Message Sequence Charts in the Development Process – Roles and Limitations

Manfred Broy

Institut für Informatik
Technische Universität München
D-80290 München, Germany

Abstract

Message Sequence Charts (MSCs) are a technique to describe patterns of interactions between the components of interactive distributed systems by specific diagrams. MSCs have evolved in telecommunication applications and become very popular in the design of software architectures and, generally, of distributed software systems. They are used frequently to describe scenarios of interactions illustrating instances of use cases. Nevertheless, both the semantics of MSCs as a technique of specification and their methodological and technical role in the development process have not been precisely and sufficiently clarified, so far. In this paper, we discuss the systematic application of MSCs in the software development process.

1 Introduction

Today message sequence charts (MSCs, see [13]) or extended event traces (EETs) have become a well-accepted and widely used description technique in software and systems engineering. They have been incorporated into a number of popular system modeling methods (see [18], [9], [15], [2], [16]). They are used to illustrate scenarios of interaction between distributed interacting components cooperating in networks. They are helpful to illustrate the fundamental patterns of cooperation and interaction in distributed systems. MSCs are widely used, for instance, in telecommunication applications for illustrating protocols as well as the cooperation of switching components. MSCs have also found their way into business application development.

Nevertheless, there remain crucial methodological questions in connection with MSCs, not answered so far in a satisfactory way. For instance, the idea of describing the complete behavior of a system by a set of MSCs seems unrealistic and naive, at least in cases of systems with complex interaction patterns and high combinatorial complexity. Too many (even infinitely many) scenarios might be necessary. Therefore, the naive idea of covering all possible interaction scenarios by MSCs may prove unrealistic. A realistic idea, however, seems to be the specification of a set of representative instances of system behavior patterns by MSCs.

The question in which development phases MSCs might be actually helpful has not been investigated systematically so far. In spite of their popularity, questions of their systematic usage
in the development process have not been tackled in a satisfactory way in the software engineering literature (see also [7], [1]).

We start with a discussion of the state of the art of MSCs, the type of systems they model, and their role in the development process. We introduce a reference model of a system in Section 2. After that we discuss shortly the semantics of MSCs, first for deterministic systems and after that for nondeterministic systems. Then we deal with control and data states in connection with MSCs in the development process and discuss the interleaving of MSCs using projections. Finally, we discuss the methodological role of MSCs.

2 The System Model for Message Sequence Charts

Typically, MSCs are used to describe a set of characteristic instances of system interactions. When used this way, MSCs describe properties of a system to be constructed or to be analyzed in requirements engineering and in system design. This view suggests to formalize MSCs as representing logical properties of systems. Consequently, we translate sets of MSCs into logical predicates describing properties of system components.

2.1 Interactive System Models

MSCs are typically used to describe patterns of interactions for distributed systems and software architectures. MSCs describe the mutual interaction between the components of a system and also their interaction with their environment. In particular, MSCs are well suited to describe systems with a concurrent control flow that communicate by message exchange.

For the interpretation of MSCs, the particular concept of interaction of the components of the described system is crucial. We identify three basic concepts of communication in distributed systems that interact by message exchange:

- **Asynchronous communication** (message asynchrony): a message is sent as soon as the sender is ready, independently of the question whether a receiver is ready to receive it. Sent messages are buffered (by the communication mechanism) and can be received by the receiver at any later time; if a receiver wants to receive a message but no message was sent it has to wait. However, senders never have to wait (see [10], [17]) until receivers are ready since messages may be buffered.

- **Synchronous communication** (message synchrony, rendezvous, handshake communication): a message can be sent only if both the sender and the receiver are simultaneously ready to communicate; if only one of them (receiver or sender) is ready for communication, it has to wait until a communication partner gets ready (see [5], [6]).

- **Time synchronous communication** (perfect synchrony): several interaction steps (signals or atomic events) are conceptually gathered into one time slot; this way systems are modeled with the help of sequences of sets of events (see [8] as a well-known example).
In principle, message sequence charts can be used to describe the interaction of components for all the three classes of systems introduced above (see [14] for a trace semantics both for synchronous and asynchronous communication).

In the following, we work with asynchronous message passing since for this model message sequence charts seem best suited as a description technique. We follow the system model of [Broy 96] and base our approach on a concept of a component that communicates messages asynchronously in a synchronous time frame with its environment via named and typed channels.

2.2 Selected System Model

We think of a system as being composed of a number of subsystems that we call its components. In fact, a composed system itself is and can be used as a component again as part of a larger system. A component is a unit with a precisely specified interface and an encapsulated state. Via its interface it is connected to and communicates with its environment. In this section, we shortly introduce a simple, very abstract mathematical notion of a system component.

2.2.1 Components

A (system) component is an active information processing unit that communicates asynchronously with its environment through a set of input and output channels. This communication takes place within a global (discrete) time frame.

Let \( I \) be the set of input channels and \( O \) be the set of output channels of the component \( f \). With every channel in the channel set \( I \cup O \) we associate a data type indicating the type of messages sent along that channel. Then by \( (I, O) \) the syntactic interface of a system component is described. A graphical representation of a component with its syntactic interface and individual channel types is shown in Fig. 1.

By \( M^\omega = M^* \cup M^\infty \) we denote the set of streams of elements from the set \( M \) which are finite or infinite sequences of elements from \( M \). A stream represents the sequence of messages sent over a channel during the lifetime of a system. Of course, in concrete systems this communication takes place in a time frame. In fact, it is often convenient to be able to refer to this time. Therefore we work with timed streams.

Our model of time is extremely simple. We assume that time is represented by an infinite sequence of time intervals of equal length. In each interval on each channel a finite, possibly empty sequence of messages is transmitted. By \( (M^*)^- \) we denote the set of infinite streams of sequences of elements of set \( M \). Mathematically, a timed stream in \( (M^*)^- \) can also be understood as a function \( IN \setminus \{0\} \to M^* \).
Throughout this paper we work with a few simple notations for streams. We use, in particular, the following notations for timed streams $x$:

- $z \cdot x$: concatenation of a sequence or stream $z$ to a stream $x$,
- $x.k$: $k$-th sequence in the stream $x$,
- $x \downarrow k$: prefix of the first $k$ sequences in the timed stream $x$,
- $x$: finite or infinite (nontimed) stream as the result of concatenating all sequences in $x$.

Let $C$ be a set of channels with types assigned by the function

$$\text{type}: C \rightarrow T$$

Here $T$ is a set of types $\tau \in T$ which are carrier sets of data elements. Let $M$ be the universe of all messages. This means

$$M = \bigcup \{\tau: \tau \in T\}$$

We define the valuations of the set $C$ of channels by functions

$$x: C \rightarrow (M^*)^\infty$$

where for each channel $c \in C$ with $\text{type}(c) = \tau$ the elements of the stream $x.c$ are of the type $\tau$ (throughout the paper we denote the application of a function $f$ to an argument $b$ not only by $f(b)$ but also by $f.b$ to save parenthesis):

$$x.c \in (\tau^*)^\infty$$

The set of valuations of the channels in $C$ is denoted by $\hat{C}$. Let in the following $I$ and $O$ be sets of typed channels.

Let $r \in (M^*)^\infty$; $\bar{r}$ is called the time abstraction of the timed stream $r$. Similarly we denote for a channel valuation $x \in C$ by $\bar{x}$ its time abstraction, defined for each channel $c \in C$ by the equation

$$\bar{x}.c = \overline{x.c}$$

The operators easily generalize to sets of streams and valuations.

We describe the black box behavior of a component by an I/O-function that defines a relation between the input streams and output streams of a component. An I/O-function is represented by a set-valued function on valuations of the input channels by timed streams. The function yields a set of valuations for the output channels for each valuation of the input channels. An I/O-function is a set-valued function

$$f: \bar{I} \rightarrow \wp(\bar{O})$$

that fulfills the following timing property, which axiomatises the time flow. It reads as follows (let $x, z \in \bar{I}$, $y \in \bar{O}$, $t \in \mathbb{IN}$):

$$x \downarrow t = z \downarrow t \Rightarrow \{y \downarrow t+1: y \in f(x)\} = \{y \downarrow t+1: y \in f(z)\}$$

Here $x \downarrow t$ denotes the stream that is the prefix of the stream $x$ and contains $t$ finite sequences. In other words, $x \downarrow t$ denotes the communication histories in the channel valuation $x$ until time interval $t$. The timing property expresses that the set of possible output histories for the first $t+1$ time intervals only depends on the input histories for the first $t$ time intervals. In other words, the processing of messages in a component takes at least one time tick. We call functions with this property time-guarded or strictly causal.
A function \( f: \mathcal{I} \rightarrow \mathcal{O}(\mathcal{O}) \) is called \textit{time independent}, if we have for all input channel valuations \( x, x' \in \mathcal{I} \):

\[
x = x' \Rightarrow f(x) = f(x')
\]

In a time independent function the timing of the input influences at most the timing of the messages in the output histories; it does not influence the choice of these messages.

By \( \text{Com} \) we denote the set of all I/O-functions for arbitrary channel sets \( \mathcal{I} \) and \( \mathcal{O} \). For any \( f \in \text{Com} \) we denote by \( \text{In}(f) \) its set of input channels and by \( \text{Out}(f) \) its set of output channels.

### 2.2.2 Composed Systems

An interactive distributed system consists of a family of interacting components (in some approaches also called \textit{agents} or \textit{objects}). These components interact by exchanging messages on their channels by which they are connected. An \textit{distributed system}, also called a \textit{system architecture}, consists of a network of communicating components. Its nodes represent components and its arcs communication lines (channels) on which streams of messages are sent.

We model distributed systems by data flow nets. Let \( N \) be a set of identifiers for components and \( \mathcal{I} \) and \( \mathcal{O} \) be sets of input and output channels, respectively. A distributed system \((\nu, \mathcal{O})\) with syntactic interface \((\mathcal{I}, \mathcal{O})\) is represented by the mapping \( \nu: N \rightarrow \text{Com} \) that associates with every node a component behavior in the form of a black box view, formally, an interface behavior given by an I/O-function.

Each data flow net describes an I/O-function. This I/O-function is called the \textit{black box view} of the distributed system described by the data flow net. We get an abstraction of a distributed system to its black box view by mapping it to a component behavior in \( \text{Com} \). This black box view is represented by the component behavior \( f \in \text{Com} \) specified by the following formula (note that \( y \in \mathcal{C} \) where \( \mathcal{C} \) is the set of all channels of the system):

\[
f(x) = \{ y|\mathcal{O}: \forall i \in N: y|\text{Out}(\nu(i)) \in \nu(i)(y|\text{In}(\nu(i))) \}
\]

Here, we use the notation of function restriction. For a function \( g: D \rightarrow R \) and a set \( T \subseteq D \) we denote by \( g|_T: T \rightarrow R \) the restriction of the function \( g \) to the domain \( T \). The formula essentially expresses that the output history of a data flow net is the restriction of a fixpoint for all the net-equations for the output channels.

### 3 Giving Meaning to MSCs

In our approach, a set of MSCs is considered as a formal specification (a predicate) of the components of a distributed system modeled by a data flow net \((\nu: N \rightarrow \text{Com}, \mathcal{O})\). In the case of a deterministic system component, each MSC describes an instance of behavior for a component that is uniquely determined for the particular input stimuli. In the case of deterministic components, MSCs can be directly translated into algebraic equations.

However, this simple view does not apply for nondeterministic systems. For them the meaning of a single MSC is less straightforward. At first sight, an MSC only describes one pattern of reaction of a component in response to a particular input pattern. According to nondeterminism, there may be other reactions possible. Only by a comprehensive set of MSCs
that comprises all possible behaviors we get a comprehensive and precise specification for nondeterministic components.

We base our discussion on a very abstract view of MSCs. Given a network of interacting components, an MSC is understood to define a predicate $Q_p$ for each component $p \in N$. We assume that the syntactic interface of each of the components of the system under consideration is given. This means that for each component $p$ the sets $I_p$ of input channels and $O_p$ of output channels are specified with their types that indicate which kind of messages can be sent along the channels. This way we obtain the following mathematical representation of the predicate represented by the I/O-function $Q_p$ (we prefer here a set notation over a logical notation)

$$Q_p : \bar{I}_p \rightarrow \mathcal{P}(\bar{O}_p)$$

By associating a predicate with sets of MSCs, we fix the formal semantics of MSCs. However, this way the methodological role of MSCs is not captured. Many question about the systematic usage of sets of MSCs in the development process remain open.

Message sequence charts describe concurrent traces of interactions, which are also called processes ("runs"), of communication actions. They represent instances of individual system runs. With this view, an MSC specifies properties of a network of components that are distributed and interact concurrently by exchanging messages. Often, MSCs are considered only as a means for illustrating representative instances of system behaviors. We understand, however, in the following a set of MSCs rather as a possibly incomplete, but nevertheless formal and thus precise specification of the behavior of the components of the system.

In the development process, MSCs are often used to specify the behavior of a network of components in terms of characteristic patterns of interaction. Complex behaviors cannot easily be specified this way. For them we need better structuring and abstraction techniques to be able to work with comprehensible sets of sufficiently short MSCs.

In this section we describe the meaning of a set of MSCs representing instances of interactions of a deterministic system. We represent the behavior of a deterministic system component by an I/O-function (in fact, it is sufficient that the time abstraction $\bar{f}$ is deterministic)

$$f : \bar{I} \rightarrow \bar{O}$$

that maps input histories to output histories.

**Example:** A simple transmission protocol

We consider the component TR that is a transmitter. It receives a message on its input channel $a$, informs the receiver by the signal ready that a message $s$ is available on its output channel $b$, and

---

1 We deliberabalry avoid the term “instance” that the authors of [MSC 95] use as a synonym for processes.
forwards the message, provided the receiver is ready to take it, which is indicated on channel c. In any case the sender is informed on channel d whether the message could be delivered or not. The syntactic interface of the component TR is shown by Fig. 2. Fig. 3 gives the two essential scenarios for the component TR.

From the two MSCs in Fig. 3 we derive the following specifying equations for the component TR:

\[ f_{TR}(\langle a:m \rangle) = \langle b:\text{ready} \rangle \]

\[ f_{TR}(\langle a:m \rangle \cdot \langle c:Y \rangle \cdot x) = \langle b:\text{ready} \rangle \cdot \langle d:Y \rangle \cdot \langle b:m \rangle \cdot f_{TR}(x) \]

\[ f_{TR}(\langle a:m \rangle \cdot \langle c:N \rangle \cdot x) = \langle b:\text{ready} \rangle \cdot \langle d:N \rangle \cdot f_{TR}(x) \]

These three equations capture all the information contained in the two MSCs.

The translation of MSCs into the equations is done fully schematically following the pattern introduced above.

Our simple interpretation of MSCs by equations does not work, in fact, when specifying nondeterministic components by sets of MSCs. In general, in this case we would get inconsistent sets of specifying equations by the simple translation technique that we defined above for the deterministic case. This is demonstrated by the following example. As a consequence, we have to work, in the nondeterministic case, either with sets of functions or with set valued functions.
**Example:** Unreliable Medium
We specify an unreliable transmission component UM. It receives messages of type M on channel a and either forwards them on channel b, sending some acknowledgment to the sender via channel c, or it may forget them sending a failure indication message (fim) to the sender. The syntactic interface of the component UM is described in Fig. 4. We provide the two cases of interaction shown in Fig. 5.

![Diagram of two MSCs for the Component UM]

If we would use naively the translation of the previous section for deterministic components, we get the following two equations for $f_{UM}$

$$f_{UM}(\langle a: m \rangle^* x) = \langle c: ack \rangle^* \langle b: m \rangle^* f_{UM}(x)$$

$$f_{UM}(\langle a: m \rangle^* x) = \langle c: fim \rangle^* f_{UM}(x)$$

These equations are obviously contradictory. Since $f_{UM}$ is required to be a function, its result on the input $\langle m: a \rangle^* x$ is required to be uniquely determined.

Hence, a set of MSCs, intended to specify a nondeterministic system component, is interpreted not by a stream-valued function but by a set valued function

$$f: \tilde{I} \rightarrow \mathcal{P}(\tilde{O})$$

Here we denote for a set valued function $f$ by the expression $y^* f(x)$ the set $\{ y^* z : z \in f(x) \}$.

**Example:** Unreliable Medium (Part 2)
For our example above we get by this translation the specifying formulas

$$\langle c: ack \rangle^* \langle b: m \rangle^* f_{UM}(x) \subseteq f_{UM}(\langle a: m \rangle^* x)$$

$$\langle c: fim \rangle^* f_{UM}(x) \subseteq f_{UM}(\langle a: m \rangle^* x)$$

For nondeterministic systems a notion of correctness as it is introduced for deterministic systems does not make much sense. If we would define:

"An I/O-function is correct if it fulfills the formulas generated from a set of MSCs."

we would call the function correct that shows a chaotic behavior (where every output is possible for every input). This is generally not what is intended by a set of MSCs. Therefore, for nondeterministic components, a more restrictive interpretation is required to be able to restrict the behavior appropriately. Such an interpretation is called **closed world assumption**.
Example: Unreliable Medium
We look at the example of the unreliable medium again. We assume the same syntactic interface as in the previous example. However, here we assume that a message has to be sent twice to be either transmitted or rejected. This is expressed by the MSCs shown in Fig. 9.

The two MSCs translate into the specification
\[
\{\cdot\} \subseteq \tilde{f}_{UM}(\langle a:m\rangle^*x)
\]
\[
\langle c:\text{ack} \rangle \langle b:m \rangle \tilde{f}_{UM}(x) \subseteq \tilde{f}_{UM}(\langle a:m \rangle^*\langle a:m \rangle^*x)
\]
\[
\langle c:\text{fim} \rangle \tilde{f}_{UM}(x) \subseteq \tilde{f}_{UM}(\langle a:m \rangle^*\langle a:m \rangle^*x)
\]

By the closed world assumption we get in addition to the first equation by a strengthening of the second two equations the much stronger specification
\[
\langle c:\text{ack} \rangle \langle b:m \rangle \tilde{f}_{UM}(x) \cup \langle c:\text{fim} \rangle \tilde{f}_{UM}(x) = \tilde{f}_{UM}(\langle a:m \rangle^*\langle a:m \rangle^*x)
\]

that fixes the set of output histories uniquely. This example demonstrates how crucially the closed world assumption is to restrict the meaning of a set of MSCs.

Each set \(S\) of MSCs defines a formal specification \(Q_s\). Given two sets \(S_1\) and \(S_2\) of MSCs we can combine them easily by combining their specifications as long as we do not work with the closed world assumption. We get
\[
Q_{S_1} \land Q_{S_2}
\]

Therefore, sets of MSCs can freely be used as specification units and combined.

Care is advised, however, in connection with the closed world assumption. Let \(CWA_s\) denote for a given set \(S\) of MSCs the specification obtained by the closed world assumption over \(S\).

As expected, we do not, in general, the validity of the equation
\[
CWA_{S_1} \lor CWA_{S_2} = CWA_{S_1 \cup S_2}
\]

which is a simple consequence of the nonmonotonicity of the closed world operator. In fact, we obtain the following much weaker relationship
\[
CWA_{S_1} \lor CWA_{S_2} \Leftarrow CWA_{S_1 \cup S_2} \Leftarrow CWA_{S_1} \land CWA_{S_2}
\]

This consideration shows that care is advisable when applying the closed world assumption in connection with collecting requirements. Here a careful selected methodology is mandatory.
4 Introducing Control and Data States

So far we worked with the following restricted interpretation of MSCs:

- *initial compact behavior*: an MSC describes a complete prefix of the input and output actions of a component. After having observed this behavior, for the described component it may be assumed to be in its initial state again.

However, in general, this style of specification is not expressive enough. We do not want to describe by an MSC only initial segments of behavior of a component that can be repeated forever. We rather want to describe behaviors in certain states where these states may occur repeatedly depending on the particular I/O-patterns.

A MSC describes only a finite interaction history (as long as we do not introduce the concept of repetition or recursion into MSCs). Hence, for the specification of systems with infinite or at least unbounded behavior, we prefer to require that an MSC holds not only for an execution sequence starting in the initial state but rather for all execution sequences starting in all those states for which the MSC is valid.

For being able to express this we introduce the notion of a control state and of a data state into a MSC. We assume that each component has a finite number of control states. The space of data states may be infinite. We add these control states and state predicates (assertions) about the local data states as labels to the threads in an MSC.

In this section we show how to enrich MSCs by control and data states and how to translate them into specifications. We use control state as labels for threads. For each component we introduce a finite set of control state elements. Each of these elements represents a control state of the component.

A data state is defined by a set of state attributes and their values. A state attribute is an identifier for a data element for which a type is given. The set of attributes together with their types defines the data state space. Every valuation of the attributes, which is an assignment of values to attributes, defines a data state $\sigma$. Let $\sigma, \sigma'$ denote data states. Given a thread, we work with predicates $P(\sigma)$ and $Q(\sigma, \sigma')$ on the data state as labels that we place at the beginning and at the end of the thread, if appropriate, and at each position between an output arrow and an input arrow. Such predicates are easily formulated as assertions on the state attributes. In general, we work with predicates that depend on the attributes of the global state. This means that they refer to local states of all the components involved. For convenience, we assume that we can derive from these given global predicates the local predicates $P$ and $Q$ that refer only to the local state.

$$\begin{array}{c}
\alpha: P(\sigma) \\
\beta: Q(\sigma, \sigma')
\end{array}$$

Fig. 6 Thread in an MSC with Control States $\alpha$ and $\beta$ and Assertions $P$ and $Q$ about the Local Data State
Syntactically we write assertions as formulas of predicate logic for $P(\sigma)$ and $Q(\sigma, \sigma')$ that contain the attributes for the data state $\sigma$ as free identifiers. We get subsections of the threads labeled by the state predicates of the form shown in Fig. 6.

This way we get an interpretation of a thread as a set of transitions of a state machine or as transition equations for stream functions indexed by control states and data state predicates. A formal description of the interpretation of MSCs along these lines is given in the following section.

From a methodological point of view, we may use the following strategy to enrich MSCs by control states. If we have a set of threads for a component, we may choose schematically the same control state identifier for the beginning and the end of each thread of the component $f$ in the set of MSCs. This means that the pattern of behavior of an MSC is repeated over and over again.

Another technique is to let the designer of the MSCs introduce the control state names explicitly into the threads of the MSCs. Then every MSC starts in an individually chosen control state and ends in an individually introduced control state. Every control state $\chi$ corresponds to a stream processing function $f^\chi_\sigma$ (given a data state $\sigma$). MSCs with control states can be represented by so called higher level MSCs, where the MSCs are represented in a structured way by an automaton (see also [12]).

**Example:** Simple Storage Cell

A storage cell is a deterministic component that can be empty or nonempty. If it is empty it can be filled; if it is filled it can be read or made empty. To fill a cell we write a data value. This value can be read repeatedly until the cell is emptied or overwritten. The syntactic interface of component Store is given in Fig. 7.

For the component Store, we work with a simple control state space:

$$\{\text{em, ne}\}$$

and with only one data attribute $\nu$. Thus the data state space is given by the set of functions

$$\sigma: \{\nu\} \rightarrow \text{Data}$$

Fig. 8 shows the MSCs that model the patterns of behavior of the component storage cell.

We obtain the following equational specifications by generating conditional equations from the MSCs in Fig. 8:

$$\bar{f}^\text{ne}_\sigma (\langle \text{a: empty}\rangle^\nu x) = \bar{f}^\text{em}_\sigma (x)$$
We assume that Store is a deterministic component and therefore we generate equations from the MSCs.

In particular, whenever for some input a corresponding input pattern is not contained in a set of MSCs, the output is arbitrary (chaos). This conforms to the well-known idea of underspecification.

5 Interleaving MSCs and Projections

Often, distributed systems have to provide a large number of quite unrelated services and respective scenarios of interactions. Then it is helpful not to show in a single MSC the pattern of all the messages that are exchanged in the distributed system related to different services but to restrict the messages shown by the MSC to those messages that are related to the use cases of the system that are of particular interest.

In such cases of independent behavior there are three possible ways to work with a set of MSCs in requirements engineering:

1. We concentrate on one particular service in a specification by studying only a specific subset of the input and output messages. That way we specify an "abstract" behavior by MSCs reflecting only the particular service. Then we relate the more complex general
behavior of the component to this abstract one. Independent services then can occur interleaved.

(2) We assume that certain I/O-patterns are independent under certain circumstances (in certain states). Then patterns can occur interleaved, too.

If there are dependencies between the different patterns that may occur interleaved, we can use the concept of a state and invariants to indicate when a certain behavior may occur and how the patterns interact with each other. This allows us to express mutual exclusion.

We may also use additional MSCs to express the data dependencies between MSCs. To do this we use overlapping MSCs. Given a set of MSCs we may interpret the threads of a component according to the following ideas:

- projection: an MSC describes a behavior in terms of a projection of the input and output,
- loose selection: an MSC describes a loose selection of messages (a freely chosen subhistory).

The second case is more difficult to formalize. Its meaning is so vague that it is most likely not suited as a precise specification method. However, even this case can be seen as a special case of a projection with a projection function that is not properly specified.

6 On the Methodological Role of MSCs

There are many ways to work methodically usefully with MSCs in the development process for interactive distributed systems. For instance, MSCs can be used to illustrate the interaction of system components as part of the decomposition of systems. Moreover MSCs help to represent system traces in tests or software inspections.

6.1 MSCs and Refinement

One way to get a clear idea of the methodological role of MSCs in systems development is to relate them to formal methods. Formal methods recommend formal specifications and system development steps based on refinement relations (see [3]). The most basic refinement relation is property refinement. In property refinement a system is developed by adding further properties (requirements) to an underspecified system description as well as by adding further system parts (enriching the signature). The basic mathematical notion of property refinement is logical implication (with respect to the logical properties) or isomorphically set inclusion (with respect to the signatures and the set of models). This allows us also to replace component specifications by logically equivalent component implementations.

For our system model property refinement is a very simple concept. A component with a behavior described by the I/O-function

\[ \hat{f} : \bar{I} \to \wp(\bar{O}) \]

is called a property refinement of a component with a behavior described by the I/O-function

\[ f : \bar{I} \to \wp(\bar{O}) \]
if for all input streams \( x \in \bar{I} \) we have:

\[ \hat{f}(x) \subseteq f(x) \]

Informally expressed, we call the component \( \hat{f} \) a property refinement of component \( f \) if for every input history \( x \) every output history that \( \hat{f} \) may generate for input \( x \) is also an output history for \( f \) on input \( x \). In other words, \( \hat{f} \) fulfills all requirements that \( f \) fulfills.

In many applications, we better distinguish between two kinds of MSCs:

- MSCs that describe the intended behavior and interaction,
- MSCs that describe unwanted behavior, modeling failure cases that may occur due to some unreliability of system parts and therefore have to be tolerated but should be avoided whenever possible.

Given such a classification for MSCs we obtain a completely different view onto a set of MSCs with respect to refinement. Positive scenarios may be dropped in system development steps only if other positive scenarios remain applicable for the same input pattern. Negative scenarios may be eliminated whenever feasible. Finally we do not accept behaviors that are composed only of patterns of negative scenarios. Positive patterns should occur infinitely often if the respective input pattern occurs infinitely often. We could also say that we require \textit{weak fairness} for the positive patterns.

6.2 \textit{MSCs in the Development Process}

Actually, in the software development process there are several phases in which MSCs prove to be a helpful concept:

1. In the early phases of requirements engineering in connection with use cases MSCs help to get first ideas which services a system should provide.
2. In the later phases of requirements engineering sets of MSCs are useful as description techniques that are part of a formal specification.
3. In the design phase the interaction between the system components can be illustrated by MSCs. This way MSCs are a key technique for designing and documenting the decomposition of a system. In particular, MSCs are helpful when describing design patterns.
4. In the implementation phase MSCs provide test cases. In particular, both the calculated result of runs of a system as well as the required behavior of test cases can be represented by MSCs.

All these scenarios for the usage of MSCs can be supported by tools. In particular, for the conception of these tools a scientific foundation of MSCs may be decisive.

From a methodological point of view, we may use a set of MSCs to describe an illustrative selection of system runs, or a complete specification of all system behaviors. A set of MSCs for the specification of the behavior of a network of components may hence be provided in a system development with the following intentions:

(a) \textit{Loose instances}: We give a loose selection of instances of runs of the system to illustrate its behavior.
(b) **Comprehensive instances:** We give a comprehensive set of MSCs that expresses all the requirements of components of the network.

These two options correspond to different ideas of using MSCs in the development process. The difference between these two ideas about the role of a set of MSCs is rather crucial. The first case may be a step in requirements engineering where we are interested in a description of a system behavior that is as general as possible (see [11]). The second case can be seen as a usage in the design process where a specific behavior has to be selected and described.

7 Conclusion

We have shown that there are several reasonable ways to understand and interpret sets of MSCs in the system development process. We see this freedom in the interpretation not necessarily as a weak but rather as a strong point of MSCs. However, we conclude from this that MSCs should never be used without an explicit indication how they should be interpreted and which role they play in the development process. One way is to annotate MSCs by control states, data state predicates and additional formal comments and hints.

There have been several proposals in the literature to give a precise semantics to MSCs. However, most of them understood MSCs as describing the meaning of composed systems in a global state view. Typical instances for this approach are found in [14]. There MSCs are translated into global state machines for the composed system. This way the set of global system traces is defined for a given MSC. Also in [4], where extended MSCs are introduced, a global system trace semantics is given.

In [11], MSCs are translated into state machines describing the behavior of the components of a system. This idea is rather close to ours since state machines can be seen as a specific form to describe I/O-functions.

For our approach to give meaning to MSCs the distinction between input messages and output messages in the communication events of an MSC proves essential. Distinguishing between input and output leads to a concept of causality for each of the involved components. This allows us to interpret MSCs in a way where we do not only describe by a set of MSCs very loosely what may happen in a system but also quite strictly what must happen.

Acknowledgements

It is a pleasure to thank Ingolf Krüger and Radu Grosu for stimulating discussions and helpful remarks on draft versions of the manuscript.

References