an MSC-BODY. Using this grammar, functions on BMSCs can be defined by means of induction on the structure of a BMSC.

```
BMSC-Syntax
sorts MSC MSC-BODY INST-DEF INST-BODY EVENT
context-free syntax
  "msc" MSCID ";" MSC-BODY "endmsc" ";"            → MSC
                                                   → MSC-BODY
  INST-DEF MSC-BODY                                → MSC-BODY
  "instance" IID ";" INST-BODY "endinstance" ";"   → INST-DEF
                                                   → INST-BODY
  EVENT INST-BODY                                  → INST-BODY
  "in" MID "from" IID ";"                          → EVENT
  "in" MID "from" "env" ";"                        → EVENT
  "out" MID "to" IID ";"                           → EVENT
  "out" MID "to" "env" ";"                         → EVENT
  "action" AID ";"                                 → EVENT
```

A syntax directed editor for BMSC is generated by the ASF+SDF Meta-environment. From the definition of the (context-free) syntax of BMSC, a scanner and a parser for BMSC is created. If the text in the editor is conform the BMSC syntax the parser generates the corresponding BMSC term. Figure 3 shows a snapshot of the syntax directed editor, containing the running example. Note, that buttons are connected to the editor for the four other tools. These buttons are created by means of the user interface language SEAL [5, 6]. When a button is selected the corresponding tool is applied to the BMSC in the editor.



```
msc example ;
 instance i ;
    out k to env ;
 endinstance ;
 instance j ;
    out l to env ;
 endinstance ;
endmsc ;
```

Figure 3. Syntax directed editor

4.3. Check

Two static requirements for Basic Message Sequence Charts are formulated in [7]. The first is that an instance may be declared only once. The second is that every message identifier occurs exactly once in an output action and once in a matching input action, or in case of a communication with the environment a message identifier occurs only once.

The *unique-instances* requirement has been specified in ASF+SDF in module Requirements by means of a function *uin* which applied to BMSCs yields a message containing both a Boolean value and a possibly empty list of error messages. The equations [1] to [6] below specify the semantics of this function applied to an MSC-BODY and its auxiliary functions.

context-free syntax

uin "(" MSC-BODY ")"	→ CHECKINFO
allinstnames "(" MSC-BODY ")"	→ IIDLIST
IID "*notin*" IIDLIST	→ CHECKINFO

equations

[1] $\mathit{uin}() \;=\; \mathit{Check}\text{: } \mathit{true}\ \mathit{Errors}\text{: } []$

[2] $\mathit{uin}(\mathit{instance}\ {<}\mathit{iid}{>};\ {<}\mathit{inst\text{-}body}{>}\ \mathit{endinstance};\ {<}\mathit{msc\text{-}body}{>}) =$
$\quad {<}\mathit{iid}{>}\ \mathit{notin}\ \mathit{allinstnames}({<}\mathit{msc\text{-}body}{>})\ \mathit{and}\ \mathit{uin}({<}\mathit{msc\text{-}body}{>})$

[3] $\mathit{allinstnames}() \;=\; []$

[4] $\mathit{allinstnames}(\mathit{instance}\ {<}\mathit{iid}{>};\ {<}\mathit{inst\text{-}body}{>}\ \mathit{endinstance};\ {<}\mathit{msc\text{-}body}{>}) =$
$\quad [{<}\mathit{iid}{>}] \cup \mathit{allinstnames}({<}\mathit{msc\text{-}body}{>})$

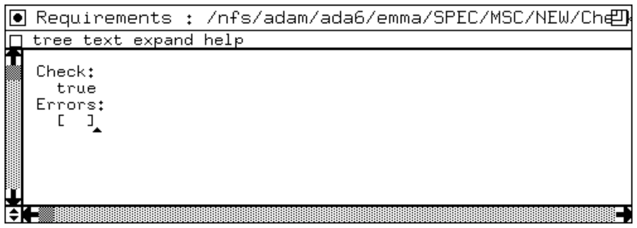
[5] ${<}\mathit{iid}{>}\ \mathit{notin}\ [{<}\mathit{iid}{>}^*_1, {<}\mathit{iid}{>}, {<}\mathit{iid}{>}^*_2] =$
$\quad \mathit{Check}\text{: } \mathit{false}\ \mathit{Errors}\text{: } [\texttt{<<duplicate␣instance␣name␣"}\ {<}\mathit{iid}{>}\ \texttt{"␣>>}]$

[6] ${<}\mathit{iid}{>}\ \mathit{notin}\ [{<}\mathit{iid}{>}^*] \;=\; \mathit{Check}\text{: } \mathit{true}\ \mathit{Errors}\text{: } []$ **otherwise**

The terms <x> indicate a variable of the sort X, <x>* and <x>$^*_i$ indicate a sequence of variables of the sort X. Equation [1] shows that the check succeeds if the MSC-BODY is empty. If the MSC-BODY is not empty, i.e. consists of an instance definition followed by an MSC-BODY, equation [2] can be applied. In this case, we check that the name of the first instance does not occur in the set of instance names of the MSC-BODY, and by a recursive call of the function *uin* the rest of the BMSC is checked. Equations [3] and [4] inductively define the auxiliary function *allinstnames*, that computes the set of instance names in an MSC-BODY. The error messages are generated by the auxiliary function *notin*. Equation [5] states that if the given instance name occurs at any position in the given set of instance names, the information from the checker consists of the Boolean value *false* and an error message. Otherwise, the checker returns the Boolean value *true* and an empty list of error messages (equation [6]).

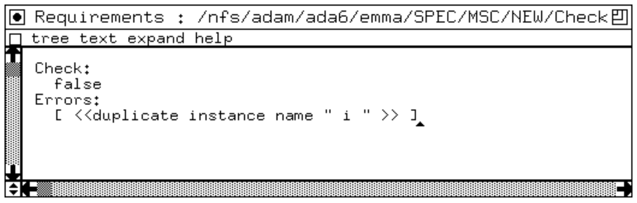
Similar functions for specifying the other requirement are given in the same module.

When the Check button in Figure 3 is selected the relevant functions are applied to the term in the editor and the generated term rewrite system is used to compute the result. A window will pop up containing this result. Figure 4 shows the result of checking the BMSC in our running example. Since this term is correct the list of error messages is empty. Next, suppose that we change the identifier `j` in the editor of Figure 3 into `i`. Selecting the check button then results in the window of Figure 5.



```
Check:
  true
Errors:
  [ ]
```

Figure 4. Result of checking a correct BMSC



```
Check:
  false
Errors:
  [ <<duplicate instance name " i " >> ]
```

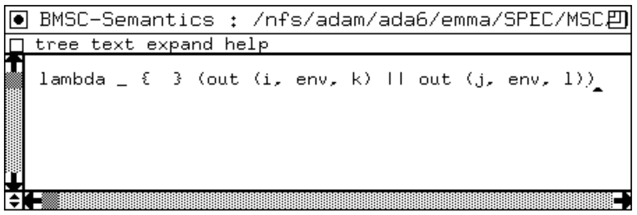
Figure 5. Result of checking a BMSC with a double occurrence of instance `i`

4.4. Semantics Calculator

In [7] Mauw and Reniers describe the translation from Basic Message Sequence Charts into the process algebra PA_{BMSC}. The definition of this translation function can be considered as an algebraic specification in a straightforward manner. We will not show the resulting specification.

The result of applying this translation to the BMSC in the editor of Figure 3 is the process algebra term $\lambda_\emptyset(out(i, env, k) \parallel out(j, env, l))$. The application of the merge operator ($\parallel$) shows that the semantics of the given BMSC is the interleaved execution of the events $out(i, env, k)$ and $out(j, env, l)$. The state operator ($\lambda_\emptyset$) in front of the expression enforces that input of message only occurs after the corresponding output. Since the example only considers communications with the environment, in this case the state operator imposes no restrictions.

Figure 6 shows the window that appears after having selected the Semantics button.



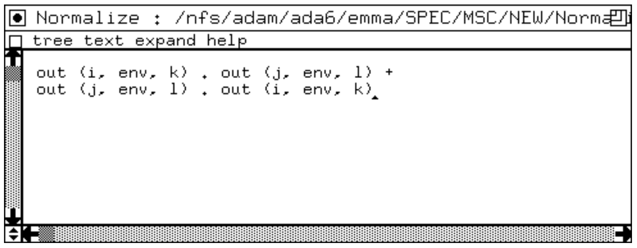
```
lambda _ {  } (out (i, env, k) || out (j, env, l))
```

Figure 6. Result of computing the semantics of a BMSC

4.5. Normalizer

The two operators in the expression of Figure 6 can be eliminated. This is called normalization. The resulting term contains the operators for sequential composition ($\cdot$) and alternative composition ($+$) only. It expresses all possible behaviors of the BMSC. Figure 7 shows the effect of pressing the normalize button in the editor of Figure 3. There are two alternative behaviors: the two events may be executed in either order.

The implementation of the normalizer is simple. The axioms defining the process algebra PA_{BMSC} can easily be interpreted as rewrite rules. Only care has to be taken not to include the axioms for commutativity, since this would give a non-terminating term rewrite system. Consequently, some extra rewrite rules had to be added, as explained in [9].



```
out (i, env, k) . out (j, env, l) +
out (j, env, l) . out (i, env, k)
```

Figure 7. Result of normalizing the semantics of a BMSC

4.6. Simulator

For large BMSCs, the expressions describing the normalized semantics as in Figure 7 become quite large and complex. Therefore, the tools offer the possibility to *walk through* the events of a BMSC in any of the admitted orders. Thus, the user can interactively simulate the behavior of a BMSC. For this purpose we used the operational semantics for BMSCs from [7]. This operational semantics defines for a given BMSC a labeled transition system. The transitions correspond with the events of the BMSC.

For the running example, represented by the term $\lambda_\emptyset(out(i,env,k) \parallel out(j,env,l))$, the set of transitions is $\{\overset{out(i,env,k)}{\rightarrow}\lambda_\emptyset(out(j,env,l)), \overset{out(j,env,l)}{\rightarrow}\lambda_\emptyset(out(i,env,k))\}$. This means that executing event $out(i,env,k)$ results in the BMSC represented by $\lambda_\emptyset(out(j,env,l))$ and that execution of the alternative action $out(j,env,l)$ results in $\lambda_\emptyset(out(i,env,k))$. Likewise, the transition set of $\lambda_\emptyset(out(j,env,l))$ is $\{\overset{out(j,env,l)}{\rightarrow}\varepsilon\}$ and of $\lambda_\emptyset(out(i,env,k))$ is $\{\overset{out(i,env,k)}{\rightarrow}\varepsilon\}$. The symbol ε means that the BMSC has terminated.

In the specification below, parts of the specification of the function *transitions* are shown. This function calculates the set of transitions for a given PA_{BMSC} expression.

Equation [TR1] states that a terminated BMSC, represented by ε has no transitions. Equation [TR2] defines the transitions for a single event a. In this case there is one single transition, namely execute a and terminate. Equation [TR5] defines the transitions of the parallel composition of two expressions. It makes use of two auxiliary functions, the union ($\cup$) as defined in equation [T1] and the merge of a transition set and a process expression (also denoted by the operator $\parallel$) as defined in the equations [T4] through [T7]. The definition in equation [TR5] states that in order to calculate the transitions of $x \parallel y$, we first take the transitions of x and the transitions of y. The result is not simply the union of these two transition sets, since, if x executes an action, y still has to be placed in parallel with the resulting process. This is expressed in equation [T5]. The symmetric case is expressed in [T7]. Finally [TR7] states that for the transition set of an expression starting with the state operator λ_M, we need to calculate the transition list of its argument and filter out the sequences in which an input occurs before the corresponding output. The definition of this *filter* function is not included in this document.

context-free syntax

"—" ATOM "→" PROCESS	→ TRANSITION
"*transitions*" "(" PROCESS ")"	→ TRANSITIONLIST
TRANSITIONLIST "∪" TRANSITIONLIST	→ TRANSITIONLIST
TRANSITIONLIST "‖" PROCESS	→ TRANSITIONLIST
PROCESS "‖" TRANSITIONLIST	→ TRANSITIONLIST

equations

[TR1] $transitions(\varepsilon) = []$

[TR2] $transitions(a) = [— a \rightarrow \varepsilon]$

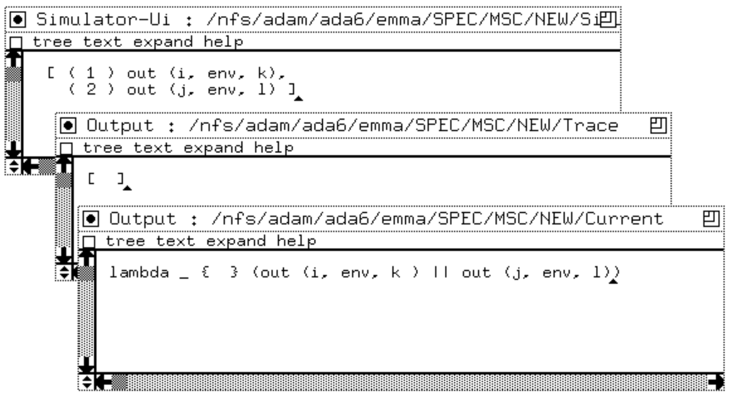
[TR3] $transitions(x + y) = transitions(x) \cup transitions(y)$

[TR5] $transitions(x \parallel y) = transitions(x) \parallel y \cup x \parallel transitions(y)$

[TR7] $transitions(\lambda_M(x)) = filter_M(transitions(x))$

$$
\begin{array}{llcl}
\text{[T1]} & [tl_1] \cup [tl_2] & = & [tl_1,\, tl_2] \\
\text{[T4]} & [] \parallel y & = & [] \\
\text{[T5]} & [\text{—}\, a \rightarrow x,\, tl] \parallel y & = & [\text{—}\, a \rightarrow x \parallel y] \cup [tl] \parallel y \\
\text{[T6]} & y \parallel [] & = & [] \\
\text{[T7]} & y \parallel [\text{—}\, a \rightarrow x,\, tl] & = & [\text{—}\, a \rightarrow y \parallel x] \cup y \parallel [tl]
\end{array}
$$

If we select the simulate button in Figure 3, we obtain three windows from Figure 8. The upper window is the selection window, in which all possible continuations of the BMSC are displayed. Either event may occur. The middle window displays the list of all events executed until now. This list is empty. The lower window shows the process algebra representation of the BMSC under consideration.



```
Simulator-Ui : /nfs/adam/ada6/emma/SPEC/MSC/NEW/Si
tree text expand help
 [ ( 1 ) out (i, env, k),
   ( 2 ) out (j, env, l) ]

Output : /nfs/adam/ada6/emma/SPEC/MSC/NEW/Trace
tree text expand help
 [  ]

Output : /nfs/adam/ada6/emma/SPEC/MSC/NEW/Current
tree text expand help
 lambda _ {  } (out (i, env, k ) || out (j, env, l))
```

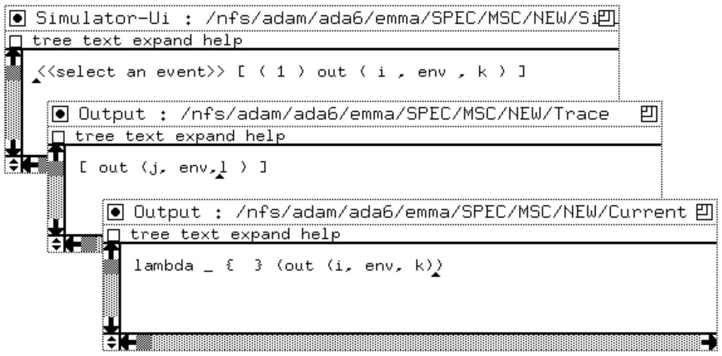
Figure 8. Starting the simulator

If the user selects the second event, all windows will be updated (see Figure 9). The selection window now contains the one remaining event. The trace window contains the chosen event and the current window contains the process algebra representation of the BMSC resulting after having executed the second event.

If we select the remaining event, we obtain the situation from Figure 10. It shows that execution of the BMSC is finished.

5. CONCLUSIONS

The main objective of this case study was to provide evidence that the formal semantics definition of Basic Message Sequence Charts can be used to derive tools in a straightforward way. The translation of the process algebra and the definitions of the semantics functions into algebraic specifications is easy, but care has to be taken when implementing them as rewrite rules. In order to obtain a nice term rewriting system, some rules have to be deleted, added or modified.

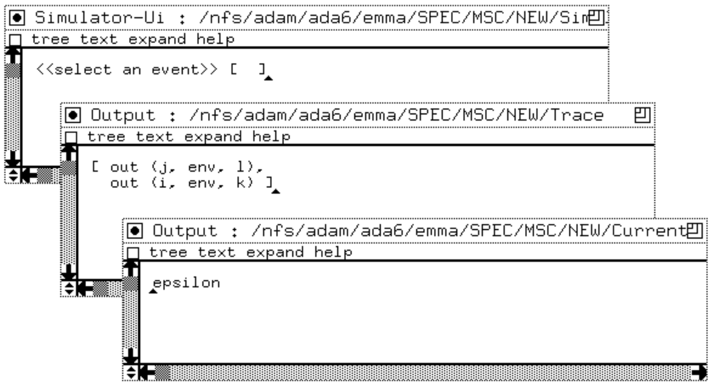


```
Simulator-Ui : /nfs/adam/ada6/emma/SPEC/MSC/NEW/Si
tree text expand help
<<select an event>> [ ( 1 ) out ( i , env , k ) ]

Output : /nfs/adam/ada6/emma/SPEC/MSC/NEW/Trace
tree text expand help
[ out (j, env, l ) ]

Output : /nfs/adam/ada6/emma/SPEC/MSC/NEW/Current
tree text expand help
lambda _ {  } (out (i, env, k))
```

Figure 9. Result after selecting event number (2)



```
Simulator-Ui : /nfs/adam/ada6/emma/SPEC/MSC/NEW/Si
tree text expand help
<<select an event>> [  ]

Output : /nfs/adam/ada6/emma/SPEC/MSC/NEW/Trace
tree text expand help
[ out (j, env, l),
  out (i, env, k) ]

Output : /nfs/adam/ada6/emma/SPEC/MSC/NEW/Current
tree text expand help
epsilon
```

Figure 10. Result after selecting the final event

We also specified a simulator tool based on the operational semantics for Message Sequence Charts. The definition of this simulator could serve as a formal specification of such a tool. Finally, we formalized the static requirements.

By using the ASF+SDF Meta-environment we derived (prototypes of) tools for BMSCs. It proved to be a flexible programming environment whose capabilities of incremental development helped in easy prototyping. The possibilities of defining a user interface on top of the term rewrite engine enables the generation of demonstrable and usable tools.

The possibility of prototyping makes it easy to explore new versions of MSC in standardization work and to make dialects of MSC for internal use. Changes to the syntax only require minor modifications to the specification. Changes with respect to the semantics and new language features require modification of the formal semantics and a corresponding modification of the specification.

A disadvantage of the term rewriting paradigm in ASF+SDF is that, sometimes, easy to understand algebraic rules have to be transformed into a more implementation directed form.

The transformation into a TRS sometimes implies that decisions on implementation details are made, which were not expressed in the algebraic specification. For example, if we aim at complete TRSs (i.e. TRSs which are confluent and terminating, see [4]), we need to decide on the implementation of commutative operators and the implementation of sets by ordered lists. Therefore, a completely automatic implementation of an algebraically specified semantics by means of a TRS is not always feasible.

The techniques described in this paper can be easily extended to the general setting of Message Sequence Charts. Due to the modular description, the framework for Basic Message Sequence Charts can be reused almost completely.

Starting from the algebraic specifications, there are two ways to proceed with the development of real tools. The obvious way is to manually translate the functionality expressed in the equations into efficient code. The specification can then be used for validation purposes. The second way is to (semi-) automatically generate efficient programs. This is topic of ongoing research ([2]).

REFERENCES

1. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications, vol. I, Equations and Initial Semantics.* Springer-Verlag, 1985.
2. J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993.
3. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.
4. J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 1–116. Oxford University Press, 1992.
5. J.W.C. Koorn. Connecting semantic tools to a syntax-directed user-interface. Report P9222, Programming Research Group, University of Amsterdam, 1992.
6. J.W.C. Koorn. *Generating uniform user-interfaces for interactive programming environments.* PhD thesis, University of Amsterdam, 1994. ILLC Dissertation series 1994-2.
7. S. Mauw and M.A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The computer journal*, 37(4):269–277, 1994.
8. S. Mauw and M.A. Reniers. An algebraic semantics of Message Sequence Charts. Experts Meeting SG10, Turin, TD9009, ITU-TS, 1994. Report CSN94/23, Eindhoven University of Technology, 1994.
9. S. Mauw and E.A. van der Meulen. Generating tools for Message Sequence Charts. Study Group Meeting SG10, Geneva, TD60, ITU-TS, 1994.
10. M.A. Reniers. Syntax requirements of Message Sequence Charts. Study Group Meeting SG10, Geneva, TD59, ITU-TS, 1994.
11. Z.120 (1993). *Message sequence chart (MSC).* ITU-T, 1994.
12. Z.120 B (1995). *Message sequence chart algebraic semantics.* ITU-T, Publ. sched. 1995.