Some Results on the Expressive Power and Complexity of LSCs *

David Harel**, Shahar Maoz, and Itai Segall

The Weizmann Institute of Science, Rehovot, Israel
{dharel,shahar.maoz,itai.segall}@weizmann.ac.il

This paper is dedicated to Prof. Boaz Trakhtenbrot,
with deep admiration and respect.

Abstract. We survey some of the main results regarding the complexity and expressive power of Live Sequence Charts (LSCs). We first describe the two main semantics given to LSCs: a trace-based semantics and an operational semantics. The expressive power of the language is then examined by describing translations into various temporal logics. Some limitations of the language are also discussed. Finally, we survey complexity results, mainly due to Bontemps and Schobbens, regarding the use of LSCs for model checking, execution, and synthesis.

1 Introduction

Live Sequence Charts (LSCs, or LSC for the language) [9] constitute a visual formalism for inter-object scenario-based specification and programming. The language extends classical Message Sequence Charts (MSC) [21], mainly by adding universal and existential modalities. LSC distinguishes between behaviors that may happen in the system (existential, cold) and those that must happen (universal, hot). A universal chart contains a *prechart*, which specifies the scenario which, if successfully executed, forces the system to satisfy the scenario given in the actual chart body.

An executable (operational) semantics for LSC was defined in [18]. Thus, LSC can be viewed not only as a specification language but also as a high-level programming language for reactive systems.

Since its original definition, the language has been the subject of much work, e.g., in the contexts of scenario-based testing [25, 26], synthesis [3, 13, 15], execution (play-out) [18], formal verification [22, 33], specification and verification of hardware [6], telecommunication systems [8], biological systems [11], specification mining [27], and compilation into aspects [12, 29]. Also, recently, in [16], a

* The research was supported in part by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science and by a Grant from the G.I.F., the German-Israeli Foundation for Scientific Research and Development.

** Part of this author's work carried out during a visit to the School of Informatics at the University of Edinburgh, which was supported by a grant from the EPSRC.

UML2 compliant and slightly generalized variant of LSC was defined, allowing the embedding of LSC into the UML standard [35].

In this paper we survey some results regarding the expressive power and succinctness of the language, as well as complexity results for various problems related to using LSC for specification and programming.

2 Language Overview

The LSC language was originally defined by Damm and Harel in [9]. The language has two types of charts: *universal* (annotated by a solid borderline) and *existential* (annotated by a dashed borderline). Universal charts are used to specify restrictions over all possible system runs. A universal chart typically contains a *prechart*, that specifies the scenario which, if successfully executed, forces the system to satisfy the scenario given in the actual chart body. Existential charts specify sample interactions between the system and its environment, and must be satisfied by at least one system run. They thus do not force the application to behave in a certain way in all cases, but rather state that there is at least one set of circumstances under which a certain behavior occurs. Existential charts can be used to specify system tests, or simply to illustrate longer (non-restricting) scenarios that provide a broader picture of the behavioral possibilities to which the system gives rise.

Most constructs in the language, e.g., messages and conditions, also have a hot/cold modality. Hot behaviors are mandatory and must be satisfied by any system run. Cold behaviors, on the other hand, are provisional, and may be satisfied. For example, a hot message must eventually be sent, while a cold message may or may not be sent.

An example of a universal LSC is given in Figure 1. The chart in the example is adopted from [24], and is part of a specification for a cellular phone. The chart requires that whenever the user closes the `Cover`, the `Chip` will send the message `StartRing(Silent)` to the `Speaker` and later the speaker will turn silent as designated by the self message `Sound(Silent)`. The `Display` will set its state to `Time` and later set its background to `Green`. An LSC induces a partial order that is determined by the order along an instance line, by the fact that a message can be received only after it is sent, and by taking into account that a synchronous message blocks the sender until receipt. Thus in Figure 1, the message `ChangeBackground(Green)` must occur after message `SetState(Time)`, but both are unordered with respect to the messages `StartRing(Silent)` and `Sound(Silent)`.

An example of an existential LSC is given in Figure 2. The chart states that there is a possible run of the system where the user presses `Click` on the `Send Key` and eventually the `Chip` receives an `ACK` from the environment `ENV`. The `SYNC` condition restricts the order between the two messages, which are otherwise unordered.

We give here a restricted and simplified trace-based semantics for a kernel subset of LSC. The original LSC semantics was given in [9]. In subsequent work

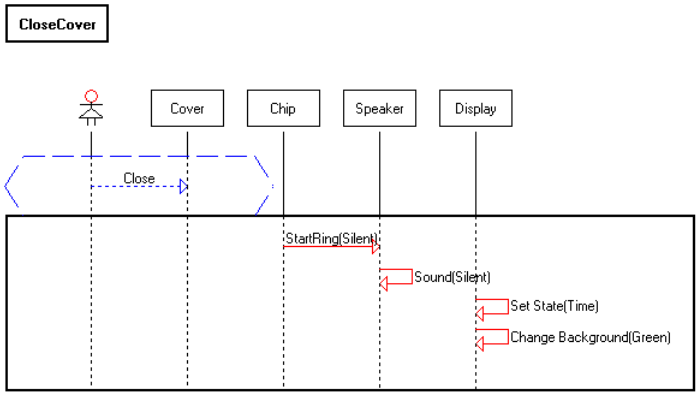

Fig. 1. An example of a universal chart

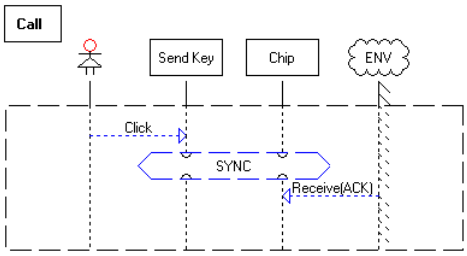

Fig. 2. An example of an existential chart

the semantics of (restricted subsets or extensions of) the language was given using temporal logics (see, e.g., [24]) or various types of automata (see, e.g., [16, 23]). An operational semantics, explicated in the play-out algorithm, was given in [18].

2.1 Basic Definitions

The following definitions are adopted from [24]. We assume the LSC specification relates to an *object system* composed of a set of objects $\mathcal{O} = \{O_1 \dots O_n\}$. An object system corresponds to an implementation, and our goal in providing semantics for LSCs is to define when a given object system satisfies an LSC specification. The instance identifiers in the LSC charts refer to objects from $\mathcal{O}$, and possibly also the environment, denoted *env*. The LSC specifies the behavior of the system in terms of the message communication between the objects in the system. We want to define the notion of satisfiability of an LSC specification. In other words, we want to capture the languages $\mathcal{L} \subseteq A^* \cup A^\omega$ generated by the object systems that satisfy the LSC specification. The alphabet A used de-

fines message communication between objects, $A = \mathcal{O} \times (\mathcal{O}.\Sigma)$, where Σ is the alphabet of messages.

An LSC chart is constructed from a set of instances, a set of locations in those instances, a set of messages, and a mapping from messages to locations. Each chart also has an activation mode, either universal or existential. Similarly, each message has a *temp* function, defining its temperature, as either hot or cold. For now, a chart is assumed to have a single message acting as the activation message. Later on this notion will be extended to a full prechart.

Let $inst(m)$ be the set of all instance-identifiers referred to in chart m. With each instance i we associate a finite number of locations $dom(m, i) \subseteq \{0, \ldots, l\_max(i)\}$. We collect all locations of m in the set

$$dom\,(m) = \{\langle i, l\rangle \mid i \in inst(m) \wedge l \in dom(m, i)\}.$$

The messages appearing in m are triples

$$Messages(m) = dom(m) \times \Sigma \times dom(m),$$

where $(\langle i, l\rangle, \sigma, \langle i', l'\rangle)$ corresponds to instance i, while at location l, sending σ to instance i' at location l'. Each location can appear in at most one message in the chart. The relationship between locations and messages is given by the mapping

$$msg(m) : dom(m) \rightarrow Messages(m)$$

The msg function induces two Boolean predicates *send* and *receive*. The predicate *send* is true only for locations that correspond to the sending of a message, while the predicate *receive* is true only for locations that correspond to the receiving of a message. We define the binary relation $R(m)$ on $dom(m)$ to be the smallest relation satisfying the following axioms and closed under transitivity and reflexivity:

- order along an instance line:

$$\forall \langle i, l\rangle \in dom(m), l < l\_max(i) \Rightarrow \langle i, l\rangle R(m) \langle i, l + 1\rangle$$

- order induced from message sending:

$$\forall msg \in Messages(m), msg = (\langle i, l\rangle, \sigma, \langle i', l'\rangle) \Rightarrow$$

$$\langle i, l\rangle R(m) \langle i', l'\rangle$$

- messages are synchronous; they block the sender until receipt:

$$\forall msg \in Messages(m), msg = (\langle i, l\rangle, \sigma, \langle i', l'\rangle) \Rightarrow$$

$$\langle i', l'\rangle R(m) \langle i, l + 1\rangle$$

We say that the chart m is *well-formed* if the relation $R(m)$ is acyclic. We assume all charts to be well-formed, and use $\leq_m$ to denote the partial order $R(m)$.

We denote the *preset* of a location $\langle i, l\rangle$ containing all elements in the domain of a chart smaller than $\langle i, l\rangle$ by

$$^\bullet\langle i, l\rangle = \{\langle i', l'\rangle \in dom(m) | \langle i', l'\rangle \leq_m \langle i, l\rangle\}.$$

We denote the partial order induced by the order along an instance line by $\prec_m$; thus $\langle i, l\rangle \prec_m \langle i', l'\rangle$ iff $i = i'$ and $l < l'$.

A *cut* through m is a set c of locations, one for each instance, such that for every location $\langle i, l\rangle$ in c, the preset $^\bullet\langle i, l\rangle$ does not contain a location $\langle i', l'\rangle$ such that $\langle j, l_j\rangle \prec_m \langle i', l'\rangle$ for some location $\langle j, l_j\rangle$ in c. A cut c is specified by the locations in all of the instances in the chart:

$$c = (\langle i_1, l_1\rangle, \langle i_2, l_2\rangle, ..., \langle i_n, l_n\rangle)$$

For a chart m with instances $i_1, ..., i_n$ the *initial cut* c_0 has location 0 in all the instances. Thus, $c_0 = (\langle i_1, 0\rangle, \langle i_2, 0\rangle, ..., \langle i_n, 0\rangle)$. We denote $cuts(m)$ the set of all cuts through the chart m.

For chart m, some $1 \leq j \leq n$ and cuts c, c', with

$$c = (\langle i_1, l_1\rangle, \langle i_2, l_2\rangle, ..., \langle i_n, l_n\rangle), c' = (\langle i_1, l'_1\rangle, \langle i_2, l'_2\rangle, ..., \langle i_n, l'_n\rangle)$$

we say that c' is a $\langle j, l_j\rangle$-*successor* of c, and write $succ_m(c, \langle j, l_j\rangle, c')$, if c and c' are both cuts and

$$l'_j = l_j + 1 \ \wedge \ \forall i \neq j, l'_i = l_i$$

Notice that the successor definition requires that both c and c' are cuts, so that advancing the location of one of the instances in c is allowed only if the obtained set of locations remains unordered.

A *run* of m is a sequence of cuts, $c_0, c_1, \ldots, c_k$, satisfying the following:

- c_0 is an initial cut.
- for all $0 \leq i < k$, there is $1 \leq j_i \leq n$, such that $succ_m(c_i, \langle j_i, l_{j_i}\rangle, c_{i+1})$.
- in the final cut c_k all locations are maximal.

$Runs(m)$ is the set of all runs of m.

Assume the natural mapping f between $(dom(m) \cup env) \times \Sigma \times dom(m)$ to the alphabet A, defined by

$$f(\langle i, l\rangle, \sigma, \langle j, l'\rangle) = (O_i, O_j.\sigma)$$

Intuitively, the function f maps a location to the sending object and to the message of the receiving object. With this notation in mind, $f(Messages(m))$ will be used to denote the letters in A corresponding to messages that are restricted by chart m:

$$f(Messages(m)) = \{f(v) \mid v \in Messages(m)\}$$

Definition 1. *Let $c = c_0, c_1, ..., c_k$ be a run. The* execution trace*, or simply the* trace *of c, written $w = trace(c)$, is the word $w = w_1 \cdot w_2 \cdots w_k$ over the alphabet A, defined by:*

$$w_i = \begin{cases} f(msg(m)(\langle j, l_j \rangle)) & \text{if } succ_m(c_{i-1}, \langle j, l_j \rangle, c_i) \ \wedge \ send(\langle j, l_j \rangle) \\ \epsilon & \text{otherwise} \end{cases}$$

We define the trace language *generated by chart m, $\mathcal{L}_m^{trc} \subseteq A^*$, to be*

$$\mathcal{L}_m^{trc} = \{w \mid \exists (c_0, c_1, ..., c_k) \in \text{Runs(m) s.t. } w = trace(c_0, c_1, ..., c_k)\}$$

There are two additional notions that we associate with an LSC, its *mode* and its *activation message*. These are defined as follows:

$$mod : m \rightarrow \{existential, universal\}$$

$$amsg : m \rightarrow dom(m) \times \Sigma \times dom(m)$$

The activation message of a chart designates when a scenario described by the chart should start, as we describe below. The charts and the two additional notions are now put together to form a specification. An *LSC specification* is a triple

$$LS\langle M, amsg, mod \rangle,$$

where M is a set of charts, and $amsg$ and mod are the activation messages and modes of the charts, respectively.

The *language* of the chart m, denoted by $\mathcal{L}_m \subseteq A^* \cup A^\omega$, is defined as follows:

For an existential chart, $mod(m) = existential$, we require that the activation message is relevant (i.e., sent) at least once, and that the trace will then satisfy the chart:

$$\begin{aligned} \mathcal{L}_m = \big\{ & w = w_1 \cdot w_2 \cdots \mid \exists i_0, i_1, ..., i_k \text{ and } \exists v = v_1 \cdot v_2 \cdots v_k \in \mathcal{L}_m^{trc}, \text{ s.t.} \\ & (i_0 < i_1 < ... < i_k) \wedge (w_{i_0} = f(amsg(m))) \wedge \\ & (\forall j, 1 \leq j \leq k, w_{i_j} = v_j) \wedge \\ & (\forall j', i_0 \leq j' \leq i_k, j' \notin \{i_0, i_1, ..., i_k\} \Rightarrow w_{j'} \notin f(Messages(m)))\} \end{aligned}$$

The formula requires that the activation message is sent once ($w_{i_0} = f(amsg(m))$), and then the trace satisfies the chart; i.e., there is a subsequence belonging to the trace language of chart m ($v = v_1 \cdot v_2 \cdots v_k = w_{i_1} \cdot w_{i_2} \cdots w_{i_k} \in \mathcal{L}_m^{trc}$), and all the messages between the activation message until the end of the satisfying subsequence ($\forall j', i_0 \leq j' \leq i_k$) that do not belong to the subsequence ($j' \notin \{i_0, i_1, ..., i_k\}$) are not restricted by the chart m ($w_{j'} \notin f(Messages(m))$).

For a universal chart, $mod(m) = universal$, we require that each time the activation message is sent the trace will satisfy the chart:

$$\begin{aligned} \mathcal{L}_m = \{ & w = w_1 \cdot w_2 \cdots \mid \forall i, w_i = f(amsg(m)) \Rightarrow \exists i_1, i_2, ..., i_k \text{ and} \\ & \exists v = v_1 \cdot v_2 \cdots v_k \in \mathcal{L}_m^{trc}, \text{ s.t. } (i < i_1 < i_2 < ... < i_k) \wedge \\ & (\forall j, 1 \leq j \leq k, w_{i_j} = v_j) \wedge \\ & (\forall j', i \leq j' \leq i_k, j' \notin \{i_1, ..., i_k\} \Rightarrow w_{j'} \notin f(Messages(m)))\} \end{aligned}$$

The formula requires that after each time the activation message is sent ($\forall i, w_i = f(amsg(m))$), the trace will satisfy the chart m (this is expressed in the formula in a similar way to the case for an existential chart.)

Now come the main definitions, which finalize the semantics of the language by connecting it with an object system:

Definition 2. *A system* S satisfies *the LSC specification* $LS = \langle M, amsg, mod \rangle$, *written* $S \models LS$, *if:*

1. $\forall m \in M, \quad mod(m) = universal \Rightarrow \ \mathcal{L}_S \subseteq \mathcal{L}_m$
2. $\forall m \in M, \quad mod(m) = existential \Rightarrow \ \mathcal{L}_S \cap \mathcal{L}_m \neq \emptyset$

In this short introduction, we assumed that a chart has an activation message. The extension of this notion to a prechart is omitted here. Informally, a chart containing a prechart must be satisfied whenever its prechart is satisfied. We also assumed all messages are hot, therefore all cuts must progress. However, when introducing cold messages, a cut containing only cold messages may progress, but need not.

The kernel language of LSC, introduced in [9], contains several constructs, in addition to the messages formally introduced above. These include:

- *Conditions*, which act as requirements on the state of the system at a given point in time. Like messages, conditions too can have a hot/cold modality, defining the effect of a false condition. A false hot condition is a violation of the requirements, whereas a false cold condition merely induces an immediate normal exit from the chart (or enclosing subchart).
- *Subcharts* are the main structuring mechanism in the LSC language. A subchart is a well-formed fragment of a chart. Along with conditions, it can also be used to define branching constructs like if-then-else.
- *Variables*, whose scope is local to an LSC. One can use assignments to assign values to variables. Expressions within conditions may include variables.

2.2 Different Variants and Additional Constructs

The above definitions constitute a kernel subset of the LSC language. A number of variants and extensions have been suggested and used in different kinds of work and in different contexts. We list some of these variants and extensions below.

The first variation to be discussed refers to the question of how often a universal LSC should be activated. The most general case is that of an *invariant* LSC, which calls for the LSC to be activated whenever the prechart is completed, regardless of the state of the system. This means that multiple copies of the same chart may be active simultaneously, if the prechart is completed several times. Two restrictions to this mode are *initial* and *iterative* (see, for example, in [5]). 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 iterative mode

allows only one incarnation of the chart at a time, i.e., as long as a chart is active its prechart is not monitored for further satisfactions.

Another variant, suggested in [18], is that of *strict* vs. *tolerant* (or *weak*) semantics. A strict LSC restricts the occurrence of the messages used in the LSC to exactly those points in time where they are supposed to occur according to the scenario. Any message appearing out-of-order in a strict LSC is considered a violation. In the weak interpretation, the specification is satisfied if each necessary message occurs at least once where it is supposed to, and additional occurrences of it are ignored.

A variety of extensions have also been suggested to the kernel subset of the language. We now list some of them.

- *Symbolic messages* were introduced in [30]. In a symbolic message, the arguments passed by the message are symbolic, thus a single message in an LSC can stand for several different instantiations of it in the system. The actual arguments used in a specific run can be stored in LSC-local variables, so that they can be used again in the same chart. See also Chapter 7 in [18].
- In a real system, multiple objects can be instances of the same class. A *symbolic instance* in an LSC represents an entire class, or rather, any instance of the class, instead of a single concrete object. Symbolic instances were first suggested in [30], and are also covered in Chapter 15 of [18].
- A *co-region* is a sequence of locations belonging to the same instance, in which the partial order requirement is relaxed, i.e., locations within a co-region may appear in any order.
- *Forbidden messages and conditions* were introduced in [18], allowing one to state behaviors that are forbidden while an LSC (or a part of it) is active. Similarly, one may add restrictions on message sending, besides the ones derived from the LSC's partial order, using a *restricts* clause.
- *Timing constraints* on LSCs are considered in [23] and [17]. In [23], LSCs can be annotated by timers and by delay intervals, thus allowing one to express timing constraints on pairs of events that are either on the same instance line, or are connected by a message. In [17], on the other hand, a single clock object with one property, `Time`, and a single method, `Tick`, are introduced. This, together with the rich LSC language, suffices for specifying a wide variety of timing constraints (see Chapter 16 of [18]).

2.3 Scenario-Based Execution

The semantics described so far is a trace-based semantics, defining when a trace of events is in the language of the LSC specification. However, in [18, 19], the play-out approach is presented. In this approach, the LSC specification can be directly executed, without any intermediate steps. Play-out is implemented in the *Play-Engine* tool. The play-out process calls for the Play-Engine to continuously monitor the applicable precharts of all universal charts, and whenever successfully completed, to execute their bodies. A full operational semantics is supplied in Appendix A of [18], defining how an LSC specification can be executed. We quote some of the main definitions from there.

The operational semantics is given as a transition system

$$\mathcal{S}em(\mathcal{S}) = \langle \mathcal{V}, V_0, \mathcal{S}_D, \mathcal{S}_M, \Delta[\mathcal{S}_O, \mathcal{S}_C] \rangle$$

where $\mathcal{V}$ is the set of possible configurations (states) of $\mathcal{S}em(\mathcal{S})$, V_0 is the initial configuration, $\mathcal{S}_D \subseteq \mathcal{S}_\mathcal{U}$ is the set of driving LSCs, $\mathcal{S}_M \subseteq \mathcal{S}_\mathcal{U} \cup \mathcal{S}_E$ is the set of monitored LSCs, and $\Delta \subseteq \mathcal{V} \times (\mathcal{E} \cup \bigcup_{L \in \mathcal{S}} E_L) \times \mathcal{V}$ is the set of allowed transitions. We require that $\mathcal{S}_D \cap \mathcal{S}_M = \emptyset$.

A state $V \in \mathcal{V}$ is defined as

$$V = \langle \mathcal{RL}, \mathcal{ML}, \mathit{Violating} \rangle$$

where $\mathcal{RL}$ is a set of live copies of 'driving' LSCs, $\mathcal{ML}$ is a set of live copies of monitored LSCs, and *Violating* indicates by $True$ or $False$ whether the state is a violating one.

The initial configuration contains no copies of driving LSCs and no copies of monitored LSCs, and is defined as:

$$V_0 = \langle \emptyset, \emptyset, False \rangle$$

The transition relation Δ is parameterized by two sets. The first, $\mathcal{S}_O$, is the set of original LSCs to which Δ should be applied. The second, $\mathcal{S}_C$, is the set of live copies that currently exist. This set contains only copies of LSCs from $\mathcal{S}_O$. The two sets are instantiated with either $(\mathcal{S}_D, \mathcal{RL})$ or $(\mathcal{S}_M, \mathcal{ML})$.

Δ is described as a set of rules to its set parameters and to *Violating* with respect to a given event e. Since the set parameters are instantiated also by $\mathcal{RL}$ and $\mathcal{ML}$, which are taken from a state V, the result of applying Δ is a new state V' consisting of the modified components. In other words, Δ defines the result of executing an event e in a given system state. We skip the formal definition of Δ. The idea behind its rules is to advance any cut that needs to be advanced by executing e, to open new live copies of charts for which the prechart has become relevant, and to update *Violating* to state whether there has been a violation.

The same definitions are used in [18] both for describing how a specification can be used for testing (quite similarly to the trace-based semantics described above), and for actual execution. The execution mechanism works in phases of *step* and *super-step*. The input to a step is a system event e. The procedure for a step phase consists of applying the transition relation onto the event e and, if the event represents a property change, changing the state of the object model according to the new value in the message.

In the super-step phase, the Play-Engine continuously executes the steps associated with internal events — i.e., those that do not originate with the user, the environment, the *Clock* or external objects — until it reaches a 'stable' state where no further such events can be carried out.

The execution algorithm proposed in [18] is naïve, in the sense that when facing multiple choices for a step, none of them causing an immediate violation, it chooses one arbitrarily. Its choice might lead to a contradiction in the future, while perhaps there could have been a different choice that would have avoided

it. This problem is addressed by the smart play-out algorithm proposed in [14], in which a legal super-step is found using a model checker. The specification is translated into a model, and the model checker is fed with this model along with the claim that no legal super-step exists. If one does exist, it will be given as a counter-example to the claim. In [20] the problem is translated into an AI planning problem, and an extended planner is used in order to find all legal supersteps from a given system state, up to a predefined length.

The operational semantics given above expresses the same ideas as the trace-based semantics of section 2.1, but in a manner more suitable for execution. The operational semantics somewhat restricts the trace-based semantics to those cases that are interesting in the context of execution. In a sense, all "interesting" traces can be generated by the operational semantics. When equipped with the smart play-out approach, it is also sound, in the sense that every supserstep generated by it is also a legal trace in the trace-based semantics.

The Play-Engine [18] is an interpreter based execution engine for an LSC specification. The specification is executed directly, with no intermediate code being generated. An implementation of play-out by compilation into aspects was suggested in [29] and is implemented in a compiler called S2A [12]. This work is defined for the slightly generalized and UML2-compliant variant of LSC given in [16], in which, unlike the version supported by the Play-Engine where precharts are monitored and main-charts are executed, the hot/cold modality is orthogonal to a new monitor/execute modality.

3 Expressive Power

The expressive power of LSC was studied in [3, 10, 13, 24] by suggesting translations from fragments of the language into various Temporal Logics.

A first embedding of a kernel subset of the language (which omits variables, for example) into CTL* was given in [13]. For this kernel subset the embedding is a strict inclusion, since given the single level quantification mechanism of LSCs, the language cannot express general formulas with alternating path quantifiers.[1]

This embedding was improved in [24] to support a wider subset of the language and in a more efficient way. Specifically, it was shown that existential charts can be expressed using the branching temporal logic CTL, while universal charts are in the intersection of linear temporal logic and branching temporal logic LTL $\cap$ CTL. Below we give the basic and then the improved explicit translations from [24].

Definition 3 ([24]). *Let $w = m_1 m_2 m_3 \dots m_k$ be a finite trace. Let $R = \{e_1, e_2, e_3 \cdots e_l\}$ be a set of events. The temporal logic formula ϕ_w^R is defined as:*

$$\phi_w^R = NU\left(m_1 \wedge \left(X\left(NU\left(m_2 \wedge \left(X(NU(m_3 \dots))\right)\right)\right)\right)\right),$$

[1] It shouldn't be too difficult to extend LSCs to allow certain kinds of quantifier alternation, as noted in [9]. However, as in [9], this was not done there either, since it was judged to have been too complex and unnecessary for real world usage of sequence charts.

where the formula N is given by $N = \neg e_1 \wedge \neg e_2 \ldots \wedge \neg e_l$.

Definition 4 ([24]). *Let $LS = \langle M, amsg, mod \rangle$ be an LSC specification. For a chart $m \in M$, we define the formula ψ_m as follows:*

- *If $mod(m) = universal$, then $\psi_m = AG\left(amsg(m) \rightarrow X\left(\bigvee_{w \in \mathcal{L}_m^{trc}} \phi_w^R\right)\right)$.*
- *If $mod(m) = existential$, then $\psi_m = EF\left(\bigvee_{w \in \mathcal{L}_m^{trc}} \phi_w^R\right)$.*

(for a universal chart m, R includes the events appearing in the prechart and in the main chart.)

In the above, the formula for a universal chart is in LTL. However, it can be large, due to the possibility of having many different traces for the chart, which affects the number of clauses in the disjunction, and also due to the similarity of clauses at the different sides of the implication operator. In the improved translation given below, the resulting temporal logic formulas are much more succinct, i.e., polynomial vs. exponential in the number of locations.

We consider the case where both the prechart and the main chart consist only of message communication, and denote by $p_1, \cdots p_k$ the events appearing in the prechart, and by $m_1, \cdots m_l$ the events appearing in the main chart. Denote by e_i any of these events, either in the prechart or in the main chart. We write $e_i \prec e_j$ if e_i precedes e_j in the partial order induced by the chart, and $e_i \not\prec e_j$ if e_i and e_j are unordered.

Definition 5 ([24]).

$$\psi_m = G\left(\left(\bigwedge_{p_i \prec p_j} \phi_{p_i,p_j} \wedge \bigwedge_{\forall p_i, m_j} \phi_{p_i,m_j} \wedge \bigwedge_{p_i \not\prec p_j} \neg\chi_{p_j,p_i}\right) \rightarrow \left(\bigwedge_{m_i \prec m_j} \phi_{m_i,m_j} \wedge \bigwedge_{m_j \ is \ maximal} Fm_j \wedge \bigwedge_{\forall e_i, m_j} \neg\chi_{e_i,m_j}\right)\right)$$

$\phi_{x_i,x_j} = \neg x_j U x_i$
$\chi_{x_i,x_j} = (\neg x_i \wedge \neg x_j) U (x_i \wedge X((\neg x_i \wedge \neg x_j) U x_i))$

Here the formula ϕ_{x_i,x_j} specifies that x_j must not happen before x_i, which eventually occurs. The formula $\neg\chi_{x_i,x_j}$ specifies that x_i must not occur twice before x_j occurs.

Note that this translation is polynomial in the number of messages appearing in the chart, while the translation in Definition 4 may be exponential in that number. However, the above translation assumes that a message does not appear more than once in the same chart. Whether an efficient translation exists for the most general case is left open in [24]. A construction given in [3] provides a polynomial translation for the more general case of deterministic LSCs, i.e., where a message may occur more than once in a chart but all appearances of the same message are ordered.

Using a characterization by Maidl for the common fragment of LTL and CTL [28] and a theorem by Clarke and Draghicesku [7], it is shown in [24] that the formulas given in Definition 5 have equivalent CTL formulas. Finally, [24] considers also the extension of the above to support conditions and bounded iterations. An explicit translation that supports these, however, is left in [24] for future work.

A different translation of LSC into TL, which supports variables but considers activation only by *activation condition* and not the general case of precharts, was given by Damm, Toben, and Westphal in [10]. To support variables, the work defines a translation of LSC into a fragment of first-order CTL*. Specifically, a translation is defined from bounded LSCs (i.e., where conditions and local invariants only appear in simultaneous regions with messages) into *(deterministic) communication sequence first-order prenex CTL** (DCSCTL), a syntactically characterized fragment of CTL*. The translation is shown to be tight, i.e., a translation back from DCSCTL into LSC is constructively defined, thus establishing an equivalence.

Restricted to messages, the two pieces of work surveyed above [10, 24] coincide. They consider different subsets of the LSC language. Neither of them handles explicit time.

3.1 Limitations

As mentioned above, given the single level quantification mechanism of LSCs, the language cannot express general formulas with alternating path quantifiers. However, as shown in [10], the embedding of LSC into CTL* is strict even without resorting to the nesting of path quantifiers. The question of whether adding constructs not included in the above work (e.g., bounded iterations, specifically within precharts) will make LSCs equivalent in expressive power to LTL remains open in [10]. A similar result is provided in [1] where it is shown that the language $\Sigma^* aa \Sigma^\omega$ that is expressible using a deterministic Büchi automaton (DBA) and by an LTL formula ($F(a \wedge Xa)$) is not expressible in LSC.

4 Complexity Results

In this section we survey the complexity results for the three main applications of LSCs, i.e., model checking, execution (play-out), and synthesis. Essentially, all results mentioned are due to Bontemps and Schobbens in [2] and [3].

4.1 Model Checking

Our first problem is that of model-checking. In this setting, one is given a system implementation (either centralized or distributed) in some formal language, e.g., I/O automata, and an LSC specification, and we want decide whether the system satisfies the specification. The complexity of this problem grows along two axes: centralized vs. distributed systems, and closed vs. open environments (i.e., whether the system is a stand-alone one or interacts with an environment).

Theorem 1 ([2]). *Closed Centralized Model Checking (CCMC) is complete for co-NP.*

Proof. Membership in co-NP is proved by guessing a counter-example, which is a path in the system automaton that violates an LSC.

Hardness is proved by reducing the complement of the traveling salesman problem (CoTSP) (see [31]) to CCMC. Given a weighted graph, an automaton is built such that a tour in the graph corresponds to a set of automata transitions. The automaton is equipped with a counter that sums the weights of the edges in the tour. The fact that all tours have length $\geq k$ is encoded in an LSC. Its prechart is matched when all vertices have occurred exactly once, and the main chart makes sure the value of the counter is $\geq k$. A tour of length $< k$ exists iff the automaton violates the LSC. □

Theorem 2 ([2]). *Open centralized model checking (OCMC), closed distributed model checking (CDMC) and open distributed model checking (ODMC) are all complete for PSPACE.*

Proof. (For CDMC) Membership is proved by building a nondeterministic PSPACE Turing machine deciding on the complement of the distributed model checking problem, and relying on coPSPACE=PSPACE, according to Savitch's theorem [32].

The hardness proof takes a DPSPACE Turing machine and builds a set of automata, A_i, one for each cell tape. Each automaton records the letter in its cell, and whether the tape head is located on it or not. Each transition of the Turing machine is encoded by transitions in the relevant automaton. The LSC states that whenever a run starts it must halt. This causes the system to satisfy the LSC iff the Turing machine halts. □

4.2 Reachability and Smart Play-Out

When considering the complexity of play-out, there are two main problems to be considered, reachability, and smart play-out. In the reachability problem, an LSC specification and a single existential LSC are given, and one wants to decide whether, under the constraints of the former, the latter can be satisfied. In smart play-out, the environment has executed several steps, and the system should find a superstep, i.e., a series of steps that satisfies the specification.

Theorem 3 ([2]). *Reachability is PSPACE-complete.*

Proof. Membership is proved by transforming the LSC specification into an LTL formula, Φ_u, and the claim that the existential formula can not be satisfied into another LTL formula, ϕ_e, and checking whether $\Phi_u \rightarrow \phi_e$ is valid. This solves the complement of the reachability problem. The solution for LTL is in PSPACE according to [34]. Note that membership can also be proved by considering [14], in which the problem is reduced to model-checking, which is known to be in PSPACE.

Hardness is proved by encoding the execution of a DPSPACE Turing machine on the blank input as an LSC specification. The existential LSC calls for the execution to start and to halt. A halting run of the Turing machine exists iff the existential LSC can be satisfied. □

Theorem 4. *Smart play-out is PSPACE-complete.*

Proof. The theorem can be proved by adapting the reachability proof above. In other work, not yet published, the same claim is proved by a reduction from QBF. □

4.3 Synthesis and Consistency

The most complex class of problems considered here is that of synthesis. In this class of problems, we would like to know whether the objects participating in the LSC specification can actually be implemented consistently. This problem is also termed "agent design".

A related problem is that of consistency; i.e., deciding whether the specification has no internal contradictions. A formal definition of a consistent system is given in [13]. Informally, a system is consistent if there exists a non-empty regular language, $\mathcal{L}$, s.t. (1) all universal charts are satisfied by all traces in $\mathcal{L}$; (2) every trace in $\mathcal{L}$ is extendible if a new message is sent from the environment; and (3) each existential chart is satisfied by some trace in $\mathcal{L}$. In [13] it is shown that a system is consistent if and only if it is satisfiable (i.e., can be synthesized).

As in previous sections, two versions of the synthesis problem are considered; a centralized one, in which a single automaton is built, and a distributed one, in which each object has its own automaton.

Theorem 5 ([1]). *Centralized synthesis is EXPTIME-complete.*

Proof. Membership follows from the exponential time algorithms proposed in [4] and [15].

Hardness is proved by encoding an alternating PSPACE Turing machine as an LSC, similar to the construction in Theorem 3, in which existential and universal moves are distinguished. □

An interesting question regarding the centralized synthesis problem deals with the size of the synthesized automaton. This is also answered in [3], where it is shown that there exists a family $(\phi_n)_{n>0}$ of LSC specifications, such that any implementation of ϕ_n requires memory of size $2^{\Omega(n \log n)}$. It is worth noting that this proof uses *co-region* constructs, which relax the ordering of events. A co-region succinctly encodes an exponential number of orderings.

Finally, [3] considers the problem of distributed synthesis, in which each object has to be synthesized separately. The question of whether such a synthesis exists is undecidable. This is proved by reducing Post's correspondence problem to the problem of deciding whether the specification is not distributively implementable.

5 Conclusion

The language of LSC has been a subject of much work. We surveyed here some theoretical results regarding the expressive power of the language and the complexity of some of its main applications.

References

1. Y. Bontemps. *Relating Inter-Agent and Intra-Agent Specifications (The Case of Live Sequence Charts)*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique (University of Namur, Computer Science Dept), April 2005.
2. Y. Bontemps and P.-Y. Schobbens. The Complexity of Live Sequence Charts. In V. Sassone, editor, *FoSSaCS*, volume 3441 of *LNCS*, pages 364–378. Springer, 2005.
3. Y. Bontemps and P.-Y. Schobbens. The Computational Complexity of Scenario-Based Agent Verification and Design. *J. Applied Logic*, 5(2):252–276, 2007.
4. Y. Bontemps, P.-Y. Schobbens, and C. Löding. Synthesis of Open Reactive Systems from Scenario-Based Specifications. *Fundam. Inform.*, 62(2):139–169, 2004.
5. M. Brill, W. Damm, J. Klose, B. Westphal, and H. Wittke. Live Sequence Charts: An Introduction to Lines, Arrows, and Strange Boxes in the Context of Formal Verification. In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, editors, *SoftSpez Final Report*, volume 3147 of *LNCS*, pages 374–399. Springer, 2004.
6. A. Bunker, G. Gopalakrishnan, and K. Slind. 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 2005.
7. E. M. Clarke and I. A. Draghicescu. Expressibility Results for Linear-Time and Branching-Time Logics. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *REX Workshop*, volume 354 of *LNCS*, pages 428–437. Springer, 1988.
8. P. Combes, D. Harel, and H. Kugler. *Modeling and Verification of a Telecommunication Application Using Live Sequence Charts and the Play-Engine Tool*, volume 3707 of *LNCS*, pages 414–428. Springer, 2005.
9. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001. Preliminary version in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, 1999, pp. 293-312.
10. W. Damm, T. Toben, and B. Westphal. On the Expressive Power of Live Sequence Charts. In T. W. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation*, volume 4444 of *LNCS*, pages 225–246. Springer, 2006.
11. J. Fisher, D. Harel, E. J. A. Hubbard, N. Piterman, M. J. Stern, and N. Swerdlin. Combining State-Based and Scenario-Based Approaches in Modeling Biological Systems. In V. Danos and V. Schächter, editors, *CMSB*, volume 3082 of *LNCS*, pages 236–241. Springer, 2004.
12. D. Harel, A. Kleinbort, and S. Maoz. S2A: A Compiler for Multi-modal UML Sequence Diagrams. In *Proc. Fundamental Approaches to Software Engineering (FASE'07)*, volume 4422 of *LNCS*, pages 121–124. Springer, 2007.
13. D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *Int. J. of Foundations of Computer Science*, 13(1):5–51, February 2002. (Also in *Proc. 5th Int. Conf. on Implementation and Application of Automata*

(CIAA 2000), Springer-Verlag, pp. 1–33. Preliminary version appeared as technical report MCS99-20, Weizmann Institute of Science, 1999.).
14. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-out of Behavioral Requirements. In *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD '02)*, pages 378–398, London, UK, 2002. Springer-Verlag.
15. D. Harel, H. Kugler, and A. Pnueli. *Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements*, volume 3393 of *LNCS*, pages 309–324. springer, 2005.
16. D. Harel and S. Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and Systems Modeling (SoSyM)*, 2007. To appear.
17. D. Harel and R. Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *MASCOTS*, pages 193–202. IEEE Computer Society, 2002.
18. D. Harel and R. Marelly. *Come , Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
19. D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. *Software and Systems Modeling (SoSyM)*, 2(2):82–107, 2003.
20. D. Harel and I. Segall. Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In O. Grumberg and M. Huth, editors, *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *LNCS*, pages 485–499. Springer, 2007.
21. ITU. International Telecommunication Union Recommendation Z.120: Message Sequence Charts. Technical report, 1996.
22. J. Klose, T. Toben, B. Westphal, and H. Wittke. Check It Out: On the Efficient Formal Verification of Live Sequence Charts. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 219–233. Springer, 2006.
23. J. Klose and H. Wittke. An Automata Based Interpretation of Live Sequence Chart. In T. Margaria and W. Yi, editors, *Proc. 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*. Springer, 2001.
24. H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In *Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05)*, volume 3440 of *LNCS*, pages 445–460, 2005.
25. H. Kugler, M. J. Stern, and E. J. A. Hubbard. Testing Scenario-Based Models. In *Proc. Fundamental Approaches to Software Engineering (FASE'07)*, volume 4422 of *LNCS*, pages 306–320. Springer, 2007.
26. M. Lettrari and J. Klose. Scenario-Based Monitoring and Testing of Real-Time UML Models. In M. Gogolla and C. Kobryn, editors, *UML*, volume 2185 of *LNCS*, pages 317–328. Springer, 2001.
27. D. Lo, S. Maoz, and S.-C. Khoo. Mining Modal Scenario-Based Specification from Execution Traces of Reactive Systems. In *Proc. 22nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'07)*, 2007.
28. M. Maidl. The Common Fragment of CTL and LTL. In *FOCS*, pages 643–652, 2000.
29. S. Maoz and D. Harel. From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ. In *Proc. 14th Int. ACM/SIGSOFT Symp. Foundations of Software Engineering (FSE-14)*, Portland, Oregon, November 2006.

30. R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *Proc. 17th ACM Conf. on Object-Oriented Prog., Systems, Lang. and App. (OOPSLA'02)*, pages 83–100, Seattle, WA, 2002.
31. C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.
32. W. J. Savitch. Relationships Between Nondeterministic and Deterministic Tape Complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970.
33. I. Schinz, T. Toben, C. Mrugalla, and B. Westphal. The rhapsody uml verification environment. In *Proc. of the 2nd Int. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.
34. A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *J. ACM*, 32(3):733–749, 1985.
35. UML. Unified Modeling Language Superstructure Specification, v2.0. OMG spec., OMG, August 2005. Available from http://www.omg.org.