# An Automata Based Interpretation of Live Sequence Charts*

Jochen Klose[1] and Hartmut Wittke[2]

[1] University of Oldenburg
Jochen.Klose@Informatik.Uni-Oldenburg.de
[2] OFFIS
Wittke@Offis.de

**Abstract.** The growing popularity of sequence charts, first of all Message Sequence Charts and UML Sequence Diagrams, for the description of communication behavior has evoked criticism regarding the semantics of the charts which led to extensions of these standardized visual formalisms. One such extension are Live Sequence Charts which allow to distinguish mandatory and possible behavior in protocol specifications. In the original language definition for LSCs the semantics are only described informally, although a sketch for a possible formalization has been provided as well. In this paper we intend to fill in the semantic blanks of the original LSC definition. Following the sketched path we define the semantics of an LSC by deriving a Timed Büchi Automata from it. We also consider qualitative and quantative timing aspects. We finally show how LSCs are integrated into a verification tool set for STATEMATE designs.

## 1 Introduction

In recent years the use of Embedded Control Units (ECUs) has become more and more widespread in industry, especially in automotive and avionics applications. Many of these ECUs have to satisfy safety critical requirements. Developing such systems requires non-trivial effort to ensure correctness of the design. Therefore many companies have come to realize the usefulness of (semi-)formal methods in the development process of safety critical ECUs.

One example of a semi-formal specification technique are Message Sequence Charts (MSCs), a graphical formalism which is concerned with the communication behavior of protocols. MSCs have been standardized by the ITU (International Telecommunications Union) in Recommendation Z.120 ([IT96b]). Designed to capture protocol scenarios in the telecommunication area, MSCs can also be used to sketch scenarios of general interprocess communication.

Notwithstanding the fact that there exists a formal semantics [IT96a], we consider MSCs only a semi-formal specification technique, because there are

* Research in part supported by the German Research Council (DFG) within the USE project as part of the SPP Integration of Specification Techniques with Engineering Applications

T. Margaria and W. Yi (Eds.): TACAS 2001, LNCS 2031, pp. 512–527, 2001.
© Springer-Verlag Berlin Heidelberg 2001

still a lot of questions unanswered. For example, one MSC only specifies one scenario, i.e. one possible communication sequence of the system. But: What does a collection of MSCs for some system mean? Is progress along the instance axis enforced? What happens when a condition is evaluated to false? These and other open questions have been identified by several researchers (see e.g. [Krü99], [DH99]).

We follow the Live Sequence Charts (LSC) approach of [DH99][1] which is an conservative extension of standard MSCs. As explained in [DH99] LSCs can be used to distinguish between accepted and non-accepted sequences of variable valuations of systems (*runs*). LSCs extend the formalism of MSCs along the following lines:

- conditions are interpreted, they are not only treated as comments as in MSCs
- distinction between possible (standard MSC) and mandatory behavior. This includes the ability to
  - enforce progress along each instance axis,
  - specify if a message has to be received or not,
  - distinguish between LSCs which show only one possible communication (existential interpretation) and LSCs which show mandatory communication, i.e. it has to be exhibited by all system runs (universal interpretation),
  - distinguish between conditions that have to be satisfied and those that may be violated without generating an error.
- specification of activation conditions guarding the activation point of the LSC, and whether an LSC should be activated only at system start (*initial* activation mode) or whenever the activation condition evaluates to true (*invariant* activation mode)

The parts of an LSC which may be interpreted either as mandatory or possible are assigned a temperature, *hot* for mandatory and *cold* for possible elements. Thus we have messages, conditions and instance locations with temperatures. Graphically cold elements are depicted by dashed lines whereas hot ones are represented as solid lines (see figure 1).

We will substantiate the original paper [DH99] in two ways: We will on one hand provide a more concrete semantics for a subset of the original features using Timed Büchi Automata. On the other hand we will introduce timing annotations which are not covered by [DH99]. Our notion of time is discrete as we base our time model on the steps of a system exhibiting time-discrete behavior.

Before we go into the details of the process of transforming the LSC into an automata format (which we call *unwinding*) we need to explain the context in which we want to use LSCs. At the University of Oldenburg/OFFIS[2] the Statemate Verification Environment (STVE) has been developed over the last years which allows to verify safety-critical properties of Statemate designs (see [BBea99], [DDK99] or [DK01] for details). The Statemate tool from i-Logix

[1] A newer version of this paper is to appear this year: [DH01]

[2] Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme

is a commercial case tool which is based on Harel's statecharts [HP96] and is widely used in industry. The STVE translates the STATEMATE design into the input format of the underlying symbolic modelchecker and also lets the designer specify the properties to be verified in a graphical way. For this purpose *Symbolic Timing Diagrams* (STDs) are used, which have been developed at OFFIS as well. STDs allow to state constraints on the changes of values of inputs and outputs of the system under design; see [Fey96], [FJ97] for more information. The STD specification is translated into propositional temporal logic (see [Fey96]) which serves as input for the modelchecker.

The algorithm of unwinding LSCs explained in this paper is already implemented in a tool (cf. section 5), which is integrated into the STVE at the moment of writing. The roles of LSCs and STDs are complementary, STDs talk about one component (*black box view*) whereas LSCs are obviously much better suited to express properties about the interactions of components (*white box view*).

The paper is structured as follows: In section 2 we define the subset of LSC features considered here. Section 3 describes how the unwinding structure for an LSC is constructed and section 4 shows the subsequent transformation into a Timed Büchi Automaton. In section 5 we give an overview about the tool environment for verification of STATEMATE designs against LSC specifications. We give a summary and identify directions of further research in section 6.

## 2 LSC Subset

LSCs as presented in [DH99] provide a rich set of features. We will only treat a subset of these in the present paper in order to focus on the core concepts. The integration of LSCs into the STVE entails some other restrictions which are caused by STATEMATE. But we also add two features which were not covered in the original paper: *timing annotations* which allow to specify a lower and an upper bound between two subsequent locations on one instance. We borrow the interval notation used in both STDs [Fey96] and MSCs [AHP96]. Timer durations are consequently interpreted as a multiple of our discrete base time unit. Our interpretation of timing annotations and timers allows the user to specify runs of a system not only in a qualitative manner, but also to constrain event sequences by quantitative time bounds. The second new concept are *simultaneous regions*, which allow to specify the simultaneous observation of several events. In contrast, the non-deterministic symbolic transition system presented in [DH99], implements a pure interleaving interpretation of LSCs: only a single instance is allowed to proceed at a time. We feel that such an interpretation is not powerful enough in the context of STATEMATE, where the communicating activities run in parallel and can change arbitrary many variable values at the same time. Besides explicit simultaneous regions, our interpretation considers unordered events of a coregion or of different instances observable in any order including *simultaneity*.

In this paper we do not treat existential LSCs, because the universal interpretation seems to be the natural choice for formal verification as we want to prove that the entire system fulfills the specification. We feel that the intention of using an existential LSC is to get a satisfying run as a witness. This would

entail a modified unwinding algorithm which is out of scope for the present paper. Our algorithm can also handle sub-charts, the details of which we omit here due to limited space.

The following concepts are contained in our approach (see figure 1 for the graphical representation of the concepts): Hot and cold locations, hot and cold messages, hot and cold conditions, coregions, *simultaneous regions*, timer, *timing annotations*.

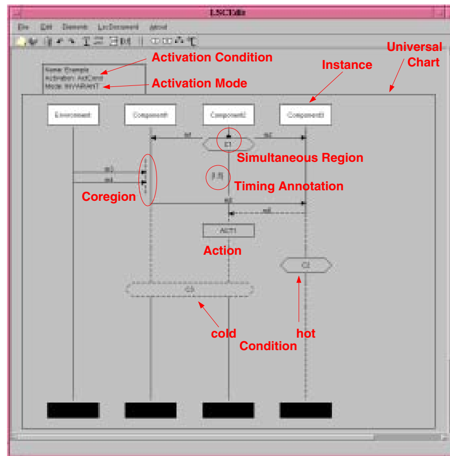



**Fig. 1.** LSC example

We restrict the setting of a timer to be bound (via a simultaneous region) to some sort of event. This is what we feel is the intention which was so far not expressible in MSCs: A timer is set when some event is observed and we then wait for some subsequent event.

## 3 Constructing the Unwinding Structure

In order to generate the unwinding structure from an LSC we first need to identify its building blocks. They are those elements of the chart which have to happen simultaneously, i.e which are indivisible. In our case the simultaneous region construct covers the majority of these elements since it encompasses both regular messages and regular conditions. The other two elements left, the instance head and the instance end, are of a more auxiliary nature. They are not unwound explicitly but the set of all instance heads/instance ends forms the start state/end state for the unwinding structure. The other elements of our LSC subset are either irrelevant for the structure or can be stated using the simultaneous region. Timer and timing annotations are not treated in the unwinding, as their timing information is considered later when transforming the unwinding structure into an automaton. Actions are disregarded because they hold no information relevant for the run. Coregions are not treated as constructs

of their own but as the separate simultaneous region constructs of which they are comprised. Neither do sub-charts form a separate construct as they can be expressed by their enclosed simultaneous region constructs as well.

The construction of the unwinding structure borrows the basic technique from the unwinding of Symbolic Timing Diagrams (cf. [Fey96]) using *Phases*. Informally a *Phase* shows how far the unwinding of the LSC has progressed. Starting with the *Initial Phase* all possible successor *Phases* are computed and this is iterated until the *Final Phase* is reached.

For the formal definition of the unwinding procedure we first introduce the concept of a *position* in an LSC: A *position* is simply a graphical point in an LSC. The atomic building blocks of an LSC are *events* which may be one of the following: (1) instance head , (2) instance end, (3) sending a message, (4) receiving a message, (5) the valuation of a condition, (6) setting a timer, (7) expiration of a timer or (8) the reset of a timer

In order to formally define events we introduce a number of sets. The first group of sets contains elements which are related to sets of positions, whereas the second group contains elements which are related to single positions.

| **Sets of objects of LSC $l$** | | | |
|---|---|---|---|
| **related to sets of positions** | | **related to single positions** | |
| $Instances(l)$ | set of instances | $Msgsend(l)$ | set of message sendings |
| $Messages(l)$ | set of messages | $Msgrecv(l)$ | set of message receipts |
| $Conditions(l)$ | set of conditions | $Set_Timer(l)$ | set of timer settings |
| | | $Reset_Timer(l)$ | set of timer resets |
| | | $Timeouts(l)$ | set of timeouts |
| | | $Timer(l)$ | $Set_Timer(l) \cup Reset_Timer(l) \cup Timeouts(l)$ |

Analogously to the chart-oriented sets of the second group we define the same sets for each instance $i$ of the LSC $l$. For conditions we define the set of condition valuations which are local to instance $i$ as the restriction of the shared condition for the whole chart to instance $i$: $Conds(i) := Conditions(l) \downarrow i$. We write $Conds(l)$ for $\bigcup_{i \in Instances(l)} Conds(i)$. We denote the instance head of instance $i$ by $\perp_i$ and the instance end of $i$ by $\top_i$.

**Events** Given these basic definitions we now formalize events:

**For LSC $l$ :**

$$Events(l) := \{ \perp_i \mid i \in Instances(l) \} \cup Msgsend(l) \cup Msgrecv(l) \cup Timer(l) \cup Conds(l) \cup \{ \top_i \mid i \in Instances(l) \}$$

**For instance $i$ :**

$$Events(i) := \{\perp_i\} \cup Msgsend(i) \cup Msgrecv(i) \cup Timer(i) \cup Conds(i) \cup \{\top_i\}$$

With each event $e \in Events(i)$ we associate its position given by the function $position(e)$. In order to handle simultaneous observation of multiple events we introduce for instance $i$ the maximal set:

$$Sim_Regions(i) := \{sr \subseteq Events(i) \mid \forall x, y \in sr : position(x) = position(y)\}.$$

We also associate a position with each $sr \in Sim_Regions(i)$ by the function $position(sr)$. Note that simultaneous regions are sets of *basic events*. Single events are treated as singleton simultaneous regions. Positions along one instance axis are totally ordered.

We now consider the set $Coregions(i)$ of coregions of instance $i$. A coregion $cr \in Coregions(i)$ starts at the graphical position $x$ of instance $i$ and ends at graphical position $y$. We define for $cr$ : $startpos(cr) := x$ and $endpos(cr) := y$ and $position(cr) := startpos(cr)$. For each $cr \in Coregions(i)$ we then define $contents(cr) := \{sr \in Sim_Regions(i) \mid position(cr) \leq position(sr) \leq endpos(cr)\}$.

In addition to a graphical position we associate a logical position with each simultaneous region which is used to determine the order along the instance axis. We call this logical position the *location* of the simultaneous region. For $sr \in Sim_Regions(i)$ we define :

$$location(sr) := \begin{cases} position(sr), & if\ \neg\exists\ cr \in Coregions(i) : sr \in contents(cr) \\ position(cr), & if\ \exists\ cr \in Coregions(i) : sr \in contents(cr) \end{cases}$$

Let $Locations(i) := \{\ location(sr) \mid sr \in Sim_Regions(i)\}$ be the set of locations of instance $i$. Note that for distinct $sr, sr' \in Sim_Regions(i)$ not located in the same coregion either $location(sr) < location(sr')$ or $location(sr') < location(sr)$ holds. Moreover, for distinct $sr, sr' \in Sim_Regions(i)$ located in the same coregion: $location(sr) = location(sr')$. Thus, with respect to coregions locations along one instance axis are ordered only partially. Concerning simultaneous regions and coregions we formulate the following well-formedness rules:

- Simultaneous regions located in a coregion must be singleton sets, or in other words, must be single events, because otherwise they would impose an order (simultaneity) on some of the events in the coregion.
- At most one condition valuation may be located in a simultaneous region. Several condition valuations can always be merged into one.
- Timer settings must only occur in simultaneous regions together with at least one non-timer setting event, because there has to be some reference point for the timer.

For the unwinding procedure we furthermore need to know the *predecessors* of each simultaneous region which are determined by the following function $(sr \in Sim_Regions(i))^3$:

$$predecessor(sr) := \begin{cases} \emptyset & , if\ sr = \{\perp_i\} \\ \{sr' \mid sr' \in Sim_Regions(i)\ \wedge \\ \quad location(sr') < location(sr) \wedge \\ \quad \neg\exists f \in Sim_Regions(i) : \\ \quad location(sr') < location(f) \\ \quad < location(sr)\} & , else \end{cases}$$

[3] Note that the set of predecessors usually contains just one element. Only a coregion as predecessor produces a set containing several elements.

This definition of a predecessor set for each location allows us to determine sets of legal event sequences along one instance axis. But simultaneous regions of one instance can be bound to other simultaneous regions on other instances. For example, shared conditions involve a simultaneous region on each instance sharing the condition. If one of the simultaneous regions of this set occurs, the other simultaneous regions of the set must occur simultaneously. This leads to the definition of

**Simultaneous Classes** Let $\leftrightarrow_{LSC}$ denote the equivalence relation "*has to happen simultaneously*", then:

$$Sim_Classes(l) := \{scl \subseteq \bigcup_{i \in Instances(l)} Sim_Regions(i) \mid \forall sr, sr' \in scl : sr \leftrightarrow_{LSC} sr'\}$$

The simultaneous classes impose the ordering between different instances. Here the constructs that satisfy $\leftrightarrow_{LSC}$ are

- shared condition valuations $c \in Conditions(l)$ which form a synchronization barrier, since $c$ has to be evaluated *simultaneously* at each involved instance
- sending and receiving a message $m \in Messages(l)$. This is only legal for models with zero delay communication like Statemate. For delayed (synchronous or asynchronous) communications sending and receipt of a message have to be treated separately. For simplicity's sake we only consider non-delayed communication in the remainder of this paper.
- a singleton simultaneous region

Note that $\forall sr \in \bigcup_{i \in Instances(l)} Sim_Regions(i) \exists^1 scl \in Sim_Classes(l) : sr \in scl$. For $scl \in Sim_Classes(l)$ we now define the set of simultaneous classes which have to be unwound before $scl$. We call this set the *prerequisites* for $scl$.

$$prerequisite(scl) := \begin{cases} \emptyset & , if\ scl \in \{\wp(\bigcup_{i \in Instances(l)} \{\bot_i\})\} \\ \{scl' \mid scl' \in Sim_Classes(l) \wedge \exists sr \in scl\ \exists\ sr' \in scl' : \\ \quad sr' \in predecessor(sr)\ \}, else \end{cases}$$

With these definitions we are now fully equipped to formally define the unwinding procedure. The procedure unwinds a LSC step by step by constructing sets of *simultaneous classes.*

**Unwinding Sets** Each step in the unwinding process is characterized by three sets:

- $History \subseteq Sim_Classes(l)$, the set of simultaneous classes which have already been unwound
- $Ready \subseteq Sim_Classes(l)$, the set of simultaneous classes whose prerequisites have already been unwound:
- $Fired \in \wp^+(Ready)$, the set of simultaneous classes which are unwound in the current phase[4]

[4] $\wp^+$ denotes the power set without the empty set.

In addition to these three sets we introduce the *Cut* through the LSC which keeps track of the progress of the unwinding, i.e. the Cut identifies the border line between elements which have already been unwound and those that still have to be considered. A cut can be visualized as a piece of rope lying across the LSC touching exactly one location of each instance. More formally we define the set of all cuts of an LSC $l$ as:

$$Cuts(l) := \{(x_1, .., x_n) \mid x_j \in Sim_Regions(j), 1 \leq j \leq n = |Instances(l)| \}$$

An unwinding phase $Phase_i$ consists of the sets $Ready_i$, $History_i$ and the vector $Cut_i$. Each phase is represented as a node in the unwinding structure which is therefore annotated with the the triple $(Ready_i, History_i, Cut_i)$; the phases are connected by edges annotated with elements of $Fired$. Thus we have the following sets:

- $Phases(l)$ - set of all possible phases for LSC $l$
- $Fireds(l)$ - set of all possible fired-sets for LSC $l$
- $Cuts(l)$ - set of all possible cut-vectors for LSC $l$

**Computing the Phases** Each unwinding step entails the computation of the successor phase(s) from the present one starting with the initial phase and ending with the final phase. The ready set of the initial phase contains all simultaneous classes, which have only instance heads as prerequisites, its history and cut contain all instance heads:

$Phase_0 = (Ready_0, History_0, Cut_0)$ , where
$Ready_0 = \{a \in Sim_Classes(l) | \; prerequisite(a) \in \{\wp^+(\bigcup_{i \in Instances(l)} \{\perp_i\})\}$
$History_0 = \{\bigcup_{i \in Instances(l)} \{\{\perp_i\}\}\}$
$Cut_0 = (\perp_1, .., \perp_n)$, where $n$ is the Number of instances of LSC $l$

A non-initial and non-final phase $Phase_j$ is characterized by

$Phase_j = (Ready_j, History_j, Cut_j)$ , where
$Ready_j = \{a \in Sim_Classes(l)|$
$\quad \forall b \in prerequisite(a) : b \in History_j \;\wedge\; a \notin History_j\}$
$History_j \subseteq Sim_Classes(l)$
$Cut_j = (x_1, .., x_n)$, $\forall k \in Instances(l) : x_k \in Sim_Regions(k)$

The final phase $Phase_{final}$ is characterized by

$Phase_{final} = (Ready_{final}, History_{final}, Cut_{final})$, where
$Ready_{final} = \{\bigcup_{i \in Instances(l)} \{\{\top_i\}\}\}$
$History_{final} = Sim_Classes(l) \setminus \{\bigcup_{i \in Instances(l)} \{\{\top_i\}\}\}$

An unwinding-step from $Phase_i$ to $Phase_j$ is thus defined by the function

$$Step : Phases(l) \times \wp^+(Sim_Classes(l)) \rightarrow Phases(l)$$

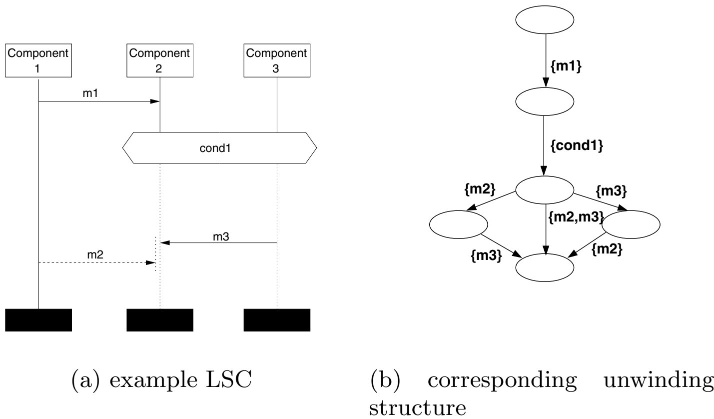
where $Step((Ready_i, History_i, Cut_i), Fired_i) =$
$(Ready_j, History_i \cup Fired_i, upd(Cut_i, Fired_i))$, with

$upd : Cuts(l) \times Fireds(l) \rightarrow Cuts(l)$ and

$upd(Cut_i, Fired_i) := (x_1, .., x_n)$, where

$$x_k = \begin{cases} x'_k \; \exists\, z \in Fired_i \; \exists\, scl \in z : x'_k \in scl \\ x_k \; else \end{cases}, k = 1, .., n$$

For each subsequent unwinding step first a subset of the ready set is selected. This subset represents the simultaneous classes which are unwound in the current step. All possible subsets except the empty set are considered to determine the set of next nodes in the unwinding structure. The source node is connected to all its successor nodes by an edge annotated with the set of simultaneous classes unwound in this step. The new ready set is then computed for all successor nodes in the following manner: First the simultaneous classes which have just been unwound have to be removed from the ready set, then all simultaneous classes whose prerequisites are now fulfilled are added. In the history the just unwound simultaneous classes are recorded, whereas for the Cut the simultaneous classes that have just been unwound are added and the ones just left are removed. The unwinding structure concludes with the final node. Notice that the resulting structure contains one path for each possible ordering — including simultaneity! — of unordered events.




(a) example LSC (b) corresponding unwinding structure

**Fig. 2.** Simple unwinding example

Figure 2 shows a simple unwinding example. We have omitted the node annotations to increase readability. `m2` and `m3` are located in a coregion - they may be observed in any order and in our interpretation also simultaneously[5].

**Optimizing the Structure** The unwinding structure for an LSC can become very broad if the LSC contains large coregions or many elements which may be executed in parallel. As a consequence it may contain identical substructures in different branches. In order to streamline the structure these identical substructures should be merged. But this merge may only be performed if the substructures represent the same unwinding step, i.e. they must have the same history and ready sets.

**Temperatures** Each location is annotated with a temperature indicating if progress is enforced along the instance axis. When the events associated with a location are observed, a hot temperature at the location requires the events located at the following location to occur eventually. A cold temperature allows the events located at the following location not to occur at all, but if they are observed they must be observed after the events of the previous location.
Let $temp(i, x) \rightarrow \{hot, cold\}$ be the temperature of location $x$ at instance $i$. We now define the temperature of a cut by

$$temp(Cut_j) = \begin{cases} hot, \;\; if\ \exists\ i \in Instances(l)\ \exists\ x_i \in Locations(i) : x_i \in Cut_j \\ \qquad \wedge\ temp(i, location(x_i)) = hot \\ cold,\ else \end{cases}$$

The temperature of a phase is defined to be the temperature of its Cut. We can now define the set of cold phases of LSC $l$:

$ColdPhases(l) := \{ph \in Phases(l) \mid ph = (Ready_{ph}, History_{ph}, Cut_{ph}) \wedge temp(Cut_{ph}) = cold\}$. By definition: $temp(Phase_{final}) = cold$.

## 4 From the Unwinding Structure to a Timed Büchi Automaton

The unwinding structure is an intermediary data structure which is incomplete since it cannot express time. This motivates the following transformation of the unwinding structure into a Timed Büchi Automaton (TBA) which serves as intermediate format to the STVE. There exists a translation of the TBA format into propositional temporal logic (for details see [Fey96]).

Before we formally introduce Timed Büchi Automata we describe how timing information is added, since this is the key procedure in transforming the unwinding structure into a TBA.

[5] For delayed communication we would have to put the send and receive events of the messages in separate simultaneous classes.

### 4.1 Adding Timing Information

Hot locations can be annotated by *timing annotations* which are interval notations of the form $[n, m]$, where $n, m$ are non-negative integers and $n \leq m$. The notation is similar to the one used for constraint intervals in STDs [Fey96] or for specifying delays in MSCs [AHP96]. The meaning of a timing annotation is: After having observed the events located at the annotated location at least $n$ and at most $m$ steps later the events located at the following location are observed.

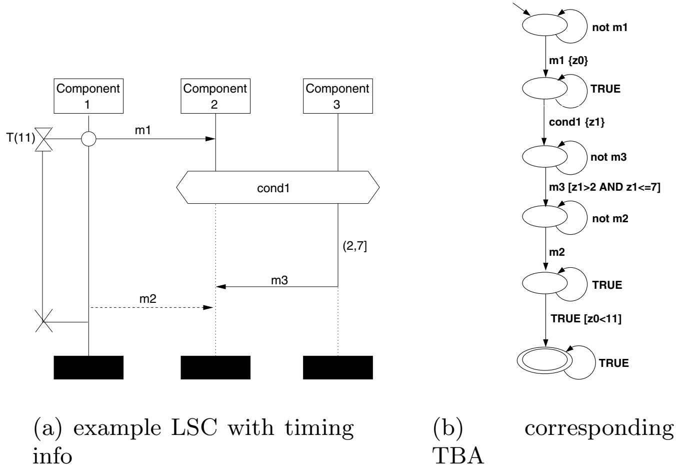
For each location of the LSC an integer clock is introduced. This clock is reset when the location is unwound, i.e. when an event located at this location is observed. A boolean expression constrains the clock value to be in the specified range when the next location along the instance axis is entered (i.e. when an event located at the following location is observed). The boolean expression is simply `true` if the location does not have a timing annotation. For treatment of timing annotations we therefore define:

- For $i \in Instances(l)$ and $x \in Locations(i)$ let $clk(i, x)$ be the unique clock identifier denoting the clock associated with the location $x$ of instance $i$
- The set of clock names is given by
$Clocks(l) := \bigcup_{i \in Instances(l)} \bigcup_{x \in Locations(i)} clk(i, x)$.
- For $i \in Instances(l)$ and $x \in Locations(i)$ let $t_ann(i, x)$ be the timing annotation for location $x$ of instance $i$. Note that $t_ann(i, x) = \epsilon$ if location $x$ of instance $i$ is not annotated with a timing annotation. Otherwise $t_ann(i, x)$ is of the form $[n, m]$, with $n \leq m$, $n, m$ integers.
- Let $clk_resets(Fired_j) := \{\ clk(i, x) \mid \exists\ z \in Fired_j\ \exists\ scl \in z\ \exists\ sr \in scl : location(sr) = x \ \wedge\ x \in Locations(i)\}$, $scl \in Sim_Classes(l)$, $sr \in Sim_Regions(i)$ be the set of names of clocks which are reset when $Fired_j$ is unwound. I.e. for each location reached with $Fired_j$ the corresponding clocks are reset.
- Let $Clk_Resets(l) := \bigcup_{f \in Fireds(l)} clk_resets(f)$ be the set of all clocks which are reset in LSC $l$.
- Let $clk_conds(Fired_j) := \{\ t_ann(i, x) \mid \exists z \in Fired_j\ \exists\ scl \in z\ \exists\ sr \in scl : x \in predecessor(location(sr))\}$, $scl \in Sim_Classes(l)$, $sr \in Sim_Regions(i)$ be the set of timing annotations to be considered when unwinding $Fired_j$.
- Let $Clk_Conds(l) := \bigcup_{f \in Fireds(l)} clk_conds(f)$ be the set of all timing annotations in LSC $l$.

Finally let us note that timers may be treated in a way quite similar to timing annotations. Figure 3 shows an example; note that we are dealing with a TBA instead of an unwinding structure here. The TBA definition as well as the exact treatment of clocks will be demonstrated in section 4.2.

### 4.2 Timed Büchi Automaton Definition

A Timed Büchi Automaton ([Alu98],[AD92]) $A$ is a tuple
$A = (\Sigma, S, s_0, C, \longrightarrow_{TBA}, F)$, where




(a) example LSC with timing info

(b) corresponding TBA

**Fig. 3.** Unwinding timing annotations and timers

- $\Sigma$ is the alphabet
- $S$ is the set of states
- $s_0 \in S$ is the initial state
- $C$ is a set of clocks
- $\longrightarrow_{TBA}$: $S \times Pred \times \wp^+(C) \times \wp(Conds(C)) \rightarrow S$ is the transition function. $Pred$ are predicates ranging over $\Sigma$. $Conds(C)$ are predicates constraining clocks of $C$. A transition $(s, p, c, cond) \longrightarrow_{TBA} s'$ represents the change from state $s$ to state $s'$ for observation $p$. The clocks in $c$ are reset when taking the transition and the transition can only be taken if the clock constraints in *cond* hold.
- Finally $F \subseteq S$ is the set of accepting states

Informally the TBA for an LSC is derived from its unwinding structure by

- renaming the nodes with a fresh set of phase names
- changing the edge annotation to the conjunction of the elements of the corresponding fired set
- adding a dedicated *Exit* state for violated cold conditions
- determining the set of accepting states according to the Büchi acceptance condition
- adding self loops on each state and labeling them with the condition which has to hold while the TBA stays in the associated state[6].

A TBA $A_{LSC}$ for an LSC is a tuple $A_{LSC} = (\Sigma, S, s_0, C, \longrightarrow_{TBA}, F)$, where

- $\Sigma := Sim_Classes(l)$

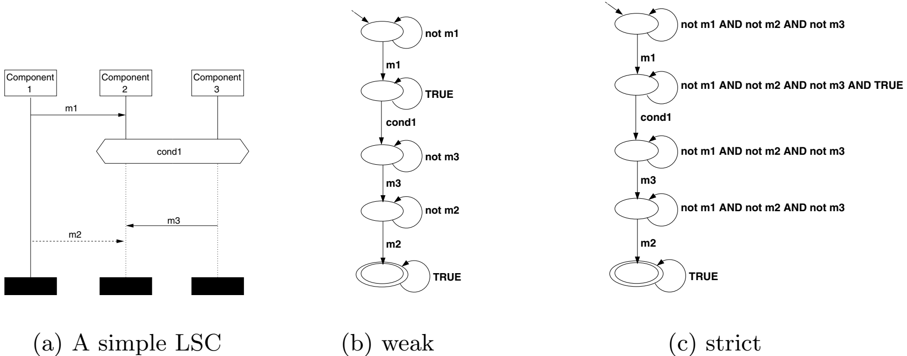
[6] The self loops are needed because time only passes when a transition is taken. Otherwise time could not progress in a state.

- $S := Ph_names(l) \cup \{Exit\}$, where $Ph_names(l)$ is a set of fresh identifiers and *phname* is the function that associates a name from $Ph_names(l)$ with each phase of the unwinding structure, i.e. $\forall\ p \in Phases(l)\ \exists^1 p' \in Ph_names(l) : p' = phname(p)$.
- $s_0 := phname(Phase_0)$
- $C := Clocks(l)$
- $\longrightarrow_{TBA}$: $S \times Pred \times \wp^+(Clk_Resets(l)) \times \wp(Clk_Conds(l)) \rightarrow S$, where the *Pred* is built from $f \in Fireds(l)$ by conjunction (and negation) of the elements of $f$.
- $F := \{\ phname(cph) \mid cph \in ColdPhases(l))\}$

Up to here we did not mention how we handle activation mode and activation condition of a LSC. The language definition [DH99] provides the activation modes *initial* and *invariant*. An initial LSC is activated at system initialization, while an invariant LSC is activated whenever its activation condition evaluates to true; note that the kernel automata are identical in both cases. Activation mode and condition must be regarded when generating the temporal logic formulae from the TBAs[Fey96]. Thus we need to preserve this information in the TBA format and extend the it with an activation predicate.

### 4.3 Determinism in the TBA

Adding the self loops for each state in the TBA raises the question of what annotation should be put on the self loop. This is closely related to the question of determinism of the TBA. There are three options of what the transition annotation should be: First, the annotation could be omitted altogether – this would be equivalent to a *true* annotation – resulting in a very non-deterministic TBA. The *true* annotation does not require the TBA to take a transition when the corresponding message has been observed. This non-deterministic behavior is obviously too weak, so we need a stronger interpretation.




(a) A simple LSC (b) weak (c) strict

**Fig. 4.** Different annotation types for self loops

The interpretation corresponding to [DH99], where each occurrence of a message has to be explicitly noted in the LSC and no other occurrence is allowed, we call *strict*. This is achieved by annotating each self loop with the conjunction of the negation of all messages appearing in the LSC (cf. figure 4(c)). This interpretation may be too strong in certain cases where we do not care if messages visible in the LSC occur anytime else, as long as the ordering imposed by the LSC is satisfied. This leads to the *weak* interpretation where the self loop annotation only contains the negation of the next message(s). This forces the TBA to react to the first occurrence of the message that is expected next, but does not restrain the occurrence of this message at other times (cf. figure 4(b)).

These different degrees of determinism only concern messages. For conditions we always annotate the self loop with *true* (cf. figures 4(c) and 4(b)), because we do not know when to evaluate a condition. This problem is inherent in the LSCs where there is only the possibility to specify if a condition has to be reached – and therefore evaluated – at all. Even if all locations before a condition are hot this does not tell us anything about when exactly the condition is evaluated. The designer would have to use a timing annotation to force a condition to be evaluated at a certain time or within a certain time interval.

## 5 Integration with STVE

The further incorporation of LSCs into the STVE is currently under way. The transformation of LSCs into TBAs described above is only one issue when connecting LSCs to the tool set. We have also developed an editor and a mapping tool for LSCs. Since LSCs should not only be used in the context of Statemate, the LSC identifiers for instances, messages and those used in conditions are only symbolic names, i.e. place holders for concrete identifiers of the model to be verified. Based on the internal representation of Statemate designs in the STVE (cf. [BBea99]) the mapping tool allows the user to associate the activities of the Statemate design with the instances of a LSC. The interfaces of the selected activities are computed and also offered to the user for identification of messages and conditions with certain variables and their valuations. The mapping of LSC objects to design items yields boolean expressions as atomic propositions of the temporal logic formulae.

Figure 5 gives an overview over the LSC tools in the Statemate context. For a given Statemate design LSC requirements are created with the LSC editor. The requirements thus created are then translated by the LSC compiler which implements the unwinding procedure described in this paper into the intermediate TBA format from which the temporal logic formula is generated. Since only symbolic names are used in LSCs, the TL formula only contains propositions which regard these symbolic names. Therefore we need the LSC mapper to relate the Statemate identifiers to the LSC identifiers. The result is a table which gives for each proposition (which consists of symbolic names) the concrete model elements. This table together with the formula generated from the LSC form the input for the modelchecker ($\Phi$ in figure 5). The STVE also translates the Statemate model into the input format for the modelchecker which then determines, if the model satisfies the requirement.

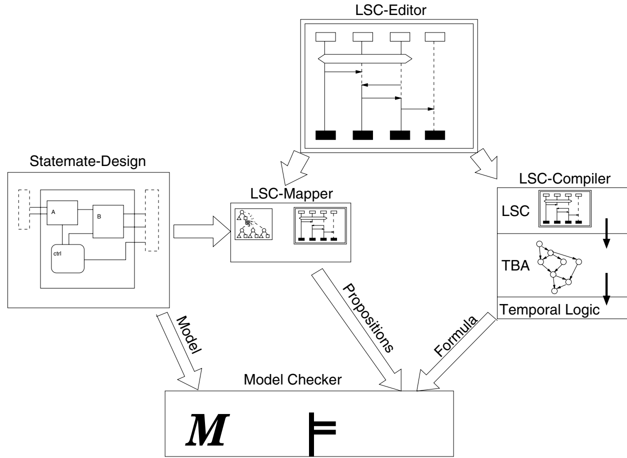



**Fig. 5.** Integration of LSCs with STVE

## 6 Conclusion

We have shown in this paper how the rather high-level semantics for LSCs presented in [DH99] can be elaborated. We only considered a subset of LSC features, which consists of what we feel are the LSC core concepts. We made some further restrictions either for simplicity's sake or due to limitations imposed by STATEMATE (zero-delay messages). We then demonstrated how this subset may be efficiently transformed into an automaton with the focus on the technical procedure of unwinding the LSC. Having arrived at the TBA format the gateway to the STVE is wide open. This format is also used for code generation from STDs in conjunction with a STATEMATE design and to synthesize state charts from STDs. These routes are possible for LSCs as well, although at the moment we have only used LSCs for formal verification as described in section 5.

The verification of STATEMATE models against LSCs has at the moment of writing not been tested extensively, so that more experience is needed in this respect. Especially the issue of complexity needs further investigation. While STDs are used to specify properties of components in a black box view, the benefit of LSCs is the ability to specify protocols in a glass box view of the system. Therefore it is quite natural to consider the whole model at once, while STDs allow the user to scale down the verification task by system verification (cf. [BBea99]). To verify large models against LSC specifications we will need powerful abstraction techniques to reduce the state space for the verification. Complexity also has to be investigated on the requirement side, where the formula may become very large depending on the size of the LSC and the degree of parallelism it contains.

In the future we plan to extend both LSCs and the verification tool set to also cope with UML models. In this respect not only the concept of synchronous and asynchronous communication has to be reflected in LSCs, but we will also need to develop strategies for verification in the object-oriented world.

**Acknowledgments** We would like to thank Werner Damm and David Harel for their comments on the first version of this paper and the following discussions. We also thank Uwe Higgen, Rainer Koopmann and Olaf Bär, who played a major role in implementing the tools described here.

## References


[AD92] R. Alur and D. Dill. The Theory of Timed Automata. In de Bakker, Henzinger, and de Roever, editors, *Proceedings of Rex 1991: Real Time in Theory and Practice*, number 600 in LNCS. Springer Verlag, 1992.

[AHP96] R. Alur, G. J. Holzmann, and D. Peled. An analyzer for Message Sequence Charts. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 35–48, Passau, Germany, 1996. Springer-Verlag.

[Alu98] R. Alur. Timed Automata. In *NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems*, 1998.

[BBea99] Tom Bienmüller, Udo Brockmeyer, and Werner Damm et. al. Formal Verification of an Avionics Application using Abstraction and Symbolic Model Checking. In Felix Redmill and Tom Anderson, editors, *Towards System Safety – Proceedings of the Seventh Safety-critical Systems Symposium, Huntingdon, UK*, pages 150–173. Safety-Critical Systems Club, Springer Verlag, 1999.

[DDK99] W. Damm, G. Döhmen, and J. Klose. Secure Decentralized Control of Railway Crossings. In S. Gnesi and D. Latella, editors, *Fourth International ERCIM Workshop on Formal Methods in Industrial Critical Systems*, 1999.

[DH99] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.

[DH01] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 2001. to appear.

[DK01] Werner Damm and Jochen Klose. Verification of a Radio-based Signaling System Using the Statemate Verification Environment. *Formal Methods in System Design*, 2001. to appear.

[Fey96] Konrad Feyerabend. Realtime Symbolic Timing Diagrams. Technical report, Carl von Ossietzky Universität Oldenburg, 1996.

[FJ97] Konrad Feyerabend and Bernhard Josko. A visual formalism for real time requirement specifications. In *Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrentand Distributed Software, ARTS'97, Lecture Notes in Computer Science 1231*, pages 156–168, 1997.

[HP96] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.* Part No. D–1100–43. i-Logix Inc., Three Riverside Drive, Andover, MA 01810, June 1996.

[IT96a] ITU-T. *ITU-T Annex B to Recommendation Z.120 Formal Semantics of Message Sequence Charts.* ITU-T, Geneva, 1996.

[IT96b] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC).* ITU-T, Geneva, October 1996.

[Krü99] Ingolf Krüger. Towards the Methodical Usage of Message Sequence Charts. In Katharina Spies and Bernhard Schätz, editors, *Formale Beschreibungstechniken für verteilte Systeme (FBT99)*, 9. GI/ITG Fachgespräch, pages 123–134. Herbert Utz Verlag, June 1999.