Specification of
Tools for Message Sequence Charts

S. Mauw[1] E. A. van der Meulen[2]

[1]Dept. of Mathematics and Computing Science, Eindhoven
University of Technology, P.O. Box 513, 5600 MB Eindhoven,
The Netherlands, email: sjouke@win.tue.nl
[2]Dept. of Mathematics and Computing Science, University of
Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The
Netherlands, email: emma@fwi.uva.nl

Abstract. 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. In this paper we present the complete specification of the tools.

1 Introduction

Message Sequence Charts (MSCs) are a graphical method for the description of the interaction between system components [IT94]. Due to the recent formalization [MR94a, MR94b, IT95] 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, for which algebraic operators are considered) 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 is to be considered 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 [MR94a] the semantics of BMSCs is defined by a translation into process algebra. This translation is defined by means of equations and the axioms defining process algebra are also equations. Therefore, the obvious way of implementing the semantics of MSCs is by using algebraic specifications [EM85].

Proceedings of ASF+SDF95. A workshop on Generating Tools from Algebraic Specifications. May 11 & 12, 1995, CWI, Amsterdam, M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman & E. Visser (eds.) Technical Report P9504, Programming Research Group, University of Amsterdam

We used the ASF+SDF Meta-environment [Kli93] 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.

The implementation consists of three parts. The first part consists of an implementation of the static requirements for BMSCs expressed informally in [IT94] and formalized in [Ren94]. The second part is the translation of BMSCs into process algebra expressions. This is based on the definition of the semantic functions in [MR94a]. The third 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 [MR94a]. In fact the description of the simulator can be regarded as a formal specification of a simulation tool.

Figure 1 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 [MR94a]) and BPA is the sub-language of PA_{BMSC} that only contains the *normalized* PA_{BMSC} expressions. The generated

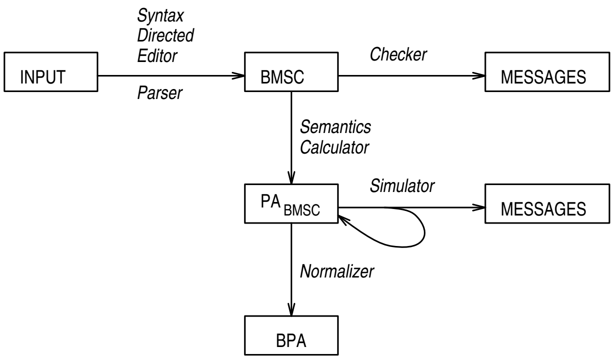

Figure 1: Structure of the tools

tools are considered as transformation tools, described by algebraic specifications. 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 (or semantics function) 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.

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 the specification of the static requirements. In Section 5 we define the process algebra and its specification in ASF+SDF. The Semantics Calculator and Normalizer are defined in Section 6 and the simulator in Section 7.

Although this paper covers the complete semantics of BMSCs, it is not intended as a self-contained explanation of these semantics. Refer to [MR94a] 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 [Kli93] 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 [Koo92, Koo94].

2.1 Algebraic Specifications

An algebraic specification consists of a signature, a set of variables and a set of equations. The signature describes a number of sorts and functions over these sorts. Using these functions and the variables one can construct terms. The equations define equalities between these terms.

Consider, e.g., an algebraic specification of the data-structure Booleans. The signature defines one sort BOOL, two constants of sort BOOL, namely true $\rightarrow$ BOOL and false $\rightarrow$ BOOL, one unary function not: BOOL $\rightarrow$ BOOL, and two binary functions and: BOOL # BOOL $\rightarrow$ BOOL and or: BOOL # BOOL $\rightarrow$ BOOL. Let's assume that a variable *Bool* over sort BOOL is declared. From the signature and the variables terms like true, false, *Bool*, not(true), not(*Bool*), and(true,false), and(*Bool*,true), etc. can be derived. The semantics of the functions "*not*", "*and*" and "*or*" are defined by equations. We have, for instance, *not(true) = false*. A complete specification of this data-structure in ASF+SDF notation will be given in section 2.2.

The most common strategy for implementing algebraic specifications is via term rewrite systems (TRSs). An algebraic specification can be transformed into a TRS by interpreting the equations as rewrite rules from left to right. An algebraic specification of the Booleans can thus be used to compute the value of a function by rewriting a term to its normal form, *true* or *false*.

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 [Klo92]), 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.

2.2 The formalism ASF+SDF

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.

An ASF+SDF specification consists of a sequence of modules. Each module may contain

Imports of other modules.

Sort declarations defining the sorts of a signature.

Lexical syntax defining layout conventions and lexical tokens.

Context-free syntax defining the concrete syntactic forms of the functions in the signature.

Variables to be used in equations. In general, each variable declarations has the form of a regular expression and defines the class of all variables whose name is described by the regular expression.

Equations Conditional equations define the meaning of the functions defined in the contex-free syntax.

Sort declarations, lexical functions, context-free syntax and variables are either part of an **export** section in a module or can be declared as **hidden**. When a module imports another module the export sections in the syntax definition as well as the equations of the imported module are visible in the importing module. Hidden sorts, functions or variables cannot be referred to by the importing module.

Example A simple example specification is given below consisting of the modules Layout and Booleans. Module Booleans defines the Boolean values and operators. In order to avoid ambiguities in parsing terms attributes {**left**} or {**right**} can be added to function declarations, defining a function to be left associative or right associative. For instance, the attribute {**left**} in BOOL "&" BOOL → BOOL indicates that the term *true & false & true* should be parsed as *(true & false) & true* rather than *true & (false & true)*. Moreover, the specifier can indicate by means of priorities rules how terms should be parsed. For instance, the priority rule in module Booleans states that the operator "*not*" binds stronger than the operator "*&*" which in turn binds stronger than the operator "|". Hence, the term *not true & false* will be parsed as *(not true) & false* rather than *not (true & false)*. Likewise, the term *true & false* | *true* should be parsed as *(true & false)* | *true*. The function with the attribute {**bracket**} is added only for grouping and disambiguation, it is not included in the abstract syntax.

The variable declaration "*Bool*[1-9']* → BOOL" declares an infinite number of variables of sort BOOL. All variables start with the letters *Bool* followed by zero or more (*) occurences of numbers and quotes [1-9']. In the equations we use only one variable.

Module Booleans imports module Layout which consists of two lexical functions for the predefined sort LAYOUT. The upper one defining that spaces (\␣), tabs (\t) and newlines (\n) are layout, and thus separate tokens. The lower function defines a comment convention by stating that any string starting with two percentage signs ("%%") followed by any number of characters other than a newline (~[\n]), and concluded by a newline ([\n]) is to be considered layout as well.

2.2.1 Booleans

imports Layout$^{2.2.2}$
exports
  sorts BOOL
  context-free syntax
    true → BOOL
    false → BOOL
    BOOL "|" BOOL → BOOL {**left**}
    BOOL "&" BOOL → BOOL {**left**}
    "*not*" BOOL → BOOL
    "(" BOOL ")" → BOOL {**bracket**}
hiddens
  variables
    Bool [*1-9'*]* → BOOL
  priorities
    BOOL "|"BOOL → BOOL < BOOL "&"BOOL → BOOL <
    "*not*"BOOL → BOOL
equations

[1] $\quad true \mid Bool = true$

[2] $\quad false \mid Bool = Bool$

[3] $\quad true \,\&\, Bool = Bool$

[4] $\quad false \,\&\, Bool = false$

[5] $\quad not\ true = false$

[6] $\quad not\ false = true$

2.2.2 Layout

exports
  lexical syntax
    [␣\t\n] → LAYOUT
    "%%"~[\n]*[\n] → LAYOUT

Other features Other features of ASF+SDF will be explained when necessary. In particular, the use of default equations and the description of list sorts is explained in section 4.1. More advanced priority rules are referred to in section 5.2.

3 Message Sequence Charts

Message Sequence Charts provide a graphical method for the description of the communication behavior of system components. The ITU-TS (the Telecommunication Standardization Section of the International Telecommunication Union, the former CCITT) maintains recommendation Z.120 [IT94] 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 [MR94b]. This proposal has been accepted for standardization by the ITU [IT95].

3.1 Basic Message Sequence Charts

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 [MR94a].

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. The Basic Message Sequence Chart in Figure 2 defines the communication behavior between instances a and b. This will be the running example in the remainder of this paper. 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. In Figure 2 we consider message m from instance a to instance b and message k which is sent from a to the environment.. The behavior of the environment is not specified. For instance b we also define a local action p.

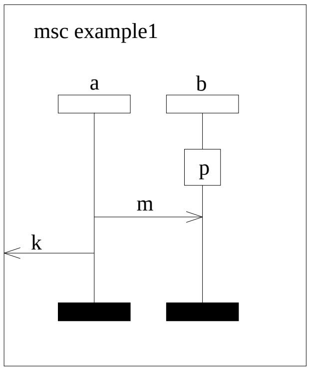

Figure 2: Example Basic Message Sequence Chart

Although the activities along one 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. In Figure 2 this implies for example that message m is received by b only after it has been sent by a. Furthermore, it is required that action p is executed before message m is received, and that message m is sent before message k. For the sending of k and the reception of m no ordering is specified.

3.2 BMSC syntax

The grammar defining the syntax of textual Basic Message Sequence Charts as presented in [MR94a] is given in Table 1. The nonterminals `<mscid>`, `<iid>`, `<mid>` and `<aid>` represent identifiers. The symbol `<>` denotes the empty string. The following identifiers are reserved keywords: `action`, `endinstance`, `endmsc`, `env`, `from`, `in`, `instance`, `msc`, `out` and `to`. The language generated by a non-terminal $< x >$ is denoted by $\mathcal{L}(< x >)$.

The Basic Message Sequence Chart of Figure 2 has the following textual representation.

```
msc example1;
  instance a;
    out m to b;
    out k to env;
  endinstance;
```

Table 1: The BNF grammar of Basic Message Sequence Charts

```
<msc>         ::=  msc <mscid>;
                   <msc body> endmsc;
<msc body>    ::=  <> |
                   <inst def> <msc body>
<inst def>    ::=  instance <iid>;
                   <inst body> endinstance;
<inst body>   ::=  <> |
                   <event> <inst body>
<event>       ::=  in <mid> from <iid>; |
                   in <mid> from env;   |
                   out <mid> to <iid>;  |
                   out <mid> to env;    |
                   action <aid>;
```

```
    instance b;
      action p;
      in m from a;
    endinstance;
  endmsc;
```

The context free syntax for BMSCs is expressed in ASF+SDF in the following specification. It is easily derived from the BNF grammar.

The first module below defines the Identifiers, which consist of a character followed by characters and digits. It states that there is an (invisible) mapping from elements of the sort ID to the sorts MSCID, IID, MID and AID. The second module defines the syntax of BMSCs.

3.2.1 Identifiers

imports Layout[2.2.2]
exports
  sorts ID MSCID IID MID AID
  lexical syntax
    [*a-z*][*a-z0-9*]∗ → ID
  context-free syntax
    ID → MSCID
    ID → IID
    ID → MID
    ID → AID

3.2.2 BMSC-Syntax

imports Identifiers[3.2.1]
exports
  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

3.3 Example

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 of figure 2. 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 [Koo92, Koo94]. When a button is selected the corresponding tool is applied to the BMSC in the editor.

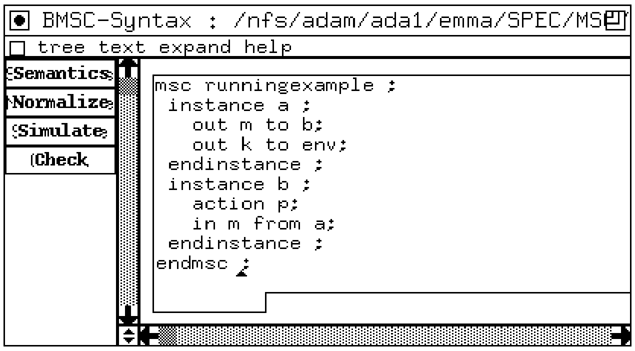

Figure 3: Syntax directed editor

4 Requirements

Two static requirements for Basic Message Sequence Charts are formulated in [MR94a]. 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. In addition we will check whether all instances that are referred to in messages have been declared. Module Requirements imports two auxiliary modules: Xevents and Messages.

4.1 Xevents

The specification of the requirements is facilitated when an MSC is represented by a list of its events. We therefore introduce the sort XEVENT. This is an event extended with the name of the instance it belongs to. E.g., when instance *a* sends a message *m* to instance *b*, *out m to b*, the corresponding extended event is *out m from a to b*.

The declaration “[” {XEVENT “;”}∗ “]” → XEVENTLIST declares a list of zero or more (∗) XEVENTs separated by semicolums (“;”) and surrounded by square brackets (“[” “]”). The variables declaration “<”*xevent*“>” “∗”[0-9]∗ → {XEVENT ”;”}∗ declares variables over such lists of XEVENTs. Any list of XEVENTs matches variables like <*xevent*>∗, <*xevent*>∗1, <*xevent*>∗13 etc.. Equation 6 defines the union of two lists of XEVENTs as a list containing the events of the first list followed by those of the second list.

The functions *xevent*, $xevent_{body}$, $xevent_{list}$ and *extend* are introduced to derive a list of extended events from an MSC. Equations 1 through 11 specify the behavior of these functions.

The function *message-name* returns an MIDLIST, a list containing the message identifier of an XEVENT. Equations 12 through 15 specify the application of this function to all xevents describing input or output actions. Equation 16 is a so called *default equation*, marked with the keyword **otherwise**. Such an equation is only used for rewriting a given term if no other equation applies. Hence, equation 16 states that the *message-name* of any xevent not describing an input or output action equals the empty list [].

Equations 17 and 18 specify that two xevents match, if one of them represents the sending of a message to an instance and the other one is the xevent for the reception of that message. The default equation 19 states that applying the predicate *matching-xevents* to any other pair of xevents equals false.

4.1.1 Xevents

```
imports BMSC-Syntax[3.2.2] Booleans[2.2.1]
exports
  sorts XEVENTLIST XEVENT MIDLIST
  context-free syntax
    "[" {XEVENT ";"}* "]"                    → XEVENTLIST
    xevents(MSC)                             → XEVENTLIST
    xevents "_" body "(" MSC-BODY ")"        → XEVENTLIST
    xevents "_" inst "(" IID "," INST-BODY ")" → XEVENTLIST
    XEVENTLIST "∪" XEVENTLIST                → XEVENTLIST {left}

    in MID from IID to IID                   → XEVENT
    in MID from env to IID                   → XEVENT
    out MID from IID to IID                  → XEVENT
    out MID from IID to env                  → XEVENT
    action AID by IID                        → XEVENT
    extend(IID, EVENT)                       → XEVENT

    message-name(XEVENT)                     → MIDLIST
    "[" {MID ","}* "]"                       → MIDLIST
    matching-xevents(XEVENT, XEVENT)         → BOOL
hiddens
  variables
    "<"msc-body">"            → MSC-BODY
    "<"inst-def">""*"[0-9]*   → {INST-DEF ";"}*
    "<"inst-body">"           → INST-BODY
    "<"event">"[0-9]*         → EVENT
    "<"xevent">""*"[0-9]*     → {XEVENT ";"}*
    "<"xevent">"[0-9]*        → XEVENT
    "<"mscid">"               → MSCID
    "<"iid">"[0-9]*           → IID
    "<"mid">"                 → MID
    "<"aid">"                 → AID
equations
```

[1] $xevents(msc\ <mscid>;\ <msc\text{-}body>\ endmsc;) = xevents_{body}\ (<msc\text{-}body>)$

[2] $xevents_{body}\ () = []$

[3] $$\mathit{xevents}_{body}\ (\mathit{instance}\ <\!iid\!>;\ <\!\mathit{inst\text{-}body}\!>\ \mathit{endinstance};\ <\!\mathit{msc\text{-}body}\!>) =$$
$$\mathit{xevents}_{inst}\ (<\!iid\!>,\ <\!\mathit{inst\text{-}body}\!>) \cup \mathit{xevents}_{body}\ (<\!\mathit{msc\text{-}body}\!>)$$

[4] $$\mathit{xevents}_{inst}\ (<\!iid\!>,\ ) = []$$

[5] $$\mathit{xevents}_{inst}\ (<\!iid\!>,\ <\!event\!>;\ <\!\mathit{inst\text{-}body}\!>) =$$
$$[\mathit{extend}(<\!iid\!>,\ <\!event\!>)] \cup \mathit{xevents}_{inst}\ (<\!iid\!>,\ <\!\mathit{inst\text{-}body}\!>)$$

[6] $$[<\!xevent\!>^*_1] \cup [<\!xevent\!>^*_2] = [<\!xevent\!>^*_1;\ <\!xevent\!>^*_2]$$

[7] $\mathit{extend}(<\!iid\!>_1,\ \mathit{in}\ <\!mid\!>\ \mathit{from}\ <\!iid\!>_2) = \mathit{in}\ <\!mid\!>\ \mathit{from}\ <\!iid\!>_2\ \mathit{to}\ <\!iid\!>_1$

[8] $\mathit{extend}(<\!iid\!>_1,\ \mathit{in}\ <\!mid\!>\ \mathit{from\ env}) = \mathit{in}\ <\!mid\!>\ \mathit{from\ env\ to}\ <\!iid\!>_1$

[9] $\mathit{extend}(<\!iid\!>_1,\ \mathit{out}\ <\!mid\!>\ \mathit{to}\ <\!iid\!>_2) = \mathit{out}\ <\!mid\!>\ \mathit{from}\ <\!iid\!>_1\ \mathit{to}\ <\!iid\!>_2$

[10] $\mathit{extend}(<\!iid\!>_1,\ \mathit{out}\ <\!mid\!>\ \mathit{to\ env}) = \mathit{out}\ <\!mid\!>\ \mathit{from}\ <\!iid\!>_1\ \mathit{to\ env}$

[11] $\mathit{extend}(<\!iid\!>_1,\ \mathit{action}\ <\!aid\!>) = \mathit{action}\ <\!aid\!>\ \mathit{by}\ <\!iid\!>_1$

[12] $\mathit{message\text{-}name}(\mathit{in}\ <\!mid\!>\ \mathit{from}\ <\!iid\!>_1\ \mathit{to}\ <\!iid\!>_2) = [<\!mid\!>]$

[13] $\mathit{message\text{-}name}(\mathit{in}\ <\!mid\!>\ \mathit{from\ env\ to}\ <\!iid\!>_2) = [<\!mid\!>]$

[14] $\mathit{message\text{-}name}(\mathit{out}\ <\!mid\!>\ \mathit{from}\ <\!iid\!>_1\ \mathit{to}\ <\!iid\!>_2) = [<\!mid\!>]$

[15] $\mathit{message\text{-}name}(\mathit{out}\ <\!mid\!>\ \mathit{from}\ <\!iid\!>_1\ \mathit{to\ env}) = [<\!mid\!>]$

[16] $\mathit{message\text{-}name}(<\!xevent\!>) = []$ **otherwise**

[17] $$\begin{aligned}\mathit{matching\text{-}xevents}(&\mathit{out}\ <\!mid\!>\ \mathit{from}\ <\!iid\!>_1\ \mathit{to}\ <\!iid\!>_2,\\ &\mathit{in}\ <\!mid\!>\ \mathit{from}\ <\!iid\!>_1\ \mathit{to}\ <\!iid\!>_2) = \mathit{true}\end{aligned}$$

[18] $$\begin{aligned}\mathit{matching\text{-}xevents}(&\mathit{in}\ <\!mid\!>\ \mathit{from}\ <\!iid\!>_1\ \mathit{to}\ <\!iid\!>_2,\\ &\mathit{out}\ <\!mid\!>\ \mathit{from}\ <\!iid\!>_1\ \mathit{to}\ <\!iid\!>_2) = \mathit{true}\end{aligned}$$

[19] $\mathit{matching\text{-}xevents}(<\!xevent\!>_1,\ <\!xevent\!>_2) = \mathit{false}$ **otherwise**

4.2 Messages

Module Messages defines the general syntax of the error messages used in module Requirements. Note, that these are not messages in the sense of MSC, but messages to inform the user of the system. Four kinds of messages are distinguished in the lexical syntax. (1) Opening brackets ≪ followed by a string without the symbol > and concluded by ≫, (2) Opening brackets ≪ followed by a string without quotes or >, and concluded by quotes. (3) Quotes followed by a string without > and concluded by ≫. (4) Quotes followed by a string without quotes or > and concluded by quotes.

This syntax allows for composed messages of sort MESSAGELIST like `[≪ in instance "a" an error has been found ≫]`.

The operator ∪ specifies the union of lists of messages.

4.2.1 Messages

imports Layout$^{2.2.2}$
exports
  sorts MESSAGE MESSAGELIST
  lexical syntax
    “≪”~[>]*“≫” → MESSAGE
    “≪”~[\” >]*“\”” → MESSAGE
    “\””~[>]*“≫” → MESSAGE
    “\””~[\” >]*“\”” → MESSAGE
  context-free syntax
    “[” MESSAGE* “]” → MESSAGELIST
    MESSAGELIST “∪” MESSAGELIST → MESSAGELIST {**left**}
hiddens
  variables
    *m[0-9]**“*” → MESSAGE*
equations

[1] $[m_1^*] \cup [m_2^*] \ = \ [m_1^* \ m_2^*]$

4.3 Requirements specification

The main function in module Requirements is the function *check* for MSCs. The result of checking an MSC is CHECKINFO, composed of a boolean value and a possible empty list of error messages. Two CHECKINFOs can be added by means of the function CHECKINFO *and* CHECKINFO. Equation 1 specifies how this is done. The sort MESSAGE is extended so that identifiers and xevents can be referred to in error messages.

As mentioned before, three requirements will be checked. Equation 2 states that the function *check* invokes the functions *unique-instance-names*, *inst-declared* and *check-message-names*. Equations 3 to 5 specify the semantics of the functions *unique-instances* and uin_{body}. According to equation 4 an empty MSC-BODY is correct, i.e. all instance names are unique. If the MSC-BODY consists of an instance definition followed by an MSC-BODY, we check that the name of the first instance does not occur in the set of declared instance names of the remaining MSC-BODY. By a recursive call of the function uin_{body} the rest of the BMSC is checked (equation 5). The error messages are generated by the auxiliary function *notin*. Equation 6 states that if a given instance name occurs at any position in a list of instance names, the Boolean value *false* and an error message are returned. Otherwise, the Boolean value *true* and an empty list of error messages are returned (equation 7). Equations 8, 9 and 10 inductively define the auxiliary function *declared-instnames*, which computes the set of instance names in an MSC-BODY.

The function *inst-declared* checks whether instances referred to by input and output actions have been declared (equation 11). The auxiliary functions *refinsts* select the names of all instances referred to by input or output actions of an MSC (equations 12 unto 17). The function *included-in* checks if all IIDs in a list do occur in another list of IIDs. If not, an error-message is generated (equations 18 – 20).

Application of the function *check-message-names* to an MSC invokes the application of the functions *unique-message-names* and *check-nonmatching-messages* to the corresponding list of xevents. Equation 23 specifies that *unique-message-names* selects all pairs of xevents that mistakenly have the same message name. If such a pair is present, an error message is generated and the function is recursively applied to the rest of the list. Lists without such pairs are correct according to equation 24.

Equation 25 specifies that *check-nonmatching-messages* removes all matching pairs of input output actions from a list of xevents. If no such pairs are left in the list the function *aux-nonmatching-messages* is invoked (equation 26). Applying *aux-nonmatching-messages* to an empty list yields the boolean value *true* and an empty list of error messages. If the first xevent in the list represents receiving a message from an instance or sending a message to an instance, no matching action will be present in the rest of the list. Therefore, an error message is generated and the rest of the list is checked. If the first xevent is any other action, it is correct and the rest of the list is checked (equations 27 – 30).

4.3.1 Requirements

```
imports Xevents^4.1.1 Messages^4.2.1
exports
  sorts CHECKINFO IIDLIST
  context-free syntax
    check(MSC)                                    → CHECKINFO
    "Check:" BOOL "Errors:" MESSAGELIST           → CHECKINFO
    CHECKINFO and CHECKINFO                       → CHECKINFO  {left}

    MESSAGE IID MESSAGE                           → MESSAGE
    MESSAGE MID MESSAGE                           → MESSAGE
    MESSAGE XEVENT MESSAGE                        → MESSAGE

    unique-instance-names "(" MSC ")"             → CHECKINFO
    uin "_" body "(" MSC-BODY ")"                 → CHECKINFO
    IID notin IIDLIST                             → CHECKINFO
    declared-instnames "(" MSC ")"                → IIDLIST
    declared-instnames "(" MSC-BODY ")"           → IIDLIST

    inst-declared(MSC)                            → CHECKINFO
    refinsts(MSC)                                 → IIDLIST
    refinsts(XEVENTLIST)                          → IIDLIST
    refinsts(XEVENT)                              → IIDLIST
    IIDLIST includedin IIDLIST                    → CHECKINFO

    "[" {IID ","}* "]"                            → IIDLIST
    IIDLIST "∪" IIDLIST                           → IIDLIST    {left}

    check-message-names(MSC)                      → CHECKINFO
    unique-message-names(XEVENTLIST)              → CHECKINFO
    check-nonmatching-messages(XEVENTLIST)        → CHECKINFO
    aux-nonmatching-messages(XEVENTLIST)          → CHECKINFO
hiddens
  variables
    b[0-9]*                      → BOOL
    ml[0-9]*                     → MESSAGELIST
    "<"msc">"                    → MSC
    "<"msc-body">"               → MSC-BODY
    "<"inst-def">"               → INST-DEF
    "<"inst-body">"              → INST-BODY
    "<"event">" "*"[0-9]*        → {EVENT ";"}*
    "<"event">"                  → EVENT
    xel                          → XEVENTLIST
    "<"xevent">" "*"[0-9]*       → {XEVENT ";"}*
    "<"xevent">"[0-9]*           → XEVENT
    "<"mscid">"                  → MSCID
    "<"iid">" "*"[0-9]*          → {IID ","}*
    i                            → IID
```

“<” *iid*“>” *[0-9]*∗ → IID
“<” *mid*“>” → MID
“<” *aid*“>” → AID

equations

[1] *Check*: b_1 *Errors*: ml_1 *and Check*: b_2 *Errors*: ml_2 = *Check*: b_1 & b_2 *Errors*: $ml_1 \cup ml_2$

[2] *check*(<*msc*>) = *unique-instance-names*(<*msc*>)
and inst-declared(<*msc*>)
and check-message-names(<*msc*>)

[3] *unique-instance-names*(*msc* <*mscid*>; <*msc-body*> *endmsc*;) = uin_{body} (<*msc-body*>)

[4] uin_{body} () = *Check*: *true Errors*: []

[5] uin_{body} (*instance* <*iid*>; <*inst-body*> *endinstance*; <*msc-body*>) =
<*iid*> *notin declared-instnames*(<*msc-body*>) *and* uin_{body} (<*msc-body*>)

[6] <*iid*> *notin* [<*iid*>$^*_1$, <*iid*>, <*iid*>$^*_2$] =
Check: *false Errors*: [<<duplicate␣instance␣name␣" <*iid*> "␣>>]

[7] <*iid*> *notin* [<*iid*>*] = *Check*: *true Errors*: [] **otherwise**

[8] *declared-instnames*() = []

[9] *declared-instnames*(*instance* <*iid*>; <*inst-body*> *endinstance*; <*msc-body*>) =
[<*iid*>] ∪ *declared-instnames*(<*msc-body*>)

[10] *declared-instnames*(*msc* <*mscid*>; <*msc-body*> *endmsc*;) =
declared-instnames(<*msc-body*>)

[11] *inst-declared*(<*msc*>) =
refinsts(<*msc*>) *includedin declared-instnames*(<*msc*>)

[12] *refinsts*(<*msc*>) = *refinsts*(*xevents*(<*msc*>))

[13] *refinsts*([]) = []

[14] *refinsts*([<*xevent*>; <*xevent*>*]) =
refinsts(<*xevent*>) ∪ *refinsts*([<*xevent*>*])

[15] *refinsts*(*in* <*mid*> *from* <*iid*>$_1$ *to* <*iid*>$_2$) = [<*iid*>$_1$]

[16] *refinsts*(*out* <*mid*> *from* <*iid*>$_1$ *to* <*iid*>$_2$) = [<*iid*>$_2$]

[17] *refinsts*(<*xevent*>) = [] **otherwise**

[18] [] *includedin* [<*iid*>*] = *Check*: *true Errors*: []

```
[19] [<iid>*1, <iid>, <iid>*2] includedin [<iid>*3, <iid>, <iid>*4] =
      [<iid>*1, <iid>*2] includedin [<iid>*3, <iid>, <iid>*4]
[20] [<iid>, <iid>*1] includedin [<iid>*2] =
      Check: false Errors: [<<instance␣" <iid> "␣used␣but␣not␣declared>>]   otherwise
      and [<iid>*1] includedin [<iid>*2]

[21] [<iid>*1] ∪ [<iid>*2]  =  [<iid>*1, <iid>*2]

[22] check-message-names(<msc>) = unique-message-names(xel)
                                  and check-nonmatching-messages(xel)
                                  when
                                    xel = xevents(<msc>)

     unique-message-names([<xevent>*1; <xevent>2;
                           <xevent>*3; <xevent>4;
[23]                       <xevent>*5])
                                  = Check: false
                                    Errors: [<<duplicate␣message␣name␣" <mid> "␣>>]
                                    and unique-message-names([<xevent>*1;
                                                              <xevent>*3;
                                                              <xevent>*5])
                                  when
                                    [<mid>] = message-name(<xevent>2),
                                    [<mid>] = message-name(<xevent>4),
                                      false = matching-xevents(<xevent>2,
                                                               <xevent>4)

[24] unique-message-names([<xevent>*])  =  Check: true Errors: []   otherwise

     check-nonmatching-messages([<xevent>*1; <xevent>2;
                                 <xevent>*3; <xevent>4;
[25]                             <xevent>*5])
                   = check-nonmatching-messages([<xevent>*1;
                                                 <xevent>*3;
                                                 <xevent>*5])
                 when
                   matching-xevents(<xevent>2, <xevent>4) = true
[26] check-nonmatching-messages([<xevent>*]) =
      aux-nonmatching-messages([<xevent>*])   otherwise

[27] aux-nonmatching-messages([])  =  Check: true Errors: []

[28] aux-nonmatching-messages([in <mid> from <iid>1 to <iid>2; <xevent>*]) =
      Check: false
      Errors: [<<no␣matching␣event␣for␣" in <mid> from <iid>1 to <iid>2 "␣>>]
      and check-nonmatching-messages([<xevent>*])
[29] aux-nonmatching-messages([out <mid> from <iid>1 to <iid>2; <xevent>*]) =
      Check: false
      Errors: [<<no␣matching␣event␣for␣" out <mid> from <iid>1 to <iid>2 "␣>>]
      and check-nonmatching-messages([<xevent>*])
[30] aux-nonmatching-messages([<xevent>; <xevent>*]) =
      Check: true Errors: []                         otherwise
      and check-nonmatching-messages([<xevent>*])
```

4.4 Example

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 message name `k` in `out k to env` of Figure 3 into `m`. Selecting the check button then results in the window of Figure 5.

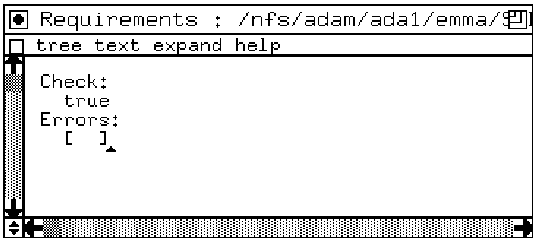

Figure 4: Result of checking a correct BMSC

Figure 5: Result of checking a BMSC with a double occurrence of message `m`

5 Process algebra

This section contains the definition of the process algebra PA_{BMSC}. First we define the atomic actions. After that we give the definition of the process algebra PA_ε and extend it with the state operator.

5.1 Atomic actions

The process algebra PA_{BMSC} is an algebraic theory for the description of process behavior based on ACP [BW90, BK84]. First we will define the set of atomic actions of PA_{BMSC}.

Every (extended) event occurring in a BMSC will be translated into an atomic action from PA_{BMSC}. Thus we have the atomic actions as displayed in Table 2.

The description in ASF+SDF of the atomic actions is given in module Atoms. Instead of defining the sets from Table 2, we define four predicates. The equations defining these predicates are straightforward.

5.1.1 Atoms

imports PA-Kernel$^{5.2.2}$ Identifiers$^{3.2.1}$ Booleans$^{2.2.1}$
exports
  context-free syntax
    in(IID, IID, MID) $\rightarrow$ ATOM

Table 2: The atomic actions of PA_{BMSC}

A_i	$=$	$\{in(s,r,m) \mid s,r \in IID, m \in MID\}$
A_o	$=$	$\{out(s,r,m) \mid s,r \in IID, m \in MID\}$
A_e	$=$	$\{out(s,env,m) \mid s \in IID, m \in MID\}$
		$\cup\{in(env,r,m) \mid r \in IID, m \in MID\}$
A_a	$=$	$\{action(i,aid) \mid i \in IID, aid \in AID\}$
A	$=$	$A_i \cup A_o \cup A_e \cup A_a$

in(*env*, IID, MID) → ATOM
out(IID, IID, MID) → ATOM
out(IID, *env*, MID) → ATOM
action(IID, AID) → ATOM

is-in-atom(ATOM) → BOOL
is-out-atom(ATOM) → BOOL
is-env(ATOM) → BOOL
is-action(ATOM) → BOOL

hiddens

variables

*atom[0-9]** → ATOM

"<" *iid* ">" [*0-9*]* → IID
"<" *mid* ">" [*0-9*]* → MID
"<" *aid* ">" [*0-9*]* → AID

equations

[1] *is-in-atom*(*in*(<*iid*>$_1$, <*iid*>$_2$, <*mid*>)) = *true*
[2] *is-in-atom*(*atom*) = *false* **otherwise**

[3] *is-out-atom*(*out*(<*iid*>$_1$, <*iid*>$_2$, <*mid*>)) = *true*
[4] *is-out-atom*(*atom*) = *false* **otherwise**

[5] *is-env*(*in*(*env*, <*iid*>$_2$, <*mid*>)) = *true*
[6] *is-env*(*out*(<*iid*>$_1$, *env*, <*mid*>)) = *true*
[7] *is-env*(*atom*) = *false* **otherwise**

[8] *is-action*(*action*(<*iid*>, <*aid*>)) = *true*
[9] *is-action*(*atom*) = *false* **otherwise**

5.2 PA_ε

The theory PA_{BMSC} is an extension of the theory PA_ε . The signature of PA_ε consists of the following functions.

1. the special constants δ and ε
2. the set of atomic actions A
3. the unary operator $\surd$

4. the binary operators $+$, $\cdot$, $\|$ and $\lfloor\!\lfloor$

The special constant δ denotes the process that has stopped executing actions and cannot proceed. This constant is called *deadlock*. The special constant ε denotes the process that is only capable of terminating successfully. It is called the *empty process*.

The atomic actions from A are the smallest processes in the description. The actual set A is defined in Table 2.

The binary operators $+$ and $\cdot$ are called the *alternative* and *sequential composition*. The alternative composition of the processes x and y is the process that either executes process x or y but not both. The sequential composition of the processes x and y is the process that first executes process x, and upon completion thereof starts with the execution of process y.

The binary operator $\|$ is called the *free merge*. The free merge of the processes x and y is the process that executes the processes x and y in parallel. For a finite set $D = \{d_1, \cdots, d_n\}$, the notation $\|_{d \in D} P(d)$ is an abbreviation for $P(d_1) \| \cdots \| P(d_n)$. If $D = \emptyset$ then $\|_{d \in D} P(d) = \varepsilon$. For the definition of the merge we use two auxiliary operators. The *termination operator* $\surd$ applied to a process x signals whether or not the process x has an option to terminate immediately. The binary operator $\lfloor\!\lfloor$ is called the *left merge*. The left merge of the processes x and y is the process that first has to execute an atomic action from process x, and upon completion thereof executes the remainder of process x and process y in parallel.

In the priorities section of module PA-Syntax one finds the line {left: PROCESS $\|$ PROCESS $\rightarrow$ PROCESS , PROCESS $\lfloor\!\lfloor$ PROCESS $\rightarrow$ PROCESS } . This means that the operators merge "$\|$" and left merge "$\lfloor\!\lfloor$" associate from left to right. Moreover, the operator $\cdot$ for sequential composition binds stronger than either of the merge operators, whereas the merge operators bind stronger than the operator for alternative composition.

5.2.1 PA-Syntax

imports Layout$^{2.2.2}$
exports
  sorts ATOM PROCESS PROCESS_LIST
  context-free syntax

ATOM	$\rightarrow$ PROCESS	
PROCESS "+" PROCESS	$\rightarrow$ PROCESS	{**right**}
PROCESS "." PROCESS	$\rightarrow$ PROCESS	{**right**}
"δ"	$\rightarrow$ PROCESS	
"ε"	$\rightarrow$ PROCESS	
PROCESS "$\|$" PROCESS	$\rightarrow$ PROCESS	{**left**}
PROCESS "$\lfloor\!\lfloor$" PROCESS	$\rightarrow$ PROCESS	{**left**}
"$\surd$" "(" PROCESS ")"	$\rightarrow$ PROCESS	
"(" PROCESS ")"	$\rightarrow$ PROCESS	{**bracket**}

  priorities
    PROCESS "."PROCESS $\rightarrow$ PROCESS > {**left**:
    PROCESS "$\|$"PROCESS $\rightarrow$ PROCESS,
    PROCESS "$\lfloor\!\lfloor$"PROCESS $\rightarrow$ PROCESS} >
    PROCESS "+"PROCESS $\rightarrow$ PROCESS

For $a \in A \cup \{\delta\}$ and processes x, y, z, the axioms of PA_ε are given in the Table 3.

Axioms A1–A9 are well known. The axioms TE1–TE3 express that a process x has an option to terminate immediately if $\surd(x) = \varepsilon$, and that $\surd(x) = \delta$ otherwise. In itself the termination operator is not very interesting, but in defining the free merge we need this operator to express the case in which both processes x and y are incapable of executing an atomic action. Axiom TM1 expresses that the free

Table 3: Axioms of PA_ε

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5
$x + \delta = x$	A6
$\delta \cdot x = \delta$	A7
$x \cdot \varepsilon = x$	A8
$\varepsilon \cdot x = x$	A9
$x \parallel y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + \surd(x) \cdot \surd(y)$	TM1
$\varepsilon \lfloor\!\lfloor x = \delta$	TM2
$a \cdot x \lfloor\!\lfloor y = a \cdot (x \parallel y)$	TM3
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	TM4
$\surd(\varepsilon) = \varepsilon$	TE1
$\surd(a \cdot x) = \delta$	TE2
$\surd(x + y) = \surd(x) + \surd(y)$	TE3

merge of the two processes x and y is their interleaving. This is expressed in the three summands. The first two state that x and y may start executing. The third summand expresses that if both x and y have an option to terminate, their merge has this option too.

Some problems arise when interpreting the axioms of Table 3 as term rewrite rules. It is clear that axiom A1 hinders termination. If we would simply delete this axiom, we would not be able to rewrite $\delta + a$ into a, so we add axiom A6a from Table 4. A second problem is that axiom A8 ($x \cdot \varepsilon = x$) is often used from right to left in calculations (e.g. $a \lfloor\!\lfloor b = a \cdot \varepsilon \lfloor\!\lfloor b = a \cdot (\varepsilon \parallel b) = \ldots = a \cdot b$). Therefore, if we give A8 an orientation from left to right, we must add the axioms TM3a and TE2a.

Finally, in order to simplify expressions we add axioms TM1a and TM1b. Note that all these axioms are provable for closed process expressions.

Table 4: Additional axioms

$\delta + x = x$	A6a
$\varepsilon \parallel x = x$	TM1a
$x \parallel \varepsilon = x$	TM1b
$a \lfloor\!\lfloor x = a \cdot x$	TM3a
$\surd(a) = \delta$	TE2a

We decided to split up the axioms of PA_ε over two separate ASF+SDF modules. The first module PA-Kernel only contains rules which deal with simplification of expressions containing the special constants. The second module PA contains the rules concerning the actual rewriting into normal form. The reason is that after translating a BMSC into a process algebra expression, one is not always interested in a complete reduction into normal form. The simulator, for example, does not need the normal forms.

It is well known that the complete state space of a parallel process may become very large. This is the so-called *state explosion* problem. The normal form of a process corresponds to its state space, so we will only calculate it when necessary.

5.2.2 PA-Kernel

imports PA-Syntax[5.2.1]
hiddens
  variables
    $x \to$ PROCESS
equations

$$
\begin{array}{lrcl}
[1] & x + \delta & = & x \\
[2] & \delta + x & = & x \\
[3] & \delta \,.\, x & = & \delta \\
[4] & x \,.\, \varepsilon & = & x \\
[5] & \varepsilon \,.\, x & = & x \\
[6] & x \parallel \varepsilon & = & x \\
[7] & \varepsilon \parallel x & = & x
\end{array}
$$

5.2.3 PA

imports PA-Kernel[5.2.2]
hiddens
  variables
    $a \to$ ATOM
    $[xyz] \to$ PROCESS
equations

$$
\begin{array}{lrcl}
[1] & (x + y) + z & = & x + y + z \\
[2] & x + x & = & x \\
[3] & (x + y) \,.\, z & = & x \,.\, z + y \,.\, z \\
[4] & (x \,.\, y) \,.\, z & = & x \,.\, y \,.\, z \\
[5] & x \parallel y & = & x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x + \surd(x) \,.\, \surd(y) \\
[6] & \varepsilon \mathbin{\lfloor\!\lfloor} x & = & \delta \\
[7] & a \,.\, x \mathbin{\lfloor\!\lfloor} y & = & a \,.\, (x \parallel y) \\
[8] & a \mathbin{\lfloor\!\lfloor} x & = & a \,.\, x \\
[9] & \delta \mathbin{\lfloor\!\lfloor} x & = & \delta \\
[10] & (x + y) \mathbin{\lfloor\!\lfloor} z & = & x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z \\
[11] & \surd(\varepsilon) & = & \varepsilon \\
[12] & \surd(a \,.\, x) & = & \delta \\
[13] & \surd(a) & = & \delta \\
[14] & \surd(\delta) & = & \delta \\
[15] & \surd(x + y) & = & \surd(x) + \surd(y)
\end{array}
$$

5.3 The state operator λ_M

A Basic Message Sequence Chart specifies a (finite) number of instances that communicate by sending and receiving messages. A message is divided into two parts: a message output and a message input. The correspondence between message outputs and message inputs has to be defined uniquely by message name identification.

A message input may not be executed before the corresponding message output has been executed. We introduce an operator λ_M that enables only those execution paths that respect the above constraint. The operator λ_M is an instance of the state operator as can be found in [BW90]. This operator remembers all message outputs that have been executed in a set M and only allows a message input if its corresponding message output is in that set.

Before specifying the signature of the state operator, we need a specification of sets of atomic actions with operators for testing, difference and union.

5.3.1 Atom-Set

imports Atoms[5.1.1]
exports
  sorts ATOM-SET
  context-free syntax
    "{" {ATOM ","}∗ "}" → ATOM-SET
    elem(ATOM, ATOM-SET) → BOOL

    ATOM-SET "\" ATOM-SET → ATOM-SET {**left**}
    ATOM-SET "∪" ATOM-SET → ATOM-SET {**left**}
    "(" ATOM-SET ")" → ATOM-SET {**bracket**}
  priorities
    ATOM-SET "\"ATOM-SET → ATOM-SET >
    ATOM-SET "∪"ATOM-SET → ATOM-SET
hiddens
  variables
    M [*0-9*]∗ → ATOM-SET
    b"∗"[*0-9*] → {ATOM ","}∗
    b"+"[*0-9*] → {ATOM ","}+
    [*ab*] → ATOM
equations

[1]	$\{b_1^*,\ a,\ b_2^*,\ a,\ b_3^*\}$	$=$	$\{b_1^*,\ a,\ b_2^*,\ b_3^*\}$	
[2]	$elem(a,\ \{b_1^*,\ a,\ b_2^*\})$	$=$	$true$	
[3]	$elem(b,\ M)$	$=$	$false$	**otherwise**
[4]	$\{b_1^*,\ a,\ b_2^*\} \setminus \{b_3^*,\ a,\ b_4^*\}$	$=$	$\{b_1^*,\ b_2^*\} \setminus \{b_3^*,\ b_4^*\}$	
[5]	$M_1 \setminus M_2$	$=$	M_1	**otherwise**
[6]	$\{b_1^*\} \cup \{b_2^*\}$	$=$	$\{b_1^*,\ b_2^*\}$	

5.3.2 State-Operator-Syntax

imports Atom-Set[5.3.1]
exports
  context-free syntax
    "λ" "_" ATOM-SET "(" PROCESS ")" → PROCESS

The axioms for the state operator are given in Table 5.

Again, some additional axioms are needed in order to get a complete term rewriting system. These are displayed in Table 6.

The axioms are again partitioned in axioms for simplification (module State-Operator-Kernel) and axioms for reduction to normal form (module State-Operator). The equations can be derived easily from Tables 5 and 6.

Table 5: Axioms for the state operator λ_M

$\lambda_M(\varepsilon) \;=\; \varepsilon$	if $M = \emptyset$	LM1
$\lambda_M(\varepsilon) \;=\; \delta$	if $M \neq \emptyset$	LM2
$\lambda_M(\delta) \;=\; \delta$		LM3
$\lambda_M(a \cdot x) \;=\; a \cdot \lambda_M(x)$	if $a \notin A_o \cup A_i$	LM4
$\lambda_M(out(i,j,m) \cdot x) \;=\; out(i,j,m) \cdot \lambda_{M \cup \{out(i,j,m)\}}(x)$		LM5
$\lambda_M(in(i,j,m) \cdot x) \;=\; in(i,j,m) \cdot \lambda_{M \setminus \{out(i,j,m)\}}(x)$	if $out(i,j,m) \in M$	LM6
	if $out(i,j,m) \in M$	
$\lambda_M(in(i,j,m) \cdot x) \;=\; \delta$	if $out(i,j,m) \notin M$	LM7
$\lambda_M(x+y) \;=\; \lambda_M(x) + \lambda_M(y)$		LM8

Table 6: Auxiliary axioms for the state operator

$\lambda_M(a) \;=\; a$	if $a \notin A_o \cup A_i$, $M = \emptyset$	LM4a
$\lambda_M(a) \;=\; \delta$	if $a \notin A_o \cup A_i$, $M \neq \emptyset$	LM4b
$\lambda_M(out(i,j,m)) \;=\; out(i,j,m) \cdot \delta$		LM5a
$\lambda_M(in(i,j,m)) \;=\; in(i,j,m)$	if $out(i,j,m) \in M$, $M \setminus \{out(i,j,m)\} = \emptyset$	LM6a
$\lambda_M(in(i,j,m)) \;=\; in(i,j,m) \cdot \delta$	if $out(i,j,m) \in M$, $M \setminus \{out(i,j,m)\} \neq \emptyset$	LM6b
$\lambda_M(in(i,j,m)) \;=\; \delta$	if $out(i,j,m) \notin M$	LM7a

5.3.3 State-Operator-Kernel

imports State-Operator-Syntax[5.3.2]
hiddens
 variables
  *M [0-9]** → ATOM-SET
  a → ATOM
  x → PROCESS
  "<" *iid* ">" *[0-9]** → IID
  "<" *mid* ">" → MID
equations

[1] $\lambda_M\ (\varepsilon) \;=\; \varepsilon$ **when** $M = \{\}$

[2] $\lambda_M\ (\varepsilon) \;=\; \delta$ **when** $M \neq \{\}$

[3] $\lambda_M\ (\delta) \;=\; \delta$

5.3.4 State-Operator

imports State-Operator-Kernel[5.3.3]
hiddens
 variables
  *M [0-9]** → ATOM-SET
  a → ATOM
  [xyz] → PROCESS
  "<" *iid* ">" *[0-9]** → IID
  "<" *mid* ">" → MID
equations

[1] $$\lambda_M\ (a\ .\ x) = a\ .\ \lambda_M\ (x)$$
$$\textbf{when}\quad \mathit{is\text{-}out\text{-}atom}(a) = \mathit{false},\ \mathit{is\text{-}in\text{-}atom}(a) = \mathit{false}$$

[2] $$\lambda_M\ (a) = a$$
$$\textbf{when}\quad M = \{\},\ \mathit{is\text{-}out\text{-}atom}(a) = \mathit{false},\ \mathit{is\text{-}in\text{-}atom}(a) = \mathit{false}$$

[3] $$\lambda_M\ (a) = a\ .\ \delta$$
$$\textbf{when}\quad M \neq \{\},\ \mathit{is\text{-}out\text{-}atom}(a) = \mathit{false},\ \mathit{is\text{-}in\text{-}atom}(a) = \mathit{false}$$

[4] $$\lambda_M\ (\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>)\ .\ x) = \mathit{out}(<iid>_1,\ <iid>_2,\ <mid>)\ .\ \lambda_{M\ \cup\ \{\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>)\}}\ (x)$$

[5] $$\lambda_M\ (\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>))\ =\ \mathit{out}(<iid>_1,\ <iid>_2,\ <mid>)\ .\ \delta$$

[6] $$\lambda_M\ (\mathit{in}(<iid>_1,\ <iid>_2,\ <mid>)\ .\ x)\ =\ \mathit{in}(<iid>_1,\ <iid>_2,\ <mid>)\ .\ \lambda_{M\ \setminus\ \{\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>)\}}\ (x)$$
$$\textbf{when}\quad \mathit{elem}(\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>),\ M) = \mathit{true}$$

[7] $$\lambda_M\ (\mathit{in}(<iid>_1,\ <iid>_2,\ <mid>))\ =\ \mathit{in}(<iid>_1,\ <iid>_2,\ <mid>)$$
$$\textbf{when}\quad \mathit{elem}(\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>),\ M) = \mathit{true},\ M \setminus \{\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>)\} = \{\}$$

[8] $$\lambda_M\ (\mathit{in}(<iid>_1,\ <iid>_2,\ <mid>))\ =\ \mathit{in}(<iid>_1,\ <iid>_2,\ <mid>)\ .\ \delta$$
$$\textbf{when}\quad \mathit{elem}(\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>),\ M) = \mathit{true},\ M \setminus \{\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>)\} \neq \{\}$$

[9] $$\lambda_M\ (\mathit{in}(<iid>_1,\ <iid>_2,\ <mid>)\ .\ x)\ =\ \delta$$
$$\textbf{when}\quad \mathit{elem}(\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>),\ M) = \mathit{false}$$

[10] $$\lambda_M\ (\mathit{in}(<iid>_1,\ <iid>_2,\ <mid>))\ =\ \delta$$
$$\textbf{when}\quad \mathit{elem}(\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>),\ M) = \mathit{false}$$

[11] $$\lambda_M\ (x + y)\ =\ \lambda_M\ (x) + \lambda_M\ (y)$$

6 Translation into process algebra

In this section, we will define a semantic function S that associates to every Basic Message Sequence Chart in textual format a closed PA_{BMSC} term. Before we give the definition of this semantic function we need to explain some auxiliary functions. The powerset of a set S is denoted by $\mathbb{P}(S)$.

The function

$$Instances : \mathcal{L}(\texttt{<msc>}) \to \mathbb{P}(\mathcal{L}(\texttt{<inst def>}))$$

that associates to a Basic Message Sequence Chart the set containing all instance definitions of the instances defined in the chart, is defined by

$$\begin{array}{l} Instances(\texttt{msc <mscid>; <msc body> endmsc;}) \;=\; \\ \quad Instances_{body}(\texttt{<msc body>}) \end{array}$$

where the function

$$Instances_{body} : \mathcal{L}(\texttt{<msc body>}) \to \mathbb{P}(\mathcal{L}(\texttt{<inst def>}))$$

is defined by

$$\begin{array}{l} Instances_{body}(\texttt{<>}) \;=\; \emptyset \\ Instances_{body}(\texttt{<inst def><msc body>}) \;=\; \\ \quad \{\texttt{<inst def>}\} \cup Instances_{body}(\texttt{<msc body>}) \end{array}$$

Next we define the following two functions

$$\begin{array}{l} Name : \mathcal{L}(\texttt{<inst def>}) \to \mathcal{L}(\texttt{<iid>}) \\ Body : \mathcal{L}(\texttt{<inst def>}) \to \mathcal{L}(\texttt{<inst body>}) \end{array}$$

These functions associate to an instance definition its name and body.

$$\begin{array}{l} Name(\texttt{instance <iid>;<inst body> endinstance;}) \;=\; \texttt{<iid>} \\ Body(\texttt{instance <iid>;<inst body> endinstance;}) \;=\; \texttt{<inst body>} \end{array}$$

6.1 The semantic function

The general idea is that the semantics of a Basic Message Sequence Chart is the free merge of the semantics of its instances. By this construction we enable all interleavings of the message outputs and message inputs. However, a message input can only be performed after its corresponding message output. In order to rule out all interleavings where a message output is preceded by the corresponding message input we use the state operator λ_M. We define the function $S : \mathcal{L}(\texttt{<msc>}) \to T(\Sigma_{PA_{BMSC}})$ by

$$S[\![msc]\!] \;=\; \lambda_\emptyset \left(\|_{\,idef \,\in Instances(msc)}\; S_{inst}[\![idef\,]\!] \right)$$

The semantic function $S_{inst} : \mathcal{L}(\texttt{<inst def>}) \to T(\Sigma_{PA_{BMSC}})$ is defined to express the semantics of one instance in separation. In the textual representation of an instance the atomic actions are specified in the order they are to be executed, thus the semantics of an instance definition is the sequential composition of its actions.

$$S_{inst}[\![idef\,]\!] \;=\; S_{body}^{Name(idef\,)}[\![Body(idef\,)]\!]$$

where for $i \in \mathcal{L}(\texttt{<iid>})$ the function

$$S^i_{body} : \mathcal{L}(\texttt{<inst body>}) \to T(\Sigma_{PA_{BMSC}})$$

is defined by

$$S^i_{body}[\![\texttt{<>}]\!] \;=\; \varepsilon$$
$$S^i_{body}[\![\texttt{<event><inst body>}]\!] \;=\;$$
$$\quad S^i_{event}[\![\texttt{<event>}]\!] \cdot S^i_{body}[\![\texttt{<inst body>}]\!]$$

and for every $i \in \mathcal{L}(\texttt{<iid>})$ the function

$$S^i_{event} : \mathcal{L}(\texttt{<event>}) \rightarrow T(\Sigma_{PA_{BMSC}})$$

is defined by

$$S^i_{event}[\![\texttt{in <mid> from <iid>;}]\!] \;=\; in(\texttt{<iid>}, i, \texttt{<mid>})$$
$$S^i_{event}[\![\texttt{in <mid> from env;}]\!] \;=\; in(env, i, \texttt{<mid>})$$
$$S^i_{event}[\![\texttt{out <mid> to <iid>;}]\!] \;=\; out(i, \texttt{<iid>}, \texttt{<mid>})$$
$$S^i_{event}[\![\texttt{out <mid> to env;}]\!] \;=\; out(i, env, \texttt{<mid>})$$
$$S^i_{event}[\![\texttt{action <aid>;}]\!] \;=\; action(i, \texttt{<aid>})$$

The translation of the semantic function into ASF+SDF is rather straightforward. The only problem is that the generalized merge construct ($\|_{\,idef\, \in Instances(msc)}$) occurring in the definition of $S[\![msc]\!]$ requires higher order functions. Therefore, we combined the generalized merge and the application of the function S_{inst} into one single function $\| S_{inst}$. This function requires the collection of all instance definitions as input and calculates the parallel composition of the semantics of these instances. The set of instances is calculated by the auxiliary function $Instances$.

Furthermore, notice that we only import the kernel of the process algebra. This means that we only have the signature and some rules for simplification, but not the defining equations.

6.1.1 BMSC-Semantics

```
imports State-Operator-Kernel^5.3.3 BMSC-Syntax^3.2.2
exports
  sorts INST-DEF-LIST
  context-free syntax
    "S" "(" MSC ")"                                  -> PROCESS
    "S" "_" "inst" "(" INST-DEF ")"                  -> PROCESS
    "S" "_" "body" "^" IID "(" INST-BODY ")"         -> PROCESS
    "S" "_" "event" "^" IID "(" EVENT ")"            -> PROCESS
hiddens
  context-free syntax
    "||S" "_" "inst" INST-DEF-LIST                   -> PROCESS
    "(" {INST-DEF ","}* ")"                          -> INST-DEF-LIST
    INST-DEF-LIST "∪" INST-DEF-LIST                  -> INST-DEF-LIST {left}
    "Instances" (MSC)                                -> INST-DEF-LIST
    "Instances" "_" "body" "(" MSC-BODY ")"          -> INST-DEF-LIST
    "Name" (INST-DEF)                                -> IID
    "Body" (INST-DEF)                                -> INST-BODY
  variables
    "<" inst-def ">" "*" [0-9]*   -> {INST-DEF ","}*
    i                             -> IID
    "<" msc ">"                   -> MSC
    "<" msc-body ">"              -> MSC-BODY
    "<" inst-def ">"              -> INST-DEF
    "<" inst-body ">"             -> INST-BODY
    "<" event ">" [0-9]*          -> EVENT
    "<" mscid ">"                 -> MSCID
    "<" iid ">"                   -> IID
    "<" mid ">"                   -> MID
```

"<" *aid* ">" $\rightarrow$ AID

equations

[1] $S(<msc>) \; = \; \lambda_{\{\}} \; (\|S_{inst} \; Instances(<msc>))$

[2] $\|S_{inst} \; () \qquad = \; \varepsilon$

[3] $\|S_{inst} \; (<inst\text{-}def>) \; = \; S_{inst} \; (<inst\text{-}def>)$

[4] $\|S_{inst} \; (<inst\text{-}def>, <inst\text{-}def>^{*}) =$

$S_{inst} \; (<inst\text{-}def>) \parallel \|S_{inst} \; (<inst\text{-}def>^{*})$

[5] $S_{inst} \; (<inst\text{-}def>) \; = \; S_{body}^{Name(<inst\text{-}def>)} \; (Body(<inst\text{-}def>))$

[6] $S^{i}_{body} \; () \qquad = \; \varepsilon$

[7] $S^{i}_{body} \; (<event>; <inst\text{-}body>) =$

$S^{i}_{event} \; (<event>) \; . \; S^{i}_{body} \; (<inst\text{-}body>)$

[8] S^{i}_{event} (in $<mid>$ from $<iid>$) $=$ $in(<iid>, i, <mid>)$

[9] S^{i}_{event} (in $<mid>$ from env) $=$ $in(\text{env}, i, <mid>)$

[10] S^{i}_{event} (out $<mid>$ to $<iid>$) $=$ $out(i, <iid>, <mid>)$

[11] S^{i}_{event} (out $<mid>$ to env) $=$ $out(i, \text{env}, <mid>)$

[12] S^{i}_{event} (action $<aid>$) $=$ $action(i, <aid>)$

[13] $Instances$(msc $<mscid>$; $<msc\text{-}body>$ endmsc;) $=$

$Instances_{body} \; (<msc\text{-}body>)$

[14] $Instances_{body} \; () \; = \; ()$

[15] $Instances_{body} \; (<inst\text{-}def>; <msc\text{-}body>) =$

$(<inst\text{-}def>) \cup Instances_{body} \; (<msc\text{-}body>)$

[16] $Name$(instance $<iid>$; $<inst\text{-}body>$ endinstance) $=$ $<iid>$

[17] $Body$(instance $<iid>$; $<inst\text{-}body>$ endinstance) $=$ $<inst\text{-}body>$

[18] $(<inst\text{-}def>^{*}_{1}) \cup (<inst\text{-}def>^{*}_{2}) \; = \; (<inst\text{-}def>^{*}_{1}, <inst\text{-}def>^{*}_{2})$

6.2 Example

The result of applying this translation to the BMSC in the editor of Figure 3 is the process algebra term $\lambda_{\emptyset}(out(a,b,m) \cdot out(a,env,k) \parallel action(b,p) \cdot in(a,b,m))$. The application of the merge operator ($\parallel$) shows that the semantics of the given BMSC is the interleaved execution of the processes $out(a,b,m) \cdot out(a,env,k)$ and $action(b,p) \cdot in(a,b,m))$. The state operator ($\lambda_{\emptyset}$) in front of the expression enforces that input of message m only occurs after the corresponding output.

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

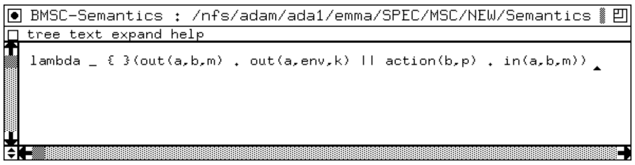

Figure 6: Result of computing the semantics of a BMSC

6.3 Normalization

The state operator and the merge operator 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. The normalizer is simply defined by combining the definitions of the semantic functions and the complete specification of the process algebra.

6.3.1 Normalize

imports BMSC-Semantics$^{6.1.1}$ PA$^{5.2.3}$ State-Operator$^{5.3.4}$

6.4 Example

Figure 7 shows the effect of pressing the normalize button in the editor of Figure 3. It expresses the branching structure of the process. First one can make a choice between executing $out(a,b,m)$ and $action(b,p)$. If one chooses the first option, another choice has to be made between $out(a,env,k)$ and $action(b,p)$. The rest of the process can be understood in a similar way.

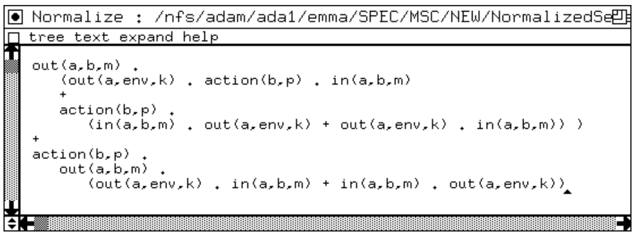

Figure 7: Result of normalizing the semantics of a BMSC

7 A simulator

For large BMSCs, the expressions describing the normalized semantics as in Figure 7 become quite large and complex. This is the so-called *state explosion* problem. 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 [MR94a]. This operational semantics defines for a given BMSC a labeled transition system. The transitions correspond with the events of the BMSC.

First, we will interpret the definition of the transition rules in an algebraic specification. After that, we define the additional functions needed to obtain a simulator.

7.1 Transitions

In this section we define a structural operational semantics of Basic Message Sequence Charts in the style of Plotkin [Plo83]. For this purpose we define action relations on closed PA_{BMSC} terms.

On the set of PA_{BMSC} terms we define a predicate $\downarrow \subseteq T(\Sigma_{PA_{BMSC}})$ and binary relations $\xrightarrow{a} \subseteq T(\Sigma_{PA_{BMSC}}) \times T(\Sigma_{PA_{BMSC}})$ for every $a \in A$. These predicates are defined by means of inference rules, which have the following form.

$$\frac{p_1, \ldots, p_n}{q}$$

This expression means that for every instantiation of variables in $p_1, \ldots, p_n, q$ we can conclude q from $p_1, \ldots, p_n$. If q is a tautology, we omit $p_1, \ldots, p_n$ and the horizontal bar.

The intuitive idea of the predicate $\downarrow$ is as follows: $t\downarrow$ denotes that t has an option to terminate immediately, i.e. ε is a summand of t. For $x, y \in T(\Sigma_{PA_{BMSC}})$, and $M \subseteq A_o$, the predicate $\downarrow$ is defined in Table 7.

Table 7: The predicate $\downarrow$

	$\varepsilon\downarrow$	
$\dfrac{x\downarrow}{(x+y)\downarrow}$	$\dfrac{x\downarrow\,,\ y\downarrow}{(x \cdot y)\downarrow}$	$\dfrac{y\downarrow}{(x+y)\downarrow}$
$\dfrac{x\downarrow}{(\surd(x))\downarrow}$	$\dfrac{x\downarrow\,,\ y\downarrow}{(x \parallel y)\downarrow}$	$\dfrac{x\downarrow}{(\lambda_M(x))\downarrow}$

The intuitive idea of the binary operator $\xrightarrow{a}$ is as follows: $t \xrightarrow{a} s$ denotes that the process t can execute the atomic action a and after this execution step the resulting process is s. For $x, x', y, y' \in T(\Sigma_{PA_{BMSC}})$, $a \in A$, $M \subseteq A_o$, $i, j \in \mathcal{L}(\texttt{<iid>})$, and $m \in \mathcal{L}(\texttt{<mid>})$, the binary relations $\xrightarrow{a}$ are defined in Table 8.

We will illustrate the use of these action relations with an example. Consider the following expression.

$$\lambda_\emptyset(out(a,b,k) \parallel in(a,b,k))$$

We have $out(a,b,k) \xrightarrow{out(a,b,k)} \varepsilon$, so we can derive $out(a,b,k) \parallel in(a,b,k) \xrightarrow{out(a,b,k)} \varepsilon \parallel in(a,b,k)$. From this we can conclude

Table 8: The action relations $\xrightarrow{a}$

$a \xrightarrow{a} \varepsilon$			
$\dfrac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'}$	$\dfrac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$	$\dfrac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$	$\dfrac{x\downarrow,\ y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$
$\dfrac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y}$	$\dfrac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'}$	$\dfrac{x \xrightarrow{a} x'}{x \mathbin{\lfloor\!\lfloor} y \xrightarrow{a} x' \parallel y}$	
$\dfrac{a \notin A_o \cup A_i\ ,\ x \xrightarrow{a} x'}{\lambda_M(x) \xrightarrow{a} \lambda_M(x')}$	$\dfrac{x \xrightarrow{out(i,j,m)} x'}{\lambda_M(x) \xrightarrow{out(i,j,m)} \lambda_{M \cup \{out(i,j,m)\}}(x')}$	$\dfrac{out(i,j,m) \in M\ ,\ x \xrightarrow{in(i,j,m)} x'}{\lambda_M(x) \xrightarrow{in(i,j,m)} \lambda_{M \setminus \{out(i,j,m)\}}(x')}$	

$$\lambda_\emptyset(out(a,b,k) \parallel in(a,b,k)) \xrightarrow{out(a,b,k)} \lambda_{\{out(a,b,k)\}}(\varepsilon \parallel in(a,b,k))$$

Next we have $in(a,b,k) \xrightarrow{in(a,b,k)} \varepsilon$, and we can derive $\varepsilon \parallel in(a,b,k) \xrightarrow{in(a,b,k)} \varepsilon \parallel \varepsilon$. Thus we have

$$\lambda_{\{out(a,b,k)\}}(\varepsilon \parallel in(a,b,k)) \xrightarrow{in(a,b,k)} \lambda_\emptyset(\varepsilon \parallel \varepsilon)$$

In order to see that this expression has the possibility to terminate, we derive $\varepsilon\downarrow$ and thus $(\varepsilon \parallel \varepsilon)\downarrow$, so

$$\lambda_\emptyset(\varepsilon \parallel \varepsilon)\downarrow$$

Finally, we conclude that the given process $\lambda_\emptyset(out(a,b,k) \parallel in(a,b,k))$ can first execute $out(a,b,k)$, then execute $in(a,b,k)$ and finally terminate. Note that this is the only execution sequence that can be derived from the inference rules.

7.2 Algebraic specification of the transition rules

The translation of the transition rules into an algebraic specification needs some explanation. In the transition rules we defined the transition predicate and the termination predicate. However, for a simulator we need to know for a given process algebra expression all possible transitions coming from this expression. Thus we are not interested in the transition relation itself, but in the function transitions which calculates for a given PROCESS a TRANSITIONLIST. A TRANSITION consists of an ATOM which is the label of the transition and a PROCESS which is the resulting process after executing the atomic action. Some additional functions are needed for calculating the list of transitions of a given process.

For example, Table 8 shows that if one wants to calculate the transitions for $x + y$, one simply has to calculate the transitions of both x and y (equation 4). The case of $x.y$ is a bit more involved. By combining the two derivation rules for sequential composition from Table 8, we obtain equations 5 and 6. If we consider the case that x does not terminate, the subtle point is that the transitions of $x.y$ are not completely equal to the transitions of x. The residue after executing an action has to be extended with y. For this purpose we use the overloaded "." function. The same procedure is carried out for the remaining operators.

The algebraic specification of the predicate terminates is straightforward.

7.2.1 Transitions

imports BMSC-Semantics[6.1.1]
exports
sorts TRANSITION TRANSITIONLIST
context-free syntax

“—” ATOM “→” PROCESS	→ TRANSITION	
“[” {TRANSITION “,”}* “]”	→ TRANSITIONLIST	
TRANSITIONLIST “∪” TRANSITIONLIST	→ TRANSITIONLIST	**{left}**
TRANSITIONLIST “.” PROCESS	→ TRANSITIONLIST	
TRANSITIONLIST “‖” PROCESS	→ TRANSITIONLIST	
PROCESS “‖” TRANSITIONLIST	→ TRANSITIONLIST	
“*filter*” “_” ATOM-SET “(” TRANSITIONLIST “)”	→ TRANSITIONLIST	
terminates(PROCESS)	→ BOOL	
transitions(PROCESS)	→ TRANSITIONLIST	
“(” TRANSITIONLIST “)”	→ TRANSITIONLIST	**{bracket}**

hiddens
variables

M	→ ATOM-SET
x	→ PROCESS
y	→ PROCESS
a	→ ATOM
*tl[0-9]**	→ {TRANSITION “,”}*
“<” *iid* “>” *[0-9]**	→ IID
“<” *mid* “>” *[0-9]**	→ MID
“<” *aid* “>” *[0-9]**	→ AID

equations

$$\begin{array}{lrcl}
[1] & transitions(\delta) & = & []\\
[2] & transitions(\varepsilon) & = & []\\
[3] & transitions(a) & = & [\text{—}\ a \to \varepsilon]\\
[4] & transitions(x + y) & = & transitions(x) \cup transitions(y)
\end{array}$$

$$\begin{array}{lrcl}
[5] & transitions(x\ .\ y) & = & transitions(x)\ .\ y \cup transitions(y)\\
 & & & \textbf{when}\\
 & & & \quad terminates(x) = true\\
[6] & transitions(x\ .\ y) & = & transitions(x)\ .\ y\\
 & & & \textbf{when}\\
 & & & \quad terminates(x) = false
\end{array}$$

$$\begin{array}{lrcl}
[7] & transitions(x \parallel y) & = & transitions(x) \parallel y \cup x \parallel transitions(y)\\
[8] & transitions(x \,\lfloor\!\lfloor\, y) & = & transitions(x) \parallel y\\
[9] & transitions(\lambda_M\,(x)) & = & filter_M\,(transitions(x))
\end{array}$$

$$\begin{array}{lrcl}
[10] & terminates(\varepsilon) & = & true\\
[11] & terminates(a) & = & false\\
[12] & terminates(\delta) & = & false\\
[13] & terminates(x + y) & = & terminates(x) \mid terminates(y)\\
[14] & terminates(x\ .\ y) & = & terminates(x)\ \&\ terminates(y)\\
[15] & terminates(\surd(x)) & = & terminates(x)\\
[16] & terminates(x \parallel y) & = & terminates(x)\ \&\ terminates(y)\\
[17] & terminates(\lambda_M\,(x)) & = & terminates(x)
\end{array}$$

$$
\begin{array}{lll}
[18] & [tl_1] \cup [tl_2] & = [tl_1,\ tl_2] \\
[19] & [\,] \,.\, x & = [\,] \\
[20] & [\text{—}\ a \to x,\ tl] \,.\, y & = [\text{—}\ a \to x \,.\, y] \cup [tl] \,.\, y \\
[21] & [\,] \parallel y & = [\,] \\
[22] & [\text{—}\ a \to x,\ tl] \parallel y & = [\text{—}\ a \to x \parallel y] \cup [tl] \parallel y \\
[23] & y \parallel [\,] & = [\,] \\
[24] & y \parallel [\text{—}\ a \to x,\ tl] & = [\text{—}\ a \to y \parallel x] \cup y \parallel [tl]
\end{array}
$$

[25] $\mathit{filter}_M\ ([\,]) = [\,]$

[26] $\mathit{filter}_M\ ([\text{—}\ \mathit{out}(<iid>_1,\ <iid>_2,\ <mid>) \to x,\ tl]) =$

$[\text{—}\ \mathit{out}(<iid>_1,\ <iid>_2,\ <mid>) \to \lambda_{M\ \cup\ \{\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>)\}}\ (x)]$
$\cup\ \mathit{filter}_M\ ([tl])$

[27] $\mathit{filter}_M\ ([\text{—}\ \mathit{in}(<iid>_1,\ <iid>_2,\ <mid>) \to x,\ tl])$

$= [\text{—}\ \mathit{in}(<iid>_1,\ <iid>_2,\ <mid>)$
$\to \lambda_{M\ \setminus\ \{\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>)\}}\ (x)]$
$\cup\ \mathit{filter}_M\ ([tl])$

when
$\mathit{elem}(\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>),\ M) = \mathit{true}$

[28] $\mathit{filter}_M\ ([\text{—}\ \mathit{in}(<iid>_1,\ <iid>_2,\ <mid>) \to x,\ tl]) = \mathit{filter}_M\ ([tl])$

when
$\mathit{elem}(\mathit{out}(<iid>_1,\ <iid>_2,\ <mid>),\ M) = \mathit{false}$

[29] $\mathit{filter}_M\ ([\text{—}\ a \to x,\ tl]) =$

$[\text{—}\ a \to \lambda_M\ (x)] \cup \mathit{filter}_M\ ([tl])$ **otherwise**

7.3 Simulation

A simulator displays the current state of the BMSC and offers the user a choice between all possible continuations. Such a STATE consists of three parts. The first component is the PROCESS under consideration. The second component, NUMBERED-TRLIST, is the list of transitions associated to this process. The transitions are numbered in order to offer the user the possibility of choosing such a transition. The third component of the state is an ATOMLIST which contains the history of the simulation session. It consists of all atomic actions chosen so far.

The function execute accepts a number and a state and calculates the resulting state after execution of the transition labeled with the given number.

Note that the imported module Naturals is not included in this paper. It defines the sort NAT with obvious properties.

7.3.1 Simulator

imports Naturals Transitions[7.2.1]
exports
sorts NUMBERED-TRLIST STATE NUMBERED-TRANSITION
ATOMLIST NUMBERED-ATOM NUMBERED-ATOMLIST
context-free syntax
initial-state(PROCESS) $\to$ STATE

“<” PROCESS “,”
NUMBERED-TRLIST “,”
ATOMLIST “>” → STATE

“[” {ATOM “,”}∗ “]” → ATOMLIST
“(” NAT “)” ATOM → NUMBERED-ATOM
“[” {NUMBERED-ATOM “,”}∗ “]” → NUMBERED-ATOMLIST
ATOMLIST “∪” ATOMLIST → ATOMLIST {**left**}
execute(NAT, STATE) → STATE

“(” NAT “)” “—” ATOM “→” PROCESS → NUMBERED-TRANSITION
“[” {NUMBERED-TRANSITION “,”}∗ “]” → NUMBERED-TRLIST
number(TRANSITIONLIST) → NUMBERED-TRLIST
number-from(NAT, TRANSITIONLIST) → NUMBERED-TRLIST
NUMBERED-TRLIST “∪” NUMBERED-TRLIST → NUMBERED-TRLIST {**left**}

hiddens

variables

tl → TRANSITIONLIST
n → NAT
a → ATOM
al → ATOMLIST
a[*0-9*]∗“∗” → {ATOM “,”}∗
tr[*0-9*]∗“∗” → {TRANSITION “,”}∗
ntr[*0-9*]∗“∗” → {NUMBERED-TRANSITION “,”}∗
x[*0-9*]∗ → PROCESS
ntl → NUMBERED-TRLIST

equations

[1] $number(tl) = number\text{-}from(1,\ tl)$

[2] $number\text{-}from(n,\ []) = []$

[3] $number\text{-}from(n,\ [\text{—}\ a \to x,\ tr^*]) = [(n)\ \text{—}\ a \to x] \cup number\text{-}from(n+1,\ [tr^*])$

[4] $[ntr_1^*] \cup [ntr_2^*] = [ntr_1^*,\ ntr_2^*]$

[5] $execute(n, < x_1, [ntr_1^*,\ (n)\ \text{—}\ a \to x_2,\ ntr_2^*],\ al >) =$
$< x_2,\ number(transitions(x_2)),\ al \cup [a] >$

[6] $[a_1^*] \cup [a_2^*] = [a_1^*,\ a_2^*]$

7.4 Example

For the running example, represented by the term

$$\lambda_\emptyset(out(a,b,m) \cdot out(a,env,k) \,\|\, action(b,p) \cdot in(a,b,m))$$

the set of transitions is

$$\{ \stackrel{out(a,b,m)}{\rightarrow} \lambda_\emptyset(out(a,env,k) \,\|\, action(b,p) \cdot in(a,b,m)),\\ \stackrel{action(b,p)}{\rightarrow} \lambda_\emptyset(out(a,b,m) \cdot out(a,env,k) \,\|\, in(a,b,m)) \}$$

This means that executing event $out(a,b,m)$ results in the BMSC represented by $\lambda_\emptyset(out(a,env,k) \,\|\, action(b,p) \cdot in(a,b,m))$ and that execution of the alternative action $action(b,p)$ results in $\lambda_\emptyset(out(a,b,m) \cdot out(a,env,k) \,\|\, in(a,b,m))$. Likewise, the transition sets of the resulting processes can be determined. If the BMSC is finished, the resulting process is ε.

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.

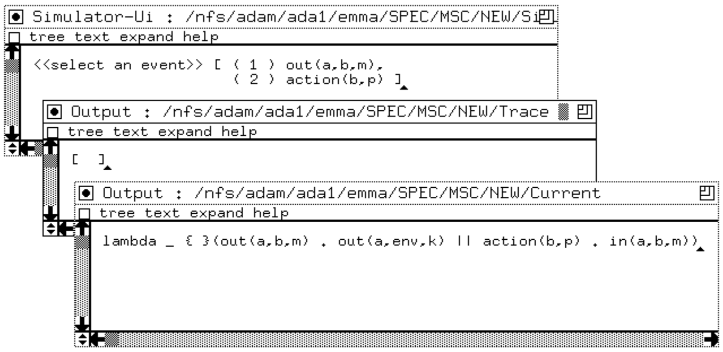

Figure 8: Starting the simulator

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

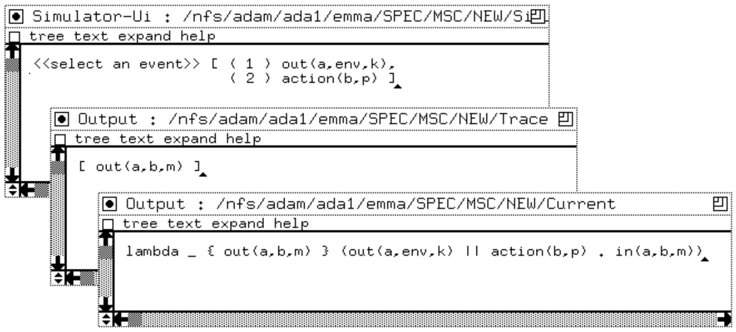

Figure 9: Result after selecting event number (1) in the previous figure

If we subsequently select the second event, we obtain the situation from Figure 10.

Next, we select the first event and obtain the situation from Figure 11.

Finally, there's only one remaining event. The result of selecting this event is in Figure 12. It shows that execution of the BMSC is finished.

8 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

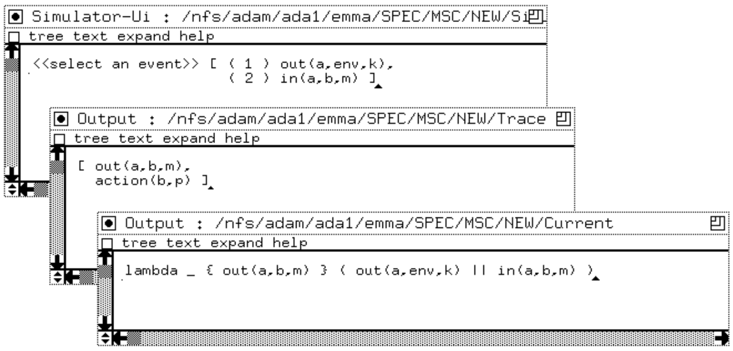

Figure 10: Result after selecting event number (2) in the previous figure

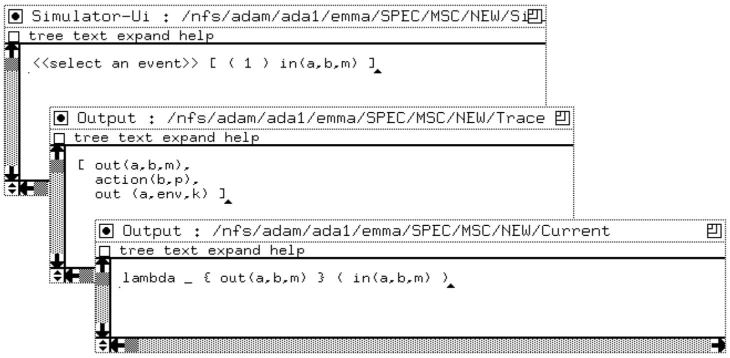

Figure 11: Result after selecting event number (1) in the previous figure

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.

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 [Klo92]), 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.

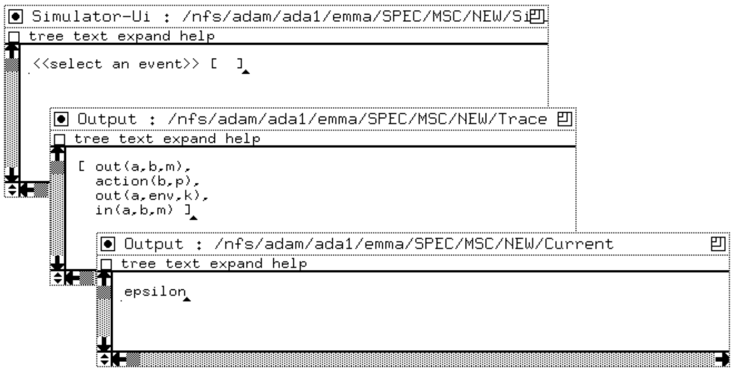

Figure 12: Result after selecting event number (1) in the previous figure

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 ([KW93]).

References

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information & Control*, 60:109–137, 1984.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications, vol. I, Equations and Initial Semantics*. Springer-Verlag, 1985.

[IT94] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, 1994.

[IT95] ITU-TS. *ITU-TS Recommendation Z.120 Annex B: Algebraic semantics of Message Sequence Charts*. ITU-TS, Geneva, Publ. Sched. 1995.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[Klo92] 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.

[Koo92] J.W.C. Koorn. Connecting semantic tools to a syntax-directed user-interface. Report P9222, Programming Research Group, University of Amsterdam, 1992.

[Koo94] J.W.C. Koorn. *Generating uniform user-interfaces for interactive programming environments*. PhD thesis, University of Amsterdam, 1994. ILLC Dissertation series 1994-2.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993.

[MR94a] S. Mauw and M.A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The computer journal*, 37(4):269–277, 1994.

[MR94b] 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.

[Plo83] G.D. Plotkin. An operational semantics for CSP. In *Proceedings of the Conference on the Formal Description of Programming Concepts*, volume 2, Garmisch, 1983.

[Ren94] M.A. Reniers. Syntax requirements of Message Sequence Charts. Study Group Meeting SG10, Geneva, TD59, ITU-TS, 1994.