The Good, the Bad and the Ugly: Well-Formedness of Live Sequence Charts*

Bernd Westphal and Tobe Toben

Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany
{westphal, toben}@informatik.uni-oldenburg.de

Abstract. The Life Sequence Chart (LSC) language is a conservative extension of the well-known visual formalism of Message Sequence Charts. An LSC specification formally captures requirements on the inter-object behaviour in a system as a set of scenarios. As with many languages, there are LSCs which are syntactically correct but insatisfiable due to internal contradictions. The authors of the original publication on LSCs avoid this problem by restricting their discussion to well-formed LSCs, i.e. LSCs that induce a partial order on their elements.

This abstract definition is of limited help to authors of LSCs as they need guidelines how to write well-formed LSCs and fast procedures that check for the absence of internal contradictions. To this end we provide an exact characterisation of well-formedness of LSCs in terms of concrete syntax as well as in terms of the semantics-giving automata. We give a fast graph-based algorithm to decide well-formedness. Consequently we can confirm that the results on the complexity of a number of LSC problems recently obtained for the subclass of well-formed LSCs actually hold for the set of all LSCs.

1 Introduction

The Live Sequence Chart (LSC) language is a scenario based specification language that is used to formalise requirements on the inter-object behaviour of distributed systems under design. It conservatively extends the well-known Message Sequence Charts [10] basically by introducing modalities. The mode of the whole chart distinguishes example scenarios from scenarios the system must always adhere to, the mode of locations allows to require progress along an instance line, and the mode of conditions, which are semantically meaningful in LSCs, allows to add so called legal exits to a scenario. The modalities make LSCs strictly more powerful than MSCs whereas the graphical appeal and intuitivity of MSCs is preserved by indicating the modalities graphically. Scenario-based approaches in general [16, 1] and LSCs in particular [2, 4, 6, 7, 12] have shown adequate for the specification of requirements on distributed systems.

After a period of experimentation and evaluation, the language has stabilised into the two dialects of [11] and [9] and is subject of active research concerning

* This work was partly supported by the German Research Council (DFG) in SFB/TR 14 AVACS and in project DA 206/7-3 (USE), SPP 1064.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 230–246, 2006.

fundamental properties of the language, e.g. decidability and complexity of problems like realisability [5], and expressive power in terms of temporal logic [13, 15]. The two dialects emerged from two objectives of LSC specification usage. The LSCs of [9] are tailored for the so called play-out approach. They employ a tool called play-engine to execute an LSC specification. Thereby there needn't be an implementation of the intra-object behaviour of the system under design; the set of LSCs *is* the implementation. To this end, they added, e.g., actions to modify the state of the system, loops, and sub-charts to the original proposal. The LSCs of [11] are tailored for a more classical approach, where an LSC specification complements the model-based development of the intra-object behaviour of a system using, e.g., Statemate state-charts or UML state-machines. Whether the intra-object behaviour adheres to the LSC specification can then automatically be established by model-checking as has been demonstrated in [2, 14]. To this end the LSCs of [11] have, e.g., local invariants to state requirements or assumptions on periods of time and activation modes (cf. Sect. 2). We introduce the LSCs of [11] in more detail in Sect. 2 and in the following we mean these LSCs if not otherwise specified. Note that although these two usages of LSCs have been investigated independently, they don't exclude each other at all. Restraining to the common sublanguage of both dialects, one may write (or play-in [9]) an LSC specification, gain confidence into the specification by playing it out, and only then start a model-based implementation of intra-object behaviour that is then formally verified against the LSC specification.

The subject of this paper is an issue that turned up in the many experiments with formal verification for LSCs. There are LSCs that are syntactically correct but that are insatisfiable due to internal contradictions. The most obvious example are instantaneous messages that cross each other like m_2 and m_3 in the LSC body in Fig. 3(a) on page 239. For instantaneous messages, the sending and reception has to be observed simultaneously thus by the order on '$inst_1$' the sending and reception of m_2 has to be observed strictly before m_3. The order on '$inst_2$' requires it the other way round. Thus the LSC shown in Fig. 3(a) is not satisfied by any model. And if it were a pre-chart (cf. Sect. 2), then any model would (trivially) satisfy the whole LSC as the premise of the main chart is equivalent to '*false*'.

These errors in LSCs are not always as obvious as the one in Fig. 3(a) because such a cyclic dependency can involve many instance lines and large numbers of all kinds of LSC elements instead of just two instance lines like in Fig. 3(a). To support the actual authoring of quality LSC specifications, we relate the notion of well-formedness for LSCs from [8] to the concrete syntax and provide a fast algorithm that decides well-formedness on the syntactical representation of an LSC. Practically, LSCs should always be checked to be well-formed, in particular before expensive [5] checks like consistency of a whole LSC specification or LSC model-checking [11] are applied. This will significantly improve the usability of LSCs for formal verification because if the outcome of a model-checking run is not as expected, either because the LSC is insatisfiable or in the case of trivial satisfaction as outlined above, it is very helpful to know that the mismatch

between expectation and reality does *not* stem from the LSC not being well-formed so one can focus on the real reason for the mismatch.

The remainder of this article is structured as follows. Section 2 uses a small example to briefly recall the intuition of the LSC language of [11] and introduces abstract syntax and semantics using the new formalisation from [15]. Section 3 discusses the issue of non-well-formedness in more detail, formally defines well-formedness of LSCs, and exactly characterises the possible reasons for non well-formedness. Thereby we can justify that it is reasonable to consider only well-formed LSCs in this sense. In Sect. 4 we provide a fast algorithm that decides well-formedness on the abstract syntax, basically an acyclicity check on a particular structure, and discuss its complexity in terms of LSC size. Section 5 discusses the impact of well-formedness on related work, namely the complexity results of [5] that happen to be established just on the set of well-formed LSCs and it identifies LSCs played-in with the play-engine [9] to be well-formed by construction. Section 6 concludes and discusses further work.

2 Live Sequence Charts

To recall the syntax and intuition of LSCs, consider the LSC 'secure_crossing' given in Fig. 1(a). It states a high-level requirement on a distributed controller for a level crossing comprising a central controller 'CrossingCtrl' and separate controllers 'LightsCtrl' and 'BarrierCtrl' for traffic lights and barriers.

As mentioned in Sect. 1, the main difference between the LSC and the MSC language is that LSCs introduce modalities for whole charts, locations on instance lines, and LSC elements. The mode of the whole chart is indicated by the frame around the LSC body. The solid frame around the LSC body in Fig. 1(a) indicates that its mode is *universal*. A system satisfies a universal LSC if it adheres to the scenario *whenever* it is activated. The LSC in Fig. 1(a) is activated if the activation condition '*securing_request*' holds. In general, activation is characterised by a *pre-chart*, i.e. a prefix of the LSC body. The suffix of the LSC (the main-chart) is activated once the pre-chart has been completely observed. An activation condition is a shortcut for a pre-chart comprising only a single condition. A dashed frame around the LSC body indicates the other chart mode *existential*. A system satisfies an existential LSC if it has *at least one* run where the whole scenario (including the pre-chart) is observed *at least once*. This is the typical interpretation of MSCs.

The mode of a location in an LSC can be *hot* or *cold* and is indicated by the style of the instance line segment below the location. In Fig. 1(a) the topmost locations are cold on all instance lines but the one of the central controller 'CrossingCtrl', which is hot. The latter requires progress, that is, whenever the LSC is activated, a system only satisfies the LSC if the location is finally left by observing the element(s) at the following location. In the original proposal [8], the authors give the figurative intuition that one can't stand at a hot location forever without burning one's feet, thus one wants to leave it eventually. On the second location of instance line 'CrossingCtrl', the sendings of the instantaneous

messages '*lights_on*' and '*barrier_down*' are drawn at exactly the same location. They are thereby put in a *simultaneous region* (simregion for short) that requires them to both occur at the same point in time in a system that satisfies the LSC.

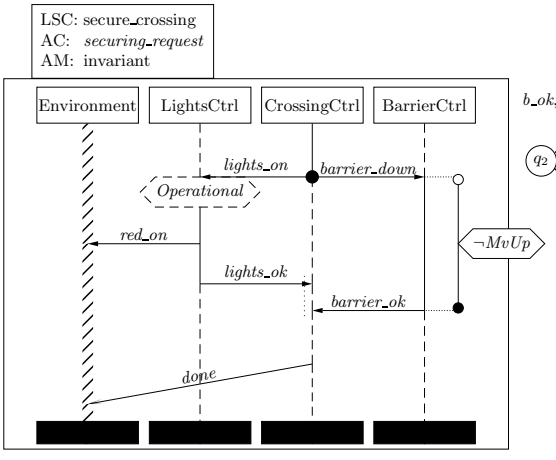
On the instance line of 'LightsCtrl', the reception of '*lights_on*' is co-located with a condition that has the mode *possible*, then also called cold condition. Thereby we express that if the lights are *not* operational, i.e. the cold condition doesn't hold at the point in time where '*lights_on*' is received, then the system needn't adhere to the rest of the chart. The chart is legally exited and immediately considered satisfied. Put the other way round, we do require that an implementation of the level crossing adheres to the remainder of the scenario whenever the lights controller is operational, i.e. if the cold condition holds. To specify requirements on the reaction of a non-operational lights controller we would provide another LSC that is activated in the situation where the lights controller is non-operational when receiving '*lights_on*'.

On the instance line 'BarrierCtrl', the reception of '*barrier_down*' is co-located with an exclusive beginning of a *mandatory* (or hot) *local invariant* that ends inclusively at the sending of message '*barrier_ok*'. By local invariants, requirements can be stated for spans of time in contrast to ordinary conditions which apply only to a single point in time. In the example we require that the barrier is not moving upwards immediately after '*barrier_down*' has been received up to and including the point in time where '*barrier_ok*' is sent. Should the barrier move upwards in this period of time, then the LSC is violated. Possible (or cold) local invariants are typically used to state assumptions. For example, if we change the lights controller's condition to a cold local invariant on the time between '*lights_on*' and '*lights_ok*' then we effectively say that a system has to adhere to the scenario unless the lights controller is not operational somewhere in between '*lights_on*' and '*lights_ok*'.

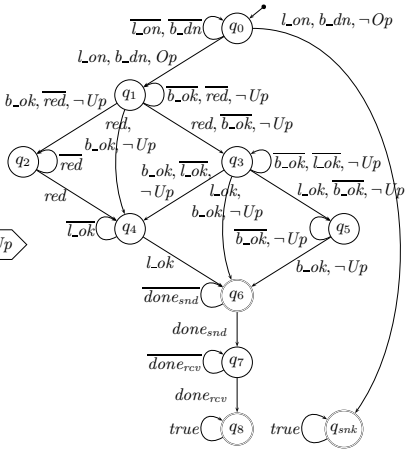
After receiving '*lights_on*', the lights controller is required to finally send a '*red_on*' message to the environment, that is, we decided that the traffic lights are not part of the model. If the lights are switched to red and the barriers are lowered, both distributed controllers should report success to the central controller by the messages '*lights_ok*' and '*barrier_ok*'. We indicate by the dotted line in parallel to the 'CrossingCtrl' instance line that we don't restrict the order of these two messages. They may occur in any order and even simultaneously. Such parts of instance lines where the order is explicitly relaxed are called *coregions*.

Note that the location before the sending of the asynchronous message '*done*', by which the central controller reports back that the crossing is secured, is cold and that we have then also reached cold locations on each other instance line. In this case, no progress is enforced. That is, a system that adheres to the LSC up to these locations but doesn't send '*done*' at all although satisfies the LSC.

In addition to giving the name and the activation condition, the *header* of the LSC in Fig. 1(a) comprises the *activation mode* that further restricts the activation. In order for a system to satisfy LSC 'secure_crossing' with activation mode *invariant*, each system run suffix that activates the LSC has to adhere to it. With activation mode *initial*, the LSC can be activated at most once per run,



(a) LSC for securing a level crossing.



(b) Symbolic Automaton $\mathcal{A}_L$ of secure_crossing's body (cf. Sect. 2.1).

$$\vec{I} \models_{LSC} L \Leftrightarrow \forall\, \vec{\iota} \in \vec{I}\, \forall\, k \in \mathbb{N}_0 : \vec{\iota}_k \models ac \Rightarrow \vec{\iota}/k \in \mathcal{L}(\mathcal{A}_L)$$

(c) Full Semantics of $L =$ 'secure_crossing' in terms of $\mathcal{A}_L$ (cf. Sect. 2.1).

Fig. 1. The securing protocol for a level crossing comprises switching on the red lights and lowering the barriers, each acknowledged by the responsible controller. For lack of space, message and condition names are abbreviated in Fig. 1(b), negation of message-observation predicates is expressed by overlining, and a comma is used for conjunction. E.g. q_0's loop fires if neither '*lights_on*' nor '*barrier_down*' are observed.

namely in the initial step. The third activation mode *iterative*, which excludes re-activation, lies outside the scope of this paper.

2.1 Syntax and Semantics

In the following, we introduce the syntax and semantics of LSCs following [11] in the formalisation of [15]. Note that we actually introduce a subset of LSCs that we call *core* LSCs. Core LSCs are missing three features that are out of the scope of this paper, namely we discuss activation only in form of activation conditions and not the general case of pre-charts, we exclude timer-set and -reset and timeout elements that LSCs inherit from MSCs, and for brevity don't consider possible messages, i.e. sending has to but reception needn't be observed. The first omission is not a restriction since the semantics of pre-charts is explained in terms of the same Symbolic Automaton construction used for main-charts thus our approach applies directly. The topic of timer consistency is orthogonal to the structural issues we discuss here.

One of the central informations of the concrete syntax of an LSC is the order of elements along a single instance line. As coregions may not be nested, the order of elements is actually a scenario order as defined in the following.

Definition 1 (LSC Instance Line). *Let P be a finite, non-empty set. The tuple $(P, \prec)$, $\prec \subseteq P \times P$, is called* instance line *iff $\prec$ is a* scenario order *(or* direct predecessor relation*) on P, that is, iff*

1. $\exists!\, a^\perp \in P\ \forall\, a \in P : a^\perp \prec^* a$ (Unique Minimum)
   where $\prec^*$ denotes the reflexive transitive closure of $\prec$.
2. $\forall\, a, a_1, a_2 \in P : a \prec a_1 \wedge a \prec a_2 \implies a_1 \not\preceq^* a_2$ (Unordered Successors)
   where $a_1 \not\preceq^* a_2$ denotes that a_1, a_2 are unordered*, i.e. $a_1 \not\prec^* a_2$ and $a_2 \not\prec^* a_1$.*
3. $\forall\, a_1, a_2 \in P : (\exists\, a_0 \in P : a_0 \prec a_1 \wedge a_0 \prec a_2)$
   $\implies (\forall\, a_3 \in P : a_1 \prec a_3 \implies a_2 \prec a_3)$. (Diamond Property)

A triple $(\mathbb{A}, \prec, \vartheta)$ with $\vartheta : \mathbb{A} \to \{hot, cold\}$ is called LSC instance line *iff $(\mathbb{A}, \prec)$ is an instance line. The elements $a \in \mathbb{A}$ are called* (tempered) atoms*. Two atoms $a_1, a_2 \in \mathbb{A}$ are called* instance co-located*, denoted by $a_1 \bowtie a_2$.* ♢

A core LSC is structured into the body and the information found in the head, namely the activation condition, the activation mode, and the quantification. The body is further structured into a set of LSC instance lines and three sets of the elements: messages, conditions, and local invariants. Each of these elements is equipped with an obligation mode from $\{mand, poss\} =: Obl$. Messages have a synchronity from $\{inst, asyn\} =: Sync$, and to each local invariant start- and end-atom a containedness from $\{incl, excl\} =: Cont$ is attached.

As the annotation of messages, conditions, and local invariants itself is not relevant in the course of this paper, we assume them to be from *Expr*, the set of boolean propositional expressions.

Definition 2 (Core LSC). *A* core LSC *is a tuple $L = (\ell, ac, am, quant)$ with* activation condition *$ac \in Expr$,* activation mode *$am \in \{initial, invariant\}$,* quantification *$quant \in \{existential, universal\}$, and $\ell = (Inst, Msg, Cond, LocInv)$ the* body *of the LSC where*

- *$Inst = \{(\mathbb{A}_1, \prec_1, \vartheta_1), \ldots, (\mathbb{A}_n, \prec_n, \vartheta_n)\}$ is a set of disjoint LSC instance lines. We set $Inst(L) := \{1, \ldots, n\}$, $\mathbb{A}_L := \bigcup_{i \in Inst} \mathbb{A}_i$, $\prec_L := \bigcup_{i \in Inst} \prec_i$, and $\vartheta_L := \bigcup_{i \in Inst} \vartheta_i$. We denote by $a_i^\perp$ the minimum of $\prec_i$, $i \in Inst(L)$, also called* instance head*, and set $\mathbb{A}_L^\perp := \{a_i^\perp \mid i \in Inst(L)\}$. By $A|_i := A \cap \mathbb{A}_i$ we denote the projection of a set $A \subseteq \mathbb{A}_L$ onto instance $i \in Inst(L)$.*

  If the LSC L is clear by context we shall simply write, e.g., $\prec$ instead of $\prec_L$.
- *$(m \in)$ $Msg =: Msg(L)$ is a set of* messages

$$m = (a_s, a_r, \varsigma, \kappa, \psi_s, \psi_r) \in \mathbb{A} \times \mathbb{A} \times Sync \times Obl \times Expr \times Expr$$

  For each message $m \in Msg$ we set $atoms(m) := \{a_s(m), a_r(m)\}$.

  By $Msg_{inst}(L) := \{m \in Msg(L) \mid \varsigma(m) = inst\}$ we denote the set of instantaneous and by $Msg_{asyn}(L) := \{m \in Msg(L) \mid \varsigma(m) = asyn\}$ the set of asynchronous messages of L.

– ($c \in$) *Cond* =: *Cond*(L) *is a set of* conditions

$$c = (A_c, \kappa, \psi_c) \in 2^{\mathbb{A}} \setminus \{\emptyset\} \times \mathit{Obl} \times \mathit{Expr}$$

where there is at most one atom per instance line, i.e. $|(A_c|_i)| \leq 1$ *for* $i \in \mathit{Inst}$. *For each condition* $c \in \mathit{Cond}$ *we set* $\mathit{atoms}(c) := A_c(c)$;

– ($l \in$) *LocInv* =: *LocInv*(L) *is a set of* local invariants

$$l = ((a_s, \gamma_s), (a_e, \gamma_e), \kappa, \psi) \in (\mathbb{A} \times \mathit{Cont}) \times (\mathbb{A} \times \mathit{Cont}) \times \mathit{Obl} \times \mathit{Expr}.$$

For each local invariant $l \in \mathit{LocInv}$ *we set* $\mathit{atoms}(l) := \{a_s(l), a_e(l)\}$.

The set $\mathit{elems}(L) := \mathit{Msg}(L) \cup \mathit{Cond}(L) \cup \mathit{LocInv}(L)$ *is called the set of* elements of L. *The function 'atoms' is canonically extended to subsets of* $\mathit{elems}(L)$ *yielding sets of atoms. We set* $\mathit{atoms}(L) := \mathit{atoms}(\mathit{elems}(L))$. ♢

In order to formally capture well-formedness of LSCs in Sect. 3, we need to introduce a number of concepts belonging to the LSC semantics definition [11]. We stop with a brief introduction of the Symbolic Automaton that is the basis of the LSC semantics definition following [11] in order to be able to study the relation between the original definition of well-formedness and a semantical definition in terms of Symbolic Automata.

The central concept in the construction of the automaton is the *cut*, i.e. a set of atoms per instance line, indicating how far the LSC has been observed. A cut is empty or comprises at least one atom for each instance line. All instance co-located atoms in a cut are pairwise unordered.

Definition 3 (Cut). *Let* L *be a core LSC. A set of atoms* $\alpha \subseteq \mathit{atoms}(L)$ *is called* cut *iff*

1. $\alpha \neq \emptyset \implies \forall\, i \in \mathit{Inst}(L) : \alpha|_i \neq \emptyset$, *and*
2. $\forall\, i \in \mathit{Inst}(L)\ \forall\, a_1, a_2 \in \alpha|_i : a_1 \bowtie a_2 \implies a_1 \not\preccurlyeq^* a_2$.

The empty cut is called initial cut of L *and denoted by* α_0, *the cut comprising all instance heads is called* instance heads cut of L *and denoted by* $\alpha_\perp(L)$, *and the maximal cut* α *with* $\forall\, a \in \alpha\ \forall\, a' \in \mathit{atoms}(L) : a \prec^* a' \implies a' = a$ *is called* final cut of L *and denoted by* $\alpha_{\mathit{fin}}(L)$.

The temperature of α *is 'cold' if* $\alpha = \alpha_{\mathit{fin}}(L)$ *or* $\vartheta(a) = \mathit{cold}$ *for all* $a \in \alpha$, *and 'hot' otherwise. The set of all cuts of* L *is denoted by* $\mathit{Cuts}(L)$. ♢

The unit by which a cut can be advanced is the simultaneous class (simclass for short). A simclass is defined by the fact that the two atoms of a synchronous message and all atoms of a condition are supposed to be observed simultaneously, and by transitivity, e.g. if two synchronous messages m_1, m_2 each use an atom which is also used by a condition c, then all atoms of these three elements m_1, m_2, and c belong to the same simclass.

Definition 4 (Simclass). *Let* L *be a core LSC. Two atoms* $a_1, a_2 \in \mathit{atoms}(L)$ *are called* simultaneous, *denoted by* $a_1 \sim a_2$, *iff*

1. $a_1 = a_2$, *or*
2. $\{a_1, a_2\} \subseteq \mathbb{A}_L^{\perp}$, *or*
3. $\exists\, e \in Cond(L) \cup Msg_{inst}(L) : \{a_1, a_2\} \subseteq atoms(e)$, *or*
4. $\exists\, a_3 \in atoms(L) : a_1 \sim a_3 \wedge a_3 \sim a_2$.

For each $a \in atoms(L)$ *we use* $[a]$ *to denote the equivalence class of 'a' wrt.* $\sim$*, i.e. the set* $\{a' \in atoms(L) \mid a' \sim a\}$*. The set* $atoms(L)/\sim$ *of all equivalence classes of atoms from* $atoms(L)$ *is also denoted by* $Simclass(L)$*, its elements are called* simclasses*. We use* $elems(scl)$ *to denote all LSC elements that share an atom with* $scl \in Simclass(L)$*, i.e. the set* $\{e \in elems(L) \mid atoms(e) \cap scl \neq \emptyset\}$. $\diamondsuit$

A simclass is intuitively enabled by a cut if each of its atoms either has its prerequisite in the cut or it belongs to a coregion and there is an atom in the cut from the same coregion. Furthermore the intuition of asynchronous messages is explicitly added in form of an additional restriction: reception of an asynchronous message shall be observed strictly after its sending.

Definition 5 (Enabled simclass). *Let* L *be a core LSC. Let* $\alpha \in Cuts(L)$ *be a cut of* L *and* $scl \in Simclass(L)$ *a simclass of* L*. A cut is said to* enable *'*scl*', denoted by* $\alpha \rhd scl$*, iff*

$$(\forall\, a' \in scl : prereq(a') \subseteq \alpha \vee \exists\, a \in \alpha : a \bowtie a' \wedge a \not\preccurlyeq^* a')$$
$$\wedge\ (\forall\, m \in Msg_{asyn}(L) \cap elems(scl) : a_r(m) \in scl \implies \exists\, a \in \alpha : a_s(m) \prec^* a)$$

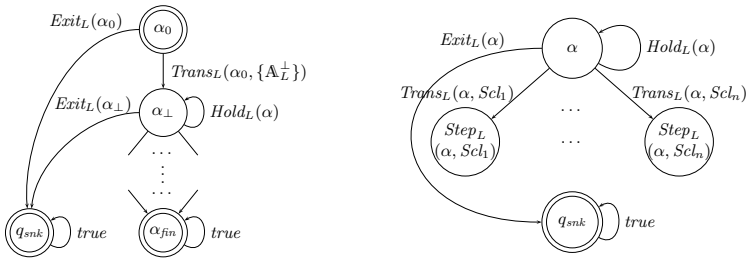
where $prereq(a) := \{a' \in atoms(L) \mid a' \prec a\}$ *is the* prerequisite *of* a*. The set of sets of simclasses* $Ready_L(\alpha) := \{\emptyset \neq Scl \subseteq Simclass(L) \mid \forall\, scl \in Scl : \alpha \rhd scl\}$ *is called the* readyset *of* α. $\diamondsuit$

Note that enabledness is a *structural* concept. It does not consider, e.g., the boolean expressions of conditions. In other words, the concept of enabledness can be seen as denoting *potential* progress. As there is no total order on the LSC elements, a single cut may enable multiple simclasses. Given a cut α and a non-empty set $Scl \in Ready_L(\alpha)$ of enabled simclasses from the readyset of α, the advancement function $Step_L(\alpha, Scl)$ denotes the (unique) follow-up cut.

We can now sketch the construction of an LSC's Symbolic Automaton which is basically a Büchi automaton where the transitions are labelled by boolean expressions. Formally, a *Symbolic Automaton* (SA) is a tuple $\mathcal{A} = (Q, q_s, \rightsquigarrow, F)$ comprising a finite set of states Q, the initial state $q_s \in Q$, the transition relation $\rightsquigarrow \subseteq Q \times Expr \times Q$, and the set of accepting states $F \subseteq Q$. We write $q_i \to q_j$ iff $(q_i, \psi, q_j) \in \rightsquigarrow$ for some ψ. An SA is called *Partially Ordered Symbolic Automaton* (POSA) if the reflexive transitive closure of $\to$ is a partial order.

The definition of an LSC's SA uses three kinds of predicates. The predicate $Hold_L$ on the self-loop characterises the condition under which a cut α is not advanced, $Exit_L$ on transitions to the legal exit characterises the conditions for legal exit from cut α, and $Trans_L$ corresponds to the observation of a given set of simclasses Scl in a particular cut α.

Definition 6 (Symbolic Automaton of an LSC). *The* Symbolic Automaton *of a core LSC* L*, denoted by* $\mathcal{A}_L$*, is the tuple* $(Q, q_s, \rightsquigarrow, F)$ *with* $Q = Cuts(L) \,\dot\cup\, \{q_{snk}\}$*,* $q_s = \alpha_0$*,* $F = \{\alpha \in Cuts(L) \mid \vartheta(\alpha) = cold\} \cup \{q_{snk}\}$*, and*



(a) Overall structure of $\mathcal{A}_L$. (b) Outgoing transitions from state α.

Fig. 2. Structure of the LSC body automaton. Double lined states are in F.

$$\begin{aligned}\rightsquigarrow = &\{(\alpha_0, \mathit{false}, \alpha_0)\} \cup \{(q_{snk}, \mathit{true}, q_{snk})\} \\ &\cup \{(\alpha, \mathit{Hold}_L(\alpha), \alpha) \mid \alpha \in \mathit{Cuts}(L) \setminus \{\alpha_0\}\} \\ &\cup \{(\alpha, \mathit{Exit}_L(\alpha), q_{snk}) \mid \alpha \in \mathit{Cuts}(L) \setminus \{\alpha_{fin}(L)\}\} \\ &\cup \{(\alpha, \mathit{Trans}_L(\alpha, \mathit{Scl}), \alpha') \mid \alpha \in \mathit{Cuts}(L), \mathit{Scl} \in \mathit{Ready}_L(\alpha), \alpha' = \mathit{Step}_L(\alpha, \mathit{Scl})\}\end{aligned}$$

Figure 2(a) shows the overall structure of the automaton, and Fig. 2(b) depicts the outgoing transitions of a single state resp. cut. Figure 1(b) gives the complete Symbolic Automaton of the LSC from Fig. 1(a). The Symbolic Automaton of an LSC is a POSA [15].

The notions introduced up to now are sufficient for the following sections that don't consider, e.g., information from the LSC header. For completeness, Fig. 1(c) exemplarily shows how the semantics of a complete LSC is expressed in terms of $\mathcal{L}(\mathcal{A}_L)$ the language accepted by the Symbolic Automaton of the body in [11]. The system model $\vec{I}$ is a set of *interpretation sequences*, that is, infinite sequences of interpretations of the predicates referred to by the LSC elements. It satisfies a universal invariant LSC L, denoted by $\vec{I} \models_{LSC} L$, iff each suffix $\vec{\iota}_k$ of a run $\vec{\iota}$ whose first snapshot $\vec{\iota}_k$ satisfies the activation condition is in the language of $\mathcal{A}_L$. In the initial activation mode, we would only consider $k = 0$.

3 Well-Formedness of Live Sequence Charts

Right from the original introduction of LSCs in [8] it has been clear that there are syntactically correct LSCs that shouldn't be considered legal. For this reason, [8] introduced a dependency relation on the LSC elements and restricted their definition of the semantics of LSCs to those with acyclic dependency relation. For the same reason, [5] restrict their investigation of the decidability and complexity of common problems, like consistency or realisability for a set of LSCs, to labelled partial orders thus they completely cover the well-formed LSCs of [8]. Up to now open is the question which syntactically correct *graphical* charts are actually ruled out by this restriction and how an author of LSCs best decides whether his LSCs are well-formed or not.

A first attempt to characterise well-formedness graphically to support authors of LSCs is given by [11]. They informally state, among others, the rule that

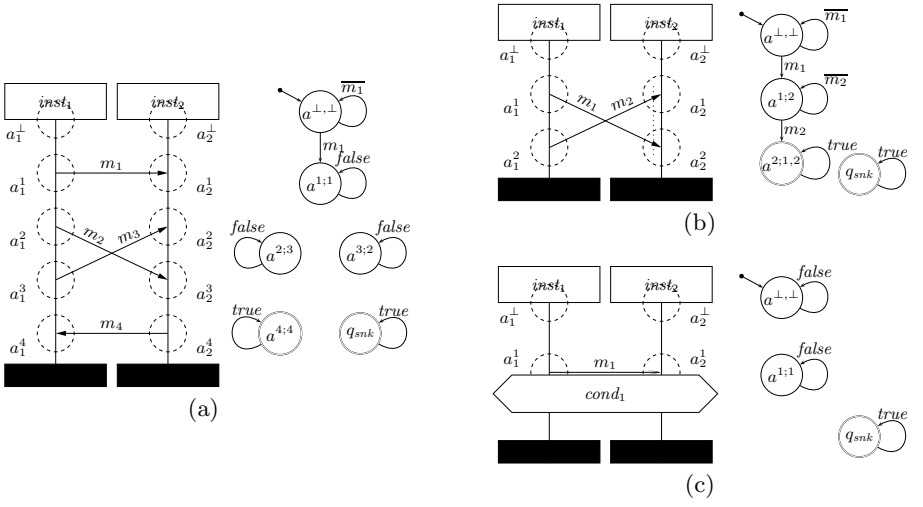


Fig. 3. Conflicting LSCs and their automata. For lack of space, we use $a^{x_1,\ldots,x_n;y_1,\ldots,y_m}$ to denote the cut $\{a_1^{x_1},\ldots,a_1^{x_n},a_2^{y_1},\ldots,a_2^{y_m}\}$.

instantaneous messages shall not cross each other. This rule is intuitively safe. For example, the LSC shown in Fig. 3(a) is obviously not well-formed since messages m_2 and m_3 cyclically depend on each other. The rule correctly identifies it as such. But it incorrectly identifies the LSC shown in Fig. 3(b) as not well-formed although the cyclic dependency is broken using a coregion, i.e. this LSC actually is well-formed.

An example that is not covered by any rule of [11] is the LSC shown in Fig. 3(c). There the initial cut doesn't enable any subsequent simclass because the condition synchronises atoms a_1^1 and a_2^1 and message m_1 being asynchronous requires a_1^1, the sending, to be observed strictly before a_2^1, the reception.

In the following Lemma 1, we establish that there are exactly two possible reasons for an LSC not to be well-formed. Firstly, that the precedence imposed by the atom order contradicts the synchronisation imposed by instantaneous messages and conditions on their atoms. Secondly, that the precedence imposed by the atom order contradicts the order between the sending and reception atom of an asynchronous message. So by Lemma 1, the two cases shown in Fig. 3(a) and 3(c) are actually all. Another conclusion we can draw is that the language extensions of [11] don't introduce new means to produce non well-formed LSCs.

As these two cases are contradictions between fundamental principles of LSCs, they can't be resolved and thereby we can see that the restriction to well-formed LSCs is just right. We don't miss any interesting graphical representations of LSCs.

The following definition introduces the precedence relation $<_L$ that captures the interdependency between atoms on different instance lines based on the scenario order $\prec$, the asynchronous messages, and the synchronising elements like conditions and instantaneous messages.

Definition 7 (Precedence Relation). *Let L be a core LSC. For two atoms $a_1, a_2 \in atoms(L)$ we say that a_1* precedes *a_2, denoted by $a_1 <_L a_2$, iff*

1. $a_1 \prec^+ a_2$, *or*
2. $\exists\, m \in Msg_{asyn}(L) : a_1 = a_s(m) \wedge a_2 = a_r(m)$, *or*
3. $\exists\, a_1', a_2' \in atoms(L) : a_1 \in [a_1'] \wedge a_1' <_L a_2' \wedge a_2 \in [a_2']$. ♢

If the precedence relation is acyclic, then it is a strict partial order and its reflexive closure is the partial order $\leq_m$ that the discussion in [8] is restricted to. In terms of our precedence relation, well-formedness is defined as follows. Lemma 1 then introduces the two possible reasons for non well-formedness.

Definition 8 (Well-formed LSCs). *A core LSC is called well-formed iff its precedence relation $<_L$ is acyclic.* ♢

Lemma 1 (Contradiction). *A core LSC L is not well-formed iff*

1. $\exists\, scl \in Simclass(L)\ \exists\, a, a' \in scl : a \neq a' \wedge a <_L a'$ *(synchrony contradiction)*
2. *or* $\exists\, m \in Msg_{asyn}(L) : a_r(m) <_L a_s(m)$ *(asynchrony contradiction)* ♢

Proof. "$\Longrightarrow$": Let L be a non well-formed LSC. Then by Def. 8 the precedence relation has a cycle, i.e. $\exists\, a \in atoms(L) : a <_L a$ $(*)$.
As '$\prec^+$' is irreflexive, we are left with two reasons for $(*)$:

- There are atoms $a_1, a_1', a_2', a_2 \in atoms(L)$ s.t.

$$a <_L a_1 \wedge a_1 \in [a_1'] \wedge a_1' <_L a_2' \wedge a_2' \in [a_2] \wedge a_2 <_L a$$

  and $a_j \neq a_j'$, $j \in \{1, 2\}$. Then we have asynchrony contradiction for $[a_j]$.
- There are messages $m_1, \ldots, m_n \in Msg_{asyn}(L)$, $n > 0$, s.t.

$$a \prec^* a_s(m_1) \wedge a_r(m_1) \prec^* a_s(m_2) \wedge \cdots \wedge a_r(m_{n-1}) \prec^* a_s(m_n) \wedge a_r(m_n) \prec^* a$$

  For m_1 we have asynchrony contradiction.

Note that the cases are exhaustive but not exclusive. There are cycles of the precedence relation that show both contradictions.

"$\Longleftarrow$": For the other direction first consider case (2.). Then there is a message $m \in Msg_{asyn}(L)$ s.t. $a_r(m) <_L a_s(m)$. By Def. 7.2 we have $a_s(m) <_L a_r(m)$ thus a cycle. In case (1.) there is a simclass $scl \in Simclass(L)$ and atoms $a, a' \in scl$ with $a \neq a'$ and $a <_L a'$. Then there are atoms $a_1, \ldots, a_n \in atoms(L)$ s.t.

$$a \in [a_1] \wedge a_1 <_L a_2 \wedge a_2 \in [a_3] \cdots \wedge a_{n-2} \in [a_{n-1}] \wedge a_{n-1} <_L a_n \wedge a \in [a_n]$$

where $a_i <_L a_{i+1}$ by Def. 7.1 or 7.2 thus $a <_L a$ by Def. 7.3. □

Considering the Symbolic Automata of the LSCs in Fig. 3 shown next to the LSCs, we observe that the two non well-formed examples have in common that there are unconnected states in the automaton. Thus the particular Symbolic

Automata in Fig. 3(a) and 3(c) don't accept any run, they necessarily get stuck.

In the following, we establish that we could've equally well started to define well-formedness semantically by calling those LSCs well-formed whose Symbolic Automaton has a *traversable structure*. This is practically relevant for the tools which compile an LSC to its Symbolic Automaton that is then used for formal verification. Alternatively to applying the stand-alone well-formedness check introduced in Sect. 4 beforehand, these tools can speculatively start to construct the Symbolic Automaton and as soon as they hit a state with empty readyset testify that they are facing a non well-formed input. In addition, the construction of a cycle in the proof of Lemma 2 can then be used to determine an actual cycle and show it to the user.

Definition 9 (Traversable Structure). *Let $\mathcal{A} = (Q, q_s, \rightsquigarrow, F)$ be a Symbolic Automaton and $q \in Q$ state. The Symbolic Automaton $\mathcal{A}$ is said to have a* traversable structure wrt. *q iff q is reachable from any state that is reachable from the initial state, i.e. $\forall\, q' \in Q : (q_s \rightarrow^* q') \implies (q' \rightarrow^* q)$.* ♢

Lemma 2 (Well-formedness and Traversable Structure). *Let L be a core LSC and $\mathcal{A}_L = (Q, q_s, \rightsquigarrow, F)$ its Symbolic Automaton with $Q = Cuts(L) \dot\cup \{q_{snk}\}$. The LSC L is* well-formed *iff $\mathcal{A}_L$ has a completely traversable structure wrt. $\alpha_{fin}(L) \in Cuts(L)$.* ♢

For the proof of Lemma 2, we observe the following relation between synchrony and asynchrony contradiction and the enabling behaviour of legal cuts, i.e. cuts which are the result of successive application of the step function $Step_L$.

Lemma 3. *Let L be a core LSC.*

1. *Let $m \in Msg_{asyn}(L)$ be an asynchronous message with $a_r(m) <_L a_s(m)$. Then there is no legal cut $\alpha \in Cuts(L)$ with $\alpha \rhd [a_r(m)]$.*
2. *Let $scl \in Simclass(L)$ be a simclass s.t. there are $a, a' \in scl$ with $a \neq a'$ and $a <_L a'$. Then there is no legal cut $\alpha \in Cuts(L)$ with $\alpha \rhd scl$.* ♢

Proof.
- Let $m \in Msg_{asyn}(L)$ with $a_r(m) <_L a_s(m)$. Assume there is a legal cut $\alpha \in Cuts(L)$ with $\alpha \rhd [a_r(m)]$. A reception is only enabled if the sending has been observed, i.e. there is $a \in \alpha$ with $a_s(m) \prec^* a$. Following backwards the precedence order chain $a_r(m) <_L a_1 <_L \ldots <_L a_n <_L a_s(m)$ there is an $a' \in \alpha$ with $a' \prec^* a_r(m)$, in contradiction to $a'' \not\prec^* a_r(m)$ or $a'' \prec a_r(m)$ for all $a'' \in \alpha$ as required by $\alpha \rhd [a_r(m)]$.
- Let $scl \in Simclass(L)$ s.t. there are $a, a' \in scl$ with $a \neq a'$ and $a <_L a'$. Assume there is a legal cut $\alpha \in Cuts(L)$ with $\alpha \rhd scl$. Following backwards the precedence order chain $a <_L a_1 <_L \ldots <_L a_n <_L a'$ yields $a \prec^* a_0$ for an atom $a_0 \in \alpha$ in contradiction to $\alpha \rhd scl$. □

Proof. (of Lemma 2)
"$\Longleftarrow$": (contraposition) Let L be a non well-formed core LSC and $\mathcal{A}_L$ its Symbolic Automaton. By Lemma 1, L has at least one contradiction:

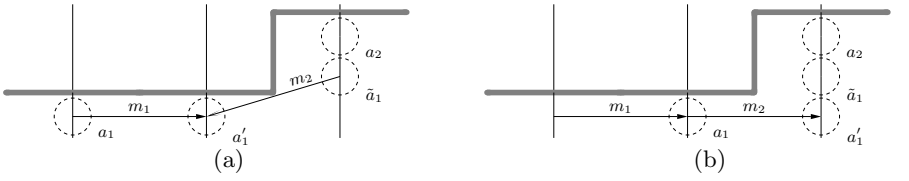


Fig. 4. Two reasons for $\alpha \not\rhd [a_1]$. The gray line shows the location of the cut α.

- *synchrony contradiction*, i.e. there is a a simclass $scl \in Simclass(L)$ and atoms $a, a' \in scl$ with $a \neq a'$ and $a <_L a'$. Then there is no cut which enables scl by Lemma 3.2.
- *asynchrony contradiction*, that is, there is an asynchronous message $m \in Msg_{asyn}(L)$ s.t. $a_s(m) <_L a_r(m)$. Then there is no cut which enables $scl := [a_r(m)]$ by Lemma 3.1.

If $\mathcal{A}_L$ had a completely traversable structure wrt. $\alpha_{fin}(L)$ then in particular $\alpha_0 \to^* \alpha_{fin}(L)$. Omitting the proof we use that in this sequence of cuts from α_0 to $\alpha_{fin}(L)$, for each atom $a \in atoms(L)$ there is a cut $\alpha \in Cuts(L)$ with $\alpha \rhd [a]$ in contradiction to the observation that scl is not enabled by any cut.

"$\Longrightarrow$": (contraposition) Let L be a core LSC whose Symbolic Automaton $\mathcal{A}_L$ is not completely traversable wrt. $\alpha_{fin}(L)$. As $\mathcal{A}_L$ is a POSA, there is a state $\alpha_{fin}(L) \neq \alpha \in Q = Cuts(L)$ that is reachable from the initial state $q_s = \alpha_0$ and doesn't enable any simclass, i.e. $Ready_L(\alpha) = \emptyset$. The cut α is not the initial cut α_0 because $\alpha_0 \rhd \alpha_\perp(L)$ by definition. By construction of $\mathcal{A}_L$ the cut α is reachable if it is the result of successive applications of the step function $Step_L$, hence α is legal. Let $A := \{a \in atoms(L) \mid \exists\, a_0 \in \alpha : a_0 \prec a \vee a_0 \not\asymp^* a\}$ be the set of atoms that are direct predecessors of an atom in α or belong to the same coregion as an atom in α. The set A comprises those atoms that can possibly be enabled by α. It is not empty because $\alpha \neq \alpha_{fin}(L)$

From A we can iteratively construct a cycle in the precedence relation as follows. Choose $a_1 \in A$. Its simclass $[a_1]$ is not enabled because α doesn't enable any simclass. There are two possible reasons for $\alpha \not\rhd [a_1]$ by the definition of '$\rhd$'. Firstly (cf. Fig. 4(b)) that there is an atom $a_1' \in [a_1]$ that is a message reception and the sending $\tilde{a}_1$ has not yet been observed $(*)$ and secondly (cf. Fig. 4(a)) that there is an atom $a_1' \in [a_1]$ whose prerequisite is not in α $(**)$, that is, $\exists \tilde{a}_1 \in atoms(L), a_0 \in A : a_0 \prec^+ \tilde{a}_1 \prec^+ a_1'$. As $\alpha \neq \alpha_0$, we can choose an atom $a_2 \in A$ with $\tilde{a}_1 \bowtie a_2$, i.e. instance co-located with $\tilde{a}_1$. If $\tilde{a}_1 \in A$, then the choice is $a_2 = \tilde{a}_1$. By $(*)$ or $(**)$ we have $a_1 <_L a_2$. For a_2 there is a similar choice of a_2', $\tilde{a}_2$, and a_3 with $a_2 <_L a_3$. Etc.

Iterate this procedure until $a_n = a_1$, $n > 1$. The procedure terminates since A is finite and we directly have $a_1 <_L \ldots <_L a_n = a_1$. □

Note that well-formedness is not exactly equivalent to being insatisfiable by structure. In the case that the cyclic dependency occurs somewhere below a cold

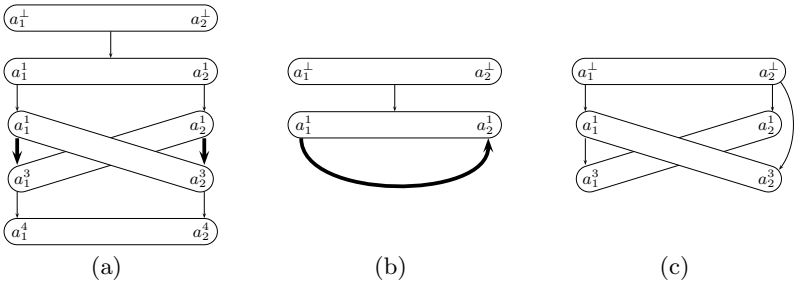


Fig. 5. Precedence Graphs of the three LSC from Fig. 3

cut of the main-chart, that is, not in the pre-chart, then a system that doesn't progress further than this cold cut properly satisfies the LSCs. Instead of refining Def. 9 to separate three cases, the well-formed and thus structurally satisfiable LSCs ("good"), the non well-formed and structurally insatisfiable LSCs ("bad"), and the non well-formed but satisfiable LSCs ("ugly") we note that for each "ugly" LSC there is a transformation to an equivalent "good" one.

4 Deciding Well-Formedness of LSCs

One result of the previous section was that well-formedness of LSCs can be checked on-the-fly when constructing its Symbolic Automaton. But all authors of LSCs need to know whether their specification is well-formed, not only the ones heading for formal verification where the Symbolic Automaton is constructed anyway. An independent mechanical check is evident from Definition 8: build up the precedence relation and check it for acyclicity. But the precedence relation is not the minimal data-structure for this purpose. Thus in the following we provide a faster well-formedness check based on the *precedence graph* of the LSC.

Definition 10 (Precedence Graph). *The* precedence graph *of a core LSC L is the directed graph $(Simclass(L), \hookrightarrow)$ with*

$$\hookrightarrow := \{(scl_1, scl_2) \in Simclass(L)^2 \mid \exists\, a_1 \in scl_1, a_2 \in scl_2 : \\ a_1 \prec a_2 \vee\ \exists\, m \in Msg_{asyn}(L) : a_s(m) \in scl_1 \wedge a_r(m) \in scl_2\}. \qquad \diamondsuit$$

The precedence graph structure reflects the order of simclasses along the instance lines and additionally tracks the dependency between the sending and reception of an asynchronous message. Fig. 5 shows the precedence graphs of the three LSCs from Fig. 3. Note that the two non well-formed LSCs exhibit cycles in Fig. 5(a) and 5(b), indicated by bold arrows.

Lemma 4. *A core LSC L is well-formed if its precedence graph is acyclic.* $\diamondsuit$

Proof. We show the equivalent claim that a cycle in the precedence relation of L implies a cycle in the precedence graph. Let $a \in atoms(L)$ with $a <_L a$, i.e. there are atoms $a_1, \ldots, a_n \in atoms(L)$ s.t.

$$a \in [a_1] \wedge a_1 <_L a_2 \wedge a_2 \in [a_3] \wedge \ldots \wedge a_{n-1} <_L a_n \wedge a \in [a_n]$$

where $a_i <_L a_{i+1}$ by Def. 7.1 or 7.2. Both cases entail $[a_i] \hookrightarrow^+ [a_{i+1}]$ by construction of the precedence graph. As $a_i \in [a_{i+1}]$ implies $[a_i] = [a_{i+1}]$ we can trace the complete precedence relation chain by the precedence graph relation and thus obtain $[a] \hookrightarrow^+ [a]$. □

Acyclicality of the graph of an LSC L is checked in $O(|Simclass(L)| + |\hookrightarrow|)$. The size of the graph relation $|\hookrightarrow|$ is bounded from above by $|\prec| + |Msg_{asyn}(L)|$. The number of edges induced by the scenario order is $\sum_{a \in \mathbb{A}_L} |prereq(a)|$. In the special case that L has no coregions, $prereq(a)$ is at most 1 thus $|\prec| \leq |\mathbb{A}_L|$. In the special case of only non-consecutive coregions we have $|\prec| \leq |\mathbb{A}_L| + |\mathbb{C}_L|$, where $\mathbb{C}_L \subseteq \mathbb{A}_L$ denotes the atoms of L that lie in a coregion. Only consecutive coregions produce a disproportional grow of $|\hookrightarrow|$ as then all possible combinations between the two coregions are reflected by $\prec$.

5 (Impact on) Related Work

Research into the LSC language is often restricted to the subset of well-formed LSCs, e.g. by abstractly representing an LSC in terms of a labelled partial order. The following subsection 5.1 discusses the impact of this restriction and extends existing results of well-formed LSCs to the set of all LSCs. Subsection 5.2 identifies all LSCs produced by the PlayEngine as being well-formed by construction.

5.1 LSCs as LPOs

As already mentioned in the previous sections, recent analysis of the complexity of Life Sequence Charts [3] uses as abstract syntax a labelled partial order, i.e. a tuple $(L, \leq, \lambda, A)$ where L is a finite set of events, $\leq \subseteq L \times L$ is a partial order on L, and $\lambda : L \to A$ is a labelling function, that is built as follows[1]:

Definition 11 (Order Relation of an LSC [3]). *Let L be a core LSC. Two simclasses $scl_1, scl_2 \in Simclass(L)$ are* directly ordered*, denoted $scl_1 <_d scl_2$, if*

1. $\exists\, a_1 \in scl_1, a_2 \in scl_2 : a_1 \prec a_2$, *or*
2. $\exists\, m \in Msg_{asyn}(L) : a_s(m) \in scl_1 \wedge a_r(m) \in scl_2$

The relation $\leq$ is the reflexive, transitive closure of $<_d$. ♢

The direct order between simclasses of Def. 11 corresponds to our precedence graph relation from Def. 10. Thus strictly speaking, the results of [3] only apply to the set of well-formed LSCs, not to the set of all LSCs since for non well-formed ones '$\leq$' is not a partial order by Sect. 3.

[1] Note that [3] consider neither simultaneous classes nor asynchronous messages. We extend their construction in the canonical way.

In the other direction we can conclude from Sect. 3 that for each well-formed LSC '$\leq$' is a partial order and thus [3] indeed applies to all practically relevant LSCs. To extend the complexity results to the set of all LSCs, including the non well-formed, we have to take the complexity of checking for well-formedness into account. Section 4 entails that the (in the best case polynomial) complexity classes identified by [3] are not left by checking for well-formedness beforehand.

5.2 Play-Engine LSCs

The PlayIn/PlayOut [9] approach employs Life Sequence Charts for specifying and executing behavioural requirements of reactive systems. LSCs are specified by "playing them in" on a prototypical GUI of the system, i.e. all interactions between the user and the GUI are recorded as LSCs. After a set of scenarios have been played in, the engine is also able to "play them out", i.e. when the GUI is operated the play-engine reacts according to the recorded specification.

Obviously, the user is not able to play in any contradictory LSC by using the GUI interface. Consequently, the results of Sect. 3 entail that every played-in LSC is well-formed by construction.

But in general it is not an option to exclusively use "played-in LSCs" in all application domains. For LSCs that are meant as requirement specification for an existing implementation, it is usually not appropriate to record every desired system run, but one rather wants to specify whole classes of scenarios at once. This is already supported by the play-engine in form of (very limited) editing capabilities. Also, the requirement that certain events happen simultaneously cannot be played in in a convenient manner.

6 Conclusion

The need to turn the abstract definition of well-formedness into something more imaginable to authors of LSC specification has already been identified by [11] and approached by an informal and incomplete list of guidelines. Our two alternative characterisations of well-formedness, in terms of concrete syntax and semantical in terms of the underlying Symbolic Automaton, provide for a better understanding of the relation between the set of syntactically correct LSC diagrams and the practically useful ones. Judging from the characterisations we can conclude that it is reasonable to restrict the discussion of LSCs to the ones that are well-formed in the sense of [8].

Further work comprises the sketched extension of the well-formedness notions and analyses to the whole LSC language of [11], in particular possible messages and timer-set/-reset and timeout elements. For LSCs with timing requirements it is also desirable to have fast syntactical sanity checks because the general case lies in the class of the timer-consistency problem of timed automata.

References

1. D. Amyot and A. Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunications Systems Journal*, 24(1):61–94, September 2003.
2. J. Bohn, W. Damm, H. Wittke, J. Klose, and A. Moik. Modelling and validating train system applications using Statemate and Live Sequence Charts. In *Proc. IDPT 2002*. Society for Design and Process Science, June 2002.
3. Y. Bontemps. *Relating Inter-Agent and Intra-Agent Specifications*. PhD thesis, University of Namur (Belgium), April 2005.
4. Y. Bontemps, P. Heymans, and H. Kugler. Applying LSCs to the specification of an air traffic control system. In *Proc. SCESM'03*, 2003.
5. Y. Bontemps and P.-Y. Schobbens. The complexity of Live Sequence Charts. In V. Sassone, editor, *Proc. FOSSACS 2005*, volume 3441 of *LNCS*, 2005.
6. A. Bunker, G. Gopalakrishnan, and K. Slink. Live Sequence Charts applied to hardware requirements specification and verification: A VCI bus interface model. *Software Tools for Technology Transfer*, 7(4):341–350, August 2004.
7. P. Combes, D. Harel, and H. Kugler. Modeling and verification of a telecommunication application using Live Sequence Charts and the play-engine tool. In *Proc. ATVA 2005*, number 3707 in LNCS, 2005.
8. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, July 2001.
9. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
10. ITU-T. *ITU-T Rec. Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, 1999.
11. J. Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior*. PhD thesis, C. v. O. Universität Oldenburg, 2003.
12. C. Knieke, M. Huhn, and U. Goltz. Modelling and simulation of an automotive system using lscs. In S. H. Houmb, J. Jürjens, and R. France, editors, *Proc. CSDUML'2005*. TUM, September 2005.
13. H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal logic for scenario-based specifications. In N. Halbwachs and L. D. Zuck, editors, *Proceedings TACAS 2005*, volume 3440 of *LNCS*. Springer-Verlag, 2005.
14. I. Schinz, T. Toben, C. Mrugalla, and B. Westphal. The Rhapsody UML Verification Environment. In J. R. Cuellar and Z. Liu, editors, *Proc. SEFM 2004*, pages 174–183, September 2004.
15. T. Toben and B. Westphal. On the expressive power of Live Sequence Charts. In *Proceedings of the SofSem 2006 Poster Session*. Matfyz Press, 2006. To appear.
16. K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenarios in system development: Current practice. *IEEE Software*, 15(2):34–45, March 1998.