Time in Message Sequence Charts: A Formal Approach
(Extended Abstract)

Piotr Kosiuczenko

Institut für Informatik, Ludwig-Maximilians-Universität, Oettingenstr. 67, D-80538 Munich, Germany.
E-mail: kosiucze@informatik.uni-muenchen.de

Abstract

Message Sequence Charts (MSC) is a graphical trace language for description and specification of communication behaviour of system components and their environment by means of message interchange. The goal of this paper is to provide a formal semantics for time aspects of MSC. Proposed semantics is based on Timed Maude which is an object-oriented real-time language. It extends a Maude semantics designed for the "untimed" part of basic MSC. We show how Timed Maude can be used to specify timers, and how they can be used to specify timing of an action, and timing imposed on message deliverance.

1. Introduction

Message Sequence Charts (MSC) have been introduced to provide a trace language for description and specification of communication behaviour of system components and their environment by means of message interchange [ITU 93, 96]. They are used as a support for object-oriented languages, in particular to formalize interaction diagrams (c. f. [AnBe 95], and [WiK 96] for an algebraic approach to interaction diagrams). MSC may be independently used for requirement specification, interface specification, validation and simulation, test case specification and documentation of real-time systems. The goal of this paper is to provide a formal semantics for time aspects of MSC (the reader is referred to [Kos 97b] for the full version of the paper). There are already some formal semantics for MSC (cf. eg. [MaRe 94]). In general proposed semantics do not allow to define time aspects either. In general, semantics mentioned above do not allow to formalize notions like decomposition, environment, local actions, and timers (c. f. [LaLe 95]). The only semantics which allows to specify timers was defined in [AHP 96], but it covers only time aspects.

The semantics introduced in [Kos 97a] is based on object-oriented language Maude [Mes 92]. It provided a complete semantics for all constructs of basic MSC except of timers. The semantics has several advantages. There is a direct correspondence between a textual MSC specification and the corresponding formal specification. The formal specification introduces explicitly objects and messages, and allows to talk about actions, message sending and receiving, objects creation and deletion. It provides a clear concept of condition, as well as flexible and natural concepts of vertical and horizontal compositions. Moreover, the semantics is compositional, because thanks to the composition operators, semantics of a composed MSC can be defined in terms its of components. Another advantage is, that a part of (Timed) Maude gains a graphical representation.

In this paper, we will extended this semantics to allow for specifying behaviour in time. As the underlying formalism, we use Timed Maude which is build on Timed Rewriting Logic (TRL) as Maude on Rewriting Logic [KoWi 97]. In our semantics, an object gets its timing from its attributes. Timing of an object (and consequently of a system) is described using timers only. Timers can be attached to states and to messages. An object can spawn multiple timers running in parallel. States of an object are multisets consisting of timers and atomic states. There are two types of atomic states: static states which can last arbitrary long or until a specified event happens, and intermediate states which last only for 0 time.

2. Formal background

In this section, we describe shortly the underlying formalisms, but the reader is referred to [Kos 97b, Kos 97a, KoWi 97, MeWi 92] for details. Timed Maude is a variant of Maude where rewriting is replaced by timed rewriting. This means also that in Timed Maude inheritance is treated in the same way as in Maude by means of subsorting. An (*object*) *class* is declared by an identifier, a list of attributes and their types. A message is a term that consists of the message´s address (the identifiers of the objects the message is addressed to) and, possibly, parameters (in mixfix notation). An object is represented by a tuple - more precisely, by a term - comprising a unique object identifier, an identifier for the class the object belongs to, and a multiset of attributes with their values constituting its state, e.g. term of the form <G : Gate | up : state> represents an object with name G belonging to the class Gate. The attribute "state" has value up. The sorts Message and Object are considered as subsorts of the sort Configuration.

A Timed Maude program makes computational progress by rewriting its global state (called "configuration"). A configuration is a multiset, or a bag, of objects and messages. It is represented by a term of the form: $o_1 \otimes .. \otimes o_k \otimes m_1 \otimes .. \otimes m_l$ where $\otimes$ is the function symbol for multiset union (will usually write $o_1..o_k\ m_1..m_l$). The empty configuration is denoted by ε. A timed step transforms a configuration into a subsequent configuration. It is of the form $t_1 -g\ r \rightarrow t_2$, where g is a label and t_1, t_2 are terms of sort configuration. An object gets its timing from its attributes. We assume, that time progress in a synchronous way in all components (cf. [KoWi 97]).

3. Model of Time

The semantics of MSC is based on following assumptions. If time constraints on an object behaviour or a message deliverance are considered, then corresponding timer is attached to object state or the message, respectively. An object can spawn multiple timers running in parallel at a given time. Therefore we treat states of an object as multisets. Each element of such a multiset can be a timer or an atomic state. An atomic state can be declared either static or dynamic. A static state can last arbitrary long, whereas a dynamic state is supposed to change in time. The duration of a dynamic state is assumed here to be 0 time units. Therefore a dynamic state can be treated as an intermediate state.

Maude is in general much more expressive than MSC and only a part of it can be illustrated with the graphical MSC notation. There are only three kinds of atomic steps a MSC instant can execute: sending or receiving a message, creating or deleting of another object, and a special action which changes object´s state only. Consequently, the set L of atomic steps is of the form {?, !, start, stop, ac_1,. ., ac_n}, and the corresponding rewrite formulas are of the form:

rl	$<O \mid state : s_1> - ! m\ 0 \rightarrow <O \mid state : s_2>\ m$.
rl	$m <O \mid state : s_1> - ?m\ 0 \rightarrow <O \mid state : s_2>$.
rl	$start(o, o_1) - start\ O_1\ 0 \rightarrow <O_1 \mid state : s>$.
rl	$stop(o, o_1) <O_1 \mid state : s> - stop\ O_1\ 0 \rightarrow \varepsilon$.
rl	$<O \mid state : s_1> - ac_i\ 0 \rightarrow <O \mid state : s_2>$.

where message start and stop are predefined message types corresponding to object creation and deletion: start, stop : $Oid \times Oid \rightarrow Msg$.

If one of the conditions above holds, then we say, that object O preforms the corresponding step.

We extend here Timed Maude by some specific rules to handle time elapse (Time elapse is denoted by τ.) A term is assumed to be dynamic, if it is not declared to be static. The first rule specifies, that static terms do not change in time:

Static term rule: For every term $t : s$ static and for every $r \in$ Time holds:

$$t - \tau\ r \rightarrow t .$$

Observe, that term T (defined below) is not static. We assume that all messages without timers are static: (i. e. m : Mes static). To be able to deal with composed states and configurations, we use a polymorphic multi-sum operation f. The following rule assures uniform time elapse in all components of a state or a system (this is due to the fact, that all variables are flexible). It is specially important, when an instance starts multiple timers which then run in parallel with the same rate.

Composition rule: For variables x and y holds:

$$x \otimes y - \tau\ r \rightarrow x \otimes y .$$

The following rule specifies, that objects get their timing from their states (let us remind, that all variables are flexible and that states are multisets which may consist timers.):

Object timing rule: For variable x : State holds:

$$<O : Oid \mid x : State> - \tau\ r \rightarrow <O : Oid \mid x : State> .$$

To be able to interleave steps which happen at the same time we add the following rule:

0-time reflexivity rule: For all terms t holds:

$$t - \tau\ 0 \rightarrow t .$$

It is worth of noticing, that in general we do not require reflexivity neither for objects nor for messages which can have timers attached, because otherwise, we would not be able to express hard real-time constraints.

3.1 Example: timers

Let us consider two simple examples: a timer attached to a state, and a timer attached to a message (see the figure). The MSC is to be read from top to bottom. To each object corresponds a vertical line defining total ordering of events. Initial (final) states of objects are indicated by empty (black) boxes. Timers are used to control time elapse in a system.

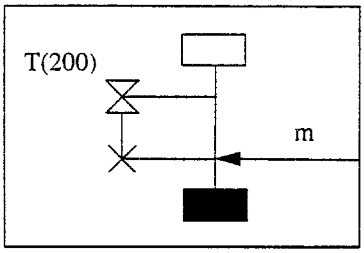

Figure. Timer

A timer can be set with a certain time value and from this moment on it counts down decreasing the value. When it reached value zero it can not be further rewritten and blocks an object until a specified event happens. We define a timer by a unary function symbol from sort Time to a new sort "Timer". This leads to the following specification:

op dynamic $T : Time \longrightarrow Timer$,

rl $T(r_1 + r_2) -\tau\, r_1 \rightarrow T(r_2)$.

Observe, that the time value of T(0) can not be decreased by any positive value $r > 0$. Timer can be reset by a specified event, like arrival of a message, sending of a message, timeout event of another timer and so on. For example one can define a rule specifying that timer T will be reset when message m arrives:

$$m <O_1 \mid T(r) \otimes st> - ?m\ 0 \rightarrow <O_1 \mid st> .$$

The figure shows timer T which was set with value 200. Before 200 seconds elapsed, message m arrived and T was reset. The double triangle indicates when the timer was set and the cross indicates when the timer was reset.

Delay caused by message transmission can be modeled using another kind of timer. Let m be a message, we can attach a timer to m. It is convenient to specify such a timer as a binary function del(m, r) which is to guarantee that message m will be delivered with delay r. The specification is as follows:

op dynamic $del: Msg \times Time \longrightarrow TMsg$

rl $del(m, r_1 + r_2) - \tau\, r_1 \rightarrow del(m, r_2)$.

where TMsg is the sort of messages with attached timers. Observe, that for a message of the form del(m, 0) there is no rule allowing time to pass, therefore such a message must be read instantaneously (in 0 time) (c.f. [OlKW 96]).

4. Concluding remarks and future research

In this paper, timing of a system was modelled using timers only. Atomic states are not capable of controlling time elapse. Timers can be attached to states and to messages to control object´s and message timing, respectively. In the paper [AHP 96], there have been proposed two possibilities to control time elapse: using timers and using direct time constraints imposed on transitions and message deliverance. It is interesting to observe, that the second possibility is superfluous, because timers are sufficient both for defining timing of an object behaviour as well as time constraints for message deliverance. State duration can be defined using timers only. One can use two timers to define, for example, that a given state lasts for 4 to 5 seconds. The first timer would be set with value 4 the second with value 5. Here, we have decided to define instantaneous steps triggered by timeouts. Such steps can be also specified using two timers set with the same time value. Another possibility is to allow non-determinism in the choice of timer value. Namely, a timer can be nondeterministically set with a value from a given interval (in the previous case, it can be any value from interval [4, 5]). This allows to avoid spawning to many timers, which would result in hardly readable specifications and figures.

Proposed semantics is still not complete. There are constructs like inline expressions and high level MSC [ITU 96] which do not posses formal semantics. Therefore, we will extend presented semantics also to this case.

References

[AnBe 95] M. Anderson, J. Bergstrand. Formalizing Use Cases with Message Sequence Charts. Master Thesis, Telelogic AB and Dep. of Communication Systems, Lund Institute of Technology, May 1995.

[AHP 96] R. Alur, G. Holzmann, D. Peled. An Analyzer for Message Sequence Charts. Proc. TACAS'96, Passau, Germany, LNCS 1055, 1996, pp 35-48.

[ITU 93] ITU-TS Recommendation Z.120. Message Sequence Charts (MSC). ITU-TS, Geneva, 1993.

[ITU 96] ITU-TS Recommendation Z.120. ITU-TS Recommendation Z.120. Message Sequence Charts (MSC). ITU-TS, Geneva, 1996.

[KoWi 97] P. Kosiuczenko, M. Wirsing. Timed Rewriting Logic with an Application to Object-Based Specification. Science of Computer Programming. 28(2-3), 1997.

[Kos 97a] P. Kosiuczenko. Formal Semantics of Basic Message Sequence Charts. Proc. of GI/ITG Fachgruppe "Kommunikation und Verteilte Systeme", Berlin, 1997.

[Kos 97b] P. Kosiuczenko. Time in Message Sequence Charts: a Formal Approach. Technical Report Nr. 9703. Ins. für Informatik, Ludwig-Maximilians-Universität, München, 1997. http://www.pst.informatik.uni-muenchen.de/.

[LaLe 95] P. Ladkin, S. Leue. Comments on a Proposed Semantics for Basic Message Sequence Charts. The Computer Journal, 37(9), January 1995.

[MaRe 94] S. Mauw, M. Reniers. An Algebraic Semantics of Basic Message Sequence Charts. The Computer Journal, 37(4), 1994.

[Mes 92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 1992, 73-155.

[OlKW 96] P. Olveczki, P. Kosiuczenko, M. Wirsing. Steamboiler specification problem: an algebraic object-oriented solution. In J. Abrial, E. Boerger, H. Langmaack (eds.): Formal Methods for Industrial Applications. LNCS 1165, 1996.

[WiK 96] M. Wirsing, A. Knapp. A Formal Approach to Object-Oriented Software Engineering. In J. Meseguer (ed.): Rewriting Logic and its Application, Proc. of the First. Intern. Workshop, Electronic Notes in TCS, Vol. 4, 1996.