Monitoring Systems with Extended Live Sequence Charts

Ming Chai[1] and Bernd-Holger Schlingloff[2]

[1] [2] Humboldt Universität zu Berlin
[2] Fraunhofer FOKUS
{ming.chai, hs}@informatik.hu-berlin.de

Abstract. A problem with most runtime verification techniques is that the monitoring specification formalisms are often complex. In this paper, we propose an extension of live sequence charts (LSCs) which avoids this problem. We extend the standard LSCs as proposed by Damm and Harel by introducing the notion of "sufficient prechart", and by adding concatenation and iteration of charts. With these extended LSCs, necessary and sufficient conditions of certain statements can be intuitively specified. Moreover, similar as for message sequence charts, sequencing and iteration allow to express multiple scenarios. We give a translation of extended LSCs into linear temporal logic formulae, and develop online monitoring algorithms for traces with respect to extended LSCs. We use our algorithm to test a concrete example from the European Train Control System (ETCS) standard, and evaluate it on several benchmarks. The results show the feasibility of our approach.

1 Introduction

Runtime verification [17,20] is a lightweight formal verification technique, where a system's behaviour is checked while the system is running. This technique involves the use of a *monitor*, which is a device or a piece of software that observes a behaviour of the system and checks the observations against a monitoring specification. Such a monitoring specification consists of a set of correctness properties formulated in some suitable formal language.

Although runtime verification techniques continue to grow more powerful, their practical application in industry is hindered by the fact that most monitoring specification languages are quite complex. A runtime verification method typically uses some form of temporal logic linear temporal logic (LTL) [22], metric temporal logic [24], time propositional temporal logic (TPTL) [7] and first-order temporal logic [2] to specify correctness properties. Although these specification languages are expressive and technically sound for monitoring, most software engineers are not familiar with them and need extensive training to use them efficiently. Therefore, many runtime verification systems support also other specification languages that are more understood by software engineers, such as regular expressions [8] and context-free grammars [21]. Unfortunately, it is difficult to specify complex properties with multiple instances in these languages, and they are not (yet) used in practice by system designers.

Graphical languages such as message sequence charts (MSCs) and UML sequence diagrams (UML-SDs) are widely used in industry for system specifications. However, as semi-formal languages, the semantics of MSCs and UML-SDs is not defined formally. One of the central questions in this context is: "does an MSC (or a UML-SD) describe all possible executions, or does it describe a set of sample executions of the system?" [3]. Since there does not seem to be an agreement on this question, these languages are not suitable for specifying monitoring correctness properties.

In this paper, we investigate the use of live sequence charts (LSCs) as proposed by Damm and Harel [12] for monitoring specifications. The LSC language is an extension of MSC. Using the notions of universal and existential chart, it can express that a behaviour of a system is necessary or possible. A universal chart specifies a necessary (i.e., required) behaviour of the system, whereas an existential chart specifies a possible (i.e., allowed) behaviour. The LSC language also introduces the notion of "temperature" of an element (i.e., hot and cold elements) for distinguishing between mandatory (hot) elements and provisional (cold) elements.

For monitoring, we focus on universal LSCs. A universal chart typically consists of two components: a *prechart* and a *main chart*. The intended meaning is that if the prechart is executed (i.e., the underlying system exhibits an execution which is specified by the prechart), then the main chart must be executed afterwards. The standard definition thus interprets the prechart as a necessary condition for the main chart.

However, for monitoring it is also important to be able to express sufficient conditions of statements. For example, consider the statement `IF a THEN b` in some programming language. It indicates that `b` is executed if `a` is true; otherwise, `b` is not executed. This is not the same as the universal chart ($prechart(a)$, $mainchart(b)$), because here the main chart b can still be executed if a is not satisfied.

As a possible specification for this statement, it has been suggested in UML 2.0 to use the *negation* operator to denote the case of not-executing `b`. As we show in this paper, sufficiency conditions of statements cannot be expressed by a finite set of negation-free universal LSCs. Since the semantics of negative LSCs is hard to define, we suggest an alternative way to specify this case. We extend LSCs to eLSCs by introducing the notion of a "sufficient" prechart. In contrast, we call the prechart of a standard universal chart a "necessary" prechart. With this extension, one can easily and intuitively express situations as above.

Alur and Yannakakis have introduced MSC-graphs to express multiple scenarios [1]. For the same reason, we introduce concatenation and iteration into the eLSC language. Since a universal chart consists of a prechart and a main chart, we define four modes of concatenation. Consequently, iteration also has four modes. In this paper, we study one mode: iteration defined on precharts.

We give a translation of eLSCs without iteration, that is, universal LSCs with necessary and sufficient precharts and concatenations, into LTL formulae.

Checking whether a system run satisfies such an eLSC specification then becomes the problem of checking an execution trace against some LTL formula.

The language of an eLSC with iteration is not necessarily regular. Therefore, an eLSC with iteration cannot be translated into an equivalent LTL formula. Thus, we develop an explicit algorithm for checking arbitrary eLSC properties.

In order to demonstrate the feasibility of these algorithms, we give a concrete example from the railway domain: We formulate properties of the RBC-RBC-handover process in the European Train Control System (ETCS) standard with our eLSC language. Then, we evaluate them with several benchmark traces and give some remarks on the complexity.

Related Work.

The MSC language and UML sequence diagrams are visual specification languages. They are widely used in industry. Alur et. al. study the model checking problems of MSCs, MSC-graphs and Hierarchical MSC-graphs [1]. They show that the complexity of model checking problems for MSCs and synchronous MSC-graphs are coNP-complete, and for asynchronous MSC-graphs are undecidable. Simmonds et. al. use UML-SD as the property specification language to monitor Web Service Conversations [23]. Ciraci et. al. propose a technique to check the correspondence between UML-SD models and implementations [9].

Damm et. al. defined the LSC language, which distinguishes between necessary and possible behaviours of a system [12]. Harel et. al. propose a play-in/play-out approach [14]. Behaviours of the system are captured by play-in; and the system is tested by play-out through executing the LSC specification directly. Bontemps et. al. prove that any LSC specification can be translated into LTL formulae [6]. Kugler et. al. [18] develop a translation of LSCs into LTL formulae, where the size of the resulting LTL formula is polynomial in the number of events appearing in the LSCs. The expressive power and complexity of LSCs are discussed in the survey [16]. Kumar et. al. extend the LSC language with Kleene star, subcharts, and hierarchical charts [19]. They translate an extended LSC based communication protocol specification into an automaton, and verify the specification with the resulting automaton. Since all existing works are based on the standard LSC language, they suffer from the same expressiveness problem as addressed in this paper.

LSCs have been used to model a variety of systems, such as railway systems [4], telecommunication systems [11], biological systems [13], and so on. The existing papers essentially build models of systems with the LSC language, and focus on model checking problems. To our knowledge, LSC based runtime verification approaches have not been studied yet.

2 Extended Universal Live Sequence Charts

2.1 Universal Live Sequence Charts

A basic chart of an LSC is visually similar to an MSC. It specifies the exchange of messages among a set of instances. Each instance is represented by a lifeline.

When an LSC is executed, for each message in the chart two events occur: the event of sending the message and the event of receiving it. The partial order of events induced by a basic chart is as follows.

- an event at a higher position in a lifeline precedes an event at a lower position in the same lifeline; and
- for each message m, the send-event of m precedes the receive-event of m.

Formally, basic charts can be defined as follows.

Let Σ be a finite alphabet of messages m, i.e., $m \in \Sigma$. An *event* e is a pair $e \triangleq (m, \beta)$ with $\beta \in \{s, r\}$, where (m, s) denotes the event of sending m, and (m, r) denotes the event of receiving m. We denote the set $(\Sigma \times \{s, r\})$ with $\mathcal{B}\Sigma$. A trace τ over $\mathcal{B}\Sigma$ is an element of $\mathcal{B}\Sigma^*$. The length of τ is $|\tau|$.

A lifeline l is a sequence of events $l \triangleq (e_1, e_2, ..., e_n)$. A basic chart $\mathfrak{c}$ is a set of lifelines $\mathfrak{c} \triangleq \{l_1, l_2, ..., l_n\}$, where each event (m, β) occurs at most once. Lifelines in a basic chart are usually drawn as vertical dashed lines, and messages as solid arrows between lifelines.

Now we present the trace semantics for basic charts. For a basic chart $\mathfrak{c}$, let $\mathscr{E}(\mathfrak{c})$ be the set of events appearing in $\mathfrak{c}$. The chart $\mathfrak{c}$ induces a partial order relation $\prec$ on $\mathscr{E}(\mathfrak{c})$ as follows:

1. for any $l \triangleq (e_1, e_2, ..., e_m) \in \mathfrak{c}$ and $1 \leq j < m$, it holds that $e_j \prec e_{j+1}$; and
2. for any $m \in \Sigma$, if (m, s) and $(m, r) \in \mathscr{E}(\mathfrak{c})$, then $(m, s) \prec (m, r)$.
3. $\prec$ is the smallest relation satisfying 1. and 2.

Let $\mathscr{P}(\mathfrak{c}) \triangleq \{(e, e') \mid e \prec e' \text{ with } e, e' \in \mathscr{E}(\mathfrak{c})\}$. A set of traces is defined by $\mathfrak{c}$ as follows:

$$\mathit{Traces}(\mathfrak{c}) \triangleq \{(e_{x1}, e_{x2}, ..., e_{xn}) \mid \{e_{x1}, e_{x2}, ..., e_{xn}\} = \mathscr{E}(\mathfrak{c});\ n = |\mathscr{E}(\mathfrak{c})|; \text{ and for all } e_{xi}, e_{xj} \in \mathscr{E}(\mathfrak{c}), \text{ if } e_{xi} \prec e_{xj}, \text{ then } xi < xj\}.$$

We call each $\sigma_{\mathfrak{c}} \in (\mathcal{B}\Sigma \backslash \mathscr{E}(\mathfrak{c}))$ a *stutter event* of $\mathfrak{c}$. For each basic chart $\mathfrak{c}$, the language $\mathcal{L}(\mathfrak{c})$ is defined by $\mathcal{L}(\mathfrak{c}) \triangleq \{(\sigma^*_{\mathfrak{c}}, e_1, \sigma^*_{\mathfrak{c}}, e_2, ..., \sigma^*_{\mathfrak{c}}, e_n, \sigma^*_{\mathfrak{c}})\}$, where $(e_1\ e_2, ..., e_n) \in \mathit{Traces}(\mathfrak{c})$ and each $\sigma^*_{\mathfrak{c}}$ is a finite (or empty) sequence of stutter events. A trace τ is *admitted* by a basic chart $\mathfrak{c}$ (denoted by $\tau \Vdash \mathfrak{c}$) if $\tau \in \mathcal{L}(\mathfrak{c})$.

A *universal chart* consists of two basic charts: a *prechart* (drawn with a surrounding hashed hexagon) and a *main chart* (drawn within a solid rectangle). It is formalized as a pair $\mathfrak{u} \triangleq (\mathfrak{p}, \mathfrak{m})$, where $\mathfrak{p}$ is the prechart and $\mathfrak{m}$ is the main chart. Intuitively, a universal chart specifies all traces τ such that, if τ contains a segment which is admitted by the prechart, then it must also contain a continuation segment (directly following the first segment) which is admitted by the main chart.

Given a universal chart $\mathfrak{u} \triangleq (\mathfrak{p}, \mathfrak{m})$, the stutter events of $\mathfrak{u}$ are $\sigma_{\mathfrak{u}} \in (\mathcal{B}\Sigma \backslash (\mathscr{E}(\mathfrak{p}) \cup \mathscr{E}(\mathfrak{m}))$. The languages $\mathcal{L}(\mathfrak{p})$ of the prechart and $\mathcal{L}(\mathfrak{m})$ of the main chart are defined with these stutter events as above.

For languages $\mathcal{L}$ and $\mathcal{L}'$, let $(\mathcal{L} \circ \mathcal{L}')$ be the concatenation of $\mathcal{L}$ and $\mathcal{L}'$ (i.e., $(\mathcal{L} \circ \mathcal{L}') \triangleq \{(\tau\tau') \mid \tau \in \mathcal{L} \ \textit{and}\ \ \tau' \in \mathcal{L}'\}$); and $\overline{\mathcal{L}}$ be the complement of $\mathcal{L}$ (i.e., for any $\tau \in \mathcal{B}\Sigma^*$, it holds that $\tau \in \overline{\mathcal{L}}$ iff $\tau \notin \mathcal{L}$). The semantics of universal charts is defined as follows (see, e.g., [5]).

Definition 1. *Given a finite alphabet Σ, the language of a universal chart* $\mathfrak{u} \triangleq (\mathfrak{p}, \mathfrak{m})$ *is*

$$\mathcal{L}(\mathfrak{u}) \triangleq \overline{\mathcal{B}\Sigma^* \circ \mathcal{L}(\mathfrak{p}) \circ \overline{\mathcal{L}(\mathfrak{m}) \circ \mathcal{B}\Sigma^*}}.$$

This formalizes the intuitive interpretation given above. An LSC specification $\mathfrak{U}$ is a finite set of universal charts. The language of $\mathfrak{U}$ is $\mathcal{L}(\mathfrak{U}) \triangleq \bigcap_{\mathfrak{u} \in \mathfrak{U}} \mathcal{L}(\mathfrak{u})$.

2.2 Expressiveness of LSC Specifications

The standard definition of a universal chart interprets the prechart as a necessary condition of the main chart, i.e., a system is allowed to adhere to any execution, as long as it does not execute the prechart. This is not sufficient for specifying some correctness properties. For instance, for two basic charts $\mathfrak{c}$ and $\mathfrak{c}'$ we can define the statement

$$\mathsf{CS} = (\ (\mathfrak{c} \textit{ is executed}) \texttt{ IF AND ONLY IF LATER } (\mathfrak{c}' \textit{ is executed})),$$

to have the semantics

$$\mathcal{L}(\mathsf{CS}) \triangleq \left(\overline{\mathcal{B}\Sigma^* \circ \mathcal{L}(\mathfrak{c}) \circ \overline{\mathcal{L}(\mathfrak{c}') \circ \mathcal{B}\Sigma^*}}\right) \cap \left(\overline{\overline{\mathcal{B}\Sigma^* \circ \mathcal{L}(\mathfrak{c})} \circ \mathcal{L}(\mathfrak{c}') \circ \mathcal{B}\Sigma^*}\right).$$

However, this can not be expressed with LSC specifications:

Lemma 1. *The language* $\left(\overline{\overline{\mathcal{B}\Sigma^* \circ \mathcal{L}(\mathfrak{c})} \circ \mathcal{L}(\mathfrak{c}') \circ \mathcal{B}\Sigma^*}\right)$ *cannot be defined by an LSC specification.*

Proof. See appendix.

2.3 Extended LSCs

One way to overcome the above expressiveness limitation is to introduce a negation operator into the LSC language. Unfortunately, the semantics of such a negation operator can be tricky, see [15]. As an alternative, we extend universal charts by introducing the notion of a "*sufficient prechart*" (drawn with a surrounding solid hexagon). This is a prechart which is interpreted as a sufficient condition for a main chart. In contrast, we label the original prechart of a universal chart as a "*necessary prechart*". Formally, the syntax of extended LSCs is as follows.

Definition 2. *An eLSC is a tuple* $\mathfrak{u} \triangleq (\mathfrak{p}, \mathfrak{m}, \mathrm{Cond})$*, where* $\mathfrak{p}$ *and* $\mathfrak{m}$ *are a prechart and a main chart, and* $\mathrm{Cond} \in \{Nec, Suff\}$ *denotes if* $\mathfrak{p}$ *is a necessary or sufficient prechart.*

For a chart $\mathfrak{u} \triangleq (\mathfrak{p}, \mathfrak{m}, Nec)$, the language is as defined in Definition 1. The language defined by a chart $\mathfrak{u} \triangleq (\mathfrak{p}, \mathfrak{m}, Suff)$ is

$$\mathcal{L}(\mathfrak{u}) \triangleq \left(\overline{\overline{\mathcal{B}\Sigma^* \circ \mathcal{L}(\mathfrak{p})} \circ \mathcal{L}(\mathfrak{m}) \circ \mathcal{B}\Sigma^*}\right).$$

The above condition statement CS can then be specified by an LSC specification $\{(\mathfrak{c}, \mathfrak{c}', \mathit{Nec}), (\mathfrak{c}, \mathfrak{c}', \mathit{Suff})\}$. As an abbreviation, we introduce an "iff" prechart (notated with a double dashed lines). An eLSC with an "iff" prechart is defined as $\mathfrak{u}^{\mathit{iff}} \triangleq \{(\mathfrak{p}, \mathfrak{m}, \mathit{Nec}), (\mathfrak{p}, \mathfrak{m}, \mathit{Suff})\}$.

2.4 Concatenations of universal LSCs

Concatenation of two eLSCs essentially introduces partial orders of executions of the charts. This feature can be inherited by eLSC specifications.

First, we define the concatenation of basic charts $\mathfrak{c}$ and $\mathfrak{c}'$, denoted with $(\mathfrak{c} \rightarrow \mathfrak{c}')$. Intuitively, a trace τ is in the language of $(\mathfrak{c} \rightarrow \mathfrak{c}')$ iff it contains two segments υ and υ' such that υ precedes υ' in τ, and υ (resp. υ') is admitted by $\mathfrak{c}$ (resp. $\mathfrak{c}'$). Formally, the language of $(\mathfrak{c} \rightarrow \mathfrak{c}')$ is given by the following clause.

$$\mathcal{L}(\mathfrak{c} \rightarrow \mathfrak{c}') \triangleq \left(\mathcal{L}(\mathfrak{c}) \cap \mathcal{L}(\mathfrak{c}') \cap \overline{\overline{\mathcal{L}(\mathfrak{c})} \circ \mathcal{L}(\mathfrak{c}')}\right).$$

Since a universal chart $\mathfrak{u}$ consists of two basic charts $\mathfrak{p}$ and $\mathfrak{m}$, there are four possibilities to define the concatenation of universal charts $\mathfrak{u}$ and $\mathfrak{u}'$: $\mathfrak{p} \rightarrow \mathfrak{p}'$, $\mathfrak{p} \rightarrow \mathfrak{m}'$, $\mathfrak{m} \rightarrow \mathfrak{p}'$ and $\mathfrak{m} \rightarrow \mathfrak{m}'$.

For monitoring, we consider only two modes of concatenation in this paper: *prechart concatenation* and *main chart concatenation*. The concatenation of two universal charts $\mathfrak{u}$ and $\mathfrak{u}'$ is defined to be a tuple $\delta \triangleq (\mathfrak{u}, \mathfrak{u}', \mathit{Mode})$, where $\mathit{Mode} \in \{\mathit{preC}, \mathit{mainC}\}$. Formally, the semantics of the two concatenation modes is given as follows.

Definition 3. *Given two eLSCs* $\mathfrak{u}$ *and* $\mathfrak{u}'$*, The language of the concatenation of* $\mathfrak{u}$ *and* $\mathfrak{u}'$ *is*

$$\mathcal{L}(\delta) \triangleq \left(\mathcal{L}(\mathfrak{u}) \cap \mathcal{L}(\mathfrak{u}') \cap \overline{\overline{\mathcal{B}\Sigma^* \circ \mathcal{L}(\mathfrak{c})} \circ \mathcal{L}(\mathfrak{c}') \circ \mathcal{B}\Sigma^*}\right),$$

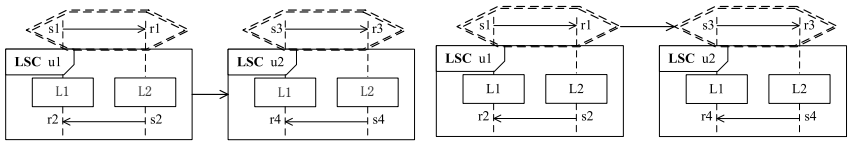
where $\mathfrak{c} = \mathfrak{p}$ *and* $\mathfrak{c}' = \mathfrak{p}'$*, if Mode = preC; and*
$\mathfrak{c} = \mathfrak{m}$ *and* $\mathfrak{c}' = \mathfrak{m}'$*, if Mode = mainC.*

It can be shown that the language of a concatenation $(\mathfrak{u}, \mathfrak{u}', \mathit{preC})$ (resp. $(\mathfrak{u}, \mathfrak{u}', \mathit{mainC})$) is the same as the language of the eLSC specification $\{\mathfrak{u}, \mathfrak{u}', (\mathfrak{p}, \mathfrak{p}', \mathit{Suff})\}$ (resp. $\{\mathfrak{u}, \mathfrak{u}', (\mathfrak{m}, \mathfrak{m}', \mathit{Suff})\}$). Figure 1 illustrates the two concatenation modes of eLSCs $\mathfrak{u}1$ and $\mathfrak{u}2$, where Fig. 1(a) presents a main chart concatenation and Fig. 1(b) presents a prechart concatenation. Fig. 1(c) and Fig. 1(d) present the partial orders of events of these concatenations, respectively.

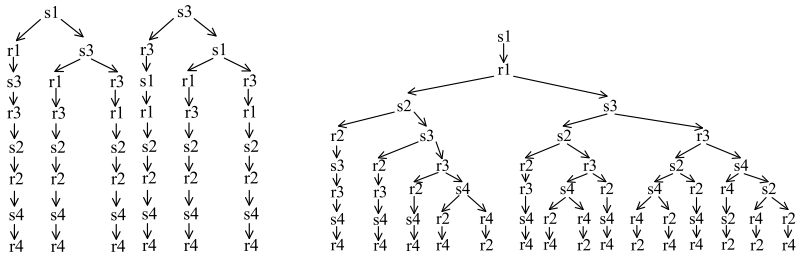
To specify a repeating execution (e.g., repeating responses to requests), an iteration operator can be introduced. Such iteration operator can be directly defined from the above concatenations; $\mathfrak{u}^+ \triangleq \mathfrak{u} \cup (\mathfrak{u} \rightarrow \mathfrak{u}) \cup (\mathfrak{u} \rightarrow \mathfrak{u} \rightarrow \mathfrak{u}) \cup ...$ Since concatenations have different modes, iteration has different modes as well.

In this paper, we consider only *iteration of necessary precharts*. Intuitively, an eLSC $\mathfrak{u}^+$ specifies that if the prechart is executed n times, then the main

chart must be executed at least n times, where the executions of the main chart can be interleaved. For instance, given the eLSC $\mathfrak{u}_1$ in Fig. 1, a trace (... s1, r1, s2, s1, r2, r1, s2, r2,...) is admitted by $\mathfrak{u}_1^+$; whereas traces (... s1, s1, r1, s2, r2, r1, s2, r2,...) and (... s1, r1, s2, r2, s1, r1) are not admitted by $\mathfrak{u}_1^+$.



(a) Mainchart Concatenation (b) Prechart Concatenation



(c) Partial orders induced by the mainchart concatenation. (d) Partial orders induced by the prechart concatenaiton.

Fig. 1. Example: a prechart concatenation and a mainchart concatenation

3 A Translation of eLSCs into LTL Formulae

3.1 Preliminaries

We now show how to translate extended LSCs into linear temporal logic formulae for online monitoring.

Definition 4. *Given the finite alphabet Σ of messages, the formulae φ of LTL are inductively formed according to the following grammar, where, $m \in \Sigma$ and $\beta \in \{s, r\}$:*

$\varphi ::= \bot \mid (m, \beta) \mid (\varphi_1 \Rightarrow \varphi_2) \mid (\varphi_1 \ \mathbf{U} \ \varphi_2) \mid \mathbf{X} \ \varphi$.

In addition, we use the following shorthand: $\neg\varphi$ stands for $(\varphi \Rightarrow \bot)$, $\top$ stands for $\neg\bot$, $\mathbf{F}\ \varphi$ stands for $(\top\ \mathbf{U}\ \varphi)$, $\mathbf{G}\ \varphi$ stands for $\neg\mathbf{F}\ \neg\varphi$ and $\varphi_1\ \mathbf{W}\ \varphi_2$ stands for $\neg\ (\neg\varphi_2\ \mathbf{U}\ \neg\ (\varphi_1 \vee \varphi_2))$. Given an event $e \triangleq (m, \beta)$, we define $\mathit{Mess}(e) \triangleq m$ and $\mathit{Beh}(e) \triangleq \beta$. We define LTL on finite traces as follows.

Definition 5. *Let $\tau \triangleq (e_1, e_2, ..., e_n) \in \mathcal{B}\Sigma^*$ with $1 \leq i \leq n$ being a position of τ. The semantics for LTL is defined inductively as follows:*

$(\tau, i) \not\models \bot$;
$(\tau, i) \models (m,\ \beta)$ *iff* $m = \mathit{Mess}(e_i)$ *and* $\beta = \mathit{Beh}(e_i)$;
$(\tau, i) \models (\varphi_1 \Rightarrow \varphi_2)$ *iff* $(\tau, i) \models \varphi_1$ *implies* $(\tau, i) \models \varphi_2$;
$(\tau, i) \models (\varphi_1\ \mathbf{U}\ \varphi_2)$ *iff there exists* $i \leq j \leq |\tau|$ *with* $(\tau, j) \models \varphi_2$,
and for all $i \leq j' < j$ *it holds that* $(\tau, j') \models \varphi_1$;
$(\tau, i) \models \mathbf{X}\ \varphi$ *iff* $i = |\tau|$ *or* $(\tau,\ i+1) \models \varphi$.

As usual, $\tau \models \varphi$ iff $(\tau, 1) \models \varphi$. Note that the logic is defined on events, and will be used for monitoring sequences of events.

3.2 Translation of Universal Charts

In this section, we show how to translate a universal chart into an LTL formula to check whether a trace is admitted. We follow the approach of Kugler et al. [18]. From a basic chart $\mathfrak{c}$, we define the LTL formula $\xi_\mathfrak{c} \triangleq \psi_\mathfrak{c} \wedge \gamma_\mathfrak{c} \wedge \mathfrak{y}_\mathfrak{c}$, where

$\psi_\mathfrak{c} \triangleq \bigwedge_{(e,\, e') \in \mathscr{P}(\mathfrak{c})} (\neg e'\ \mathbf{W}\ e)$

$\gamma_\mathfrak{c} \triangleq \bigwedge_{e \in \mathscr{E}(\mathfrak{c})} (\neg e\ \mathbf{W}\ (e \wedge \mathbf{X}\ \mathbf{G}\ \neg e))$

$\mathfrak{y}_\mathfrak{c} \triangleq \bigwedge_{e \in \mathscr{E}(\mathfrak{c})} \mathbf{F}\ e$

The formula $\psi_\mathfrak{c}$ specifies that e' cannot occur before e in a trace with $e \prec e'$. It does not specify e must occur at some point. The formula $\gamma_\mathfrak{c}$ specifies that each e can only occur at most one time in a trace. The formula $\mathfrak{y}_\mathfrak{c}$ specifies that every event appearing in the chart will eventually occur in a trace.

With these formulae, we can then obtain LTL formulae from eLSCs with necessary and sufficient precharts. From an eLSC $\mathfrak{u} \triangleq (\mathfrak{p},\ \mathfrak{m},\ \mathit{Cond})$, we define the following formulae.

$\xi_\mathfrak{p} \triangleq \psi_\mathfrak{p} \wedge \gamma_\mathfrak{p} \wedge \mathfrak{y}_\mathfrak{p}$, and $\xi_\mathfrak{m} \triangleq \psi_\mathfrak{m} \wedge \gamma_\mathfrak{m} \wedge \mathfrak{y}_\mathfrak{m}$,

$\chi \triangleq \bigwedge_{e' \in \mathscr{E}(\mathfrak{p})} ((\bigwedge_{e \in \mathscr{E}(\mathfrak{m})} (\neg e))\ \mathbf{W}\ e')$

$\varphi(\mathfrak{u}) \triangleq ((\xi_\mathfrak{p} \wedge \chi) \Rightarrow \xi_\mathfrak{m})$ if $\mathit{Cond} = \mathit{Nec}$; and

$\varphi(\mathfrak{u}) \triangleq (\neg(\xi_\mathfrak{p} \wedge \chi) \Rightarrow \neg\xi_\mathfrak{m})$ if $\mathit{Cond} = \mathit{Suff}$.

The formula χ specifies that events appearing in the main chart cannot occur until all events appearing in the prechart have occurred in a trace. In can be shown that the formula $\varphi(\mathfrak{u})$ defines the language of $\mathfrak{u}$.

Lemma 2. *A trace is admitted by a universal chart* $\mathfrak{u}$ *if and only if it satisfies* $\varphi(\mathfrak{u})$*:* $\tau \Vdash \mathfrak{u}$ *iff* $\tau \models \varphi(\mathfrak{u})$.

Proof. Follows from the definitions. Omitted in this version of the paper.

With this translation of LSCs into LTL formulae, a system can be monitored by standard methods, e.g., formula rewriting. The size of the formula $\varphi(\mathfrak{u})$ is polynomial in the number of events appearing in $\mathfrak{u}$. Therefore, the resulting LTL formula will not explode when dealing with large eLSC specifications.

As remarked above, a concatenation $\delta = (\mathfrak{u}, \mathfrak{u}', \mathit{Mode})$ of eLSCs can be expressed by a set of single eLSCs. This can be translated into an equivalent

conjunction of LTL formulae as above. Thus, concatenation does not pose any additional difficulties for monitoring.

4 An Algorithm for Checking eLSCs with Iteration

The language defined by an eLSC with iteration is incomparable with LTL. Even eLSCs cannot express the temporal "next" operator. Similar with asynchronous concatenations of MSCs, the language of an eLSC with iterated precharts is not necessary regular. Therefore, for an eLSC $\mathfrak{u}^+$, in general there is no equivalent LTL formula, and the above approach to monitoring cannot be applied. For this reason, we develop an explicit algorithm for checking traces against eLSCs with prechart iterations. In the algorithm, a trace is checked against an eLSC specification $\mathfrak{u}^+$ in two steps.

1. The trace is decomposed into a set of sub-traces and a remainder sequence according to the events appearing in $\mathfrak{p}$ and $\mathfrak{m}$. Every event is unique in each sub-trace.
2. It is checked whether all sub-traces are admitted by the corresponding basic charts $\mathfrak{p}$ and $\mathfrak{m}$, and whether the begin point and the end point of each sub-trace respect the partial order implied by $\mathfrak{u}^+$.

4.1 Decomposing Traces

A trace is decomposed by two operations $\triangleright$ and $\overline{\triangleright}$. The operation $\triangleright$ generates a sub-trace τ_s from a trace τ according to a set of events E. In the resulting τ_s, each event in E occurs at most once. The order of events in τ_s is the same as in the original trace. The operation $\overline{\triangleright}$ generates the "complement" sub-trace of τ_s. These operations are formally defined as follows.

Given a trace $\tau \triangleq (e_1, ..., e_n)$ and a formula φ, we define $\kappa(\tau, \varphi)$ to be the smallest i such that $(e_1, ...e_i) \models \varphi$ (and $\kappa(\tau, \varphi) = 0$ if there is no such i). For a set of events $E = \{x_1, ..., x_m\}$, we define a sequence of points $K(\tau, E) \triangleq (k_1, ..., k_m)$ with $1 \leq k_1 \leq ... \leq k_m \leq |\tau|$ by $\{k_1, ..., k_m\} = \{\kappa(\tau, \mathbf{F}x_1), ..., \kappa(\tau, \mathbf{F}x_m)\}$. Let $\mathscr{E}(\tau)$ be the set of events appearing in τ and let ε be the empty trace. The operations $\triangleright$ and $\overline{\triangleright}$ are defined as follows.

$\triangleright$: $\mathcal{B}\Sigma^* \times 2^{\mathcal{B}\Sigma} \mapsto \mathcal{B}\Sigma^*$ such that
$\tau \triangleright E \triangleq (e[k_1], ..., e[k_{|E|}])$ with $(k_1, ..., k_{|E|}) = K(\tau, E)$ if $E \subseteq \mathscr{E}(\tau)$;
$\tau \triangleright E \triangleq \varepsilon$ if $E \not\subseteq \mathscr{E}(\tau)$.
$\overline{\triangleright}$: $\mathcal{B}\Sigma^* \times 2^{\mathcal{B}\Sigma} \mapsto \mathcal{B}\Sigma^*$ such that
$\tau \overline{\triangleright} E \triangleq (e[1], ..., e[k_1 - 1], e[k_1 + 1], ..., e[k_{|E|} - 1], e[k_{|E|} + 1], ..., e[n])$ if $\tau \triangleright E \neq \varepsilon$;
$\tau \overline{\triangleright} E \triangleq \tau$ if $\tau \triangleright E = \varepsilon$.

Given a trace τ and a basic chart $\mathfrak{c}$, we define a tuple $\mathsf{Div}(\tau, \mathfrak{c}) \triangleq (\tau_s, Pos^I, Pos^F)$, where Pos^I is the index of the initial point of τ_s, and Pos^F is the index of the final point of τ_s. Formally, $\mathsf{Div}(\tau, \mathfrak{c})$ is defined as follows.

Algorithm 1 divide a trace according to a basic chart

procedure TraceDiv(trace τ, basic chart $\mathfrak{c}$) =
while $(\tau \rhd \mathscr{E}(\mathfrak{c})) \neq \tau$ **do**
  $\tau s \leftarrow (\tau \rhd \mathscr{E}(\mathfrak{c}))$;
  $Pos^I \leftarrow \kappa(\tau, \bigvee_{e \in \mathscr{E}(\mathfrak{c})} \mathbf{F}e)$;
  $Pos^F \leftarrow \kappa(\tau, \bigwedge_{e \in \mathscr{E}(\mathfrak{c})} \mathbf{F}e)$;
  DivSet $\leftarrow$ DivSet.**add** $(\tau s, Pos^I, Pos^F)$; //add the resulting tuple into the set DivSet
  $\tau \leftarrow (\tau \rhd \mathscr{E}(\mathfrak{c}))$;
end
return DivSet

- $\tau_s \triangleq (\tau \rhd \mathscr{E}(\mathfrak{c}))$, $Pos^I \triangleq \bigvee_{e \in \mathscr{E}(\mathfrak{c})} \mathbf{F}e$, $Pos^F \triangleq \bigwedge_{e \in \mathscr{E}(\mathfrak{c})} \mathbf{F}e$, if $\mathscr{E}(\mathfrak{c}) \subseteq \mathscr{E}(\tau)$;
- $\tau_s \triangleq \tau$ and $Pos^I = Pos^F = 0$, otherwise.

Next, we define a set $\mathsf{DivSet}(\tau, \mathfrak{c}) \triangleq \{(\tau_{s1}, Pos_1^I, Pos_1^F), ..., (\tau_{sn}, Pos_n^I, Pos_n^F)\}$, where
$(\tau_{s1}, Pos_1^I, Pos_1^F) \triangleq \mathsf{Div}(\tau, \mathfrak{c})$;
$(\tau_{si}, Pos_i^I, Pos_i^F) \triangleq \mathsf{Div}((\tau_{i-1} \rhd \mathfrak{c}), \mathfrak{c})$ for $1 < i \leq n$; and
$(\tau_{s(n+1)} \rhd \mathscr{E}(\mathfrak{c})) = \varepsilon$.
For a universal chart, we define two such sets $\mathsf{DivSet}(\tau, \mathfrak{p})$ and $\mathsf{DivSet}(\tau, \mathfrak{m})$. The calculation of these set can be done with Algorithm 1 above.

4.2 Checking Sub-traces

With the above decomposition, we can then check whether τ is admitted by $\mathfrak{u}^+$. An eLSC with iteration specifies repeated execution of a chart. A trace τ is admitted by $\mathfrak{u}^+$ if and only if

- τ is able to be decomposed into a number of sub-traces, each of which is admitted by $\mathfrak{u}$; and
- the order of execution of the prechart is respected.

According to the above rules, we develop algorithms for checking whether $\tau \Vdash \mathfrak{u}^+$, where Alg. 2 (resp. Alg. 3) checks the prechart (resp. the main chart) of $\mathfrak{u}$. The two sub-algorithms return PRes and MRes as the checking result. The satisfaction of τ against $\mathfrak{u}^+$ is (PRes $\wedge$ MRes) Let $\mathcal{F}$ be a formula, we define an interpretation operation $[\![\mathcal{F}]\!]$ that maps $\mathcal{F}$ to a boolean value. For a trace τ and an LTL formula φ, we say $[\![\tau \models \varphi]\!] \triangleq \mathit{true}$ if τ is satisfied by φ; and $[\![\tau \models \varphi]\!] \triangleq \mathit{false}$ if τ is violated by φ. The algorithm for checking traces against LTL formulae is developed according to an effective rewriting algorithm proposed by Havelund [17].

Algorithm 2 Checking the prechart of $\mathfrak{u}^+$

input : A trace τ and an eLSC $\mathfrak{u} \triangleq (\mathfrak{m}, \mathfrak{p}, \mathit{Cond})$
output: whether τ is admitted by $\mathfrak{u}^+$

$\mathsf{PRes} \leftarrow$ true; // initialize the checking result
$p \leftarrow |\mathsf{DivSet}(\tau, \mathfrak{p})|$; // the number of executions of the prechart
for $i \leftarrow 1$ **to** p **do**
  // check whether each execution of the prechart is correct
  $\mathsf{PRes} \leftarrow (\mathsf{PRes} \wedge [\![\tau s_i \models \psi_{\mathfrak{p}}]\!])$;
  // check the partial order of the prechart's executions
  $\mathsf{PRes} \leftarrow (\mathsf{PRes} \wedge [\![Pos_i^F < Pos_{i+1}^I]\!])$;
  /* if the prechart is a necessary prechart, then there is an execution of the main chart after each execution of the prechart */
  if $\mathit{Cond} == \mathit{Nec}$ **then**
    $\mathsf{PRes} \leftarrow (\mathsf{PRes} \wedge [\![m \geq p]\!])$;
    if $\exists(\tau s, \mathrm{Pos}^I, \mathrm{Pos}^F) \in \mathsf{DivSet}(\tau, \mathfrak{m})$ *s.t.* $\mathrm{Pos}^I > \mathrm{Pos}_i^F$ **then**
      $\mathsf{PRes} \leftarrow \mathsf{PRes} \wedge$ true;
    else
      $\mathsf{PRes} \leftarrow$ false;
    end
  end
end
return PRes

5 Case Study: the RBC/RBC Handover Process

In this section, we present a concrete example from the European Train Control System (ETCS). In the ETCS level 2, the radio block center (RBC) is responsible for providing movement authorities to allow the safe movement of trains. A route is divided into several RBC supervision areas. When a train approaches the border of an RBC supervision area, an RBC/RBC handover process takes place. The current RBC is called the handing over RBC (HOVRBC), whereas the adjacent RBC is called the accepting RBC (ACCRBC) [3].

The RBC/RBC handover process is performed via exchanging a sequence of messages between the two RBCs. These messages are called NRBC messages, including "Pre-Announcement" (preAnn), "Route Related Information Request" (RRIReq), "Route Related Information" (RRI) and "Acknowledgement" (Ackn). The NRBC messages are exchanged via an open communication system GSM-R.

The safety standard EN50159 identifies the following threats to an open transmission system: corruption, masquerading, repetition, deletion, insertion,

[3] Further details of this case study are provided in
`http://www2.informatik.hu-berlin.de/~hs/Publikationen/2014_RV_Ming-Schlingloff_ETCS-Case-study(description-of-RBCRBC-handover).pdf`

Algorithm 3 Checking the main chart of $\mathfrak{u}^+$

input : A trace τ and an eLSC $\mathfrak{u} \triangleq (\mathfrak{m}, \mathfrak{p}, \mathit{Cond})$
output: whether τ is admitted by $\mathfrak{u}^+$

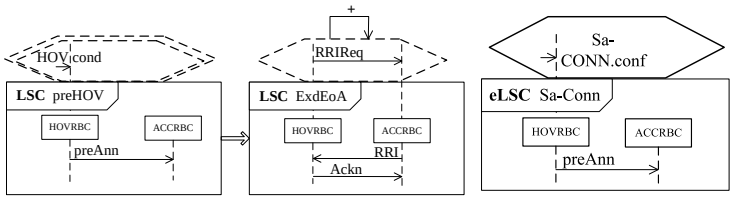
```
MRes ← true; // initialize the checking result
m ← |DivSet(τ, 𝔪)|; // the number of executions of the main chart
for j ← 1 to m do
    MRes ← (MRes ∧ [[τs_j ⊨ ψ_m]]; // check each execution of the main chart
    /* If 𝔲 is with a sufficient prechart, then there is an execution of the prechart before
    each execution of the main chart. */
    if Cond == Suff then
        MRes ← (MRes ∧ [[m ≤ p]]);
        if ∃(τs, Pos^I, Pos^F) ∈ DivSet(τ, 𝔭) s.t. Pos^F < Pos_i^I then
            MRes ← MRes ∧ true;
        else
            MRes ← false;
        end
    end
end
return MRes
```

resequencing and delay. A safety protocol is added between the application layer and the transport layer for providing safe communication between RBCs. The safety protocol provides protection against threats related to corruption and masquerading, other threats are covered elsewhere.

We use eLSC based monitors to protect against threats related to temporal relations of messages, i.e., repetition, deletion, insertion and resequencing. In this paper, we specify the following two properties with the eLSC language.

1. For a successful RBC/RBC handover process, if the train reaches the border of two RBC areas, the NRBC messages should be correctly exchanged between the two RBCs (see Fig. 2(a)).
2. The NRBC messages can only be exchanged after the two RBCs establish a safe connection (see Fig. 2(b)).

For property 1, the message preAnn is exchanged in sequence if and only if after the HOVRBC detects the handover condition. We specify the handover condition by an "HOV cond" message. Therefore, the eLSC preHOV is with an "iff" prechart, which consists of the receiving event of the message HOV cond. If HOVRBC sends an RRIReq message to ACCRBC, ACCRBC sends an RRI message to HOVRBC. HOVRBC sends an Ackn message to ACCRBC after receiving the RRI message. In fact, the accepting RBC is allowed to send an RRI without an RRI request when there is new route information. Hence, the second eLSC in Fig. 2(a) (eLSC ExdEoA) is with a necessary prechart. Since



(a) The eLSC for Pro. 1 (b) The eLSC for Pro. 2

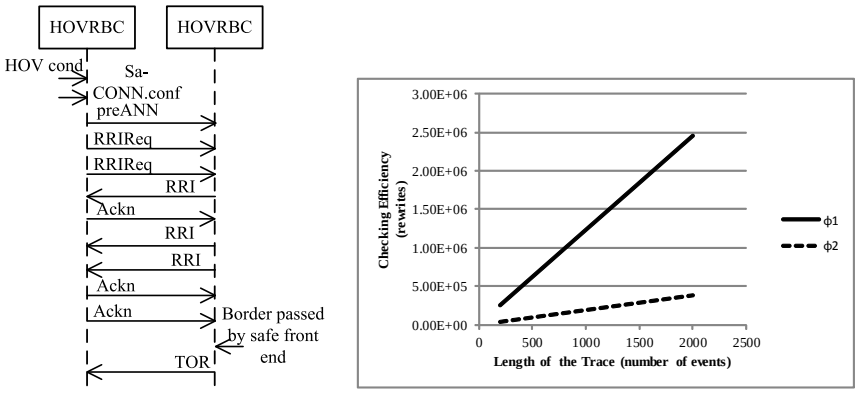
Fig. 2. Example: the RBC/RBC handover process

the HOVRBC can ask for new route information iteratively, the eLSC is with an iteration.

According to the requirements of ETCS, the messages RRIReq and RRI are allowed to be exchanged after HOVRBC receives the "preAnn" message. Thus, the eLSC ExdEoA in Fig. 2(a) cannot be executed before preHOV. The double arrow between eLSCs preHOV and ExdEoA in Fig. 2(a) denotes $\{(\mathfrak{m}, \mathfrak{m}', \mathit{Suff}), (\mathfrak{m}, \mathfrak{p}', \mathit{Suff})\}$ for $\mathfrak{u} = \mathit{preHOV}$ and $\mathfrak{u}' = \mathit{ExdEoA}$.

For property 2, the safe connection is established after HOVRBC receives a "safe connection confirm" (Sa-CONN.conf) message. As an example, we consider the message preANN: it cannot be transmitted before HOVRBC receives Sa-CONN.conf. This property is specified by an eLSC with a sufficient prechart, which consists of a receiving event of SaCONN.conf (see Fig. 2(b)).

As an example observation from the log file of RBCs (according to the specification SUBSET-039), we used the trace shown in Fig. 3(a).



(a) A trace example (b) Monitoring efficiency

Fig. 3. Evaulation results in Maude

To prove that the concept of eLSC based monitoring is feasible, we built a prototypical implementation of our algorithms. We translate eLSCs without iteration into LTL formulae, and implement the LTL model checking algorithm in Maude, see [17] and [7]. This is a high performance rewriting environment, which is able to execute millions of rewrites per second [10].

We checked the example trace with our prototypical implementation. The results show that it satisfies the two properties. In addition, we built some more traces by injecting errors, such as adding/removing events, and exchanging the occurrence order of events. The results show that the monitor can detect repetition, deletion, insertion and resequencing errors.

For our LTL translation, since the size of the formula is polynomial in the size of the eLSC, the monitoring complexity is the same as the complexity of LTL model checking. Thus, given an eLSC specification and a trace, the complexity of monitoring is linear in the length of the trace, and (worst-case) exponential in the number of events appearing in the eLSC. We repeated similar experiments several times with different traces. The checking efficiency is shown in Fig. 3(b). In this diagram, φ_1 and φ_2 are the resulting LTL formulae of property 1 and property 2, respectively. The difference in checking efficiency is caused by the sizes of the two formulae: φ_1 consists of approx. 630 sub-formulae, whereas φ_2 has only approx. 130 sub-formulae. The experimental results show that our approach is capable to detect failures in the executions of a system.

6 Conclusion and Discussion

In this paper, we have proposed a monitoring approach on basis of eLSC specifications. We introduced the notion of a sufficient prechart for specifying sufficiency conditions in correctness properties. Then we defined concatenation and iteration of LSCs. We have shown how to translate eLSCs without iteration into LTL formulae. A system can then be monitored by formula rewriting. For the full language, we developed an explicit monitoring algorithm. Finally, we presented a case study with a concrete example from the railway domain. The results show the feasibility of our implementation.

There are several interesting topics for future work. Firstly, the implementation reported in this paper was done as a proof-of-concept, showing that the approach of eLSC based monitoring is feasible. Since the sizes of resulting formulae are often large, translating eLSC into LTL formulae is not an efficient way for monitoring. In addition, to maintain monitors in deployed systems, one would not want to employ full Maude. Therefore, we are currently developing a more efficient implementation, which can check eLSC specifications directly.

Secondly, in this paper we only considered a subset of the original LSC language, excluding conditions and "cold" elements, where additionally all messages had to be unique. Even though we do not think that the full LSC language poses additional fundamental problems, this needs to be worked out. Moreover, the LSC language has been extended with timing constructs for specifying real-time

properties. We want to investigate the translation of eLSCs with such timing constructs into TPTL formulae for monitoring purposes.

Last but not least, it remains open to define an automaton concept which has exactly the same expressiveness as our eLSCs.

Reference

1. Rajeev Alur and Mihalis Yannakakis. Model Checking of Message Sequence Charts. In *CONCUR99 Concurrency Theory*, pages 114–129. Springer, 1999.
2. Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From Propositional to First-order Monitoring. In *Runtime Verification*, pages 59–75. Springer, 2013.
3. Hanene Ben-abdallah and Stefan Leue. Timing Constraints in Message Sequence Chart Specifications. In *In IFIP. Chapman*. Hall, 1997.
4. Jürgen Bohn, Werner Damm, Jochen Klose, Adam Moik, Hartmut Wittke, H Ehrig, B Kramer, and A Ertas. Modeling and Validating Train System Applications Using Statemate and Live Sequence Charts. In *Proc. IDPT*. Citeseer, 2002.
5. Yves Bontemps. *Relating Inter-Agent and Intra-Agent Specifications*. PhD thesis, PhD thesis, University of Namur (Belgium), 2005.
6. Yves Bontemps and Pierre-Yves Schobbens. The Computational Complexity of Scenario-based Agent Verification and Design. *Journal of Applied Logic*, 5(2):252–276, 2007.
7. Ming Chai and Holger Schlingloff. A Rewriting Based Monitoring Algorithm for TPTL. In *CS&P 2013*, pages 61–72. Citeseer, 2013.
8. Feng Chen and Grigore Roşu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 546–550. Springer, 2005.
9. Selim Ciraci, Somayeh Malakuti, Shmuel Katz, and Mehmet Aksit. Checking the Correspondence between UML Models and Implementation. In *Runtime Verification*, pages 198–213. Springer, 2010.
10. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. Maude Manual (version 2.6). *University of Illinois, Urbana-Champaign*, 1(3):4–6, 2011.
11. Pierre Combes, David Harel, and Hillel Kugler. Modeling and Verification of a Telecommunication Application Using Live Sequence Charts and the Play-engine Tool. *Software & Systems Modeling*, 7(2):157–175, 2008.
12. Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
13. Jasmin Fisher, David Harel, E Jane Albert Hubbard, Nir Piterman, Michael J Stern, and Naamah Swerdlin. Combining State-based and Scenario-based Approaches in Modeling Biological Systems. In *Computational Methods in Systems Biology*, pages 236–241. Springer, 2005.
14. David Harel, Hillel Kugler, Rami Marelly, and Amir Pnueli. Smart Play-out of Behavioral Requirements. In *Formal Methods in Computer-aided Design*, pages 378–398. Springer, 2002.
15. David Harel and Shahar Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software & Systems Modeling*, 7(2):237–252, 2008.
16. David Harel, Shahar Maoz, and Itai Segall. Some Results on the Expressive Power and Complexity of LSCs. In *Pillars of computer science*, pages 351–366. Springer, 2008.

17. Klaus Havelund and Grigore Roşu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, 55(2):200–217, 2001.
18. Hillel Kugler, David Harel, Amir Pnueli, Yuan Lu, and Yves Bontemps. Temporal Logic for Scenario-based Specifications. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 445–460. Springer, 2005.
19. Rahul Kumar and Eric G Mercer. Verifying Communication Protocols Using Live Sequence Chart Specifications. *Electronic Notes in Theoretical Computer Science*, 250(2):33–48, 2009.
20. Martin Leucker and Christian Schallhart. A Brief Account of Runtime Verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.
21. Patrick ONeil Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. Efficient Monitoring of Parametric Context-free Patterns. *Automated Software Engineering*, 17(2):149–180, 2010.
22. Grigore Roşu and Klaus Havelund. Rewriting-based Techniques for Runtime Verification. *Automated Software Engineering*, 12(2):151–197, 2005.
23. Jocelyn Simmonds, Marsha Chechik, Shiva Nejati, Elena Litani, and Bill OFarrell. Property Patterns for Runtime Monitoring of Web Service Conversations. In *Runtime Verification*, pages 137–157. Springer, 2008.
24. Prasanna Thati and Grigore Roşu. Monitoring Algorithms for Metric Temporal Logic Specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.

A Proof of lemma 1

Lemma 1. *The language* $\left(\overline{\overline{\mathcal{B}\Sigma^* \circ \mathcal{L}(\mathfrak{c})} \circ \mathcal{L}(\mathfrak{c}') \circ \mathcal{B}\Sigma^*}\right)$ *cannot be defined by an LSC specification.*

Proof. A universal chart $\mathfrak{u} \triangleq (\mathfrak{p}, \mathfrak{m})$ defines the language $\mathcal{L}(\mathfrak{u}) \triangleq \left(\overline{\mathcal{B}\Sigma^* \circ \mathcal{L}(\mathfrak{p}) \circ \overline{\mathcal{L}(\mathfrak{m}) \circ \mathcal{B}\Sigma^*}}\right)$ [5]. The language defined by an LSC specification $\mathfrak{U} \triangleq \{\mathfrak{u}_1, \mathfrak{u}_2, ..., \mathfrak{u}_n\}$ with $\mathfrak{u}_i \triangleq (\mathfrak{p}_i, \mathfrak{m}_i)$ is $X \triangleq \bigcap\limits_{1 \le i \le n} \left(\overline{\mathcal{B}\Sigma^* \circ \mathcal{L}(\mathfrak{p}_i) \circ \overline{\mathcal{L}(\mathfrak{m}_i) \circ \mathcal{B}\Sigma^*}}\right)$. We only consider the segments $S \triangleq \bigcap\limits_{1 \le i \le n} \left(\overline{\mathcal{L}(\mathfrak{p}_i) \circ \overline{\mathcal{L}(\mathfrak{m}_i)}}\right)$, where every word in X contains a segment in S. The complement of S is $\overline{S} \triangleq \bigcup\limits_{1 \le i \le n} \left(\mathcal{L}(\mathfrak{p}_i) \circ \overline{\mathcal{L}(\mathfrak{m}_i)}\right)$. Every word in $\overline{S}$ contains a prefix $\upsilon \in \bigcup\limits_{1 \le i \le n} \mathcal{L}(\mathfrak{p}_i)$. For the segment $S' \triangleq \left(\overline{\overline{\mathcal{L}(\mathfrak{c})} \circ \mathcal{L}(\mathfrak{c}')}\right)$ of $Y \triangleq \left(\overline{\overline{\mathcal{B}\Sigma^* \circ \mathcal{L}(\mathfrak{c})} \circ \mathcal{L}(\mathfrak{c}') \circ \mathcal{B}\Sigma^*}\right)$, a word in $\overline{S'}$ contains a segment $\upsilon' \in \overline{\mathcal{L}(\mathfrak{p}_i)}$. The language of a basic chart $\mathfrak{c}$ is defined by stutter events and a finite set $Traces(\mathfrak{c})$. Therefore, the language of $\overline{\mathfrak{c}}$ is defined by stutter events and a set $(\mathcal{B}\Sigma^* \backslash Traces(\mathfrak{c}))$, which is an infinite set. Whereas, the set $\bigcup\limits_{1 \le i \le n} \mathcal{L}(\mathfrak{p}_i)$ is finite with $n < \infty$. Therefore, there exists some υ' that is not expressed by $\overline{S}$. In other words, there are some segments of words in S' that are not expressed by S. Since *i)* X consists of S and $\mathcal{B}\Sigma^*$; *ii)* Y consists of S' and $\mathcal{B}\Sigma^*$; and *iii)* S' cannote be expressed by S, the language Y cannot be epxressed by X. ■