On the Meaning of Message Sequence Charts

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

Abstract

Message Sequence Charts (MSCs) are a technique to describe patterns of the interaction between the components of a interactive system by diagrams. They have become rather popular in the design of software architectures and distributed software systems. They are used frequently to describe scenarios of interaction illustrating use cases in software and systems development. Nevertheless, both the semantics of MSCs as means of specification and their methodological and technical role in the development process are not precisely clarified, so far. In this paper, we suggest a semantic model of MSCs in terms of stream processing functions that allows us to use them as an intuitively clear specification technique for the components of a system with a precisely defined meaning. In contrast to other semantical models for MSCs suggested in the literature (see [Ladkin, Leue 93, 95] and [Cobben et al. 98]) where the meaning of message sequence charts is explained by state transition machines and traces for the composed system, we define MSCs as a technique to specify the behavior of the components of a system. Furthermore, we discuss the systematic use of MSCs in the software development process.

1. Introduction

Today *message sequence charts* (MSCs) or *extended event traces* (EETs) are a well accepted description technique. They incorporated in a number of modeling methods used in practice (see [UML 97] or [Room 94]). They are to illustrate scenarios of interaction between distributed interacting components collected in networks. They prove to be very helpful when illustrating the fundamental ideas of the cooperation in distributed systems. MSCs are widely used, for instance, in telecommunication applications for illustrating protocols as well as the cooperation of switching components. They have found their way also into business applications. Several object oriented development methods such as Objectory, OMT, ROOM, and UML suggest use cases and their illustration by MSCs in order to document all possible scenarios of interaction. This way, the behavior of a system should be completely specified by MSCs.

In the sequel we discuss the following topics and suggest answers to the following questions:

- What is the formal meaning of an individual MSC for the components of a system?
- How can MSCs be combined?
- What is the formal meaning of a set of MSCs for the components of a system?
- What is the methodological role of MSCs and how do they fit into the development process?
- How far can MSCs serve as a precise and comprehensive mean of specification?

To answer all these questions in a precise and clear way, we have to tackle the following question, too:

- what type of systems do MSCs refer to and what properties do they specify precisely.

In the following, we deal with all these questions. We start with a discussion of the state of the art of MSCs, the type of systems they talk about, and introduce a reference model of a system. In section 3 we discuss the semantics of MSCs, first for deterministic systems and after that for nondeterministic systems. In section 4 we deal with the role of control and data states in connection with MSCs in the development process. In section 5 we discuss the meaning of MSCs that are seen as projections. In section 6 we deal with complete sets of MSCs by closed world assumptions. In section 7 and the conclusion we discuss the role of MSCs in the development process.

2. The Purpose of Message Sequence Charts

Typically, MSCs are used to describe a set of characteristic instances of system interactions in requirements engineering and in system design. When used this way, MSCs describe properties of a system to be constructed. This suggests to formalize MSCs as *logical properties of systems*. Hence we translate a set of MSCs into a predicate on system components.

To get a clear understanding of the meaning of MSCs along these lines we need to have a precise understanding what type of systems MSCs refer to. Therefore, when defining the meaning of MSCs with respect to a system model, the choice of this model gets crucial. Clearly, MSCs refer to systems that run concurrently and interact. We have to select a mathematical system model on which we base a denotational semantics of MSCs.

This approach is in contrast to [Cobben et al. 98] where MSCs are translated into the terms of a specific process algebra for which an operational semantics is provided. This way the equivalence is based on bisimulation. This gives only a very implicit model and does not provide a meaning to sets of MSCs as specifications of the individual components.

2.1 Selected System Model

We work in the following with asynchronous message passing since for this model message sequence charts seem best suited. This model also fits best with the spirit of the description language SDL in the context of which the idea of MSCs has been mainly developed (see [Broy 91] and [Hinkel 98] for a formal semantics of SDL based on this model). For synchronous message passing systems, the concept of MSCs seems less well suited since for those systems

interleaved traces of actions are more adequate (see [CSP 85]). Moreover, for synchronous message passing so-called refusals are quite important for a compositional semantics which are not shown by MSCs at all (see [Hoare et al. 81])

2.1.1 Components

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

For us, 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 in a (discrete) time frame.

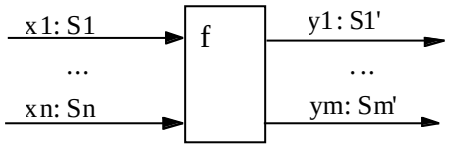

Fig. 1 Graphical Representation of a Component as a Data Flow Node
with Input Channels x1, ..., xn and Output Channels y1, ... , ym and their Respective Types

Let I be the set of input channels and O be the set of output channels. With every channel in the cannel set $I \cup O$ we associate a data type indicating the type of messages sent on that channel. Then by (I, O) the *syntactic interface* of a system component is given. 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 streams of messages of set M which are finite or infinite sequences of elements from M and by M^{ϖ} we denote the set of infinite streams of elements of set $M \cup \{\surd\}$ which contain an infinite number of time ticks. Mathematically, a timed stream in M^{ϖ} can also be understood as a function $\mathbb{N} \rightarrow M \cup \{\surd\}$.

Throughout this paper we work with some simple forms of notation for streams that are listed in the following. We use the following notations for timed streams x:

$z^\frown x$ concatenation of a sequence z to a stream x,

$x \sqsubseteq y$ x is a prefix of y, formally $x \sqsubseteq y \equiv \exists z: x^\frown z = y$

$x{\downarrow}k$ maximal initial sequence of the first i sequences in the stream x,

$S©x$ stream obtained from x by deleting all messages that are not elements of the set S,

$\bar{x}$ finite or infinite stream that is the result ofdropping all time ticks in x: $\bar{x} = M©x$.

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

$$type: C \rightarrow S$$

Here S is a set of types $s \in S$ where by $[\![s]\!]$ we denote the carrier set of data elements associated with the types. Let M be the universe of all messages. This means

$$M = \bigcup \{[\![s]\!]: s \in S\}$$

We define the valuations of the set C by the functions

$$x: C \to M^{\varpi}$$

where for each channel $c \in C$ the stream x(c) is of correct type:

$$x(c) \in [\![s]\!]^{\varpi}$$

This set of valuations of the channels in C is described by $\vec{C}$. Let in the following I and O be sets of typed channels.

Given a time stream $s \in M^{\varpi}$ we denote by $\bar{s}$ the stream in M^{ω} obtained by dropping all the time ticks in s. $\bar{s}$ is called the *time abstraction* of s. Similarly we denote for $x \in \vec{C}$ by $\bar{x}$ its time abstraction, defined for each channel $c \in C$ by the equation

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

We describe the *black box behavior* of a component by an *I/O-function*. It defines a relation between the input streams and output streams of a component that fulfills certain conditions with respect to their timing. 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. An I/O-function is a set-valued function

$$f: \vec{I} \to \wp(\vec{O})$$

that fulfills the following *timing property*. This property axiomatises the time flow. It reads as follows:

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

Here $x{\downarrow}k$ denotes the family of streams that are the largest prefixes of the streams in x that contain at most i time ticks. In other words, $x{\downarrow}k$ denotes the communication histories until the time k. The timing property expresses that the set of possible output histories for the first k+1 time intervals only depends on the input histories for the first k time histories. In other words, the processing of messages in a component takes at least one tick of time. We call functions with this property *time-guarded*.

We do not consider explicit timing properties in the following. Therefore we often work for a time-guarded function

$$f: \vec{I} \to \wp(\vec{O})$$

with its time abstraction

$$\bar{f}: (I \to M^{\omega}) \to \wp(O \to M^{\omega})$$

specified by

$$\bar{f}.x = \{\bar{y} \in O \to M^{\omega}: \exists\, z \in \vec{I}: \bar{z} = x \wedge y \in f.z\}$$

By Com[I, O] we denote the set of all I/O-functions with input channels I and output channels O. By Com we denote the set of all I/O-functions for arbitrary channel sets I and O. For any f in Com we denote by In(f) its set of input channels and by Out(f) its set of output channels.

2.1.2 Composed Systems

We model distributed systems by data flow nets. Let N be a set of identifiers for components (represented by data flow nodes) and O be a set of output channels. A distributed system (ν, O) with syntactic interface (I, O) is given by the mapping

$$\nu: N \rightarrow Com$$

that associates with every node a component behavior by a black box view (an interface behavior given by an I/O-function). As a well-formedness condition we require that for all component identifiers $i, j \in N$ (with $i \neq j$) the sets of output channels of the components $\nu(i)$ and $\nu(j)$ are disjoint. This is formally expressed by the equation

$$Out(\nu(i)) \cap Out(\nu(j)) = \emptyset$$

In other words, each channel has a uniquely specified component as its source. We denote the set of channels of the net by

$$Chan((\nu, O)) = O \cup \{In(\nu(i)): i \in N\} \cup \{Out(\nu(i)): i \in N\}$$

The set

$$I = \{In(\nu(i)): i \in N\} \backslash \{Out(\nu(i)): i \in N\}$$

denotes the set of input channels of the net. The channels in $\{Out(\nu(i)): i \in N\} \backslash O$ are called *internal*.

Each data flow net describes an I/O-function. This I/O-function is called the *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 Com[I, O] where I denotes the set of input channels and O denotes the set of output channels of the data flow net. This black box view is given by the component behavior $f \in Com[I, O]$ specified by the following formula:

$$f(x) = \{y|_O: y|_I = x \wedge \forall i \in N: y|_{Out(\nu(i))} \in \nu(i)(y|_{In(\nu(i))}) \}$$

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[1)] for all the net equations to the output channels.

2.3 Towards a Meaning of MSCs

We base our discussion on a very abstract view of MSCs. Given a network of interacting components a MSC is understood to define a predicate Q_p for each component p that is represented by a thread in the MSC.

[1)] According to the fact that we consider only time-guarded I/O-functions it can be shown that there exists always such a fixpoint; if the I/O-function is deterministic, the fixpoint is unique.

We assume that the syntactic interface of each of the components is given. This means that for each component p the sets I_p and O_p of input channels and output channels are specified as well as their types that fix which kind of messages can be sent along the channels. Therefore we obtain the following mathematical representation of the predicate represented by the I/O-function Q_p (we prefer a set notation over a logical notation)

$$Q_p: \vec{I}_p \rightarrow \wp(\vec{O}_p)$$

We show in the following how to derive the I/O-function specification Q_p given a set of MSCs. The result of the formalization depends on the following additional information:

(1) Is the component p that is described by the threads of the MSCs required to be deterministic?

(2) Is the component, after showing a pattern of behavior as specified, in a state where again MSCs are available to describe the behavior?

(3) Are the MSCs dense prefixes, projections, or free selections of instances of interactions?

(4) Is the set of MSCs meant as a loose sample or a comprehensive set of requirements?

The precise meaning of MSCs is given in section 4. We treat there first deterministic and then nondeterministic systems. Assuming a particular system description method (such as for instance the process diagrams in SDL), we may ask what it means that a described system with its components is *correct* with respect to a given set of MSCs.

This way we may fix the formal semantics of MSCs. However, this way the methodological role of MSCs is not fixed. The question about the systematic usage of sets of MSCs in the development process is still open.

3. Semantics of Message Sequence Charts

Message sequence charts describe concurrent *traces* also called *processes* ("runs") representing instances of individual system runs. A MSC specifies properties of a network of components that are distributed and interact by exchanging messages. A MSC describes an execution instance of such a system. Often, MSCs are therefore considered only as means of illustrations of representative instances of system behavior. We understand, however, in the following a set of MSCs rather as a (possible incomplete) formal requirements specification the components of the system.

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

A crucial question is what it is that should be expressed by a MSC. This tells us how to interpret a MSC and a set of MSCs as a specification.

3.1 Syntax and Structured Presentation of MSCs

In this section we introduce a structured syntactic presentation of MSCs. A MSC is a diagram with n vertical lines, one for each component of the system, that symbolize the flow of time for each component (time flows from the top to the bottom). These lines are called the *threads*. Between the threads there are horizontal arrows representing *interactions* between the respective components. These arrows point from one thread to another. Each arrow represents two communication *events* (sending and receiving) with a uniquely determined thread which is its *source* (called *sender*) and a uniquely determined *target* thread (called the *receiver*). Thus each thread consists of a finite sequence of events for which the component associated with the event is either its sender or its receiver. For convenience we assume that the events for each thread are in a linear order[1)].

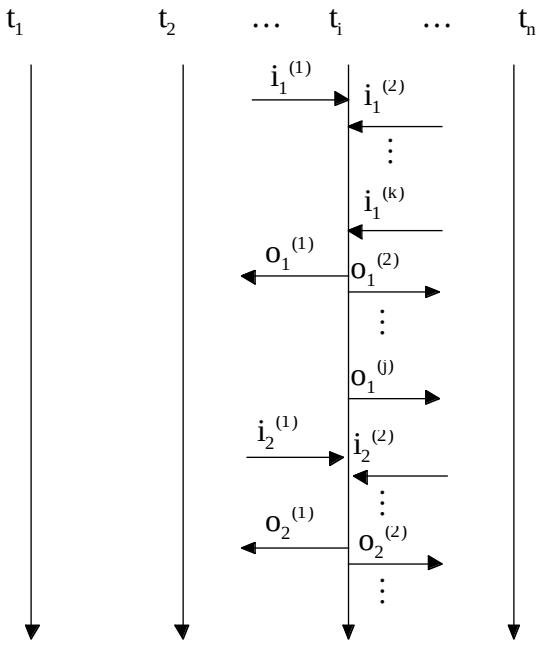

Fig. 2 General Schematic Form of a Thread for the Component t_i in a MSC Showing the Communication Events

A MSC is called *consistent*, if for all its events there is a partial ordering that contains the linear orderings of all the threads as suborderings (for a comprehensive discussion see [Alur et al. 96]). This makes sure that there are no cycles in the causality flow of the communication events (no "causal loops").

A MSC is labeled as follows. Each thread is labeled by the identifier referring to the component it stands for. Each communication arrow is labeled by a channel identifier and a message. In addition, certain points of a thread may be labeled by identifiers called the *control states* of the thread and by predicates referring to the attributes of the respective component. These predicates are called (data state) *assertions*.

[1)] A component with a behaviour described by a thread may in fact be a network itself with subcomponents operating in parallel. In this case the sequential order of the thread is understood as an interleaving of the partially ordered events of the network.

We concentrate on the threads in a MSC each of which specifies part of the behavior of one component. A crucial issue is the distinction of input and output for such a thread. In a thread, we find sequences of input arrows followed by sequences of output arrows as shown in Fig. 2. We cluster those sequences of input or output arrows into maximal subsequences. We understand the meaning of MSCs then informally as follows:

> Whenever the component gets all the input messages belonging to the input patterns up to a point in the thread, it produces all the output message as this is indicated by the output patterns.

This rule can be used to transform a MSC into a logical formula. In the following we define this formula explicitly.

Given a message sequence chart of the general form as shown in Fig. 2, we observe that every thread t_i has a sequence of alternating clusters of incoming and outgoing arcs. These clusters are combined into input and output patterns as shown in Fig. 3.

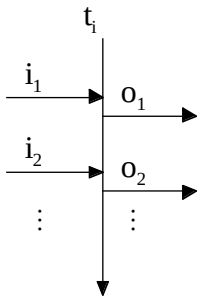

Fig. 3 General Form of a Thread in a MSC with Clustered Input/Output

One way to cope with a thread in a MSC is to cluster the incoming and outgoing messages into maximal segments of input and output as shown in Fig. 3. Each segment i_k contains only input messages (ingoing arrows) and each segment o_k contains only output messages (outgoing arrows).

Let C be the set of channels for the component t_i decomposed in input channels I and output channels O where $C = I \cup O$, $I \cap O = \varnothing$. An input or output pattern can be represented by a finite channel valuation

$$\vec{C}^*$$

for the channels in C which is defined as a mapping

$$C \rightarrow (M \cup \{\surd\})^*.$$

Following this view, a thread provides a finite sequence of valuations $i_k \in \vec{I}^*$, $o_k \in \vec{O}^*$ for the input channels and the output channels. Note that every arc labeled by i_k may represent a sequence of incoming messages and every arc labeled by o_k may represent a sequence of outgoing messages.

3.2 Semantics of Sets of MSCs Specifying Deterministic Systems

In this section we introduce two ways of describing the meaning of a set of MSCs representing instances of interaction of a deterministic system. One is to associate exactly one I/O-function with them. The other introduces a function for every pair of segments.

We represent the behavior of a deterministic system component by an I/O-function

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

that maps input histories to output histories. The MSC of Fig. 3 specifies a behavior for the function f which can be represented by the equation

$$\bar{f}(i_1 \hat{} \ldots \hat{} i_n) = o_1 \hat{} \ldots \hat{} o_n$$

and the additional liveness conditions (for all k, $1 \leq k \leq n$)

$$o_1 \hat{} \ldots \hat{} o_k \sqsubseteq \bar{f}(i_1 \hat{} \ldots \hat{} i_k)$$

This yields a liberal interpretation of threads in MSCs leading to the understanding that at least the indicated output by a MSC is guaranteed by the component, but maybe more.

3.3 The Semantics of MSCs for Nondeterministic Systems

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 as introduced above. Therefore we have to work, in the nondeterministic case, either with sets of functions or with set valued functions.

A set of MSCs intended to specify a nondeterministic system component is interpreted by a set valued function

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

For each of the MSCs of the form shown in Fig. 3 we get the specifying formulas (for all $k \in \mathbb{N}$, $1 \leq k < n$)

$$\exists\, y: o_1 \hat{} \ldots \hat{} o_k \sqsubseteq y \wedge y \in \bar{f}(i_1 \hat{} \ldots \hat{} i_k)$$

and

$$o_1 \hat{} \ldots \hat{} o_n \in \bar{f}(i_1 \hat{} \ldots \hat{} i_n)$$

We call a function f *comprehensive* with respect to a set of MSCs if it fulfills these specifying formulas. There are many functions, in general, that are comprehensive for a given set of MSCs. In particular, the "chaotic" function f with $f.x = \vec{O}$ for all input histories $x \in \vec{I}$ is comprehensive.

In contrast to the equations with which we work in the deterministic case this gives a much looser interpretation of a set of MSCs. The sets of MSCs specify this way which behavior is supposed to be possible, but they do not specify anything that restricts the set of outcomes of the system to be constructed.

For nondeterministic systems a notion of correctness as 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 not always what is intended by a set of MSCs. Therefore, for nondeterministic components, a more restrictive interpretation is appropriate to be able to restrict the behavior. This is done in the section 6 under the keyword closed world assumption.

4. Introducing Control and Data States

So far we worked with the following interpretation of MSCs:

– *initial compact behavior*: a MSC describes a complete prefix of the input and output actions.

Since a MSC is only a finite history to specify systems with infinite or at least unbounded behavior, we prefer not to require that a MSC holds only for an execution sequence starting in the initial state but rather for all execution sequences starting in particular states.

However, in general, life is not so simple. We do not want to describe by a MSC only initial segments of a component. We rather want to describe behaviors in certain situations. For being able to do this we introduce the notion of a *control state* and a *local data state*. We label threads in addition with these control states and state predicates (*assertions*) about the local data states.

4.1 Introducing Local Data States

A data state is defined by a set of attributes and their values. An attribute is an identifier for which a type is given. The set of attributes together with their types defines the data state space. Every valuation of the attributes by data values defines a data state σ. Let σ, σ' denote data states. Given a thread we assume predicates $P_i(\sigma)$ and $Q_i(\sigma, \sigma')$ on the data state as labels which we place at the beginning and at the end of the thread and at each position between an output arrow and an input arrow. Syntactically we write formulas of predicate logic for $P(\sigma)$ and $Q(\sigma, \sigma')$ that contain the attributes for the data state σ as free identifiers. We get sections of threads of the form as shown in Fig. 4.

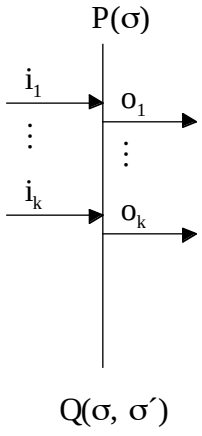

Fig. 4 General Form of a Thread in a MSC with Assertions

This way we get an interpretation of a thread as a set of transitions of a state machine.

4.2 Introducing Control States

We have already shown how to make use of control states in section 3.2.2. Now we discuss the usage of control states in more detail. There is the following strategy to add control states to MSCs. If we have a set of threads for a component, we may choose

(1) the same state identifier for the beginning and the end of each thread of the component f in the set of MSCs.

(2) if initial prefixes of threads coincide, then we choose the same state identifiers for those initial segments.

Another technique is to introduce the control state names explicitly by the designer into the thread of the MSC. Then every MSC starts in a state and ends in a state. As long as the states are all control states their meaning is simple. Every control state χ corresponds to a stream processing function (given a data state).

In addition to the control states we deal with the data states. Given a MSC as shown in Fig. 4 labeled by an assertion P about the initial data state σ and an assertion Q about the final data state σ' in relation to σ as well as by the initial control state χ and the post control state χ' we obtain the following specifying formulas

$$y \in \bar{f}^{\chi}_{\sigma}(i_1\hat{}\ldots\hat{}i_k) \quad \Leftarrow o_1\hat{}\ldots\hat{}o_k \sqsubseteq y \wedge P(\sigma) \qquad \text{for } 1 \leq k < n$$

$$o_1\hat{}\ldots\hat{}o_n\hat{}\bar{f}^{\chi'}_{\sigma'}(x) \subseteq \bar{f}^{\chi}_{\sigma}(i_1\hat{}\ldots\hat{}i_n\hat{}x) \quad \Leftarrow P(\sigma) \wedge Q(\sigma, \sigma')$$

This leads to a system of specifications of the function $\bar{f}^{\chi}_{\sigma}$.
A crucial question is the selection of the identifiers for the intermediate control states. If we choose all these control states distinct this is probably not always what is intended. However, we may use control identifiers freely and repeatedly in threads. By repeating a control state identifier at several points in a set of threads we may express that we can compose certain behaviors sequentially. This is taken care of in our translation of MSCs to predicates by the function identifiers indexed by the control states. Introducing control states explicitly can be understood as clustering (partitioning) the set of MSCs into sets (with the same initial and final states) and relating these sets (by arrows pointing of one set to the other if the final state of the first coincides with the initial state of the second).

4.3 The Universe of Unfiltered Scenarios

As a result of the logical interpretation described above we get a set of conditional formulas of the general forms

$$\text{condition}(x, y) \Rightarrow y \in \bar{f}^{\chi}_{\sigma}(x)$$

or

$$\text{condition}(x, y) \Rightarrow y = \bar{f}^{\chi}_{\sigma}(x)$$

for each set of MSCs for each of the involved components. By choosing all kinds of instantiations for the input history x and the output history y we get rid of the conditions and obtain a generally infinite set of formulas of the form

$$y \in \bar{f}^{\chi}_{\sigma}(x)$$

or

$$y = \bar{f}^{\chi}_{\sigma}(x)$$

for concrete histories $y \in \vec{O}$ and $x \in \vec{I}$. This set is called the *unfiltered universe* of *interaction patterns* (similar to the Herbrand universe in logic programming) for the component f.

5. MSCs as Projections

Often distributed systems have to handle a large number of quite unrelated scenarios of interaction. Then it is helpful not to show all messages that are exchanged in the distributed system but to restrict the presentation shown by the MSC to those messages that are related to the aspects of the system that should be clarified by the MSC. A typical example is the restriction to the message for a particular subprocess (a particular business transaction). Let us assume that we have projection functions for a stream x written in the form

$$x \mid_N$$

that restrict a stream to those elements in the message set N. Given a set of messages N for a set of MSCs we can restrict the communication shown in the MSCs to the messages in N. We specify then a set of MSCs by the formulas

$$i_1 \hat{} \ldots \hat{} i_k \sqsubseteq x \mid_N \Rightarrow \exists\, y \in \bar{f}(x): o_1 \hat{} \ldots \hat{} o_k \sqsubseteq y \mid_N$$

This allows us to talk about communication subhistories. Given a set of MSCs we may interpret the threads of a component according the following ideas:

- *projection*: a MSC describes a prefix of a projection of the input and output,
- *loose selection*: a MSC describes a loose selection of messages.

The second case is a bit difficult to deal with. Its meaning is so vague that it is not suited as a specification method. However, even this case can be seen as a special case of projection.

If a MSC describes a projection of messages, we may ask whether MSCs may overlap. In this case each MSC may describe a particular aspect of a system.

Given the unfiltered universe of interaction patterns and two filter functions

$$\varphi_I: \vec{I} \to \vec{I}$$

$$\varphi_O: \vec{O} \to \vec{O}$$

which formalize the projections by the restrictions used above in a more general way, we select for each interaction pattern

$$y \in \bar{f}^{\chi}_{\sigma}(x)$$

or

$$y = \bar{f}^{\chi}_{\sigma}(x)$$

respectively a set of patterns

$$y' \in \bar{f}^{\chi}_{\sigma}(x')$$

or

$$y' = \bar{f}^{\chi}_{\sigma}(x')$$

respectively provided $\varphi_I(x') = x$ and $\varphi_O(y') = y$. The set of the patterns obtained that way is called the *filtered universe of interaction patterns*.

In principle, it makes also sure how to combine MSCs with filters. Given a set

$$\{m_k : k \in K\}$$

of MSCs with a set U_k of unfiltered universes for each m_k and the filter function φ_I^k and φ_O^k we get a conjunction

$$\bigwedge_{k \in K} \bigwedge_{(x, y) \in U_k} (\varphi_I^k(x') = x \Rightarrow \forall\, y' \in \bar{f}(x'): \varphi_O^k(y') = y)$$

Again by instantiating the x and y in the formulas we get a universe of equations and element formulas.

6. Closed World Assumptions

From a MSC we derive, as explained in the preceding paragraph, a set of equations of the form

$$y = \bar{f}(x)$$

or in the nondeterministic case of formulas of the following form

$$y \in \bar{f}(x)$$

This set is called the *filtered universe*. So far we have not restricted the behavior of a nondeterministic component described by a set of MSCs. We obtain a more restrictive interpretation as follows. We assume that whenever we have a thread for a component with behavior function f that leads to a condition

$$y \in \bar{f}(x)$$

then for the input x only the output y and the outputs that are explicitly included by other threads for the input x are allowed. Only if no thread is included for some input x at all arbitrary output is permitted.

6.1 Associating a Canonical Behavior with a Set of MSCs

In this section we construct the closure of a set of MSCs according to the closed world assumption. Given a set of MSCs specifying a nondeterministic component by their threads we

associate with them a canonical behavior based on the closed world assumption as follows. An I/O-function

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

connects infinite input histories with infinite output histories. A thread in a MSC only talks about finite sections of histories. Therefore we define the meaning of a set of threads for the function f by considering a function on finite and infinite timed stream

$$f^*: I^{\vec{*}} \rightarrow \wp(O^{\vec{*}})$$

where for a set of channels C we define $C^{\vec{*}}$ as the set of valuations

$$C \rightarrow (M^*)^* \cup (M^*)^\infty$$

We use the prefix ordering $\sqsubseteq$ as an auxiliary construct for defining the set of downward closed subsets

$$DCS(O^{\vec{*}}) = \{Z \subseteq C^{\vec{*}}: Z \neq \emptyset \wedge \forall z \in C^{\vec{*}}: z \sqsubseteq x \wedge x \in Z \Rightarrow z \in Z\}$$

The partial ordering that we use on the set DCS is simply set inclusion. Under this ordering $Z = \{C^{\langle\rangle}\}$ is the least element where $C^{\langle\rangle} \in C^{\vec{*}}$ with (for $c \in C$) $C^{\langle\rangle}.c = \langle\rangle$.

Actually we define the function f^* such that

$$f^*: I^{\vec{*}} \rightarrow DCS(O^{\vec{*}})$$

If we are given a set of formulas $R = \{\Phi_k: k \in K\}$ where each Φ_k is of the form (we can get rid of the conditions by looking at all instances of the identifiers in the conditions)

$$y_k \in \bar{f}(x_k)$$

or of the form

$$\bar{f}(x_k \hat{} x') \subseteq y_k \hat{} \bar{f}(x')$$

we specify the function $\bar{f}^*$ as the $\subseteq$-least time guarded function that fulfills the following specifying formulas

$$y_k \in \bar{f}^*(x_k)$$

$$\bar{f}^*(x_k \hat{} x') \subseteq y_k \hat{} \bar{f}^*(x')$$

As a result of this construction we obtain a function

$$\bar{f}^*: I^{\vec{*}} \rightarrow DCS(O^{\vec{*}})$$

For an input history $x \in \vec{I}$ (which is also in $I^{\vec{*}}$) each maximal output history $y \in f^*(x)$ that is not in $\vec{O}$ does contain for certain channels $c \in C$ sequences y.c that are not infinite.

6.2 Closures on the Canonical Behavior

The function f^* constructed in the previous paragraph can be completed into an I/O-function

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

by adding to the maximal elements in $f^*(x)$ for $x \in \vec{I}$ arbitrary streams to those streams that are finite. We define the function f with based the function f^* by the following equation:

$$\bar{f}.x = \{y \in \vec{O}: \quad \forall\, k \in \mathbb{N}: \overline{y \downarrow k} \in \bar{f}^*(x) \vee$$
$$\exists\, k \in \mathbb{N}: \overline{y \downarrow k} \in \bar{f}^*(x) \wedge \forall\, y' \in \bar{f}^*(x): \overline{y \downarrow k} \sqsubseteq y' \Rightarrow \overline{y \downarrow k} = y'\}$$

This construction is called the *chaotic closure* for the set of defining formulas.

According to this construction the function f is the inclusion largest function with the following properties:

(1) f is time guarded,

(2) if there is a MSC that talks about the input x, then f(x) contains exactly those outputs explicitly described in one of the MSCs.

(3) if for an input x we can decompose x into x'^x'' such that exactly for the input pattern in x' output is specified by the MSCs we define the output for x'' as being arbitrary.

In particular, whenever for some input an input pattern is not contained in a set of MSCs, the output can be arbitrary (chaos).

The closed world construction concludes our semantic treatment of MSCs by a translation into a set of equations and is-element-formulas.

7. The Methodological Role of MSCs

There are many useful ways to work with MSCs in the development of 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 can be used to represent system traces as they are used in tests or software inspections. In the software development process there are several phases in which MSCs are a helpful concept:

(1) In the early phases of requirements engineering we may use MSCs to get a first idea which services the system should provide.

(2) In the later phases of requirements engineering sets of MSCs may be used as description techniques as part of a formal specification.

(3) In the design phase the interaction between the system parts can be illustrated by MSCs. This way MSCs are a key technique for the decomposition of a system. In particular, MSCs are helpful when describing design patterns.

(4) In the implementation phase MSCs can be used as test cases. In particular, both the result of runs of a system as well as the required behavior may be represented by test cases.

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.

In section 3 to 6 we defined a formal meaning of sets of MSCs. Strictly speaking, a set of MSCs is considered as a formal specification of the components of a distributed system modeled by a data flow net. In the case of a deterministic system component, a 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 equations. This does not hold for nondeterministic systems, however. For them the meaning of

a single MSC is much less straightforward. At first sight, a MSC only describes one way of reaction of a component in response to a particular input pattern. There may be many more different reactions possible. Only if we give a comprehensive set of MSCs that contains *all* behaviors that should be possible we get a powerful, expressive specification technique.

From a methodological point of view, we may use a set of MSCs to describe

- an illustrative selection of system runs,
- 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) *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 how to work with 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 [Klein 98]). The second case can be seen as a usage in the design process where a specific behavior has to be described.

One way to obtain a more precise idea of the methodological role of MSCs is to relate them to formal methods. Formal methods recommend formal specification and system development steps based on *refinement relations*. 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 concept of a system model *property refinement* is very simple. A component with a behavior described by the I/O-function

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

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

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

if for all input streams $x \in \vec{I}$ we have:

$$\hat{f}(x) \subseteq f(x)$$

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

At a first look, the ideas of specification and refinement and MSCs fit together very well. A MSC describes a relation between particular input/output histories and this way is a formal specification. There is, however, a *methodological obstacle* between this use of MSCs and refinement. The refinement relation allows us to get rid of nondeterministic branches and this way make a system more deterministic. This means that we may throw away interaction

scenarios in development steps as long as (for the same input) there are (nondeterministic) alternatives. This is, however, not always in the spirit of MSCs as they are understood, in practice. There the idea is to provide all runs of a system that should appear. And why should we introduce many more scenarios if they are thrown away afterwards, anyhow.

For deterministic systems the methodological obstacle does not appear. A deterministic system cannot be made "more deterministic". Consequently, as long as a described system is deterministic, the interpretation of its MSCs is rather straightforward. Every scenario corresponds to a specifying equation for the I/O-function modeling the particular component.

For nondeterministic systems and sets of scenarios including nondeterminism, there is a conflict, however, between:

- refinement steps that allow us to get rid of nondeterministic alternatives,
- the sets of MSCs that are assumed to describe all behaviors that all should be possible.

The second idea would not allow us to throw away scenarios while the first would suggest to get rid of certain scenarios. Actually, to understand sets of MSCs as a precise method of specification we have to define how to deal with the two situations characterized above.

One convincing answer to this methodological obstacle is given in [Klein 98] suggesting to understand a set of scenarios as the specification of the components that shows for input patterns covered by scenarios exactly the behavior described by the scenarios and for the input patterns not covered arbitrary behavior. This is formalized by our presentation under the keyword "closed world assumption".

Another answer to the methodological obstacle is as follows. 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 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 step only of other positive scenarios remain 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.

8. Conclusion

We have shown that there are several sensible ways to understand and interpret sets of MSCs in the system development process. We see this fact not as a weak but rather as a strong point of MSCs. However, we conclude from this that MSCs should never be used without a precise indication of the role they play in the development process and the way they should be interpreted.

There have been several proposals 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 examples for this approach are [Ladkin, Leue 93, 95]. 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 [Broy et al. 97] where extended MSCs are introduced a global system trace semantics is used.

In [Klein 98] MSCs are translated into state machines describing the behavior of the components of a system. This idea is rather close to ours.

For our approach to give meaning to MSCs the distinction between input messages and output messages in the communication events of a MSC is essential. Distinguishing between input and output leads to a concept of *causality*. Furthermore, it allows us to distinguish between deterministic and nondeterministic sets of MSCs. Typically, for an asynchronous component, input cannot be influenced and has to be accepted the way it is provided. An asynchronous component does accept any input at any time. We speak therefore of the *input enabledness of components*. In particular, the existence and the choice of input messages is not within the responsibility (requirements) of a component. However, output is. This allows us to interpret MSCs in a way where we do not only describe by an MSC very loosely what *may* happen in a system but also quite strictly what *must* happen.

References

[Alur et al. 96]
R. Alur, G.J. Holzmann, D. Peled: An Analyzer for Message Sequence Charts. Software - Concepts and Tools 17 (1996), 70-77

[Alur et al. 96]
R. Alur, G.J. Holzmann An Analyzer for Message Sequence Charts. Proc TACAS96, Passau, Germany, Lecture Notes in Computer Science, Vol. 1055, 35-48

[Ben-Abdallah, Leue 96]
H. Ben-Abdallah and S. Leue. Syntactic analysis of Message Sequence Chart specifications. Tech Report 96-12, Department of Electrical and Computer Engineering, University of Waterloo, 1996.

[Broy 91]
M. Broy: Towards a formal foundation of the specification and description language SDL. Formal Aspects of Computing 3, 1991, 21-57

[Broy 98]
M. Broy: Compositional Refinement of Interactive Systems Modelled by Relations. Malente 1997

[Kahn, MacQueen 77]
G. Kahn, D. MacQueen: Coroutines and Networks of Parallel Processes. In: Proceedings of the IFIP 77, Amsterdam: North-Holland 1977, 994-998

[Broy et al. 97]
M. Broy, C. Hofmann, I. Krüger, M Schmidt: A Graphical Description Technique for Communication in Software Architectures. Technische Universität München, Institut für Informatik, TUM-I9705, Februar 1997 URL: http://www4.informatik.tu-muenchen. de/

reports/TUM-I9705, 1997. Also in: Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC'97/ICSC'97)

[Cobben et al. 98]
J.M.H. Cobben, A. Engels, S. Mauw, M.A. Renics: Formal Semantics of Message Sequence Charts. Eindhoven University of Technology, Departement of Computing Science, Technical Report CSR 97/19

[CSP 85]
C.A.R. Hoare: Communicating Sequential Processes. Prentice Hall, 1985

[CCS 80]
R. Milner: A Calculus of Communicating Systems. Lecture Notes in Computer Science 92, Springer 1980

[Facchi 95]
Ch. Facchi: Methodik zur formalen Spezification des ISO/OSI-Schichtenmodells. Dissertation, Fakultät für Informatik, Technische Universität München 1996

[Graubmann et al. 93]
P. Graubmann, E. Rudolph, J. Grabowski. Towards a Petri Net based semantics definition for message sequence charts. In O. Faergemand and A. Sarma, editors, Proceedings of the 6th SDL Forum, SDL´93: Using Objects, October 1993

[Hinkel 98]
U. Hinkel: Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL. Dissertation, Fakultät für Informatik, Technische Universität München 1998

[Hoare et al. 81]
C.A.R. Hoare, S.D. Brookes, A.W. Roscoe: A theory of communicating sequential processes. Oxford University Computing Laboratory Programming Research Group, Technical Monograph PRG-21, Oxford 1981

[Kahn 74]
G. Kahn: The Semantics of a Simple Language for Parallel Processing. In: J.L. Rosenfeld(ed.): Information Processing 74. Proc. of the IFIP Congress 74, Amsterdam: North Holland 1974, 471-475

[Klein 98]
C. Klein: Anforderungsspezifikation durch Transitionssysteme und Szenarien. Promotion, Fakultät für Informatik, Technische Universität München, Dezember 1997

[MSC 93]
ITU-T (previously CCITT) (March 1993) Criteria for the Use and Applicability of Formal Description Techniques. Recommendation Z. 120, Message Sequence Chart (MSC), 35pgs.

[MSC 95]
ITU-T. Recommendation Z.120, Annex B: Algebraic Semantics of Message Sequence Charts. ITU-Telecommunication Standardization Sector, Geneva, Switzerland, 1995.

[Ladkin, Leue 93]
P.B. Ladkin, S. Leue. Interpreting Message Sequence Charts. Technical Report TR 101, Department of Computing Science, University of Stirling, 1993.

[Ladkin, Leue 95]
P.B. Ladkin, S. Leue. Interpreting Message Flow Graphs. Formal Aspects of Computing, 7(5): 473-509, 1995.

[Mauw, Reniers 94]
S. Mauw, M.A. Reniers. An algebraic semantics of basic message sequence charts. The Computer Journal, 37(4): 269-277, 1994.

[Selic 96]
B. Selic: Automatic generation of test drivers from MSC specs. Technical Report TR 960514 - Rev. 01, ObjecTime Limited, Kanata, Ontario, Canada, 1996.

[Zave, Jackson 97]
P. Zave, M. Jackson: Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology, January 1997

[Esterel 88]
G. Berry, G. Gonthier: The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. INRIA, Research Report 842, 1988

[Room 94]
B. Selic, G. Gullekson. P.T. Ward: Real-time Objectoriented Modeling. Wiley, New York 1994

[UML 97]
G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language for Object-Oriented Development, Version 1.0, RATIONAL Software Cooperation