

Live Sequence Charts ^{*}

an introduction to lines, arrows and strange boxes in the context of formal verification.

Matthias Brill¹, Werner Damm¹, Jochen Klose², Bernd Westphal¹, and
Hartmut Wittke³

¹ Carl von Ossietzky Universität Oldenburg, Department für Informatik,
PO Box 2503, 26111 Oldenburg, Germany

{brill,damm,westphal}@informatik.uni-oldenburg.de

² Bombardier Transportation, Braunschweig,
Wolfenbüttler Straße 86/Obergstraße 5, 38102 Braunschweig, Germany
jochen.klose@de.transport.bombardier.com

³ OFFIS, Escherweg 2, 26121 Oldenburg, Germany
wittke@offis.de

Abstract. The language of Message Sequence Charts (MSC) is a well-established visual formalism which is typically used to capture scenarios in the early stages of system development. But when it comes to rigorous requirements capturing, in particular in the context of formal verification, serious deficiencies emerge: MSCs do not provide means to distinguish mandatory and possible behavior, for example to demand that a communication is required to finally occur.

The Live Sequence Chart (LSC) language introduces the distinction between mandatory and possible on the level of the whole chart and for the elements messages, locations, and conditions. Furthermore they provide means to specify the desired activation time by an activation condition or by a whole communication sequence, called pre-chart.

We present the current stage of LSC language and a sketch of its formal semantics in terms of Timed Büchi Automata.

1 Introduction

Message sequence charts (MSCs) are a well established visual formalism for the description of inter-working of processes or objects. There is also a standard for the MSC language, which has appeared as a recommendation of the ITU [1]. The standard defines the allowed syntactic constructs, and is also accompanied by a formal semantics [2] that provides unambiguous meaning to basic MSCs. Despite the widespread use of MSCs and the foundational efforts cited above, several fundamental issues have been left unaddressed. One of the most basic of these is, quoting [3]: “*What does an MSC specification mean: does it describe all*

^{*} This research was supported by the German Research Council (DFG) within the priority program Integration of Specification Techniques with Engineering Applications under grant DA 206/7.

behaviors of a system, or does it describe a set of sample behaviors of a system?”.

In this paper we will provide an introduction to the Live Sequence Chart language, introduced by Damm and Harel in [4], as a conservative extension to the standard MSC language. In the first section, we will motivate the need for another visual formalism and describe the basic features of the LSC language informally. In the second section we will sketch the formal semantics of LSCs in terms of timed automata, and furthermore provide an idea of the relation between LSCs and a compatible reference system. The underlying complete syntax and rigorous semantics of the LSC language can be found in [5].

1.1 Shortcomings of Message Sequence Charts

Typically MSCs are used to capture sample scenarios corresponding to use-cases. While the system model becomes refined and conditions characterizing use-cases evolve, the intended interpretation often undergoes a metamorphosis from an *existential* to a *universal* view: earlier one wants to say that a condition can become true and that when true the scenario can happen, but later on one wants to say that if the condition characterizing the use-case indeed becomes true the system must adhere to the scenario described in the chart.

While the distinction between *mandatory* and *possible* behavior is one of the most urging deficiencies which needs to be addressed in order to construct semantically meaningful computerized tools for describing and analyzing use-cases and scenarios, the formal semantics for MSCs as described in [2] leave a good deal of questions beyond unanswered:

Existential or universal view. An MSC shows only one sample run of the system, one scenario, i.e. it is not possible to specify a mandatory protocol between the communicating entities.

Safety and Liveness properties. The MSC semantics offers no distinction, whether progress is enforced or not, i.e. The semantics in [2] only define permitted sequences of events; the occurrence of an event can not be enforced. Using the terms coined by Lamport in [6] MSCs can only express *safety* (nothing bad ever happens), but not *liveness* properties (something good will happen eventually).

Semantics of Conditions. Conditions in MSCs have no formal semantics. In the words of [2]: “*The semantics of a chart containing conditions is simply the semantics of the chart with the conditions deleted from it.*”. This is obviously not the way to treat conditions from a more formal point of view.

Simultaneous events. MSCs do not allow more than one event to happen exactly at the same time, i.e. there is no notation of simultaneity.

Activation time. An MSC does not state explicitly when the behavior it describes should be observed, i.e. there is no indication of when the MSC should be activated during a system run.

Time treatment. The treatment of time is only rudimentary, since quantitative timing is not covered by the semantics, i.e. timer durations are ignored. Only the correct sequence of timer events, respectively intervals is enforced.

The LSC language as described in [5] is an extension to the MSC language which addresses these shortcomings and provides a fully worked out formal semantics. LSCs can therefore accomplish the requirements for the application to more advanced use cases like formal verification as discussed in the companion paper [7].

1.2 Basic LSC Features

This section presents the key elements of the Live Sequence Chart language. The basic idea of LSCs is to allow a distinction between *mandatory* and *possible* behavior, i.e. most LSC elements can be designated to belong to either one category or the other. This distinction is also expressed graphically, which contributes largely to the easy understanding of LSC specifications. Mandatory elements are depicted by solid lines, possible ones by dashed lines.

Instances and Messages. Instances and messages are the elementary building blocks of LSCs. The graphical representation for instances has been adopted from MSCs, i.e. LSC instances consist of an instance head carrying the instance name, an instance axis and an instance end, as the example LSC in figure 1 shows.

As for MSCs, the horizontal dimension is the structural dimension and the vertical dimension corresponds to the time dimension. Deviating from MSCs we depict the environment by an instance of its own rather than the border of the LSC. When using LSCs for formal verification, an explicit environment instance offers the possibility of expressing assumptions on the behavior of the environment within the LSC by employing the same elements as for the other instances.

Concerning messages we consider two kinds: asynchronous and instantaneous ones. Asynchronous messages are visualized by half stick arrows, instantaneous messages by arrows with solid heads, as shown in figure 1. Instantaneous messages have to be drawn horizontally to indicate simultaneity of sending and receiving, while asynchronous ones are drawn slanted to indicate the passage of time between sending and receipt.

Operation calls, are represented by two instantaneous messages: the method call and the return of the method call. For method calls in LSCs we require the return message to be stated explicitly, because otherwise confusion arises whether messages and other elements following a method call receipt are part of the method body. The pairing of operation call and return is indicated graphically by widening the instance axis on the receiver side into a thin rectangle

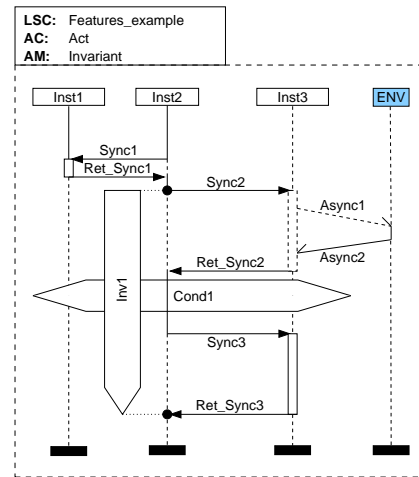


Fig. 1. Kernel LSC features example.

which marks the operation body; see `Sync2` and `Ret_Sync2` in figure 1 for example.

Liveness and Temperatures. One deficiency of MSCs is their inability to enforce progress, as mentioned in section 1.1. LSCs overcome this drawback by associating a *temperature* with both locations⁴ and messages. The temperature can be either *hot* or *cold*, the former indicating that progress is enforced. The analogy here is that one cannot remain at a hot location for an infinite amount of time, because then one would burn ones feet. This obviously requires that a hot location has to be left, i.e. the following location has to be reached. At a cold location one can stay forever without harming ones feet, i.e. the following location need not be reached. In terms of messages this means that a hot message has to be delivered, whereas a cold message may be lost along the way. Progress information is thus expressed by the temperatures of the messages and along the instance lines.

Graphically hot temperatures are represented by solid lines, cold ones by dashed lines. This means that e.g. a hot location is depicted by a solid instance axis segment, which starts at this location and ends either at the next cold location or the instance end. In figure 1 for example the location of the sending of message `Sync2` is cold and the next location (the receipt of the return message) is hot, so that the instance axis segment in between them is dashed.

Conditions and Local Invariants. In order to make statements about the state of the system boolean conditions referring to attributes or data items of the involved entities are used. Graphically, conditions are represented as in MSCs by an elongated hexagon (cf. figure 1). Conditions also come in two variants: *mandatory* and *possible*. A mandatory condition must be satisfied, i.e. the boolean expression associated with it has to hold; violation of the condition is considered an error. Possible conditions do not generate an error when they are not satisfied, but merely constitute an exit from the enclosing LSC. Mandatory conditions are denoted by solid lines (e.g. `Cond1` in figure 1) and possible ones by dashed lines.

Conditions constrain attributes or data items of entities at one point in time, but often it is desired to express validity of a condition over a period of time. This observation motivates the introduction of a fitting feature: *local invariants*, which come in two flavors: possible and mandatory, with the same interpretation as for conditions. Since local invariants cover a period of time, they need reference points for start and end and should thus always be bound to observable events. Graphically, local invariants are depicted by a condition symbol, which is rotated by 90°. A pointed end of the invariant symbol indicates the exclusion of the corresponding reference point while a planar end denotes the inclusion (e.g. in figure 1 the local invariant `Inv1` includes message `Sync2` but excludes message `Ret_Sync3`).

⁴ Locations are those points on an instance axis, where some event is attached, e.g. sending or receipt of a message, conditions, etc. In section 2.2 we give a formal definition of a location.

Simultaneous Regions and Coregions. We now have assembled all basic elements of our kernel LSC language. The default ordering of these basic elements is one after the other from top to bottom along the instance axis. Ordering between instances is induced only by messages and conditions ranging over more than one instance. *Simultaneous regions* allow to group several elements, which should be observed at the same time. This is essential for determining reference points for conditions, local invariants and timers. Graphically they are represented by enlarging the location in question into a small filled circle; see figure 1 for examples.

A coregion is used to indicate that no ordering is imposed on the events it contains, i.e. they may occur in any order. This corresponds to the classical MSC view of a coregion with the exception that – as a consequence of the simultaneous region construct – we also allow events in a coregion to take place simultaneously.

Coregions are represented graphically by a dotted line running in parallel to the instance axis. This differs from the representation in MSCs, where they are depicted as dashed portions of the instance axis.

Activation, Quantification and Pre-charts. In the preceding paragraphs the graphical elements describing the communication behavior of several interacting entities are presented. This paragraph adds information about when this behavior should be observed and whether it specifies a sample behavior or a protocol to be obeyed, therefore addressing the primary criticism outlined before.

The quantification information represents the distinction between mandatory and possible behavior on the chart level. The sample-run or scenario view of MSCs, i.e. the interpretation that there *exists* a run, which fulfills the LSC, is covered by the possible mode, which we call *existential*. The mandatory mode, which is missing in MSCs as laid out in section 1.1, expresses that the behavior specified in the LSC must be fulfilled by *all* runs, for which reason it is called the *universal* view. Graphically, the quantification information is depicted by the border style of the LSC: a solid border indicates a universal chart, a dashed border an existential one.

For universal LSCs it is vital to be able to characterize the activation point. If every run has to fulfill the universal LSC, it must be possible to state at which point(s) of the run the LSC should be considered, otherwise the behavior of the entire system has to be specified in one LSC, which is clearly undesirable. The activation point of an LSC is characterized by two complementary concepts: *activation condition* and *activation mode*. The activation condition is a boolean condition, which expresses the activation point for a chart. The activation mode specifies, how often an LSC should be activated; the offered modes are *initial*, *invariant* and *iterative*. The initial mode indicates that the LSC is activated at system start only, i.e. it is intended to describe a start-up or initialization sequence. The other two modes indicate that the LSC is activated whenever the activation condition holds. The difference between an invariant and an iterative LSC is that the invariant mode allows a reactivation of the LSC while another

incarnation of the same chart is still active, while the iterative mode allows only one incarnation of the chart at a time.

It turns out that activation condition and mode do not always sufficiently characterize the activation point of a property specification. Often it is necessary to know more about the history of a run, before being able to decide, whether the LSC should be activated. There may be e.g. more than one way for a run to arrive at a certain system state (characterized by a condition), but the LSC should only be activated, if the run has followed a specific “route”. We are for instance only interested in activating an LSC, when no errors have occurred so far. This motivates the introduction of *pre-charts* which allow to specify a prefix of a run acting as a trigger for the actual LSC. Pre-charts allow to specify a prefix or history, which must be fulfilled by a run in order to activate the LSC. A pre-chart is essentially an LSC, i.e. all language constructs can be used in a pre-chart, but its semantics is different, since the message sequence of the pre-chart is not required to hold in the system, but rather must be observed before activating the actual LSC. Pre-charts do not replace the activation condition, but extend it; the activation condition in the presence of a pre-chart indicates the starting point of the prefix.

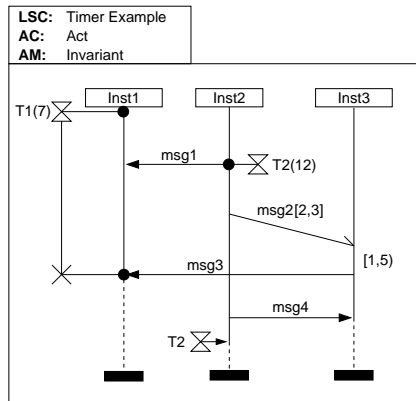


Fig. 2. Examples of LSC timing constraints.

represented by an hour glass symbol, which is connected to the instance axis by an arrow; a timer reset is represented by a \times -symbol, which is connected to the instance axis by a simple line; see figure 2 for examples.

Timing intervals express quantitative local liveness properties, since they refer to neighboring atoms⁵. They are used to give both a minimum and a maximum delay between two directly consecutive atoms. The delimiting atoms can either be located on the same instance axis, or be the sending and receipt of an asynchronous message. The intervals are placed next to the instance axis

The informal semantics of an LSC with pre-chart is consequently: If the activation condition holds *and* afterwards the pre-chart is *completed*, then the LSC is activated.

Time. LSCs allow the specification of time constraints either in form of an MSC-style timer or in interval notation, with a lower and an upper bound. The graphical representation of timers is identical to the one given in MSC-96, i.e. the setting of a timer is represented by an hour glass symbol, which is annotated by a name and a duration and is connected to the instance axis by a simple line; a timeout symbol is

⁵ Atoms are the most basic building blocks of an LSC, e.g. instance heads, instance ends, sending a message or receiving a message. In section 2.2 we give a formal definition of an atom.

between the two locations which delimit them or are attached to the identifier of the constrained asynchronous message (cf. `msg2` in figure 2).

2 Semantics of the LSC Language

The semantics for the behavior of an LSC is defined in terms of a symbolic timed Büchi automaton in [5]. The procedure of deriving an automaton from an LSC, which is called *unwinding*, is inspired by the semantics definition for Symbolic Timing Diagrams presented in [8], which has been taken as a blue print for the first version of the LSC semantics in [4].

Before describing the the unwinding procedure we introduce timed Büchi automata and define the formal syntax of LSC elements, which is the starting point for the translation from an LSC into a timed symbolic automaton.

2.1 Automata-Theoretic Foundation

The base for the definition of the formal semantics of LSCs is a variation of timed Büchi automata. We chose Büchi automata for several reasons: First, we use LSCs to describe the communication behavior of reactive systems, i.e. systems which must be able to accept and react to input signals at any time. These systems are typically designed to operate forever, at least theoretically, which means that runs of reactive systems are infinite. Therefore classical finite automata are not sufficient for our purpose. Büchi automata ([9]) are one possible solution as they accept infinite words. The second reason for choosing Büchi automata over other variants of automata on infinite words is that there is a close relationship between Büchi automata and linear time temporal logic (LTL) [9, 10]. Since the main application field for LSCs is property specification for formal verification, this is a major advantage. The third reason is that Büchi automata allow to express liveness properties easily via their acceptance criterion.

Timed Büchi Automata. The acceptance criterion for Büchi automata (and other automata on infinite words) needs to take the infiniteness of the words into account. Informally an infinite word is accepted by a Büchi automaton if its run passes infinitely often through (at least) one of the accepting states of the automaton; these states are also called *fair* states.

When the occurrence times of the letters of words are important, timed languages are used and consequently a corresponding type of automaton is needed: a timed Büchi automaton. For the definition of timed Büchi automata we loosely follow the lead of [11] here, inasmuch as we associate an occurrence time to each symbol of a word, yielding timed words. In contrast to [11], however, we only consider discrete time instead of dense time. Time is represented by a sequence of time values from the set of non-negative natural numbers: $\tau_i \in \mathbb{N}$, which has to satisfy two constraints: time is non-decreasing and time always progresses:

Definition 1 (Timed Word). *A time sequence $\tau = \tau_0\tau_1\tau_2\dots$ is an infinite sequence of time values $\tau_i \in \mathbb{N}$ for which the following holds:*

1. $\tau_i \leq \tau_{i+1}, \forall i \geq 0$ (τ is monotonically increasing)
2. $\forall t \in \mathbb{N} \exists i \geq 1 : \tau_i > t$ (time always progresses)

A *timed word* over an alphabet Σ is then a pair (σ, τ) , where $\sigma = \sigma_0 \sigma_1 \dots$ is an infinite word and $\tau = \tau_0 \tau_1 \dots$ is a time sequence. The time value τ_i denotes the occurrence time of input symbol σ_i .

In the untimed case the behavior of the automaton only depends on the input symbols, i.e. being in some state the next states of an automaton are determined by the current input symbol. In order for an automaton to also accept timed words it needs a means to count time, since the choice of the next states also depends on the occurrence time of the input symbols in question. This is realized by *clocks*, which can be set to zero on any transition of the automaton and count the time since their last reset. This allows for the introduction of clock constraints on transitions – i.e. a transition may only be taken, if all its clock constraints are satisfied – forcing the input word to obey certain timing requirements. Thus time is introduced into an automaton by adding a (finite) set of clocks and augmenting transitions by clock resets and clock constraints formulated over this set of clocks.

In the following the definition of a timed Büchi automaton, the set of clocks used in the clock constraints of the automaton is added and the transitions between states are augmented by clock resets and constraints. The acceptance criterion is adjusted accordingly, so that fair states are defined in terms of timed words.

Definition 2 (Timed Büchi Automaton). A *timed Büchi automaton TBA* is a tuple

$$\mathcal{TBA} := (\Sigma, Q, q_0, C, \longrightarrow, F), \text{ where}$$

- Σ is a finite alphabet of input symbols,
- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- C is a finite set of clocks
- $\longrightarrow \subseteq Q \times \Sigma \times \mathcal{P}(C) \times \Phi(C) \times Q$ is the transition relation. A transition $(q, \sigma, \rho, \gamma, q') \in \longrightarrow$ represents the change from state q to state q' on input symbol σ . The set $\rho \in \mathcal{P}(C)$ indicates which clocks are reset when taking the transition, the set $\Phi(C)$ is the set of clock constraints over C and $\gamma \in \Phi(C)$ is a clock constraint over C , which has to be fulfilled.
- $F \subseteq Q$ is the set of fair (accepting) states.

Let $(\sigma, \tau) = (\sigma_0, \tau_0)(\sigma_1, \tau_1) \dots$ be a timed word over Σ . A *timed run* tr of a timed Büchi automaton \mathcal{TBA} over timed word (σ, τ) is an infinite sequence of configurations q_i, ν_i , where $q_i \in Q$ is the i -th state of the automaton along the run and $\nu_i \in I$ is the clock interpretation in this state:

$$tr : (q_0, \nu_0) \xrightarrow[\tau_0]{\sigma_0} (q_1, \nu_1) \xrightarrow[\tau_1]{\sigma_1} (q_2, \nu_2) \xrightarrow[\tau_2]{\sigma_2} \dots, \text{ with}$$

- $\forall x \in C : \nu_0(x) = 0$, (initially all clocks are zero)
- $\forall i \geq 0 \exists (q_i, \sigma_i, \rho_i, \gamma_i, q_{i+1}) \in \text{Trans} : \llbracket \gamma_i \rrbracket(\nu_i) = \text{true} \wedge \forall x \in \rho_i : \nu_{i+1}(x) = 0 \wedge \forall x \notin \rho_i : \nu_{i+1}(x) = \nu_i(x) + \tau_i - \tau_{i-1}$
(the target state of each transition is the source state of the following transition, the transition respects all its clock constraints, resets all appropriate clocks and all clocks, which are not reset, correctly advance the time).

Let $\text{inf}(tr) \subseteq Q$ denote the set of states of \mathcal{TBA} which are visited infinitely often by timed run tr , i.e. $\text{inf}(tr)$ consists of those states $q \in Q$ such that $q = q_i$ for infinitely many $i \geq 0$. The language accepted by \mathcal{TBA} is defined as the set of timed words, for which there is an accepting run of \mathcal{TBA} :

$$\mathcal{L}(\mathcal{TBA}) := \{(\sigma, \tau) \mid \exists tr = (q_0, \nu_0) \xrightarrow[\tau_0]{\sigma_0} (q_1, \nu_1) \xrightarrow[\tau_1]{\sigma_1} \dots : \text{inf}(tr) \cap F \neq \emptyset\}$$

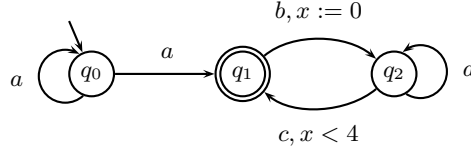


Fig. 3. Example timed Büchi automaton.

Example 1. Figure 3 shows an example of a timed Büchi automaton, which accepts the language $\{(\sigma, \tau) \mid \sigma \in a^+(ba^*c)^\omega \wedge \forall i \exists j > i : \sigma_i = b \Rightarrow \sigma_j = c \wedge \tau_j < \tau_i + 4\}$.

Symbolic Timed Automata. The timed Büchi automata described so far operate on single input symbols only, since they allow but one element of Σ per transition. In order to be able to describe the communication behavior of a system, it is necessary to allow more than one observation at a time.

Symbolic Timing Diagrams (STDs) faced a similar problem, which in [8] is solved by the extension of Büchi automata to *symbolic automata*, where a run is extended to a *computation sequence* referring to valuations of system variables and formulas are allowed as transition annotations in the automaton. We adopt this strategy for the LSC semantics definition and therefore use symbolic automata.

In [8] Schlör does not consider quantitative timing for STDs, so that symbolic automata are untimed. We consequently extend the definition of symbolic automata to also encompass (discrete) time, similar to timed Büchi automata. First, we extend the notion of a timed word to a timed symbolic word (called a computation sequence in [8]).

Definition 3 (Timed Symbolic Word).

Let AP be the set of atomic propositions. A symbolic word is an infinite sequence $\theta = \theta_0\theta_1\theta_2\dots$, where $\theta_i : AP \rightarrow \mathbb{B}$ is a valuation, which assigns to each $v \in AP$ a boolean value. Let τ be a time sequence. A timed symbolic word then is a pair $(\theta, \tau) = (\theta_0, \tau_0)(\theta_1, \tau_1)(\theta_2, \tau_2)\dots$, where τ_i denotes the occurrence time of valuation θ_i .

In order to refer to the occurrence of several communication events it is necessary to extend the timed Büchi automata to allow formulas:

Definition 4 (Formula over AP). Let AP be a set of atomic propositions. A formula ψ over AP is a boolean expression produced by the following rules:

$$psi := \sigma \mid \psi \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2, \text{ with } \sigma \in AP.$$

The set of all formulas over AP is denoted by $BExpr_{AP}$.

We can now define in extension of definition 2 on page 8 a timed symbolic automaton:

Definition 5 ((Non-deterministic) Timed Symbolic Automaton).

A timed symbolic automaton TSA is a tuple

$$TSA := (AP, Q, q_0, C, \longrightarrow, F), \text{ where}$$

- AP is a finite alphabet of input symbols (atomic propositions),
- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- C is a finite set of clocks
- $\longrightarrow \subseteq Q \times BExpr_{AP} \times \mathcal{P}(C) \times \Phi(C) \times Q$ is the transition relation. A transition $(q, \psi, \rho, \gamma, q') \in \longrightarrow$ represents the change from state q to state q' while satisfying formula ψ . The set $\rho \in \mathcal{P}(C)$ indicates which clocks are reset when taking the transition, the set $\Phi(C)$ is the set of clock constraints over C and $\gamma \in \Phi(C)$ is a clock constraint over C , which has to be fulfilled.
- $F \subseteq Q$ is the set of fair states.

A timed run tsr of a timed symbolic automaton TSA over a timed symbolic word (θ, τ) is an infinite sequence of configurations (q_i, ν_i) , where $q_i \in Q$ is the i -th state of the automaton along the run, $\nu_i \in I$ is the clock interpretation in this state:

$$tsr : (q_0, \nu_0) \xrightarrow[\tau_0]{\theta_0} (q_1, \nu_1) \xrightarrow[\tau_1]{\theta_1} (q_2, \nu_2) \xrightarrow[\tau_2]{\theta_2} \dots, \text{ with}$$

- $\forall x \in C : \nu_0(x) = 0$, (initially all clocks are zero)
- $\forall i \geq 0 \exists (q_i, \psi_i, \rho_i, \gamma_i, q_{i+1}) \in \longrightarrow : \theta_i \models \psi_i \wedge \llbracket \gamma_i \rrbracket(\nu_i) = true \wedge \forall x \in \rho_i : \nu_{i+1}(x) = 0 \wedge \forall x \notin \rho_i : \nu_{i+1}(x) = \nu_i(x) + \tau_i - \tau_{i-1}$
(the target state of each transition is the source state of the following transition, the boolean expression annotating the transition is evaluated to true, the transition respects all its clock constraints, resets all appropriate clocks and all clocks, which are not reset, correctly advance the time).

Let $\text{inf}(tsr) \subseteq Q$ denote the set of states of TSA which are visited infinitely often by timed run tsr , i.e. $\text{inf}(tsr)$ consists of those states $q \in Q$ such that $q = q_i$ for infinitely many $i \geq 0$. The language accepted by TSA is defined as:

$$\mathcal{L}(TSA) := \{(\theta, \tau) \mid \exists tsr = q_0 \xrightarrow[\tau_0]{\theta_0} q_1 \xrightarrow[\tau_1]{\theta_1} q_2 \xrightarrow[\tau_2]{\theta_2} \dots : \text{inf}(tsr) \cap F \neq \emptyset\}$$

2.2 LSC Syntax

In order to introduce the unwinding procedure we define the formal syntax of LSC elements, which is the starting point for the translation from an LSC into a timed symbolic automaton.

An LSC consists of several components: the body of the LSC – i.e. the instances and events defined on it —, the activation condition and mode, the quantification and the pre-chart. While the complete LSC Syntax is defined in [5], the focus in the remainder of this section is on the body of the LSC.

Definition 6 (LSC). *An LSC is a tuple*

$$L = (l, ac, pch, amode, quant)$$

with ‘ l ’ the body of the LSC, ‘ ac ’ the activation condition, ‘ pch ’ the pre-chart, $amode \in \{\text{initial, invariant, iterative}\}$ the activation mode and $quant \in \{\text{existential, universal}\}$ the quantification.

An LSC body consists of a number of instances, which are collected in the set $\text{Inst}(l)$. In the following let l denote the body of an LSC L , and let $i \in \text{Inst}(l)$ denote some instance of l . The basic blocks (*atoms*) of which an LSC body is comprised are the following: *instance heads*, *instance ends*, *sending a message*, *receiving a message*, *condition atom* (local to one instance), *start of a local invariant* and *end of a local invariant*.

The atoms carry the progress information of each instance (cf. section 2.3) and are organized according to their positioning on the instance axes. The atoms are thus instance-wise collected in sets:

$\text{Msgsnd}(i), \text{Msgrcv}(i)$: sets of message send/receive atoms

$\text{Conds}(i)$: set of condition atoms⁶

$\text{LI_starts}(i), \text{LI_ends}(i)$: sets of start/end atoms of local invariants

There is only one instance head atom, denoted by \perp_i , and one instance end atom, denoted by \top_i , for each instance i , so that it is not necessary to have sets for these atoms on the instance level. Collecting all atoms of the LSC body

according to type yields the following sets:

$$\begin{aligned} Instheads(l) &:= \cup_{i \in Inst(l)} \{\perp_i\}, & Instends(l) &:= \cup_{i \in Inst(l)} \{\top_i\}, \\ Msgsnd(l) &:= \cup_{i \in Inst(l)} Msgsnd(i), & Msgrcv(l) &:= \cup_{i \in Inst(l)} Msgrcv(i), \\ Conds(l) &:= \cup_{i \in Inst(l)} Conds(i), \\ LI_starts(l) &:= \cup_{i \in Inst(l)} LI_starts(i), & LI_ends(l) &:= \cup_{i \in Inst(l)} LI_ends(i) \end{aligned}$$

Collecting all atoms of an instance, resp. of the entire body yields the sets:

$$\begin{aligned} Atoms(i) &:= \{\perp_i\} \cup Msgsnd(i) \cup Msgrcv(i) \cup Conds(i) \\ &\quad \cup LI_starts(i) \cup LI_ends(i) \cup \{\top_i\} \\ Atoms(l) &:= Instheads(l) \cup Msgsnd(l) \cup Msgrcv(l) \cup Conds(l) \\ &\quad \cup LI_starts(l) \cup LI_ends(l) \cup Instends(l) \end{aligned}$$

The atoms of each instance are ordered from top to bottom as drawn in the LSC⁷ and thus have a graphical position, given by the function $position(a)$ for $a \in Atoms(i)$.

Definition 7 (Atom Position). *Let $a, a', a'' \in Atoms(i)$ and $a \prec_i a'$ denote the order of a and a' as drawn on instance i . The function $position(\cdot)$ assigns a natural number to each atom $a \in Atoms(l)$ of LSC body l . This number is called position of a .*

$$\begin{aligned} position &: Atoms(l) \longrightarrow \mathbb{N}_0 \\ position(a) &:= \begin{cases} 0 & \text{if } \forall a' \neq a : a' \not\prec_i a \\ 1 + position(a') & \text{if } a' \prec_i a \wedge \neg \exists a'' \in Atoms(i) : \\ & a' \prec_i a'' \prec_i a \end{cases} \end{aligned}$$

Remark 1 (Atom Positions). The definition of the position of an atom does not assign a unique number to each atom, but allows several atoms of one instance to share one position. This is necessary in order to correctly describe simultaneous regions (see below), which are characterized by the fact that more than one atom occupies the same position.

Atoms $a \in Atoms(i)$ are grouped into *clusters*, which are characterized by the fact that all atoms contained in them are observed simultaneously and therefore have the same position. This concept is needed to express simultaneous regions.

Definition 8 (Cluster). *The set of clusters of an instance $i \in Inst(l)$ is defined by the maximal set*

$$Clusters(i) := \{cl \subseteq Atoms(i) \mid \forall a, a' \in cl : position(a) = position(a')\}.$$

⁶ Note that a condition atom is local to one instance. A condition shared by several instances consists of one condition atom on each participating instance.

⁷ This total order can be disrupted by a coregion as discussed later.

The set of clusters of an LSC body l is defined by

$$\text{Clusters}(l) := \bigcup_{i \in \text{Inst}(l)} \text{Clusters}(i).$$

The position function is extended to cover clusters as well:

$$\text{position}(cl) := \text{position}(a), a \in cl, cl \in \text{Clusters}(l).$$

Remark 2 (Clusters). Note that each atom is contained in exactly one cluster, but that each cluster may contain several atoms. The latter case only arises, if the cluster corresponds to a simultaneous region; atoms outside of a simultaneous region result in a singleton cluster.

The clusters on each instance as defined so far are totally ordered; as introduced in chapter 1.2, coregions allow to suspend this total order. A coregion cr is thus a set of unordered clusters and contains all those clusters, which are covered by the dotted line next to the instance axis. Each coregion consists of a set of unordered clusters: $cr := \{cl_1, \dots, cl_n\} \subseteq \text{Clusters}(i)$. The set of all coregions of instance i is given by $\text{Coregions}(i)$ and the set of all coregions of an LSC body l by $\text{Coregions}(l)$. For each cluster $cl \in \text{Cluster}(l)$ we define the function $\text{coreg}(cl)$, which returns the coregion cl is part of:

$$\text{coreg}(cl) := \begin{cases} \emptyset & , \text{if } \neg \exists cr \in \text{Coregions}(l) : cl \in cr \\ cr & , \text{else} \end{cases}$$

The position does not reflect the relaxed ordering requirement imposed by a coregion: the sending of `msg3` and the receipt of `msg4` on `Inst1` in figure 4 on the next page are still ordered according to their positions. In order to correctly capture the coregion semantics a *logical* position, called *location* is needed:

Definition 9 (Location). The location of a cluster $cl \in \text{Clusters}(i)$ is given by the function

$$\text{location}(cl) := \begin{cases} \text{position}(cl), & \text{if } \text{coreg}(cl) = \emptyset \\ \min(\{\text{position}(cl_i) \mid cl_i \in \text{coreg}(cl)\}), & \text{if } \text{coreg}(cl) \neq \emptyset \end{cases}$$

$\text{Locations}(i) := \{\text{location}(cl) \mid cl \in \text{Clusters}(i)\}$ is the set of locations of instance i and $\text{Locations}(l) := \bigcup_{i \in \text{Inst}(l)} \text{Locations}(i)$ the set of all locations of LSC body l .

Remark 3 (Locations). The locations are unique only on one instance, since $\text{location}(\cdot)$ relies on $\text{position}(\cdot)$. The location of a cluster which is not part of a coregion is equal to its position and unique on its instance. All clusters in a coregion share the same location but have different positions. While Clusters on instances without coregions are totally ordered, Clusters on instances with coregions are only partially ordered.

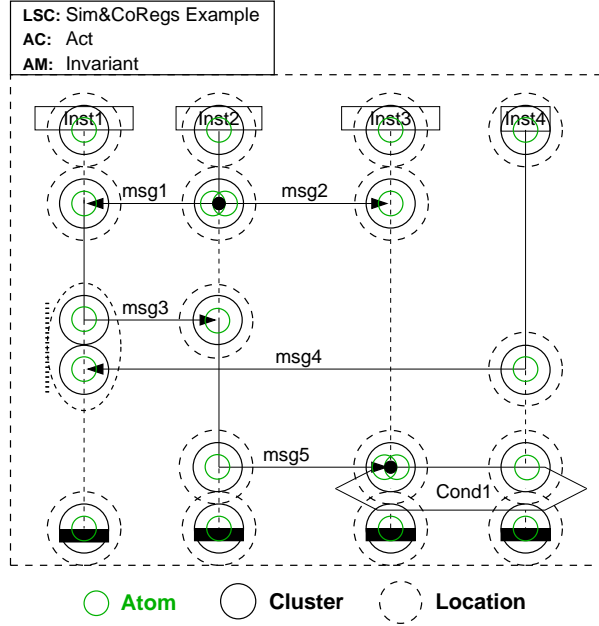


Fig. 4. The Sim&Coregs Example LSC with highlighted atoms, clusters and locations.

Example 2 (Locations). Figure 4 illustrates the location concept. Notice that the location of a cluster is identical to its position, except for clusters in coregions. All clusters in the coregion share one location.

The atoms introduced above are local to one instance. In order to be able to refer to the graphical elements used in the LSC it is necessary to establish the connection between atoms, which belong to the same LSC element but are located on different instances. Each message e.g. is made up of two atoms, the send and the receive atom, which are located on different instances. Likewise, conditions, which involve more than one instance are comprised by one condition atom on each participating instance. The relation between such atoms is established by the identifier, which designates the graphical element. For an LSC body l there are the following disjunct sets of unique identifiers: $Instances(l)$ the set of instance identifiers in l , $Messages(l)$ the set of message identifiers in l , $Conditions(l)$ the set of condition identifiers in l , $Local_Invariants(l)$ the set of identifiers of local invariants in l . The following functions associate an atom with its identifier:

$$\begin{aligned}
 instID &: Inst(l) \longrightarrow Instances(l) \\
 msgID &: Msgsnd(l) \cup Msgrcv(l) \longrightarrow Messages(l) \\
 condID &: Conds(l) \longrightarrow Conditions(l) \\
 liID &: LI_starts(l) \cup LI_ends(l) \longrightarrow Local_Invariants(l)
 \end{aligned}$$

The identification of the send and receive part of a message is achieved by the $msgID(\cdot)$ function. In the transition annotations of the automaton generated from an LSC body (cf. section 2.3) it is necessary to distinguish send and receive atoms, therefore the set of message labels $MsgLabels(l)$ is introduced, including the function $msgLabel(m)$, which associates a message label with each $m \in Msgsnd(l) \cup Msgrcv(l)$. Sending of message $m1$ is represented by the label $!m1$, receipt by $?m1$.

$$msgLabel : Msgsnd(l) \cup Msgrcv(l) \longrightarrow MsgLabels(l)$$

$$msgLabel(m) := \begin{cases} !msgID(m) & \text{if } m \in Msgsnd(l) \\ ?msgID(m) & \text{if } m \in Msgrcv(l) \end{cases}$$

Furthermore it is necessary to access the properties of messages, conditions and local invariants. For messages the relevant properties are temperature and type (instantaneous or asynchronous), for conditions and local invariants the mode (mandatory or possible). The progress information along each instance is associated with the atoms of this instance, so that the domain of function $temp(\cdot)$ below is the union of message identifiers and atoms. The properties for messages and conditions on the other hand pertain to the LSC elements, represented by the corresponding identifiers, and not the atoms. The situation for the mode of local invariants is identical to that of conditions, i.e. this information is tied to the LSC element.

$$temp : Messages(l) \cup Atoms(l) \longrightarrow \{hot, cold\}$$

$$sync_type : Messages(l) \longrightarrow \{async, instant\}$$

$$mode : Conditions(l) \cup Local_Invariants(l) \longrightarrow \{mandatory, possible\}$$

2.3 LSC Semantics

In this section we define the semantics of an LSC in terms of the language accepted by an automaton which is generated from the LSC. We first construct a proto-automaton called *unwinding structure* from the LSC body, which is then completed into a symbolic timed automaton (cf. section 2.1). The semantics of the LSC language in terms of a thorough description of the unwinding algorithm is defined in [5].

The central idea is to define a *cut* through the LSC starting at the top and moving this cut downward until we reach all instance ends, while respecting the partial order imposed by the LSC. Cuts become states in the automaton and the transition relation of the automaton encodes the successor relation among the cuts. Ordering among the atoms in an LSC is induced by the following rules:

1. atoms (clusters) along each instance axis are totally ordered (unless they are part of a coregion)
2. a message has to be sent before it can be received

3. conditions ranging over several instances (*shared conditions*) enforce synchronization between the involved instances

In order to be unwound an LSC element has to be *enabled*. An atom is enabled when:

1. all its predecessors along the instance axis have already be unwound
2. the corresponding message send atom has already been unwound (if the atom is a message receive atom of an asynchronous message) or is being unwound simultaneously (if the atom is part of an instantaneous message)
3. all other condition atoms belonging to the same condition are also enabled (if the atom is a shared condition)

The first condition demands a computation of all predecessors on the instance axis for each atom. It is actually sufficient to only conduct a local computation of the *immediate* predecessor of each atom, since this relation is transitive: If the immediate predecessor of a cluster cl has been unwound, then immediate predecessor of cl must have been unwound, and so on. The formal definition of the function computing the immediate predecessor is:

Definition 10 (Immediate Predecessor).

$$predecessor : Clusters(i) \longrightarrow \mathcal{P}(Clusters(i))$$

$$predecessor(cl) := \begin{cases} \emptyset & , \text{if } \exists a \in cl : a = \perp_i \vee a = \top_i \\ CL & , \text{else} \end{cases}, \text{ where}$$

$$CL := \{cl' \in Clusters(i) \mid location(cl') < location(cl) \wedge \neg \exists cl'' \in Clusters(i) : location(cl') < location(cl'') < location(cl)\}$$

The function $predecessor(cl)$ looks along one instance for the location, which is immediately above the cluster cl and returns the clusters bound to that location. If the element bound to the predecessor location is not a coregion, the returned set of clusters contains a single cluster. Otherwise all clusters of the coregion are returned. For clusters containing instance head or end atoms no predecessor is returned, since an instance head does not have a predecessor and instance end atoms are not unwound by the algorithm (see below).

The second and third rule above involve more than one instance and thus require to also take clusters of other instances into account. The message send and receive atom for an instantaneous message have to be unwound simultaneously, and all condition atoms of a condition must be unwound at the same time. This synchronization of clusters is expressed by the relation \approx .

Definition 11 (Equivalence of Clusters). Let $cl, cl' \in Clusters(l)$. The relation \approx on $Clusters(l)$ is defined as:

$$cl \approx cl' \Leftrightarrow \exists a \in cl \exists a' \in cl' : \\ cl = cl' \vee sharedConds(a, a') \vee instMsgs(a, a') \\ \vee transitivity(a, a')$$

where

$$\begin{aligned}
sharedConds(a, a') &:= a, a' \in Conds(l) \wedge condID(a) = condID(a') \\
instMsgs(a, a') &:= a \in Msgsnd(l) \wedge a' \in Msgrcv(l) \\
&\quad \wedge msgID(a) = msgID(a') \\
&\quad \wedge sync_type(msgID(a)) = instant \\
transitivity(a, a') &:= a, a' \in syncAtoms(l) \\
&\quad \wedge \exists cl'' \in Clusters(l) \exists a_i, a_j \in cl'' : |cl''| > 1 \\
&\quad \wedge (sharedConds(a, a_i) \vee instMsgs(a, a_i)) \\
&\quad \vee (sharedConds(a', a_j) \vee instMsgs(a', a_j)) \\
syncAtoms(l) &:= \{a \in Atoms(l) \mid a \in Conds(l) \vee \\
&\quad (a \in Msgsnd(l) \vee a \in Msgrcv(l)) \\
&\quad \wedge sync_type(msgID(a)) = instant\}
\end{aligned}$$

The classes defined by \approx are called *simultaneous classes*. $SimClasses(l) := \{ scl \subseteq Clusters(l) \mid \forall cl, cl' \in scl : cl \approx cl' \}$ is the set of simultaneous classes of l .

The simultaneous classes are the basic elements, which the unwinding algorithm operates. Similar to the case for clusters it is necessary to determine whether it is the predecessor(s) of a simultaneous class in order to determine, if it is enabled or not. This gives rise to the definition of the function $prerequisite(\cdot)$, which uses the $predecessor(\cdot)$ function defined for clusters:

Definition 12 (Simultaneous Class Prerequisite).

$$\begin{aligned}
prerequisite &: SimClasses(l) \longrightarrow \mathcal{P}(SimClasses(l)) \\
prerequisite(scl) &:= \begin{cases} \emptyset & , \text{if } \exists cl \in scl : \exists a \in cl : a \in Instheads(l) \\ SCL & , \text{else} \end{cases} , \text{ where} \\
SCL &:= \{ scl' \in Sim_Classes(l) \mid \\
&\quad \exists cl \in scl \exists cl' \in scl' : (cl' \in predecessor(cl) \vee \\
&\quad \exists a \in cl \exists a' \in cl' : a \in Msgrcv(l) \wedge \\
&\quad a' \in Msgsnd(l) \wedge msgID(a) = msgID(a') \wedge \\
&\quad sync_type(msgID(a)) = async) \}
\end{aligned}$$

The $prerequisite(\cdot)$ function is similar to the $predecessor(\cdot)$ function. It looks for the immediate predecessors of all clusters contained in the considered simultaneous class and collects the simultaneous classes for all predecessor clusters. If the current simultaneous class contains an asynchronous message receive atom, the simultaneous class containing the corresponding message send atom is added as well. It returns the empty set, if the considered simultaneous class contains an instance head atom.

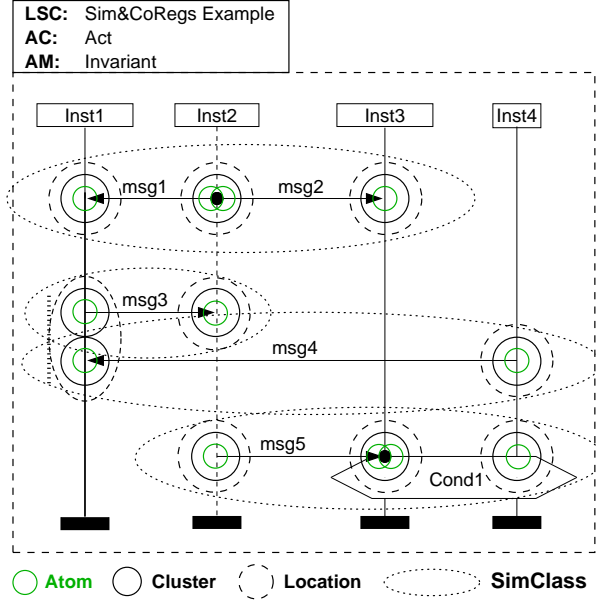


Fig. 5. The Sim&CoRegs Example LSC with highlighted atoms, clusters, locations and SimClasses.

Example 3 (Simultaneous classes).

Figure 5 illustrates the concepts of positions, clusters, locations and SimClasses. We have omitted the SimClasses for instance heads and ends for readability's sake.

A *cut* through the LSC is used to keep track of the progress of the unwinding procedure. It represents the borderline between already unwound elements and those which still have to be considered. The elements directly below the cut are those, which are currently enabled. Cuts are represented by a tuple containing one cluster of each instance:

Definition 13 (Cut). A cut $Cut \subseteq Clusters(i_1) \times \dots \times Clusters(i_n)$, for $Inst(l) = \{i_1, \dots, i_n\}$ is a tuple $(cl_{i_1}, \dots, cl_{i_n})$, $cl_{i_j} \in Clusters(i_j)$, $1 \leq j \leq n = |Inst(l)|$.

Let $Cuts(l)$ be the set of all possible cuts of LSC body l .

The unwinding algorithm requires a number of other auxiliary sets and constructs, which are collected in a tuple called *phase*. Each phase characterizes one unwinding step and consequently corresponds to a state in the resulting automaton. Possible transitions from one phase to another correspond to transitions of the automaton.

Definition 14 (Phase). A phase is a tuple $Phase := (Ready, History, Cut)$, with

- $History \subseteq SimClasses(l)$: the set of simultaneous classes which have already been unwound
- $Ready \subseteq SimClasses(l)$: the set of simultaneous classes, which are currently enabled to be unwound
- Cut : the current cut.

Let $Phases(l)$ be the set of all phases of LSC body l .

Since several simultaneous classes may be enabled concurrently, e.g. due to a coregion, each ready set may contain more than one simultaneous class. In this case all combinations of elements of the ready set lead to valid successor phases and thus have to be taken into consideration. All possible combinations correspond to the powerset of the ready set; the set of simultaneous classes, which is currently chosen to be unwound, is called the *fired set* $Fired \subseteq \mathcal{P}(SimClasses(l))$. Thus, for each phase $Phase_i$ there exist k fired sets $Fired_{i_k}$, with $k = |\mathcal{P}(Ready_i)|$. Let $Firedsets(l)$ be the set of all fired sets in l .

The unwinding algorithm starts at the top of the LSC body and computes the initial phase $Phase_0$, which is given by

$$\begin{aligned}
 Phase_0 &= (Ready_0, History_0, Cut_0), \text{ with} \\
 Ready_0 &:= \left\{ scl \in SimClasses(l) \mid prerequisite(scl) \in \right. \\
 &\quad \left. \mathcal{P}(\{scl' \in SimClasses(l) \mid \forall cl \in scl' : \forall a \in cl : \right. \\
 &\quad \quad \left. a \in (Instheads(l) \cup LI_starts(l))\}) \setminus \emptyset \right\} \\
 History_0 &:= \{\cup_{i \in Inst(l)} \{\{\perp_i\}\}\} \\
 Cut_0 &:= (\{\perp_1\}, \dots, \{\perp_n\})
 \end{aligned}$$

Starting with the initial phase the construction of the automaton considers every phase and computes the fired set(s) for it. For each phase a state is generated in the automaton and for each fired set of the current phase the successor phase is computed and the corresponding state is generated. For each fired set a transition is inserted, which is annotated with the simultaneous class(es) of the fired set. The successor phases for a phase is computed by the function $Step(\cdot, \cdot)$:

Definition 15 (Step).

$$\begin{aligned}
 Step &: Phases(l) \times Firedsets(l) \longrightarrow Phases(l) \\
 Step(Phase_i, Fired_{i_k}) &= Phase_j
 \end{aligned}$$

where

$$\begin{aligned}
 History_j &:= History_i \cup Fired_{i_k} \\
 Ready_j &:= \{scl \in SimClasses(l) \setminus \{\cup_{i \in Inst(l)} \{\{\top_i\}\}\} \mid \\
 &\quad \forall scl' \in prerequisite(scl) : scl' \in History_j \wedge scl \notin History_i\} \\
 Cut_j &:= (cl'_1, \dots, cl'_n), \text{ with} \\
 cl'_t &= \begin{cases} cl''_t & \exists f \in Fired_{i_k} \exists scl \in f : cl''_t \in scl \\ cl_t & \text{else} \end{cases}, \text{ for } t = 1, \dots, n
 \end{aligned}$$

The $Step(\cdot, \cdot)$ function is applied to all phases until the entire LSC body has been unwound, i.e. until the final phase is reached:

$$\begin{aligned} Phase_{final} &= (Ready_{final}, History_{final}, Cut_{final}), \text{ with} \\ Ready_{final} &:= \emptyset \\ History_{final} &:= SimClasses(l) \setminus \{\cup_{i \in Inst(l)} \{\{\top_i\}\}\} \end{aligned}$$

Example 4 (Primitive unwinding structure). Application of the $Step(\cdot, \cdot)$ function to the LSC in figure 5 on page 18 results in the (incomplete) automata shown in figure 6 on the facing page. The sets of the corresponding phases are listed in the following:

$$\begin{aligned} q_0 : History_0 &= \{\{\perp_{Inst1}\}, \{\perp_{Inst2}\}, \{\perp_{Inst3}\}, \{\perp_{Inst4}\}\} \\ Cut_0 &= (\perp_{Inst1}, \perp_{Inst2}, \perp_{Inst3}, \perp_{Inst4}) \\ Ready_0 &= \{\{\{?msg1\}, \{!msg1, !msg2\}, \{?msg2\}\}\} \\ q_1 : History_1 &= History_0 \cup \{\{\{?msg1\}, \{!msg1, !msg2\}, \{?msg2\}\}\} \\ Cut_1 &= (\{?msg1\}, \{!msg1, !msg2\}, \{?msg2\}, \perp_4) \\ Ready_1 &= \{\{\{!msg3\}, \{?msg3\}\}, \{\{?msg4\}, \{!msg4\}\}\} \\ q_2 : History_2 &= History_1 \cup \{\{\{!msg3\}, \{?msg3\}\}\} \\ Cut_2 &= (\{!msg3\}, \{?msg3\}, \{?msg2\}, \perp_4) \\ Ready_2 &= \{\{\{?msg4\}, \{!msg4\}\}\} \\ q_3 : History_3 &= History_1 \cup \{\{\{!msg4\}, \{?msg4\}\}\} \\ Cut_3 &= (\{?msg4\}, \{!msg1, !msg2\}, \{?msg2\}, \{!msg4\}) \\ Ready_3 &= \{\{\{?msg3\}, \{!msg3\}\}\} \\ q_4 : History_4 &= History_2 \cup \{\{\{!msg4\}, \{?msg4\}\}\} \\ Cut_4 &= (\{?msg4\}, \{?msg3\}, \{?msg2\}, \{!msg4\}) \\ Ready_4 &= \{\{\{!msg5\}, \{?msg5, cond1\}\}\} \\ q_5 : History_5 &= History_1 \cup \{\{\{!msg3\}, \{?msg3\}\}, \{\{!msg4\}, \{?msg4\}\}\} \\ Cut_5 &= (\{!msg3, ?msg4\}, \{?msg3\}, \{?msg2\}, \{!msg4\}) \\ Ready_5 &= \{\{\{!msg5\}, \{?msg5, cond1\}\}\} \\ q_6 : History_6 &= History_3 \cup \{\{\{!msg3\}, \{?msg3\}\}\} \\ Cut_6 &= (\{!msg3\}, \{?msg3\}, \{?msg2\}, \{!msg4\}) \\ Ready_6 &= \{\{\{!msg5\}, \{?msg5, cond1\}\}\} \\ q_7 : History_7 &= History_4 \cup \{\{\{!msg5\}, \{?msg5, cond1\}\}\} \\ Cut_7 &= (\{?msg4\}, \{!msg5\}, \{?msg5, cond1\}, \{!msg4\}) \\ Ready_7 &= \emptyset \\ q_8 : History_8 &= History_5 \cup \{\{\{!msg5\}, \{?msg5, cond1\}\}\} \\ Cut_8 &= (\{?msg4\}, \{!msg5\}, \{?msg5, cond1\}, \{!msg4\}) \\ Ready_8 &= \emptyset \\ q_9 : History_9 &= History_6 \cup \{\{\{!msg5\}, \{?msg5, cond1\}\}\} \\ Cut_9 &= (\{?msg4\}, \{!msg5\}, \{?msg5, cond1\}, \{!msg4\}) \\ Ready_9 &= \emptyset \end{aligned}$$

A closer look at the automaton reveals some redundant states and transitions: The phases for states q_7, q_8, q_9 are identical and could therefore be represented by a single state. The same is true for the phases for states q_4, q_5, q_6 , with the

exception that their cuts differ. The cuts are not directly relevant for the $Step(\cdot, \cdot)$ function, thus these states could be merged into one as well. The rule for being able to combine states is thus that they have identical history and ready sets.

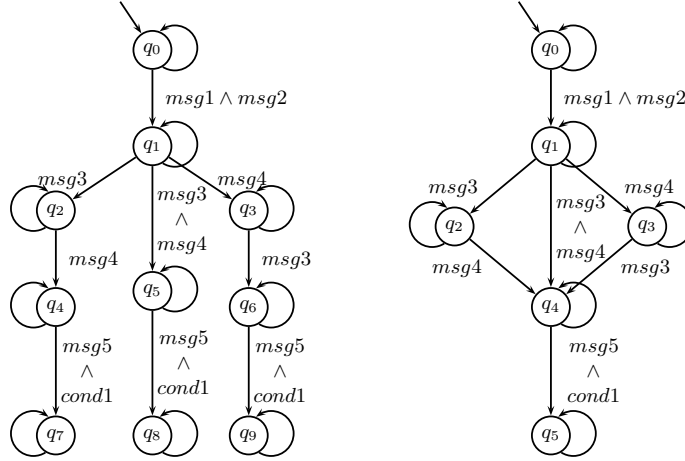


Fig. 6. Incomplete automata for `Sim&Coregs Example` LSC (c.f. figure 5 on page 18). The left automaton is gained directly from the primitive unwinding structure above; for the right automaton, the identical phases for states q_7, q_8, q_9 and q_4, q_5, q_6 are unified.

Self loop annotation and Interpretation. The question of how to annotate the self loops is influenced by the question whether duplicate messages are allowed, i.e. should it be considered an error, if a message, which is contained in an LSC body, is observed more than once during the activation of the LSC? The interpretation assumed in [4] is that duplicate messages result in an error and are thus prohibited. Applying this *strict* interpretation to the incomplete automaton generated so far results in an automaton where every self loop is annotated with the conjunction of the negation of all messages of the LSC; all non-self-loop messages similarly need to be extended by the conjunction of the negation of all messages of the LSC, which are not unwound in the current step. It is not immediately clear if this interpretation is too restrictive.

This motivates a second interpretation, which we call *weak* and which allows duplicate messages as long as the messages contained in the LSC are observed also at the correct times. This corresponds to annotating the self loops in the automaton only with the conjunction of the negation of all messages, which are in the ready set associated with the current state.

Both interpretations remove the non-determinism in the automaton due to the empty self loop annotation. The annotation for the self loop of the final state is identical for both variants: *true*, because once the behavior specified in the LSC has been observed entirely no further restriction applies to the system.

Example 5 (Sim&Coregs LSC and the corresponding symbolic automaton).

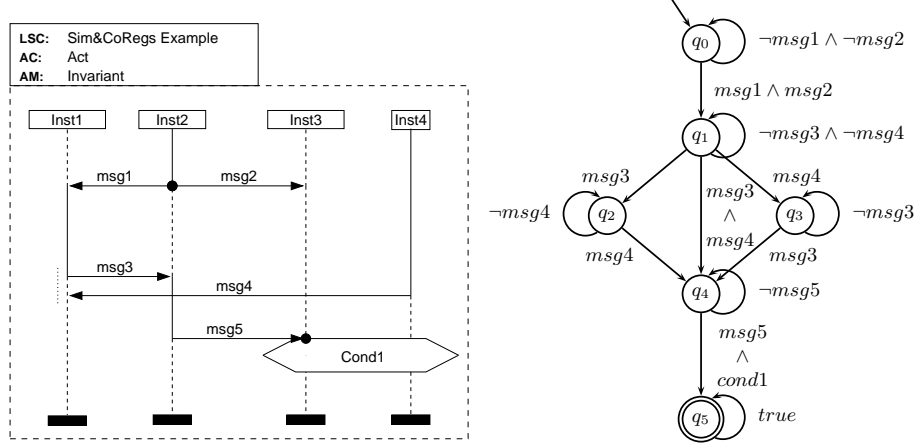


Fig. 7. The Sim&Coregs Example LSC and the corresponding symbolic automaton with annotated self loops using weak interpretation.

Treatment of Liveness and Temperatures. So far the location temperatures, i.e. the progress requirements on the instance axes, have not been considered. This information is collected by the cuts, which indicate up to which locations the unwinding has progressed. The local liveness information of each instance, given by the location temperature, is used to compute the global liveness requirements, which is expressed by the cut temperature defined below.

The LSC syntax as presented in section 2.2 needs to be extended in order to propagate the local temperature of the observed atoms. The atom temperature is first lifted to the cluster level and thereafter to the location level. Analogously to the cut temperature computation below, one hot atom overrides the temperature of all other atoms in a simultaneous region.

$$temp : Clusters(l) \longrightarrow \{hot, cold\}$$

$$temp(cl) := \begin{cases} cold, & \text{if } \forall a \in cl : temp(a) = cold \\ hot, & \text{else} \end{cases}$$

The temperature function is extended analogously to locations, so that coregions can be treated correctly. Since no ordering exists between the atoms of a coregion, no progress between them can be enforced, so that a single temperature is associated with a coregion.

$$temp : Locations(l) \longrightarrow \{hot, cold\}$$

$$temp(loc) := \begin{cases} cold, & \text{if } \forall cl \in Clusters(l) : location(cl) = loc \wedge \\ & temp(cl) = cold \\ hot, & \text{else} \end{cases}$$

Definition 16 (Cut Temperature).

$$\begin{aligned} cut_temp : Cuts(l) &\longrightarrow \{hot, cold\} \\ cut_temp(cut) &:= \begin{cases} hot & \text{if } \exists cl_j \in cut : temp(location(cl_j)) = hot \\ cold & \text{else} \end{cases}, \text{ with} \\ &cut = (cl_1, \dots, cl_n), 1 \leq j \leq n \end{aligned}$$

Thus phases containing a hot cut mean that the corresponding states in the automaton have to be left within a finite amount of time — analogously to hot locations on an instance. A cold cut indicates that the corresponding state needs not to be left. In conjunction with the self loop this means that such a state is fair in the sense of the Büchi acceptance criterion. Note that the final and the exit state are always fair, since no further requirements are posed by the LSC once it has been completed or exited.

Treatment of possible Conditions. Up to now conditions are only partly treated by the unwinding procedure. The unwinding so far only covers the case that the condition is fulfilled, violations are not considered.

For mandatory conditions no further steps are necessary, since evaluating a condition to false results in a non-accepting run (there exists no transition in the automaton for this case). As illustrated by the automaton in figure 7 condition **Cond1** must be satisfied, when message **msg5** occurs. For possible (cold) conditions, whose violations should result in an exit from the LSC, a special *exit state* is introduced, which is entered whenever a possible condition is violated. This state is similar to the final state inasmuch as no further restrictions apply after entering this state, i.e. the self loop annotation is *true* as well⁸.

Treatment of Time Constraints. So far the semantics of the presented LSC elements were expressible in an untimed symbolic automaton, whereas a timed symbolic automaton (c.f. section 2.1 on page 9) is needed when dealing with time constrained LSCs.

The idea is to associate a clock in the symbolic automaton with every timer and timing interval in the LSC, which is set to zero when the corresponding timer is set or the location of the timing interval is reached. When the timer is either reset or a timeout is observed or the location following the one with the timing interval is reached, a clock constraint is placed on the corresponding transitions in the automaton, which reflects the nature and duration of the timer or timing

⁸ Exit and final state can effectively be merged as a further optimization of the symbolic automaton.

interval. For a timeout atom this means that the value of the associated clock must be equal to the duration of the timer. For a timer reset atom the clock must be strictly less than the timer duration, because no timeout has been observed so far, i.e. the clock must not have reached its maximal value. For timing intervals the clock must be equal to or greater than the lower bound and less than or equal to the upper bound depending on the type of interval (open, closed).

2.4 Relating LSCs to a reference system

Informally the semantics of an LSC L is defined in terms of the timed runs (cf. definition 5 on page 10), which are produced by the reference system S and accepted by the automaton of L . The unwinding algorithm operates on the identifiers, which are associated with the unwound LSCs elements, and therefore the generated symbolic automaton is annotated with formulas over these identifiers. The final step in relating the LSC to the reference system is taken by mapping identifiers to concrete design elements.

Each *mapping function* for an LSC L and a reference system S , $map \in Maps(S, L)$, assigns a functional unit of S to each instance identifier, and a proposition over signals of S to each message label and condition and local invariant identifier. The concrete definition of $map(\cdot)$ depends on the reference system. If the reference system has a *static* structure, that is, the number and inter-relation of functional units is fixed in all system runs, then $Maps(S, L)$ is a singleton set. Otherwise, $Maps(S, L)$ provides a means to explain *dynamic binding*, i.e. having symbolic instance lines which do not have a one-to-one correspondence to a functional unit but represents *all* functional units of a kind.

For a symbolic automaton TSA we thus denote the substitution of all identifiers by their corresponding proposition by $map(TSA)$, the accepted language is consequently denoted by $\mathcal{L}(map(TSA))$.

The definition of *Satisfaction* for LSCs depends on the activation mode and quantification. Initial LSCs are activated at system start, whereas invariant and iterative LSCs are activated whenever the activation condition is true. Iterative LSCs allow only one incarnation of an LSC at a time, i.e. such an LSC may not be reactivated.

Definition 17 (Satisfaction of an LSC).

Let $L = (l, ac, pch, amode, quant)$ be an LSC, TSA_l the timed symbolic automaton generated for LSC body l by the unwinding algorithm, and S the corresponding reference system.

L is existentially satisfied by S , denoted $S \models_{\exists} L$, iff $\text{quant} = \text{existential} \wedge \exists tr = tr_0, tr_1, \dots \in \text{Runs}(S) : tr \models_{\exists} L$, where $tr \models_{\exists} L$ iff

$$\exists \text{map} \in \text{Maps}(S, L) : \begin{cases} tr_0 \models ac \wedge \\ \overrightarrow{tr_1} \in \mathcal{L}(\text{map}(\mathcal{TSA}_l)) & \text{amode} = \text{initial} \\ \exists i : tr_i \models ac \wedge \\ \overrightarrow{tr_{i+1}} \in \mathcal{L}(\text{map}(\mathcal{TSA}_l)) & \text{amode} = \text{invariant} \\ \exists i : tr_i \models ac \wedge \neg \text{active}(L) \wedge \\ \overrightarrow{tr_{i+1}} \in \mathcal{L}(\text{map}(\mathcal{TSA}_l)) & \text{amode} = \text{iterative} \end{cases}$$

L is universally satisfied by S , denoted $S \models_{\forall} L$, iff $\text{quant} = \text{universal} \wedge \forall tr = tr_0, tr_1, \dots \in \text{Runs}(S) : tr \models_{\forall} L$, where $tr \models_{\forall} L$ iff

$$\forall \text{map} \in \text{Maps}(S, L) : \begin{cases} tr_0 \models ac \Rightarrow \\ \overrightarrow{tr_1} \in \mathcal{L}(\text{map}(\mathcal{TSA}_l)) & \text{amode} = \text{initial} \\ \forall i : tr_i \models ac \Rightarrow \\ \overrightarrow{tr_{i+1}} \in \mathcal{L}(\text{map}(\mathcal{TSA}_l)) & \text{amode} = \text{invariant} \\ \forall i : tr_i \models ac \wedge \neg \text{active}(L) \Rightarrow \\ \overrightarrow{tr_{i+1}} \in \mathcal{L}(\text{map}(\mathcal{TSA}_l)) & \text{amode} = \text{iterative} \end{cases}$$

By $\overrightarrow{tr_i}$ we denote the suffix of the run tr starting at index i . The predicate $\text{active}(L)$ is true if there is another active incarnation of L , which has not yet reached its final or exit state.

3 Conclusion and Related Work

In the paper at hand, we have pointed out the deficiencies of the current state of the MSC language and emphasized the need for a formally well founded visual formalism. We presented the LSC language introduced by Damm and Harel in [4], which overcomes the observed shortcomings. We have sketched a part of the formal semantics of the LSC language as defined in [5] by providing the unwinding procedure, which generates a symbolic timed automaton from an LSC body, and have given an idea of the relation between LSCs and reference systems.

While our primary objective is the application of LSCs in the context of formal system verification, Harel and Marelly have proposed a different use case for LSCs described in [12]. The basic idea of this approach, called *Play-In/Play-Out*, is to play-in the desired interactions and use LSCs to record them. The key point is that no model representation exists at this time, so that the play-in procedure is carried out via a mock-up graphical user interface. Once a set of LSCs has been recorded in this way they can be used as behavior specifications, which monitor a user-guided simulation (*play-out*). This approach is intended to be used at the beginning of the development process to generate the basic system interactions in an easy and intuitive way. The view of this approach is an

operational one, whereas the purpose of LSCs as presented here is a denotational description of the behavior of system.

Another advanced use case for LSCs, described by Harel and Kugler in [13], is bridging the gap between requirements, specified by LSCs, and a behavioral model by automatically synthesizing a first-cut model from LSCs.

References

1. ITU-T: ITU-T Recommendation Z.120: Message Sequence Chart (MSC). ITU-T, Geneva (1993)
2. ITU-T: ITU-T Annex B to Recommendation Z.120: Algebraic Semantics of Message Sequence Charts. ITU-T, Geneva (1995)
3. Ben-Abdallah, H., Leue, S.: Expressing and analyzing timing constraints in message sequence chart specifications. Technical Report 97-04, Department of Electrical and Computer Engineering, University of Waterloo (1997)
4. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. *Formal Methods in System Design* **19** (2001) 45–80
5. Klose, J.: Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior. PhD thesis, Carl von Ossietzky Universität Oldenburg (2003)
6. Lamport, L.: Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering* **3** (1977)
7. Brill, M., Buschermöhle, R., Damm, W., Klose, J., Westphal, B., Wittke, H.: Formal verification of LSC's in the development process. In: this volume. (2004)
8. Schlör, R.: Symbolic Timing Diagrams : A Visual Formalism for Model Verification. PhD thesis, Carl von Ossietzky Universität Oldenburg (2000)
9. Thomas, W.: Automata on infinite objects. In van Leeuwen, J., ed.: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier Science Publishers (1990)
10. Emerson, E.A.: Temporal and modal logic. In van Leeuwen, J., ed.: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier Science Publishers (1990) 995–1072
11. Alur, R., Dill, D.: A theory of timed automata. *Theoretical Computer Science* **126** (1994) 183–236
12. Harel, D., Marelly, R.: *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer Verlag (2003)
13. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. *International Journal of Foundations of Computer Science* **13** (2002) 5–51