Towards a Harmonization of UML-Sequence Diagrams and MSC

Ekkart Rudolph[a], Jens Grabowski[b], and Peter Graubmann[c]

[a]Technical University of Munich, Institute for Informatics, Arcisstrasse 21, D-80290 München, Germany, eMail: rudolphe@informaik.tu-muenchend.de

[b]Institute for Telematics, University of Lübeck, Ratzeburger Allee 160, D-23538 Lübeck, Germany, eMail: jens@itm.mu-luebeck.de

[c]Siemens AG, ZT SE 2, Otto-Hahn-Rin 6, D-81739 München, Germany, eMail: Peter.Graubmann@mchp.siemens.de

Sequence Diagrams as part of UML play an important role within use case driven object oriented (OO) software engineering. They can be seen as OO variants of the ITU-T standard language Message Sequence Chart (MSC) which is very popular mainly in the telecommunication area. Both notations would benefit from a harmonization. A more formal and powerful notation for Sequence Diagrams may arise, on the one hand. On the other hand, the application area of MSC might be considerably enlarged. In this context, it has to be noted that the acceptance of a language in the OO community essentially depends on a clear visualization of constructs typical for OO modelling. It is argued that Sequence Diagrams can be transformed into MSC diagrams if some enhancements of MSC are introduced. Such a transformation demonstrates the big advantage of MSC concerning composition mechanisms, particularly, in comparison with the rather obscuring branching constructs in Sequence Diagrams. At the same time, such a transformation may be used for a formalization of Sequence Diagrams in UML since MSC has a formal semantics based on process algebra.

Keywords: MSC, UML, OO, software engineering, distributed systems, real time systems, telecommunication

1. Introduction

Sequence Diagrams in UML [10] resulted from two sources: Ivar Jacobson's interaction diagrams (Objectory) [3] and the 1992 version of the MSC language (MSC-92)[1] [11]. From MSC-92 first an OO variant, called OMSC, was developed at Siemens [2] which essentially combined a subset of MSC with constructs typical for OO design, in particular, the construct for method calls. Sequence Diagrams are a further development and adjustment of OMSC. They do not claim to have the same degree of formality yet as MSC. This refers

[1]The terms MSC-92, MSC-96 and MSC-2000 refer to the 1992, 1996 and 2000 versions of the MSC recommendation Z.120. MSC without version indication refers to the actual 1996 language definition [11].

to both syntax and semantics. The syntax is not equally fixed in UML as in the ITU-T Recommendation Z.120 [12]. Therefore, different authors referring to UML use slightly different variants and take over some more constructs from MSC.

Sequence Diagrams and use cases are closely related within UML [1,10]. Sequence Diagrams are derived from use cases. A use case diagram shows the relationship among actors and use cases within a system. A use case diagram is a graph of actors, a set of use cases enclosed by a system boundary, communication associations between the actors and the use cases, and generalizations among the use cases. A given use case is typically characterized by multiple scenarios. Scenarios are described by means of Sequence Diagrams.

Sequence Diagrams are used whenever dynamic aspects are considered. The interaction between objects always arises from methods or processes being attached to objects. Such processes need time, have to be ordered possibly with respect to other processes, can be called only under certain guards, need parameters and provide results. Sequence Diagrams are useful whenever such correlations shall be visualized without showing the concrete programming code of a special programming language. Thereby, an abstraction of details of the later implementation is essential. Often, only a rather coarse overall view of the interplay of the concerned objects is important. Nevertheless, in its strongest refinement, Sequence Diagrams in OO systems can take over a similar role which in a procedural paradigm flow diagrams can play.

MSC is an ITU-T standard trace language for the specification and description of the communication behaviour of system components and their environment by means of message exchange [12]. In general, MSC applications can be attached to the area of reactive and distributed systems, and its main application area lies in the field of telecommunication systems. Traditionally, MSC is used primarily in connection with SDL. Indeed, it also arose from the same ITU-T study group as SDL. Contrary to Sequence Diagrams, MSC is a rather advanced language with a well defined syntax and semantics [4,6,9,12,13].

After the 1996 edition, MSC has been considered several times by Rational as a candidate for the trace description of object interaction in UML. A main obstacle essentially was the missing notion of flow of control in MSC. A harmonization of MSC and Sequence Diagrams certainly will enhance the application area of MSC by bringing it more to the attention of the OO community. However, the introduction of flow of control into MSC is more than a pure marketing strategy. It also pays attention to the fact that traditional telecommunication developing methods and OO techniques grow together. Beyond that, the introduction of flow of control to MSC appears as a challenging and promising subject in itself. The explicit representation of flow of control, in addition to the message flow representation, offers a completely new view of the implicit event trace which may contribute considerably to transparency and expressiveness of the description. It is also a natural place to introduce new communication mechanisms into MSC, e.g., synchronous communication, remote procedure call, etc. [8]. Though the role of flow of control in MSC appears to be not yet completely settled, it may be looked at already as a description, supplementary to the message flow. In this paper the role of flow of control within MSC will be clarified and the benefits of its explicit representation will be explained. A particular problem is how and on which level flow of control patterns can be embedded into

the MSC language. Last not least, an intuitive graphical representation is crucial.

Within Chapter 2, the constructs of Sequence Diagram are presented and compared with corresponding constructs in MSC. In Chapter 3, a proposal for the introduction of flow of control into MSC is given and an interpretation of flow of control based on event structures is presented. The possible interpretation as critical region in case of shared resources is mentioned. Chapter 4 contains concluding remarks and an outlook.

2. Sequence Diagrams and MSC

Sequence Diagrams are an advanced form of Ivar Jacobson's interaction diagrams [3]. Sequence Diagrams, however, now go much beyond the purely sequential interaction diagrams. The following sections provide a more detailed comparison of Sequence Diagrams and MSC.

2.1. General remarks

Sequence Diagrams and MSCs represent different views of system modelling and also refer to different application areas. Sequence Diagrams offer a software oriented view where the target program normally is running on one processor and the flow of control is passing between various program modules.

Contrary to that, MSCs are typically applied to the description of logically and physically distributed systems with asynchronous communication mechanisms. The specification by means of MSCs normally refers to a fairly high level of abstraction focusing on the causal relations between the contained events. The introduction of control flow to these highly concurrent behaviour descriptions appears much less obvious than in the case of Sequence Diagrams. The harmonization of MSC and Sequence Diagrams is intended to connect the software oriented view of Sequence Diagrams with the distributed conception of MSC.

The MSC example in Figure 2 which is obtained by translating the UML example in Figure 1 into MSC-96 clearly shows that the information of control flow is missing. The complete diagram in Figure 2 seems to be rather disconnected in comparison with Figure 1. In addition, the return message is not distinguished graphically from the calling message (a corresponding construct is missing in MSC-96). An advantage is that the MSC diagram is not overloaded by symbols.

2.2. Diagram area

For describing the interworking of several entities in a graphical form, the constructs of Sequence Diagrams and MSC have to be arranged according to the syntax rules in a diagram area.

Sequence Diagrams: The diagram area of Sequence Diagrams (Figure 1) has two dimensions: the vertical dimension represents time, in the horizontal dimension different objects are described. Usually, merely the ordering of events in time is shown. However, for real time applications, the time axis may actually show a metric. In comparison with MSC, the diagram area of Sequence Diagrams is not bounded by a frame.

MSC: Also the diagram area of MSC, e.g., Figure 2, is two-dimensional with the vertical dimension representing time and the horizontal dimension representing different instances.

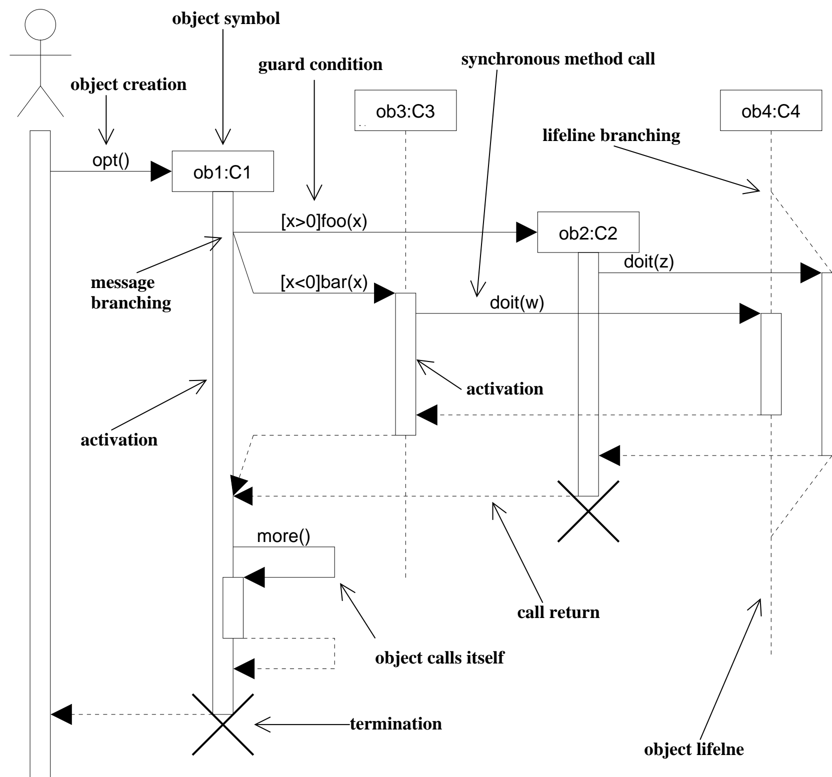


Figure 1. A Sequence Diagram example from the UML1.1 reference manual

Within MSC-96, only time ordering is provided. In MSC-2000, means for real time descriptions is expected. The diagram area of MSC is surrounded by a frame which represents the system environment. In contrast to instances (Section 2.3), no ordering of communication events is assumed at the environment.

2.3. Objects and instances

The entities which interwork are called *objects* in Sequence Diagrams and *instances* in MSC.

Sequence Diagrams: An object denotes a unique instance of an object class and is meant to be an entity to which a set of operations can be applied and which has a state that stores the effects of the operations. Graphically, an object in Sequence Diagrams is described by an *object symbol* and an *object lifeline*.

An object symbol is represented by a rectangular box which is drawn on top of a life line containing the object name and class name separated by a colon (e.g., ob1:C1 in Figure 1). An object lifeline is shown as a vertical dashed line. The lifeline represents the existence of the object at a particular time. In contrast to MSC instance axes, object lifelines do not possess an explicit end symbol unless the termination of the respective object shall be expressed.

Objects may create other objects and may terminate. If an object is created within a Sequence Diagram, then the message that causes the creation is drawn with its arrow head

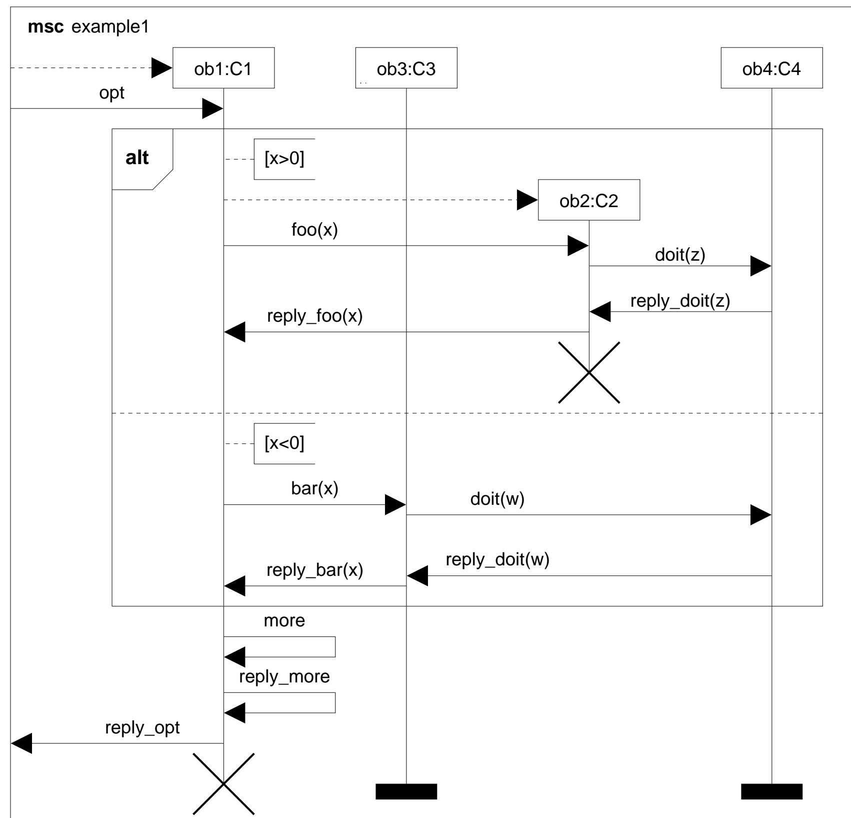


Figure 2. Sequence Diagram example in Figure 1 translated into MSC-96

attached to the object symbol of the created object. Contrary to MSC, create messages in Sequence Diagrams have the same syntax as normal messages, i.e., they show as solid lines with, possibly, names and parameters. An example of object creation can be found in Figure 1, the creation of ob1:C1 is the result of the creation message opt() coming from the system environment.

An object may terminate itself or its termination may be caused by a destroy operation called from some other object. The termination of an object is indicated by a cross symbol at the end of the object lifeline. In Figure 1 the objects ob1:C1 and ob2:C2 terminate themselves. In order to indicate the call of a destroy operation the termination symbol may be the target of a message arrow.

MSC: An MSC instance consists of an *instance head*, an *instance axis* and either an *instance end* or an *instance stop symbol*. The instance head is graphically represented by a rectangular box. Within the instance head, an entity type may be specified in addition to the instance name. An MSC instance axis corresponds to an object lifeline in Sequence Diagrams. Instances are shown by vertical lines or, alternatively, by columns. The instance axis line is solid to indicate the total ordering of events. A coregion may be attached to the instance axis in form of a dashed line. The instance end in an MSC diagram is indicated explicitly by an instance end symbol, i.e., a small solid rectangular box.

An instance can terminate itself by executing a stop event. The termination of an instance is graphically represented by a stop symbol in form of a cross at the end of the instance axis. In contrast to Sequence Diagrams, the stop event is not allowed to be the target of a message. This is due to the fact that the termination of an MSC instance by another instance is not considered a valid concept within MSC.

An MSC instance may be created by another MSC instance. The MSC *create symbol* is a dashed horizontal arrow which may be associated with a parameter list. A create arrow originates from a parent instance and points at the instance head of the child instance.

2.4. Communication

Communication among the entities in Sequence Diagrams and MSC are described in form of arrows. Different types of arrows are used to denote different types of communication. Generally, we may distinguish between synchronous and asynchronous communication. Synchronous means that the involved parties have to meet and during the meeting the communication is performed. Asynchronous means that the communication partners exchange messages which are buffered, i.e., the sending of messages and their consumption are decoupled.

Sequence Diagrams: The communication in Sequence Diagrams is based on *method calls*. Methods are attached to objects and can be compared to procedures in imperative paradigms. The call of a method is only possible from another method, but, calling and called method can be attached to different objects.

Method calls may be synchronous or asynchronous. In case of a synchronous method call, the calling object is suspended until the called object has performed the called method and has given back the result of the method. This can be seen as passing program control from the calling object to the called object and back. In case of an asynchronous method call, the calling object will not be suspended, i.e., may perform other actions during the execution of the called method, and may not wait for an answer about the success of the called method from the corresponding object.

In Sequence Diagrams, a synchronous method call is described by a solid arrow with a full arrow head. The return is shown as a dashed arrow with a thin arrow head. On the called side, both, method call and return message normally are connected by a vertical bar (thin vertical rectangle) representing the activated method. There are several variants admitted. Method call and return message may be combined to one bi-directional arrow with one full arrow head in the direction of the call and one thin arrow head in the return direction in case where no special activity is shown. In a procedural flow of control, the return message may be omitted, otherwise it is mandatory. In case of an asynchronous method call, the arrow describing the call message is solid and drawn with a half arrow head. The arrows of synchronous and asynchronous method calls are labelled with the call name and argument values (in parentheses).

Method calls in general are nested. An object may call itself directly or indirectly. In this case, the nested method symbols (vertical bars) are drawn slightly to the right of the previous one.

A method call in a Sequence Diagram normally is shown as a horizontal solid arrow from the lifeline of one object to the lifeline of another object. The horizontal arrow indicates that the duration required to call the method is atomic, that means, it is brief

compared to the granularity of the interaction and nothing else can happen during the call transmission. If the method call requires some time to arrive, during which something else can occur (such as a method call in the opposite direction) then the call arrow is slanted downward.

Method calls in Sequence Diagrams contain some additional concepts which are not contained in the MSC message concept: a call in Sequence Diagrams may contain sequence numbers (to show the sequence of the method call in the overall interaction) and guard conditions by placing Boolean expressions in braces.

MSC: MSC only provides a means to describe asynchronous communication. A construct for synchronous communication is still missing.

Communication in MSC is performed by the exchange of *messages*. One message represents two events, namely message sending and reception. A message is graphically represented by a solid arrow with a full arrow head. A message name is attached to the message arrow. To the message name, a parameter list may be assigned in parentheses.

Messages in MSC may be horizontal or have a downward slope and may be bent. The special form has no additional semantics, no concrete timing semantics is connected with the special graphical form. The message name, if necessary, with a parameter list in parantheses, is attached to the message arrow.

2.5. Methods and activation

The concepts of *methods* and the *activation of methods* are related to the OO view of the world. Denoting that a method is activated means that it has control and is performing some tasks. In case of a synchronous call the activation ends when the method decides to give the control back to the calling object, i.e., during the activation the calling party is suspended. For the asynchronous case, the called object decides itself when it returns into passive state.

Sequence Diagrams: The activation of a method graphically is shown as a tall thin rectangle whose top is aligned with its initiation time and whose bottom is aligned with its completion time.

The top of the activation symbol is attached to the arrow representing the method call, and, if there is a return message, the base of the symbol is attached to the tail of the return message. For example, in Figure 1 the object ob1:C1 is activated by the call opt() from the environment and remains activated throughout the entire Sequence Diagram. The object ob1:C1 terminates itself and returns some results to the environment. During the described communication behaviour, ob1:C1 may call a method in ob3:C3.

Activations may be nested, if a method of an object with already existing activations is called. In this case, the new activation symbol is drawn slightly to the right of the previous one, so that it appears to stack up visually. This can be seen on the lifeline of ob1:C1 in Figure 1. By means of the call more() the object calls a method which is owned by itself. Please note, that the existence of nested activations does not necessarily describe concurrency.

Actions being performed during an activation may be described in text next to the activation symbol or in the left margin, alternately the method call may indicate the action with its name.

MSC: The UML-terms method and activation do not have an immediate counterpart in MSC. However, for inclusion of such concepts into MSC-2000 a corresponding proposal has been made. It will be presented in Section 3 [8].

2.6. Branching, iteration and other structural concepts

Both in Sequence Diagrams and MSCs, some structural concepts have been introduced for a compact representation of either complex situations or more than one communication behaviour. Contrary to MSC, the structural concepts contained in Sequence Diagrams are rather limited and concern alternatives and iteration only.

Sequence Diagrams: A Sequence Diagram can exist in two forms: a *generic form* and an *instance form*. In the generic form which may contain branches and loops it describes several communication behaviours. In the instance form it describes one actual behaviour.

In generic Sequence Diagrams, object life-lines may branch and merge, thus showing alternatives (e.g., ob4:C4 in Figure 1). The branching of a method call is also allowed: it is represented by multiple arrows emanating from one common origin. The guards for messages included in Sequence Diagrams can be interpreted as an if-statement, without else-part. An example of a method call branching protected by guards can be seen in Figure 1. The choice of the method calls foo(x) and bar(x) by ob1:C1 depends on the fulfilment of the gate conditions [X>0] and [X<0].

A connected set of method calls in a Sequence Diagram may be enclosed and marked as an iteration.

MSC: Within MSC, the branching and iteration constructs can be described by inline expressions. Their clarity is an advantage of MSC. A corresponding construct for method call guards, however, is missing. The inclusion of guards for alternatives is planned together with the inclusion of formal data descriptions for MSC-2000. Even then, as is shown by the example in Fig. 6, local data attached to one object cannot be simply translated into guards within the inline expression which in general refers to several objects. The problem of non-locality of choices in MSC has been pointed out already before [5].

2.7. Real time specification

In Sequence Diagrams and MSCs, usually only time ordering of events is described. For real time applications, however, the time axis may have an actual metric and time specifications and constraints may be added to the diagram.

Sequence Diagrams: Various labels indicating timing marks can be shown either in the margin or near the transitions or activations that they refer to. These labels may be used to indicate the transition time for messages: One label (e.g., a in Figure 3) may be attached to the tail of a method call representing the sending time, another label (e.g., b) may be attached to the arrow head indicating the reception time. These labels may be used in constraint expressions (e.g., [b - a < 5 sec]).

MSC: A corresponding explicit notation for the indication of time and time constraints does not exist in MSC-96. Time annotations can be provided in form of comments, but from a formal point of view the time annotations in Sequence Diagrams may also be seen as some sort of comments. A syntax definition for time descriptions in MSC-2000 is in preparation.

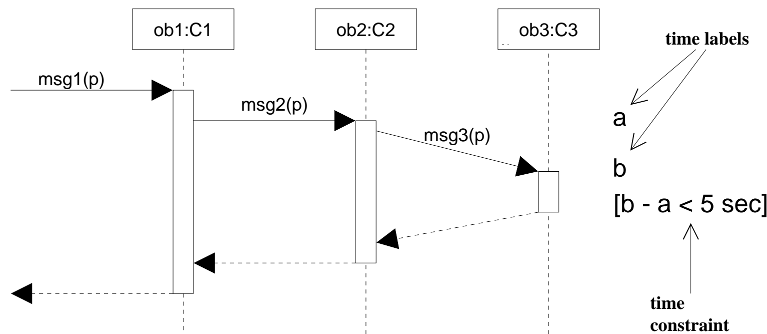


Figure 3. Sequence Diagrams with time labels

2.8. MSC concepts, not supported in Sequence Diagrams

MSC concepts, not or only partially supported in Sequence Diagrams, are *MSC reference*, *High level MSC* (HMSC), *inline expression*, *coregion*, *generalized ordering*, *instance decomposition*, *gates*, *special timer constructs*, *action symbols* and *conditions*.

MSC reference, HMSC, inline expression: MSC references and HMSC are not supported by Sequence Diagrams in UML 1.1. Inline expressions partially enter Sequence Diagrams. According to the UML 1.1 manual, a connected set of messages may be enclosed and marked as an iteration which is very close to the idea of inline expressions. For a scenario, the iteration indicates that the set of messages can occur multiple times. However, no concrete syntax showing the graphical representation of this iteration construct is provided.

Coregion and generalized ordering: Sophisticated ordering constructs like coregion and generalized ordering are not supported in Sequence Diagrams. A total ordering of events along each object lifeline is assumed.

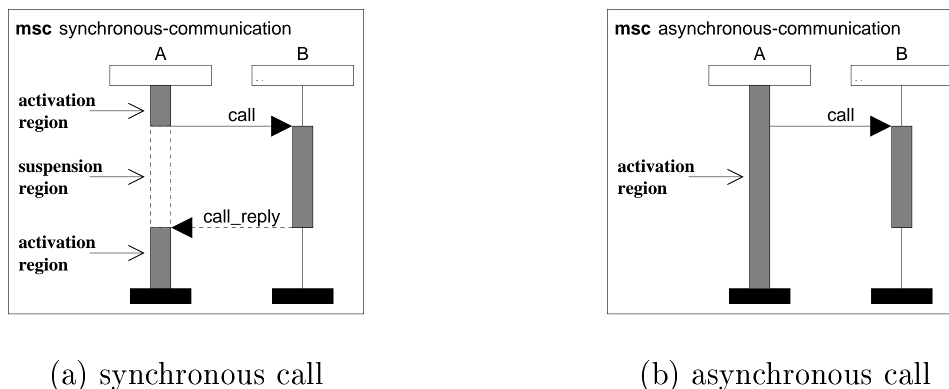
Instance decomposition: Instance decomposition admitting refinement of instances is not supported.

Gates: As already said before, Sequence Diagrams do not contain an explicit environment and therefore also no explicit gates. The environment may be represented by an additional environmental lifeline, like the leftmost lifeline in Figure 1, or by messages with no lifeline as target or source.

Special timer constructs: Special timer constructs are not part of the UML Sequence Diagram syntax. MSC includes language constructs for expressing the setting, resetting and time-out of timers.

Action: An explicit action symbol containing informal text is not included in Sequence Diagrams. An action being performed may be labelled in text next to the activation symbol or in the left margin, alternately the name of a method call may indicate an action.

Condition: Condition symbols indicating initial, final, intermediate, global and non-global states do not exist in Sequence Diagrams.



(a) synchronous call

(b) asynchronous call

Figure 4. Synchronous and asynchronous calls in MSC

3. Introduction of flow of control into MSC

In this section we make a proposal for the introduction of flow of control into MSC. For this we assume that objects and method calls of Sequence Diagrams can be translated into instances and messages in MSC.

3.1. Definition of flow of control for plain MSCs

Within Sequence Diagrams the notion of *flow of control* plays an important role. A strict distinction is made between passive objects which do not have control and active objects which have control. "Having control" means that an object which has control is able to execute its program code independently from other objects. An active object may calculate some data, may communicate asynchronously with other objects or may call other methods.

Thus, only active objects can call a method whereas passive objects can be activated by a method call. In the simplest case of a sequential program and synchronous communication, only one object has the control at a time. By means of a synchronous method call, the control is passed from one object to another.

In concurrent systems, the control may be distributed over several objects, i.e., several objects may have control at the same time. Depending on the level and purpose of specification the control flow in Sequence Diagrams may be indicated explicitly in graphical form whereby the active part is represented by the activation construct (Section 2.5).

The possibility to explicitly indicate the flow of control in a similar way seems to be essential for the acceptance of the MSC language for OO modelling. It certainly offers a significant new visualization of the processes occurring in the system. Because of that, a corresponding activation construct, the *activation region*, has been proposed recently within ITU-T for MSC-2000 [8]. An activation region indicates the activity carried out due to a synchronous or an asynchronous method call. It also indicates the flow of control in an MSC since the activation denotes which instance has control.

It turns out that the activation region alone is not sufficient for a clear definition of flow of control in MSC. For the detailed specification of the flow of control, two regions are employed (Figure 4a):

Activation region: The *activation region* may be explicitly represented by a tall thin vertical shadowed rectangle like in Sequence Diagrams. An activation region describes that an instance has been activated.

Suspension region: The *suspension region* indicates the blocking state of a synchronous call which is explicitly represented by a tall thin vertical white rectangle with dotted vertical border lines. The suspension region has no counterpart in Sequence Diagrams. It has been suggested to clearly distinguish the blocking state within a synchronous call from the activation region. In Sequence Diagrams, a suspension is graphically indicated as an activation which is rather misleading.

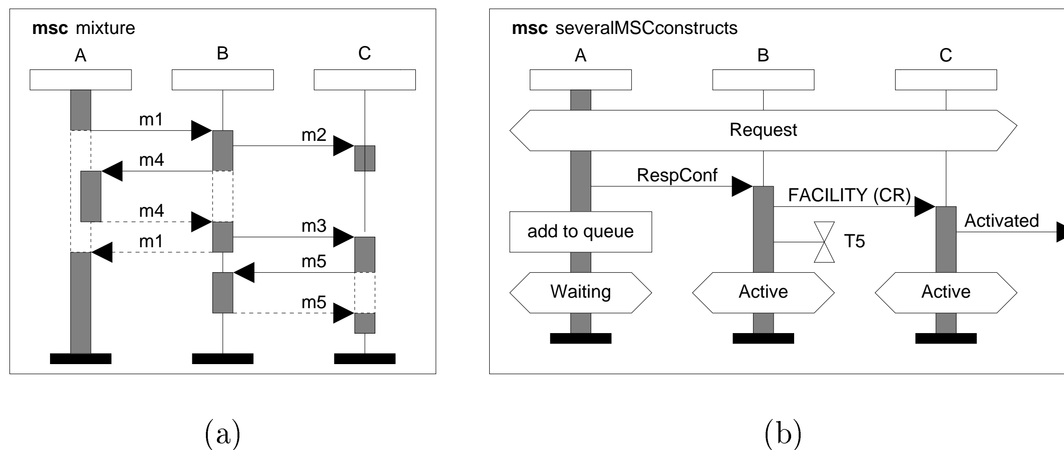
For modelling method calls in MSC, two kinds of calls are used: synchronous calls with blocking mechanism for the calling instance and asynchronous calls without blocking mechanism. It should be noted that the term *synchronous call* used in this context refers to the blocking mechanism and does not mean that the communication is atomic. A synchronous call is represented by a normal message and a message reply in form of a dashed arrow (Figure 4a). The asynchronous call is represented by a normal message (Figure 4b). Due to the suspension region, synchronous and asynchronous calls are sufficiently distinguished so that no special arrow head symbols are necessary as is the case in Sequence Diagrams. As shown in Figure 5a the mixture of synchronous and asynchronous calls is also possible and does not lose transparency for the reader.

It should be noted that the specification of the control flow should be left as an option to the user depending on the purpose and on the stage of design. In particular, no uniform representation is demanded, i.e., even within one MSC diagram it should be admitted to indicate the flow of control only for certain parts.

On *normal* instances, i.e., outside of activation and suspensions regions, we assume the usual syntax rules for message events [12] with few extensions concerning synchronous calls: Synchronous calls are also allowed from and to normal instances. As a static semantics rule, it is requested that no events may occur between a synchronous call and the corresponding call reply.

The description of the flow of control for plain MSCs is governed by few relatively simple rules:

1. An activation region starts with a call input or with the instance begin (see also Rule 13).
2. In case of a synchronous call, the activation region is terminated by the sending of the corresponding call reply.
3. In case of an asynchronous call, the activation region terminates before the reception of the subsequent new call input.
4. Both, asynchronous and synchronous calls can start from an activation region.
5. If a synchronous call starts from an activation region, the region goes over to a suspension region.
6. If an asynchronous call starts from an activation region, the region remains active.



(a) (b)

Figure 5. Use of flow of control in MSC

7. Call inputs must not enter activation regions.
8. A suspension region starts with a synchronous call from an activation region.
9. A suspension region is terminated by the reception of the corresponding call reply.
10. No calls may start from a suspension region.
11. Call inputs must not enter suspension regions apart from a direct or indirect synchronous call of the instance to itself in which case the suspension region goes over to an activation region. The activation region is then drawn slightly shifted above the suspension region which indicates that the suspension region is only interrupted.
12. Other orderable events like create output, action, timer set and reset may be attached to an activation region. Time-out should be treated like call inputs.
13. We allow activation and suspension regions to be split between different MSCs, in analogy to timer events. If an activation or suspension region is continued in a subsequent MSC, it finishes at the instance end of the first MSC and starts at the instance begin of the second MSC. As a special case, an instance may start with an activation region from the very beginning (*start instance*).

In order to get an intuitive idea, the activation region may be compared with SDL state transitions triggered by message inputs (Figure 5b). The SDL start transition corresponds to the *start instance*. A formal syntax and semantics description may be derived from these rules.

In the following, the integration of a syntax description for the flow of control into the MSC syntax is sketched only. For the synchronous communication, a return message (graphically represented by a dashed arrow line) has to be introduced in addition to the existing message construct. Incomplete return messages should be introduced in analogy to incomplete messages. The return message events should be added to the list of orderable events. Return message events follow the same syntax rules as the normal message events,

but a number of new static semantics rules has to be added, e.g., a return message input has to occur always after the corresponding message output. Activation regions and suspension regions may be integrated into the MSC syntax description similarly to coregions, i.e., in the graphical syntax an activation symbol has to be introduced containing an activation event area, and a suspension symbol containing a suspension event area. Activation regions optionally may follow a message input, suspension regions always have to follow synchronous message outputs within activation regions. Activation regions also may contain higher order language constructs like inline expressions and MSC references. Finally, activation regions and suspensions regions have to be added to coregions. The inclusion of flow of control into the structural concepts of MSC will be discussed in Section 3.3.

3.2. Interpretation of flow of control

Within the ITU-T MSC standardization group, the formal definition of flow of control and its semantics has attracted special attention. The following definition has been given in terms of event structures abstracting from special regions:

> *"Flow of control can be understood as a sequence of events also distributed over several instances, but still representing the accomplishment of one behavioral purpose.*
>
> *The sequence of events should be such that two events following each other in the flow of control either are events following each other on the same instance axis or are the two events of the same message.*
>
> *Flow of control may be forked to a flow of control tree / graph, and several flow of controls may be present in an MSC. A fork then 'produces' a new flow of control that needs a new identification.*
>
> *One instance may be involved in more than one flow of control. Independence of flow of control must be defined. It is clear that two flow of controls do not share the same event (other than possibly the forking event). Other restrictions do not seem to be needed or desirable."*

This formulation of flow of control certainly gives a completely new view within MSC: A flow of control appears as a special kind of event flow whereby emphasis is put on the *independence* of different flows of control. At present, this *independence* only refers to the exclusion of shared events since MSC is defined in terms of event structures. Intuitively, control structures also have the interpretation of *critical regions* with respect to shared resources. Depending on the role which in the future the inclusion of formal data concepts within MSC will play, this provides a special semantics of control structures already on the MSC level or only on later stages of design and implementation. Without data concepts, flow of control seems to have no special dynamic semantics but it implies several static semantic rules and it contributes strongly to the intuition.

3.3. Inclusion of structural concepts

In the following, a generalization of the flow of control to structural concepts is sketched. It should be noted that the inclusion of structural concepts certainly needs further elabora-

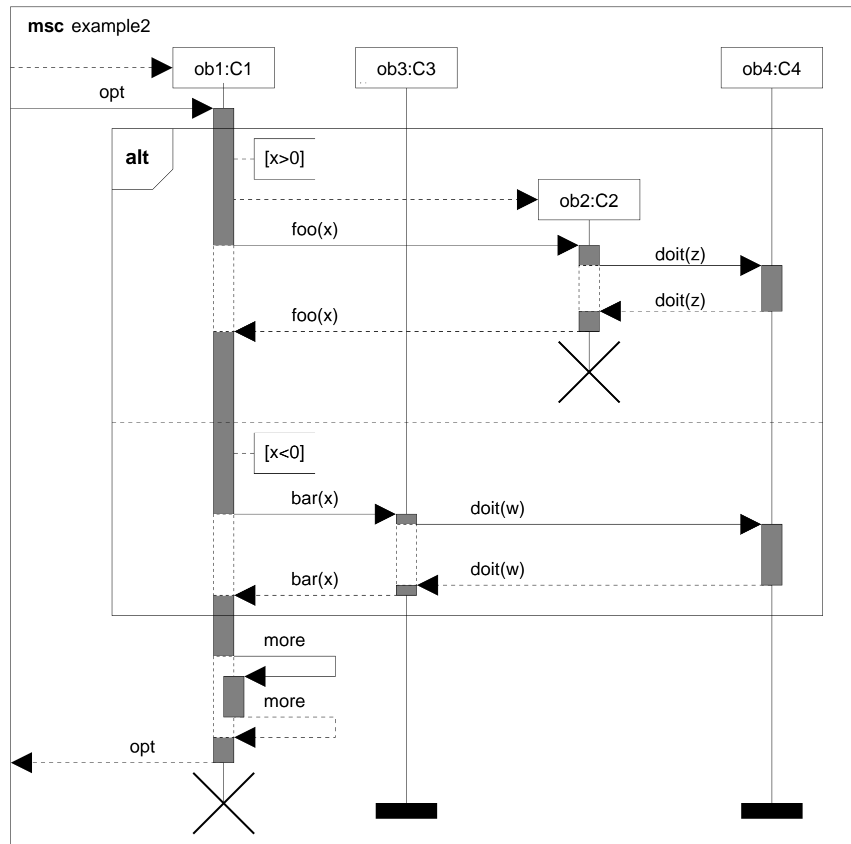


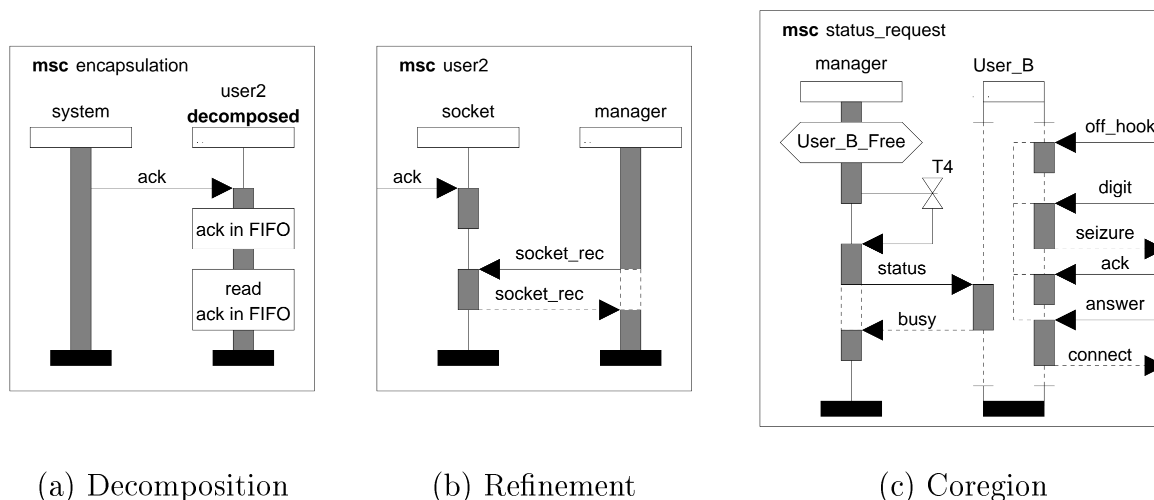
Figure 6. MSC in Figure 2 extended by flow of control

tion. In particular, the representation of flow of control within nested structural concepts, e.g., inline expressions, poses special problems since it soon loses transparency.

MSC references and inline expressions: The definition of flow of control for plain MSCs can be generalized to MSC references and inline expressions in a straightforward manner. The rules stated in Section 3.1 have just to be applied to individual inline sections or to the plain MSCs referred to by MSC references. An illustrations for the inclusion of flow of control is provided for inline expressions by Figure 6.

Instance decomposition: MSC offers the possibility of instance refinement by means of decomposed instances. This mechanism can be used within OO modelling to describe objects including other objects and communication behavior caused by the encapsulation, i.e., reading and changing encapsulated data structures by calling special access functions [7]. By that, the behavior of systems can be modelled on different levels of abstraction. The flow of control on the decomposed instance may be specified according to the same rules as on normal instances (Figure 7a). However, no formal mapping to the flow of control of the refining MSC (Figure 7b) is assumed.

Coregion and generalized ordering: MSC provides several means for the specification of parallel activities within one instance. Apart from parallel operator expressions, the



(a) Decomposition (b) Refinement (c) Coregion

Figure 7. Instance decomposition, instance refinement and coregion (generalized ordering)

coregion and generalized ordering constructs are tailored for this purpose. Such constructs allow the specification of logically and causally independent tasks in a form where no time ordering between them is provided. Such a specification is ideally suited on early stages of design where decisions concerning implementation are not yet made. An introduction of the concept of flow of control to coregions and generalized ordering constructs imposes some additional restrictions due to the static semantics rules stated in Section 3.1. In case of an explicit specification of activation and suspension regions, we assume a total event ordering along these regions. Figure 7c shows the case where activation regions are embedded in a coregion. The activation regions are partially ordered by means of generalized ordering constructs graphically represented by dotted lines. The example demonstrates that one object may be able to cope with several method calls in parallel. The order of execution of events is left open apart from the prescribed generalized ordering and the total order imposed by the activation regions themselves.

In case of non-explicit representation of flow of control, some ordering consistency rules have to be obeyed within the coregion since otherwise reception and sending of method calls might be arbitrarily interchanged. These ordering rules can be derived from the static semantics rules provided in Section 3.1.

4. Conclusion and outlook

It the preceding chapters, it has been shown that flow of control can be integrated consistently into MSC. Therefore, the representation of the flow of control should be included in MSC-2000. Within this paper, the most important features of flow of control have been tackled. As can be seen from Figure 6 in Chapter 3, some constructs in Sequence Diagram have no immediate counterpart in MSC: At present, there are no formal means in MSC to express guard conditions, a deficiency which hopefully will be removed in MSC-2000. Since the create construct in Sequence Diagrams implies a method call, e.g., in Figure 6 the creation of object ob2:C2 has been described by a create message

followed by a MSC method call message. Possibly, a better way of harmonization with UML may be found in the future. In addition, the different kinds of messages need further elaboration. Certainly, one also needs special MSC language constructs to discriminate between asynchronous and synchronous messages in the sense of atomic or non-atomic events. Concerning the two regions proposed in Section 3, we have restricted ourselves to simple flow of control descriptions governed by basic syntax rules. For the specification of flow of control within highly concurrent systems, these rules may turn out to be too stringent. Therefore, a generalization has been proposed by the ITU-T MSC standardization group whereby also call inputs and asynchronous replies are allowed to enter the activation region. The usefulness and graphical form of such more complex flow of control descriptions certainly need further investigations. Beyond that, the development of a formal semantics for flow of control is still in a beginning state.

REFERENCES

1. M. Andersson, J. Bergstrand. *Formalizing Use Cases with Message Sequence Charts.* Master Thesis, Lund Institute of Technology, 1995.
2. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A System of Patterns, Pattern Oriented Software Architecture.* John Wiley & Sons, New-York 1996
3. I. Jacobson et al. *Object-Oriented Software Engineering, A Use Case Driven Approach.* Addison-Wesley, 1994.
4. O. Haugen. *The MSC-96 Distillery.* In SDL'97: Time for Testing - SDL, MSC and Trends, Proceedings of the 8th SDL Forum in Evry, France (A. Cavalli and A. Sarma editors), North-Holland, Sept. 1997.
5. P.B. Ladkin, S. Leue. *Four issues concerning the semantics of message flow graphs.* Formal Description Techniques VII, FORTE 94 (D. Hogrefe and S. Leue editors), Chapman & Hall, 1995.
6. S. Mauw. The formalization of Message Sequence Charts, In: Computer Networks and ISDN Systems - SDL and MSC) (Guest editor: O. Haugen), Volume 28 (1996), Number 12, June 1996.
7. A. Rinkel. *MSC: A universal tool for describing and analysing communication structures.* Technical Report, ITU-T Experts Meeting St. Petersburg, April 1995.
8. E. Rudolph. *Control flow for synchronous and asynchronous calls, a unifying approach (UMSC).* Technical Report, ITU-T Experts Meeting Sophia Antipolis, October 1998.
9. E. Rudolph, P. Graubmann, J. Grabowski. *Tutorial on Message Sequence Charts (MSC-96).* Forte/PSTV'96, Kaiserslautern, October 1996.
10. J. Rumbaugh, I. Jacobson, G. Booch. *The Unified Modelling Language, Reference Manual Version 1.1.* Rational 1997.
11. ITU-T Rec. Z.120 (1993). *Message Sequence Chart (MSC).* Geneva, 1994.
12. ITU-T Rec. Z.120 (1996). *Message Sequence Chart (MSC).* Geneva, 1996.
13. ITU-T Rec. Z.120 (1998). *Message Sequence Chart (MSC)*, Annex B. Geneva, 1998.