# Semantics of Message Sequence Charts

A.A. Letichevsky[1], J.V. Kapitonova[1], V.P. Kotlyarov[2], V.A. Volkov[1], A.A. Letichevsky Jr.[1], and T. Weigert[3]

[1] Glushkov Institute of Cybernetics, National Academy of Science of Ukraine
Kiev, Ukraine
[2] Global Software Group, Motorola ZAO
St. Petersburg, Russia
[3] Global Software Group, Motorola, Inc.
Schaumburg, Illinois
thomas.weigert@motorola.com

**Abstract.** The language of MSC diagrams is widely used for the specification of communicating systems, the design of software and hardware for real time and reactive systems, and other industrial applications. Often it is used as an abstraction of systems specified in SDL or UML (in the form of sequence diagrams). In this paper, a novel representation of the semantics of message sequence charts is described. This formulation has been developed to enable the implementation of tools aimed at the verification of requirements for interactive systems. Our definition of the formal semantics of the language of MSC diagrams relies on the theory of interaction of agents and environments. This approach helped to simplify the definition of the semantics in comparison to other approaches based on highly sophisticated process algebras and it brought the definition of the semantics closer to possible implementations.

## 1 Introduction

The language of Message Sequence Charts (MSC diagrams) is widely used for the development of communicating systems, and for the design of software and hardware for real time and reactive systems [1]. MSC is an asynchronous language without an explicit notion of time. To enable the development of tools for the verification of highly reliable systems, a formal semantics of the language which can be easily implemented must be available. To define the formal semantics of MSC diagrams, we relied on a new semantic approach based on the theory of interaction of agents and environments [3–5]. This approach greatly simplified the definition of the semantics in comparison to conventional approaches based on highly sophisticated process algebras, such as that of Reniers [6]. In addition, the definition of our semantics is much closer to possible implementations and thus simplifies its realization in tools. Moreover, this semantics is easy to modify and new features can be added easily. (For example, we have also developed extensions of this semantics representing timed message sequence charts and an algorithm for checking time consistency, albeit these are not discussed in this paper.)

Our definition is represented in the form of a calculus which defines transitions for an untimed MSC environment. In Section 2, we review the theory of agents and environments. Section 3 gives the semantics of MSC diagrams. In section 4 we briefly outline the translation of MSC diagrams to MSC processes. Finally, in section 5, we give an example of computing the meaning of an MSC diagram following the presented semantics.

## 2 Agents and Environments

MSC diagrams describe interacting entities or *instances*. Mathematically, these entities are represented by means of labeled transition systems with divergence and termination, considered up to bisimilarity. These transition systems, which we shall refer to as *agents*, execute in an environment. *Environments* are agents supplied with an insertion function, which describes the change of the behavior of an environment after inserting an agent into this environment.

### 2.1 Agents

Agents are objects which can be recognized as separate from other agents and their environment. They can change their internal states and interact with other agents and their environments, performing observable actions. The notion of an agent formalizes such diverse objects as software components, programs, users, clients, servers, active components of distributed knowledge bases, or similar.

Agents with the same behavior are considered as equivalent. The equivalence of agents is characterized in terms of an algebra of behaviors $F(A)$ which is a free continuous algebra (algebra with approximation) with two sorts–actions $a \in A$ and behaviors $u \in F(A)$. The operations of this algebra are non-deterministic choice $u + v$, $u, v \in F(A)$, which is an associative, commutative and idempotent binary operation, and prefixing $a.u \in A$, $a \in A$, $u \in F(A)$. The approximation relation $\sqsubseteq$ on a set of behaviors is a partial order such that these two operations are continuous. The algebra $F(A)$ is closed relative to the limits (least upper bounds) of the ordered sets of finite behaviors. Consequentially, the minimal fixed point theorem can be used for the definitions of infinite behaviors. Finite elements are generated by three termination constants: $\Delta$ (successful termination), $\perp$ (the minimal element of the approximation relation), and the deadlock element 0.

Behaviors can be considered as states of a transition system by interpreting a transition $u \xrightarrow{a} u'$ to mean that $u = a.u' + v$ for some behavior $v$. Compositions described by the various kinds of process algebras (including parallel and sequential composition) can be defined through continuous functions over the behavior of agents. For example, to define parallel composition, an algebraic structure on the set of actions is leveraged to define synchronization operations in the action algebra. In this paper we shall use only interleaving for the definition of parallel composition; thus the set of actions is a flat set (without synchronizing operations).

Each behavior $u \in F(A)$ over an action algebra $A$ can be represented in the form

$$u = \sum_{i \in I} a_i.u_i + \varepsilon$$

where $a_i$ are actions, $u_i$ are behaviors, the set $I$ is a finite or infinite set of indices, and the termination constant $\varepsilon$ is either $\Delta$, $\perp$, $\Delta + \perp$, or 0. If all summands in this representation are different, then this representation is unique up to the associativity and commutativity of non-deterministic choice.

## 2.2 Environments

An *environment* $E$ is an agent over an *algebra of actions* $C$ with an *insertion function*. The insertion function is a function of two arguments written as $e[u]$. The first argument $e$ is a behavior of an environment, the second is a behavior of an agent over an action algebra $A$ in a given state $u$ (the action algebra of agents may be a parameter of the environment, if needed). An insertion function is an arbitrary function continuous in both of its arguments. Its result is a new behavior of the same environment.

Using the notion of an environment, we define a new type of agent equivalence which is in general weaker than bisimilarity: *insertion equivalence* depends on an environment and its insertion function. Two agents (in given states) or behaviors $u$ and $v$ are insertion equivalent with respect to an environment $E$, written $u \sim_E v$, if for all $e \in E$, $e[u] = e[v]$. After the insertion of an agent into an environment, the new environment is ready to accept new agents to be inserted.

## 2.3 Insertion Function

To define insertion we use rewriting rules in the algebra of behaviors. Each rule has one of two forms:

$$F(x)[G(y)] \longrightarrow d.F'(z)[G'(z)]$$

and

$$F(x)[G(y)] \longrightarrow F'(z)[G'(z)]$$

where $x = (x_1, x_2, \ldots)$, $y = (y_1, y_2, \ldots)$, $z = (x_1, x_2, \ldots, y_1, y_2, \ldots)$, $x_1, x_2, \ldots$, $y_1, y_2, \ldots$ are action or behavior variables, $d \in C$ ($C$ is an action algebra), and $F, G, F', G'$ are expressions of a behavior algebra, that is, expressions built by nondeterministic choice and prefixing. The first kind of rule defines labeled transitions, the second kind of rule defines unlabeled transitions. The latter are not observable; the definition of environment behavior includes the following rule

$$\frac{e[u] \xrightarrow{*} e'[u'],\ e'[u'] \xrightarrow{d} e''[u'']}{e[u] \xrightarrow{d} e''[u'']}$$

where $\xrightarrow{*}$ denotes the transitive closure of unlabeled transition. Rewrite rules must be left linear with respect to their behavior variables, that is, no behavior

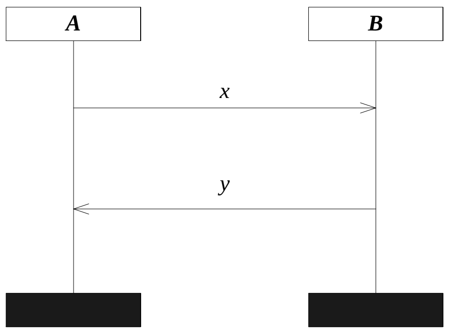


**Fig. 1.** An MSC diagram.

variable may occur more than once in the left hand side. We add the obvious rules for terminal and divergent states, as well as the following condition:

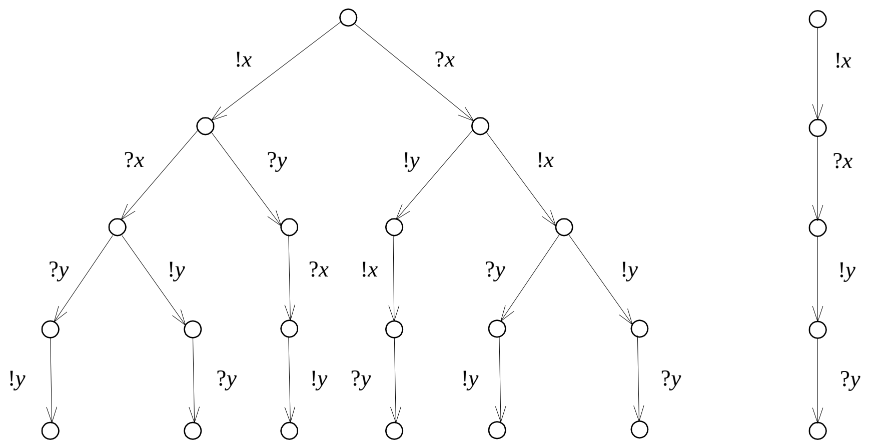
$$e[u] \xrightarrow{d} e'[u'],\ v \sqsubseteq u,\ f \sqsubseteq e,\ f[v] \not\rightarrow \quad \Rightarrow f[v] = \perp$$

where $f[v] \not\rightarrow$ means that there are no transitions from the state $f[v]$. Under these conditions, the insertion function defined will be continuous even if there are infinitely many rules. This is because to compute the function $e[u]$ one needs to know only some finite approximations of $e$ and $u$.

Rewriting rules define a non-deterministic transition relation if two different left hand sides can be matched with the same state of an environment $e[u]$ (critical pairs are allowable).

## 3 The Environment for MSC Diagrams

An MSC agent corresponds to a set of MSC diagrams; the insertion of an MSC agent into an MSC environment correspond to MSC references, which are the only way to transfer control of execution between different diagrams. The environment itself can be considered as an execution engine for a set of MSC diagrams. It synchronizes the interaction of the agents defined by those MSC diagrams and generates all possible traces for a given set of diagrams. Figure 1 shows a simple MSC diagram: Two instances, $A$ and $B$, represented by the vertical lines, interact with each other by sending asynchronous messages, indicated by the arrows running between the lines representing the instances. A message is characterized by two events: the sending of the message, and the receiving of the message. Reduced to its bare essentials, an instance is characterized by the sequence of events that occur during its lifetime. Events on an instance line are considered to be temporarily ordered, but no ordering is assumed between events occurring on different instance lines. That is, each instance executes each event on its instance line in the order in which it occurs on the line, from top to



(a) According to conventional process algebra semantics (b) MSC semantics

**Fig. 2.** Traces for the MSC diagram in Figure 1

bottom. Each instance executes its events independently of any other instance on a diagram. If we label the event of sending message $x$ as $!x$, the receiving of message $x$ as $?x$, and so on, and interpret the diagram in Figure 1 based on ordinary process algebra semantics, then the traces in Figure 2(a) are induced.

However, an MSC diagram describes the transitive closure of the orderings between the events, subject to the constraints that a message must be received before it is sent. Consequentially, the explication of the semantics of MSC diagrams must yield the single trace shown in Figure 2(b) as the meaning of the MSC of Figure 1. (An MSC diagram may contain events other than message interactions, and there are special constructs which remove the ordering constraint for an instance line, which are discussed below.)

The doctoral thesis of M.A. Reniers [6] which is based on the Algebra of Communicated Processes [7] is probably the best known explication of the semantics of MSC. Reniers defined the operators of his algebra operationally in the style of [8]. He introduced the following operators to capture the semantics of MSC diagrams: Delayed choice expresses the meaning of MSC alternatives, generalized parallel composition allows to combine several instances into a diagram, generalized weak sequential composition gives the vertical composition of events on an instance line, and iteration and unbounded repetition express looping. Altogether, 41 rules are required to specify these operators (and two additional rules for constants). The result is a highly complex semantic description which is very difficult to implement; to our knowledge, no tool has been able to implement Reniers' semantics.

In contrast, our approach begins with the simple set of traces induced by conventional process algebra semantics. Taken by itself, the agent representing the MSC diagram of Figure 1 would generate the traces shown in Figure 2(a).

However, we define an environment such that, when the agent is inserted into this environment, only the trace shown in Figure 2(b) is permitted. We call such environments *MSC environment*. After insertion into the MSC environment, the agent derived from Figure 1, $u = !x.?y \,.\, \Delta \parallel ?x.!y \,.\, \Delta$, is equivalent to a much simpler agent, $v = !x.?x.!y.?y \,.\, \Delta$. We say that $u$ and $v$ are insertion-equivalent.

In addition, our semantics for MSC is different from the formulation by Reniers in the use of additional synchronization for conditions, references, and in-line expressions.

### 3.1 The Structure of MSC Environments

To create the environment one must define both the actions of agents and environment and the insertion function. The insertion function will be described through a calculus for the transition relations of the environment. To textually express message sequence charts we rely largely on the event-oriented syntax of MSC defined in the Z.120 standard [1], as shown in Table 3.

| Feature | Textual syntax |
|---|---|
| Send message | $i :$ $m$ **from** $j$ |
| Receive message | $i :$ $m$ **to** $j$ |
| Local action | $i :$ **action** $b$ |
| Set timer | $i :$ **set** $t$ |
| Reset timer | $i :$ **reset** $t$ |
| Timeout | $i :$ **timeout** $t$ |
| Instance start | $i :$ **instance** |
| Instance creation | $i :$ **create** $j$ |
| Instance stop | $i :$ **stop** |
| Condition | $J :$ **condition** $b$ |
| Reference | $J :$ **reference** $z$ |
| Local condition | $i :$ **cond** $b(J)$ |
| Local reference | $i :$ **ref** $z(J)$ |

**Fig. 3.** MSC textual syntax

In Table 3, $i$ and $j$ are instance names, $J$ is a set of instance names, $m$ is a message expression, $b$ is an expression describing an action or condition, $t$ is a timer expression, and $z$ is an MSC reference expression. (The detailed syntax of the expressions is not relevant to this paper. For example: A message expression $m$ may contain parameters describing structural components of the message. A timer expression $t$ may contain a duration. While in MSC conditions and actions are not further interpreted, in many practical applications these are given specific meaning and syntax.) Note that the last two forms are not part of [1] and are not user-visible.

We assume, without loss in generality, that all names used in an MSC diagram are distinct. Each instance is executing in the context of an agent deriving from an MSC diagram. If $i$ is an instance, **agent**$(i)$ is the agent executing this instance. Each event belongs to some instance, and if $a$ is an event, then **inst**$(a)$ denotes the instance to which this event belongs.

**Message events** represent communication events in the system: Instances may receive messages, or they may send messages to other instances. The names **lost** and **found** are always-defined instance names to allow representation of incomplete events: A lost message is a message that is sent but will never be received by another instance. A found message is a message where the sender is unknown. The name **env** is always defined and refers to the environment of the instance containing this event; messages may be send to the environment or may be received from the environment.

**Local events** do not impact other instances. They may describe arbitrary, not further defined actions, the setting or expiration of timers, as well as timer resets.

**Instance events** describe the life-time of an instance. An instance may be created by another instance, and an instance may stop its execution. The start of a sequence of events belonging to an instance is indicated by the instance start event.

**Control events** synchronize conditions and references across a set of instances. These events are not observable from the outside environment and establish that events preceding a condition or reference on a given instance have completed. In [1], control events are represented as "multi-instance events", while we represent the occurrence of a control event on each instance as a separate event (referred to as *local condition* and *local reference* in Figure 3; the conversion from multi-instance events to local events is performed by the translation rules in Section 4). The set of instances $J$ indicates all instances that are synchronized by this control event, where each instance $i \in J$.

Due to space limitations, in the following discussion gates and causal orderings are not considered, but the corresponding extensions are straightforward.

Note that while [1] speaks of instances as comprised of a partially ordered sequence of events, in the explication of the semantics of message sequence charts we will speak of sequences of actions (to remain consistent with the terminology of process algebra). We shall use the terms "event" and "action" interchangeably, when there is no danger of confusion.

**Agents** are composed from actions (events) by the standard algebraic operations (such as prefixing, sequential and parallel compositions, or non-deterministic choice), considered up to bisimilarity. The transition rules for these operations are as usual, but parallel composition is interpreted as interleaving.

Using this syntax, we can express the instance $A$ in Figure 1 as

$$A:\ \textbf{instance}\,.\,A:\ x\ \textbf{to}\ B\,.\,A:\ y\ \textbf{from}\ B\,.\,A:\ \textbf{stop}\,.\,\Delta$$

and the instance $B$ as

$$B:\ \textbf{instance}\,.\,B:\ x\ \textbf{from}\ A\,.\,B:\ y\ \textbf{to}\ A\,.\,B:\ \textbf{stop}\,.\,\Delta$$

The environment states of an MSC environment will be represented by the tuple of functions $\langle \mathcal{O}, \mathcal{S}, \mathcal{R}, \mathcal{U} \rangle$.

$\mathcal{O}$ is a partial function of three arguments $m$, $i$, $j$, where $m$ is a message expression, and $i$ and $j$ are instances. This function yields values in the set of positive integers. $\mathcal{O}(m, i, j) = k$ means that earlier $k$ message events $i :\ m$ **to** $j$ occurred for which there are no corresponding receiving message events pending. (If $\mathcal{O}(m, i, j)$ is undefined, there are no in message events pending.)

$\mathcal{S}$ is a partial function of two arguments $y$ and $J$. The first argument is a condition or reference expression, the second is a set of instances. $\mathcal{S}(y, J)$ represents a nonempty subset of the set $J$. $\mathcal{S}(y, J) = I$ means that earlier a control event $i :$ **cond** $y(J)$ or a reference event $i :$ **ref** $y(J)$ had been executed, for all instances $i \in I$. The condition or reference event is attached to all instances in $J$. In other words, $\mathcal{S}(y, J)$ is the set of all instances which have already been synchronized by the condition or reference.

$\mathcal{R}$ is a partial function from a set of reference names. If $\mathcal{R}(x) = J$, where $J$ is a set of instances, then the agent corresponding to reference expression $x$ attached to the instances in $J$ is currently executing. Several references to the same MSC diagram can be executed at the same time.

$\mathcal{U}$ is a function defined on the set of reference names. $\mathcal{U}(x) = I$ is a set of all instances active in the agent deriving from the MSC diagram denoted by reference expression $x$. The condition $\mathcal{U}(x) = \emptyset$ is a termination condition for the agent defined by the MSC diagram.

The states of the MSC environment are expressions $e[P]$ where $e$ is an environment state and $P$ is an MSC agent. The insertion function is defined so that $(e[P])[Q] = e[P \parallel Q]$. The environment in a state $e[\Delta]$ is called the empty environment if $e$ is an environment state. Initial states are state expressions $e[P]$ where $P$ is an MSC agent. The set of environment states is restricted to the set of states reachable from the possible initial states. This restriction is consistent with the insertion function, because if $e'[P']$ is reachable from $e[P]$ then $e'[P' \parallel Q]$ is reachable from $e[P \parallel Q]$.

## 3.2 Insertion function for MSC environments

An environment state consists of partial functions; in the following we make extensive use of partial functions transformations: Let $f$ be a partial function, then $\mathrm{Dom}(f)$ denotes the domain of this function. For any $x$, $f(x) = \perp$ if $\perp \notin \mathrm{Dom}(f)$, that is, if $f(x)$ is not defined. The operator $[x := y]$ transforms $f$ to a new function $f'$ such that for all $z$, if $z \neq x$, $f'(z) = f(z)$, and $f'(x) = y$. Note that $\mathrm{Dom}(f[x := y]) = \mathrm{Dom}(f) \cup \{x\}$, as $x$ may or may not be in $\mathrm{Dom}(f)$. The operator $[\mathrm{Dom}\backslash E]$ deletes the set $E$ from the domain of $f$, that is, it transforms $f$ to a new function $f'$ such that $\mathrm{Dom}(f') = \mathrm{Dom}(f)\backslash E$ where $f$ is an extension of $f'$.

**Suspended instances and actions.** Let $e = \langle \mathcal{O}, \mathcal{S}, \mathcal{R}, \mathcal{U} \rangle$. Instance $k$ is called suspended in environment state $e$ if one of the following conditions is true, for some name $y$:

1. $k \in \mathcal{S}(y, J)$, where $k \in J$ or
2. $k \in \mathcal{R}(y)$

Suspended instances are synchronized by either a condition or by a reference. Action $a$ is suspended in an environment state $e$ if one of the following conditions is true:

1. $\mathbf{inst}(a)$ is suspended in $e$,
2. $i = \mathbf{inst}(a)$ and $k \notin \mathcal{R}(\mathbf{agent}(i))$.
3. for some message $m$ and some instances $i$ and $j$, $a = i :\ m$ **from** $j$ and $\mathcal{O}(m, j, i) \not> 0$.

Each reference is invoked from the main diagram or some other uniquely identifiable executing diagram (invoked through another reference). All references are executing in parallel. The following restriction must be satisfied: If an instance $i$ is active in an executing reference $x$, it must be suspended in all other currently executing references. This restriction will be satisfied if each instance used in the reference is attached to this reference and if an instance that is shared by two references is attached to both.

In the rules below, the environment state for $e$ is $\langle \mathcal{O}, \mathcal{S}, \mathcal{R}, \mathcal{U} \rangle$; the environment state for $e'$ is $\langle \mathcal{O}', \mathcal{S}', \mathcal{R}', \mathcal{U}' \rangle$.

The necessary conditions for the existence of an environment state $e'$ such that $e \xrightarrow{a} e'$ is that action $a$ is not suspended in $e$ for $x$. This condition is assumed for all rules below.

**General rules.**

$$s \longrightarrow e' \Rightarrow e[P] \longrightarrow e'[P] \tag{1}$$

$$e[P + Q] = e[P] + e[Q] \tag{2}$$

$$\frac{e[\Delta] \xrightarrow{a} e'[Q],\ P \xrightarrow{a} P'}{e[P] \xrightarrow{a'} e'[q \parallel P']} \tag{3}$$

for an observable action $a$ and

$$\frac{e[\Delta] \xrightarrow{a} e'[\Delta],\ P \xrightarrow{a} P'}{e[P] \longrightarrow e'[P']} \tag{4}$$

for a non-observable action. (An action is non-observable if it is an incomplete control action, see below, otherwise an action is observable. Note that the notion of observability depends on the state of the environment.) In rules (3) and (4), we assume that there are no hidden transitions for the environment state $e[\Delta]$. Action $a'$ in (3) is different from $a$ only if $a$ is a control action. In this case, if $a = i : \mathbf{cond}\ y(J)$ or $a = i : \mathbf{ref}\ y(J)$, then the observed action $a' = J : \mathbf{reference}\ y$ or $a' = J : \mathbf{condition}\ y$, respectively.

**Rules for messages and local actions.** For a non-suspended message action or local action $a$, the transition

$$e[\Delta] \xrightarrow{a} e'[\Delta]$$

is always possible. Only function $\mathcal{O}$ is changed as a result of this action, as follows:

$$\left.\begin{array}{l} a = i:\ \textbf{action}\ b \\ a = i:\ m\ \textbf{to lost} \\ a = j:\ m\ \textbf{from found} \end{array}\right\} \Rightarrow \mathcal{O}' = \mathcal{O}$$

$$\begin{array}{l} a = i:\ m\ \textbf{to}\ j, j \neq \textbf{lost} \Rightarrow \mathcal{O}' = \mathcal{O}[(m,i,j) := n+1] \\ a = j:\ m\ \textbf{from}\ i, i \neq \textbf{found} \Rightarrow \mathcal{O}' = \mathcal{O}[(m,i,j) := n-1] \end{array}$$

where $n = 0$ if $\mathcal{O}(m,i,j) = \bot$, otherwise $n = \mathcal{O}(m,i,j)$.

**Rules for instance actions.** For a non-suspended instance action $a$ the transition

$$e[\Delta] \xrightarrow{a} e'[\Delta]$$

is possible under the conditions below. Only function $\mathcal{U}$ is changed by this action. Let $x = \textbf{agent}(i)$, where $i$ is an instance, then

$$\begin{array}{l} a = i:\ \textbf{instance}, i \notin \mathcal{U}(x) \Rightarrow \mathcal{U}' = \mathcal{U}[x := \mathcal{U}(x) \cup \{i\}] \\ a = i:\ \textbf{create}\ j, j \notin \mathcal{U}(x) \Rightarrow \mathcal{U}' = \mathcal{U}[x := \mathcal{U}(x) \cup \{j\}] \\ a = i:\ \textbf{stop}, i \in \mathcal{U}(x) \Rightarrow \mathcal{U}' = \mathcal{U}[x := \mathcal{U}(x) \backslash \{j\}] \end{array}$$

The cases which are not covered by these conditions are forbidden.

**Rules for control actions.** Finally, consider control actions $i : \textbf{cond}\ y(J)$ and $i : \textbf{ref}\ y(J)$, where $y$ is a name. A control action is complete in a state $e$ if $J = \{i\}$ or $\mathcal{S}(y,J) \cup \{i\} = J$. Otherwise the action is incomplete. For a non-suspended incomplete control action $a$ a transition

$$e[\Delta] \xrightarrow{a} e'[\Delta]$$

is always possible. Only function $\mathcal{S}$ is changed as follows:

$$\mathcal{S}' = \mathcal{S}[(y,J) := \mathcal{S}(y,J) \cup \{i\}]$$

For a complete control action $a = i : \textbf{cond}\ y(J)$, a transition

$$e[\Delta] \xrightarrow{a} e'[\Delta]$$

is always possible, and $\mathcal{S}$ is changed as follows:

$$\mathcal{S}' = \mathcal{S}[\mathrm{Dom} \backslash (y,J)]$$

For a complete control action $a = i : \textbf{ref}\ y(J)$, the transition is

$$e[\Delta] \xrightarrow{a} e'[Q]$$

where $Q$ is a new agent derived from the MSC reference expression $y$. The functions $\mathcal{R}$ and $\mathcal{S}$ are changed as follows:

$$\begin{array}{l} \mathcal{R}' = \mathcal{R}[x := J], \text{ where } x = \textbf{agent}(i) \\ \mathcal{S}' = \mathcal{S}[\mathrm{Dom} \backslash (y,J)] \end{array}$$

Other functions do not change.

**Terminate reference execution rule.** The following transition is always possible

$$\mathcal{U}(x) = \emptyset \Rightarrow e[\Delta] \longrightarrow e'[\Delta]$$

Only functions $\mathcal{U}$ and $\mathcal{R}$ are changed when this transition is executed:

$$\begin{array}{l} \mathcal{U}' = \mathcal{U}[\mathrm{Dom} \backslash x] \\ \mathcal{R}' = \mathcal{R}[\mathrm{Dom} \backslash x] \end{array}$$

$$\begin{aligned}
&\mathcal{F}(\mathbf{loop}\langle m,n\rangle(E),P_1,P_2,\ldots) &&= \mathbf{loop}(m,n,\mathcal{F}(E,P_1,P_2,\ldots))\\
&\mathcal{F}(E_1\ \mathbf{alt}\ E_2\ \mathbf{alt}\ldots,P_1,P_2,\ldots) &&= \mathcal{F}(E_1,P_1,P_2,\ldots)+\mathcal{F}(E_2,P_1,P_2,\ldots)+\ldots\\
&\mathcal{F}(\mathbf{opt}\ E,P_1,P_2,\ldots) &&= \mathcal{F}(E,P_1,P_2,\ldots)+\Delta\\
&\mathcal{F}(E_1\ \mathbf{par}\ E_2\ \mathbf{par}\ldots,P_1,P_2,\ldots) &&= \mathcal{F}(E_1,P_1,P_2,\ldots)\parallel\mathcal{F}(E_2,P_1,P_2,\ldots)\parallel\ldots\\
&\mathcal{F}(E_1\ \mathbf{seq}\ E_2\ \mathbf{seq}\ldots,P_1,P_2,\ldots) &&= \mathcal{F}(E_1,P_1,P_2,\ldots)\,;\mathcal{F}(E_2,P_1,P_2,\ldots)\,;\ldots\\
&\mathcal{F}(\mathbf{exc}\ E,P_1,P_2,\ldots) &&= (\mathcal{F}(E,P_1,P_2,\ldots)\,;0)+\Delta\\
&\mathcal{F}(x_i,P_1,P_2,\ldots) &&= P_i\\
\\
&\mathbf{loop}(0,0,G) &&= \Delta\\
&\mathbf{loop}(0,\mathbf{inf},G) &&= (G\,;\mathbf{loop}(0,\mathbf{inf},G))+\Delta\\
&\mathbf{loop}(m,\mathbf{inf},G) &&= (G\,;\mathbf{loop}(m-1,\mathbf{inf},G))\\
&\mathbf{loop}(0,n,G) &&= (G\,;\mathbf{loop}(0,n-1,G))+\Delta\\
&\mathbf{loop}(m,n,G) &&= (G\,;\mathbf{loop}(m-1,n-1,G))
\end{aligned}$$

**Fig. 4.** Translation rules for MSC reference expressions.

## 4 Translation of MSC Diagrams to MSC Environment Expressions

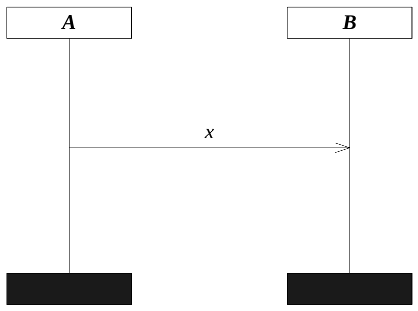
Each MSC diagram in a specification is considered and translated individually. The result of a diagram translation is an expression $P$ describing the agent representing the MSC diagram. We begin by translating all HMSC diagrams to the equivalent MSC diagrams involving in-line expressions. Then we can assume that the body of each MSC diagram contains only event definitions. We change all in-line expressions to named reference expressions introducing separate MSC diagrams for the MSC bodies. After this change, the only instance events involving multiple instances are conditions and references.

The condition $i_1, i_2, \ldots$ : **condition** $y$ attached to instances $i_1$, $i_2$, …is changed to the set of events $i_1$ : **cond** $y(J)$, $i_2$ : **cond** $y(J)$, …, where $J = \{i_1, i_2, \ldots\}$ and these new events are attached to the corresponding instances $i_1$, $i_2$, and so on.

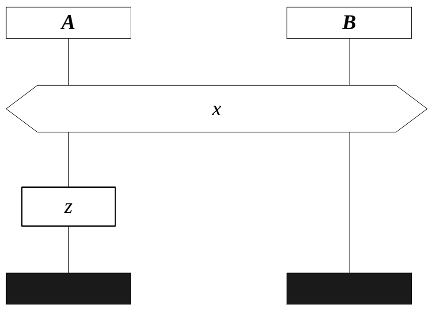
To translate a reference event $i_1, i_2, \ldots$ : **reference** $y$ attached to instances $i_1$, $i_2$, …, where $y$ is an MSC reference expression containing names of the MSC diagrams $x_1$, $x_2$, …, first translate the diagrams $x_1$, $x_2$, … obtaining the set of MSC agents $P_1$, $P_2$, …, and then compute the function $\mathcal{F}(y, P_1, P_2, \ldots)$ using the definitions shown in Figure 4.

After this translation a name $z$ is used as the name of the new agent resulting from the translation of this diagram by evaluating $\mathcal{F}(y, P_1, P_2, \ldots)$ and each reference event is changed to the set of local events $i_1$ : **ref** $z(J)$, $i_2$ : **ref** $z(J)$, …, where $J = \{i_1, i_2, \ldots\}$, and these actions are attached to the corresponding instances $i_1$, $i_2$, and so on.

The translation of a diagram $x$ without in-line expressions containing only local events on the instances $i_1$, $i_2$,…, is the agent $p_1 \parallel p_2 \parallel \ldots$, where $p_k$ is a sequential composition of local events belonging to the instance $i_k$ if there are no coregions on this instance. For coregions a parallel composition is used instead



(a) Message passing



(b) Condition

**Fig. 5.** Two simple MSC diagrams

of sequential composition. Actions $x:$ **instance** and $x:$ **stop** are added at the beginning and the end of this agent, respectively.

The names of agents are used to create the initial state $e$ and the environment expression $e[P]$, where $P$ is the translation of the main diagram. If there is no main diagram, the translation is $e[\Delta]$. The function **loop** used above is computed at execution time.

We use strict sequential composition instead of weak sequential composition as it was defined in Reniers' semantics. First, in engineering practice events are usually considered strictly ordered. Secondly, the computation of a weak product has high complexity and is unsolvable for recursive diagrams.

## 5 Examples

Consider the MSC diagram in Figure 5(a). In algebraic notation, we can express the instances $A$ and $B$ as

$$A = A:\ x\ \mathbf{to}\ B\,.\,\Delta$$
$$B = B:\ x\ \mathbf{from}\ A\,.\,\Delta$$

To be more precise, $A$ and $B$ begin with an instance start event and end in a stop event, but as these do not impact the examples they are omitted for conciseness.

The semantics of the example MSC is $P = e[A \parallel B]$, that is

$$P = e[A:\ x\ \mathbf{to}\ B\,.\,\Delta \parallel B:\ x\ \mathbf{from}\ A\,.\,\Delta]$$

We can expand this to

$$P = e[A:\ x\ \mathbf{to}\ B\,.\,B:\ x\ \mathbf{from}\ A\,.\,\Delta + B:\ x\ \mathbf{from}\ A\,.\,A:\ x\ \mathbf{to}\ B\,.\,\Delta]$$

which by rule (2) can be rewritten to

$$P = e[A:\ x\ \mathbf{to}\ B\,.\,B:\ x\ \mathbf{from}\ A\,.\,\Delta]$$
$$+\, e[B:\ x\ \mathbf{from}\ A\,.\,A:\ x\ \mathbf{to}\ B\,.\,\Delta]$$

Now let $a$ be $A:\ x$ **to** $B$. Since this is an output action, and $a$ is not lost, the transition

$$e[\Delta] \xrightarrow{a} e'[\Delta]$$

is always possible. Let $Q$ be $\Delta$, then we can apply rule (3), and obtain

$$\begin{aligned} P = {} & A:\ x\ \mathbf{to}\ B\ .\ e'[\Delta \parallel B:\ x\ \mathbf{from}\ A\ .\ \Delta] \\ & + e[B:\ x\ \mathbf{from}\ A\ .\ A:\ x\ \mathbf{to}\ B\ .\ \Delta] \end{aligned}$$

where $e'$ is obtained from $e$ by setting $\mathcal{O}' = \mathcal{O}[(x, A, B) := 1]$ as given by the rules for message actions. Using the algebraic law $\Delta \parallel P = P$,

$$\begin{aligned} P = {} & A:\ x\ \mathbf{to}\ B\ .\ e'[B:\ x\ \mathbf{from}\ A\ .\ \Delta] \\ & + e[B:\ x\ \mathbf{from}\ A\ .\ A:\ x\ \mathbf{to}\ B\ .\ \Delta] \end{aligned}$$

We can now apply the rule for input actions together with rule (3) and obtain

$$\begin{aligned} P = {} & A:\ x\ \mathbf{to}\ B\ .\ B:\ x\ \mathbf{from}\ A\ .\ e''[\Delta] \\ & + e[B:\ x\ \mathbf{from}\ A\ .\ A:\ x\ \mathbf{to}\ B\ .\ \Delta] \end{aligned}$$

where $e''$ is obtained from $e'$ by setting $\mathcal{O}'' = \mathcal{O}'[(x, A, B) := 0]$. By $e[\Delta] = \Delta$,

$$\begin{aligned} P = {} & A:\ x\ \mathbf{to}\ B\ .\ B:\ x\ \mathbf{from}\ A\ .\ \Delta \\ & + e[B:\ x\ \mathbf{from}\ A\ .\ A:\ x\ \mathbf{to}\ B\ .\ \Delta] \end{aligned}$$

Note that $x$ is suspended in $e$, as there is no $(x, A, B) \in \mathrm{Dom}(\mathcal{O})$ such that $\mathcal{O}(x, a, b) > 0$. Therefore, there is no transition possible for the agent state $e[B:\ x\ \mathbf{from}\ A\ .\ A:\ x\ \mathbf{to}\ B\ .\ \Delta]$, and so $e[B:\ x\ \mathbf{from}\ A\ .\ A:\ x\ \mathbf{to}\ B\ .\ \Delta] = 0$. Thus we obtain

$$P = A:\ x\ \mathbf{to}\ B\ .\ B:\ x\ \mathbf{from}\ A\ .\ \Delta + 0$$

and by $P + 0 = P$, we finally arrive at

$$P = A:\ x\ \mathbf{to}\ B\ .\ B:\ x\ \mathbf{from}\ A\ .\ \Delta$$

In other words, when we insert the agent $A \parallel B$ into the MSC environment, the behavior of that agent is restricted to the one allowed by the MSC semantics: the instance $A$ first sends message $x$, and then instance $B$ receives this message. The alternative behavior of the agent $A \parallel B$, namely $B$ receiving the message $x$ before $A$ has sent $x$, is not permitted by the environment.

The MSC diagram in Figure 5(b) is expressed in algebraic notation as

$$\begin{aligned} A &= A : \mathbf{cond}\ x(\{A, B\})\ .\ A:\ \mathbf{action}\ z\ .\ \Delta \\ B &= B : \mathbf{cond}\ x(\{A, B\})\ .\ \Delta \end{aligned}$$

The expanded semantics of this diagram, as rewritten by rule (2) is

$$\begin{aligned} P = {} & e[A : \mathbf{cond}\ x(\{A, B\})\ .\ A:\ \mathbf{action}\ z\ .\ B : \mathbf{cond}\ x(\{A, B\})\ .\ \Delta] \\ & + e[A : \mathbf{cond}\ x(\{A, B\})\ .\ B : \mathbf{cond}\ x(\{A, B\})\ .\ A:\ \mathbf{action}\ z\ .\ \Delta] \\ & + e[B : \mathbf{cond}\ x(\{A, B\})\ .\ A : \mathbf{cond}\ x(\{A, B\})\ .\ A:\ \mathbf{action}\ z\ .\ \Delta] \end{aligned}$$

If we apply the rule for control actions together with rule (4) to the first summand, we obtain

$$\begin{aligned} P = {} & e'[A:\ \mathbf{action}\ z\ .\ B : \mathbf{cond}\ x(\{A, B\})\ .\ \Delta] \\ & + e[A : \mathbf{cond}\ x(\{A, B\})\ .\ B : \mathbf{cond}\ x(\{A, B\})\ .\ A:\ \mathbf{action}\ z\ .\ \Delta] \\ & + e[B : \mathbf{cond}\ x(\{A, B\})\ .\ A : \mathbf{cond}\ x(\{A, B\})\ .\ A:\ \mathbf{action}\ z\ .\ \Delta] \end{aligned}$$

where $\mathcal{S}(x, \{A, B\}) = \{A\}$ in $e'$. Note that this action is not observed, by rule (4). However, now $A \in \mathcal{S}(x, \{A, B\})$, and therefore, $A$ is suspended in environment $e$, and no further behavior is possible for instance $A$. If we instead begin with the second summand, we have

$$\begin{aligned} P = {} & e[A : \mathbf{cond}\ x(\{A, B\})\ .\ A:\ \mathbf{action}\ z\ .\ B : \mathbf{cond}\ x(\{A, B\})\ .\ \Delta] \\ & + e'[B : \mathbf{cond}\ x(\{A, B\})\ .\ A:\ \mathbf{action}\ z\ .\ \Delta] \\ & + e[B : \mathbf{cond}\ x(\{A, B\})\ .\ A : \mathbf{cond}\ x(\{A, B\})\ .\ A:\ \mathbf{action}\ z\ .\ \Delta] \end{aligned}$$

where $\mathcal{S}(x, \{A, B\}) = \{A\}$ in $e'$. Note that $B$ : **cond** $x(\{A, B\})$ is complete in state $e'$, and therefore we can transition to

$$\begin{aligned} P = {} & e[A : \textbf{cond } x(\{A,B\}) \,.\, A : \textbf{ action } z \,.\, B : \textbf{cond } x(\{A,B\}) \,.\, \Delta] \\ & + A, B : \textbf{condition } x \,.\, e''[A : \textbf{ action } z \,.\, \Delta] \\ & + e[B : \textbf{cond } x(\{A,B\}) \,.\, A : \textbf{cond } x(\{A,B\}) \,.\, A : \textbf{ action } z \,.\, \Delta] \end{aligned}$$

where $\mathcal{S}(x, \{A, B\}) = \emptyset$ in $e''$. Now we can continue execution with the local action:

$$\begin{aligned} P = {} & e[A : \textbf{cond } x(\{A,B\}) \,.\, A : \textbf{ action } z \,.\, B : \textbf{cond } x(\{A,B\}) \,.\, \Delta] \\ & + A, B : \textbf{condition } x \,.\, A : \textbf{ action } z \,.\, e'''[\Delta] \\ & + e[B : \textbf{cond } x(\{A,B\}) \,.\, A : \textbf{cond } x(\{A,B\}) \,.\, A : \textbf{ action } z \,.\, \Delta] \end{aligned}$$

leaving the state of $e'''$ unchanged. Applying the same reasoning to the final summand, and using algebraic laws as in the example above, we determine the meaning of this MSC diagram to be

$$P = A, B : \textbf{condition } x \,.\, A : \textbf{ action } z \,.\, \Delta$$

Again we can see that when we inserted the agent for MSC diagram $P$, the behaviors not licensed by the MSC semantics have been disallowed.

## 6 Conclusion

In [9–11] we have presented an environment to verify the consistency and completeness of behavioral descriptions expressed in the form of message sequence charts. This environment has been deployed for the verification of telecommunications applications and has enabled us to verify systems that have not been amenable to other techniques, such as model checking, due to the large state space induced by the specifications of these systems. In this environment, automated reasoning is performed over the semantic representation of message sequence charts.

In this paper, we explicated the formal semantics of message sequence charts as leveraged in our environment. The use of the semantic representation in tools imposed two constraints on the formal definition: The presentation had to be close to feasible and efficient implementations, as this makes a correct implementation of the semantics more likely. More importantly, the presentation had to be flexible to introduce variations into the semantics with relative ease. Different subject domains require variation in the interpretation of MSC diagrams to account for domain-specific differences. For example, while many telecommunication applications interact with their environment asynchronously, when specifying embedded processors or applications interacting with the system bus, communication is synchronous. While in many situations, the agents comprising a system are executing independently and in parallel, when modeling applications on an embedded operating system, these agents are executing in a sequential environment. Our experience has taught us that reasoning about these systems is more efficient (which is crucial in light of the large state spaces) when the underlying concurrency semantics and interaction semantics are represented in a manner close to the characteristics of the subject domain. Separating the presentation of the semantics in a small but well-understood process algebra core and the insertion function allowed us to adjust the detailed semantics of a system specification to the actual behavior of the represented systems.

We have further developed a number of extensions to standard message sequence charts. We have added temporal concepts to message sequence charts, such as time intervals between events and the specific timing of events. We have further developed different communication styles between instances of message sequence charts, such as queuing behavior or bounded buffers. By specifying appropriate environments and an insertion function it was straightforward to capture the meaning of these extensions and immediately integrate them in our tools.

We believe that it would have been significantly more difficult to implement our tools on a conventional semantic model, such as the process algebra presentation by Reniers.

## References

1. ITU-T. Recommendation Z.120: Message Sequence Charts (MSC). Geneva, October 1996.
2. ITU-T. Recommendation Z.120: Message Sequence Charts (MSC). Geneva, October 2000.
3. A. Letichevsky and D. Gilbert. A general theory of action languages. Cybernetics and System Analysis, 1, 1998.
4. A. Letichevsky and D. Gilbert. A Model for Interaction of Agents and Environments. In D. Bert, C. Choppy, P. Moses, editors. Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer Science 1827, Springer, 1999.
5. A. Letichevsky and D. Gilbert. A Model for Interaction of Agents and Environments. In Selected papers from the 14th International Workshop on Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer Science, 1827, 2004.
6. M.A. Reniers. Message Sequence Charts: Syntax and Semantics. PhD Thesis, Eindhoven University of Technology, June 1999.
7. J.A. Bergstra, A. Ponse, S.A. Smolka, editors. Handbook of Process Algebra. North-Holland, 2001.
8. G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report, DIAMI FN-19, Aarhus University, 1981.
9. S. Baranov, C. Jervis, V. Kotlyarov, A. Letichevsky, and T. Weigert. Leveraging UML to Deliver Correct Telecom Applications. In L. Lavagno, G. Martin, and B. Selic, editors. UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Publishers, Amsterdam, 2003.
10. A. Letichevsky, J. Kapitonova, A. Letichevsky Jr., V. Volkov, S. Baranov, V. Kotlyarov, T. Weigert. Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. Computer Networks, 47, forthcoming in 2005.
11. J. Kapitonova, A. Letichevsky, V. Volkov, and T. Weigert. Validation of Embedded Systems. In R. Zurawski, editor. The Embedded Systems Handbook. CRC Press, Miami, forthcoming in 2005.