Fundamenta Informaticae 153 (2017) 173–198
DOI 10.3233/FI-2017-1536
IOS Press

Monitoring with Parametrized Extended Life Sequence Charts

Ming Chai*
National Engineering Research Center of Rail Transportation Operation and Control System, Beijing Jiaotong University, China &
Institut für Informatik, Humboldt Universität zu Berlin, Germany
chaiming@bjtu.edu.cn

Bernd-Holger Schlingloff
Institut für Informatik, Humboldt Universität zu Berlin, Germany &
Fraunhofer FOKUS, Berlin, Germany
hs@informatik.hu-berlin.de

Abstract. Runtime verification is a lightweight formal method that checks whether an execution of a system satisfies a given property. A challenge in building a runtime verification system is to define a suitable monitoring specification language, i.e., a language that is expressive, of reasonable complexity, and easy to understand. In this paper, we extend live sequence charts (LSCs, [1]) for the specification of properties in systems monitoring. We define Parametrized extended LSCs (PeLSCs) by introducing the notions of necessary prechart, concatenation, and condition- and assignment-structure. With these PeLSCs, necessary and sufficient conditions of certain observations, and parametric properties can be specified in an intuitive way. We prove some results about the expressiveness of extended LSCs. In particular, we show that LSCs with necessary precharts are strictly more expressive than standard LSCs, and that iteration-free extended LSCs have the same expressive power as linear temporal logic (LTL). To generate monitors, we develop translations of PeLSCs into hybrid logic. We show that the complexity of the word problem of PeLSCs is linear with respect to the length of input traces, thus our formalism is well-suited for online monitoring of communicating systems.

Keywords: Runtime verification, Live sequence charts, Parameterized property, LTL, Hybrid logic

*Address for correspondence: National Engineering Research Center of Rail Transportation Operation and Control System, Beijing Jiaotong University, China.

Received December 2015; revised April 2017.

1. Introduction

Even with most advanced quality assurance techniques, correctness of complex software is hard to be guaranteed within the development phases. To solve this problem, *runtime verification* has been proposed to provide on-going protection during the operational phase. Runtime verification checks whether an execution of a computational system (formalized by a sequence of events) satisfies or violates a given correctness property. It is performed using a *monitor*. This is a device or a piece of software that observes the system under monitoring (SuM) and generates a certain verdict (*true* or *false*) as the result. Compared to model checking and testing, in runtime verification one does not try to cover all possible executions of the SuM. It detects failures of an SuM directly in its actual running environment. This avoids some shortcomings of other techniques, such as imprecision of the model in model checking, and inadequateness of the artificial environment in testing.

One of the main challenges in building a runtime verification system is to define a suitable specification language for monitoring properties. A monitoring specification language should be *expressive* and *attractive* [2]: The language should be able to express all interesting monitoring properties, and it should keep the formulations simple for simple properties. A simple formulation means that the size of the formula is small, and the formula is easily understood by users (e.g., system designers).

Over the last years, various runtime verification systems have been developed, using some forms of temporal logic, including linear temporal logic (LTL), metric temporal logic (MTL), time propositional temporal logic (TPTL) and first-order temporal logic (LTL^{FO}). Also, regular expressions and context-free grammars are supported by many runtime verification systems. Although these specification languages are expressive and technically sound for monitoring, software engineers are not familiar with them and need extensive training to use them efficiently.

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 correctness properties to be monitored.

In this paper, we investigate the use of live sequence charts (LSCs) as proposed by Damm and Harel [1] for monitoring specifications. The language of LSCs is an extension of MSCs. 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 want to specify required behaviour in order to detect failures of a system. Thus, 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 sufficient condition for the main chart. However, standard LSCs are not expressive enough for monitoring.

Consider a client/server system that allows clients to access a server, and consider the following properties, where a monitor can observe the propositional events *login* and *logout*.

($\mathbf{P1}$)*: Whenever there is a login to the server, it must be followed by a logout.*
($\mathbf{P2}$)*: A logout event can not occur, unless it is preceded by a login.*
($\mathbf{P3}$)*: Whenever a client performs a login to the server, it must logout within 200 sec.*

Observing a *login* is a sufficient condition for a *logout* in $\mathbf{P1}$, whereas it is a necessary condition in $\mathbf{P2}$. Property $\mathbf{P3}$ involves parameters *time* and *clients*, which are on infinite and unspecified domains. The expected executions of the first two properties can be formalized by the following regular expressions $\mathbf{L1}$ and $\mathbf{L2}$, respectively. Let Σ_τ be $\Sigma \setminus \{login, logout\}$, i.e., the set of all events which are not a login or logout.

$$\mathbf{L1} \triangleq \overline{\Sigma_\tau^* \circ login \circ \Sigma_\tau^* \circ \overline{\Sigma_\tau^* \circ logout \circ \Sigma_\tau^*}}$$
$$\mathbf{L2} \triangleq \overline{\Sigma_\tau^* \circ login \circ \Sigma_\tau^* \circ \Sigma_\tau^* \circ logout \circ \Sigma_\tau^*}$$

For the property ($\mathbf{P3}$), assume that each of the *login* and *logout* events carries a client name and a time stamp. An execution of this system can be formalized by a sequence of parametrized events. Each of the propositional events has two parameters *client_id* (`id`) and *time_stamp* (`time`). With these definitions, property (**P3**) can be written more formally as follows:

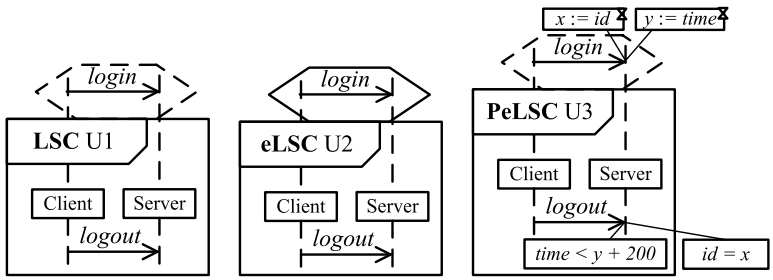
Whenever a *login* event happens with $(\texttt{id} = x)$ and $(\texttt{time} = y)$, a *logout* event with $(\texttt{id} = x')$ and $(\texttt{time} = y')$ should occur afterwards, where $(x' = x)$ and $(y' \leq (y + 200))$.

As has been proved in our previous work [4], necessary conditions of statements (i.e., with the language of $\mathbf{L2}$) cannot be expressed by a finite set of negation-free universal LSCs. Furthermore, standard LSCs cannot express parametric properties (e.g., the property $\mathbf{P3}$), where the correctness depends on both the temporal relations of events and data carried by the events [5]. One possible workaround for this shortage is to formalize each assignment of data with a unique atomic proposition. However, since the domain of data can be infinite or unknown, this approach is not feasible in general.

To express properties with necessary conditions, we define extended LSCs (eLSCs) by introducing modal precharts. That is, we distinguish between precharts that are *necessary* and those that are *sufficient* conditions of main charts. For dealing with parametric properties, we further define parametrized eLSCs (PeLSCs) by introducing *assignment-* and *condition-structures*. An assignment structure is used to store an arbitrary parameter value, and a condition structure is used to express constraints on such values.

The PeLSCs of Fig. 1 specify the three properties above. The chart $U1$ is a standard LSC formalizing ($\mathbf{P1}$). Property $\mathbf{P2}$ cannot be formalized with LSCs; an eLSC for it is $U2$. Property **P3** involves parameters `id` and `time` and is expressed with the PeLSC $U3$.

In this paper, we analyze the expressiveness of various sub-languages of PeLSCs. For generating monitors from such specifications, we translate PeLSCs into Hybrid Logic (HL) [6]. A monitor essentially solves the word-problem: given a trace, decide whether the trace is in the language defined



(a) LSC for **P1** (b) eLSC for **P2** (c) PeLSC for **P3**

Figure 1. Examples: extended LSCs for properties of a client/server system

by a monitoring property. As one main result of this paper, we prove that the complexity of the word-problem of PeLSCs is linear if the propositions in the condition structures express only comparisons of parameter values. Thus, monitoring can be done on-line, while the SuM is running.

The rest of the paper is organized as follows. Section 2 outlines related work. Section 3 introduces parametrized eLSCs (PeLSCs), including their syntax and trace-based semantics. Section 4 presents results on the expressiveness of eLSCs. Section 5 gives a translation of PeLSCs into HL, and proves the complexity of the word problem of PeLSCs . Section 6 contains some conclusions and hints for future work.

2. Related work

The problem of analysing MSCs and related formalisms has a long tradition. Alur et. al. study the model checking problems of MSCs, MSC-graphs and hierarchical MSC-graphs [7]. They show that the complexity of model checking for MSCs and synchronous MSC-graphs is coNP-complete, and for asynchronous MSC-graphs it is undecidable. There are some runtime verification systems using UML-SDs for specifying monitoring properties. Simmonds et. al. use UML-SD to monitor Web Service Conversations [8]. Ciraci et. al. propose a technique to check the correspondence between UML-SD models and implementations [9]. Unfortunately, MSCs and UML-SDs are not expressive enough for monitoring because they cannot distinguish between required and allowed behaviours of a system.

Harel et. al. propose a play-in/play-out approach [10] for modelling with LSCs. 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 [11]. Kugler et. al. [12] 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 [13]. Kumar et. al. extend the LSC language with Kleene star, subcharts, and hierarchical charts [14]. They translate an extended LSC based communication protocol specification into an automaton, and verify some properties with

the resulting automaton. Since all existing work is based on the standard LSC language, none of the above papers addresses the expressiveness problem discussed in this paper.

Our parametric extension of LSCs is inspired by the treatment of time in live sequence charts proposed by Harel et. al. [15]. There, a time constraint in LSCs is defined by a combination of assignment- and condition structures. In contrast, we provide a more general notation for arbitrary data parameters. There are several other runtime verification approaches for handling parametrized events. The EAGLE logic [16], which is a linear μ-calculus, is one of the first logics in runtime verification for specifying and monitoring data-relevant properties. Although EAGLE has a rather high expressiveness, it also has high computational costs [17]. To avoid this problem, other rule-based methods have been introduced. They are based on MetateM [18] and the Rete algorithm [19]. MetateM provides a framework of executing temporal formulae and Rete is an efficient algorithm for matching patterns with objects. Inspired by MetateM, RuleR is an efficient rule-based monitoring system that can compile various temporal logics [17]. LogicFire is an internal domain specification language for artificial intelligence on basis of Rete [20]. These rule-based runtime verification systems have a high performance. However, they are not attractive for practical applications. The language of PeLSCs has comparable expressiveness, and monitors can be generated automatically with the translation algorithm.

TraceMatches [21] is essentially a language of regular expressions. It extends the language of AspectJ [22] by introducing free variables in the matching patterns. TraceContract is an API for trace analysis, implemented in Scala, which is able to express parametric properties with temporal logic [23]. Monitoring oriented programming (MOP) is an efficient and generic monitoring framework that integrates various specification languages [24]. In particular, JavaMOP deals with parametric specification and monitoring using TraceMatches [25]. TraceMatches and JavaMOP are defined on the basis of *trace slicing*, which translates parametrized events into propositional events. With trace slicing, the problem of checking parametrized event traces is translated into a (standard) propositional word problem. Although JavaMOP has a high performance, to our opinion its expressiveness is insufficient. As pointed out in [26], trace slicing can only handle traces where all events with the same name carry the same parameters. Our PeLSCs based approach overcomes this shortage by using formula rewriting algorithms.

Various extensions of LTL have been proposed for parametric monitoring. If time is the only parameter, properties can be formalized with real-time logics such as TLTL [27], MTL [28] and TPTL [29]. For other parameters, first order extensions of LTL have been introduced. Parametrized LTL [30] contains a binary binding operator, and is further translated into parametrized automata for monitoring. First-order temporal logic LTL^{FO} includes both first-order and temporal connectives [31]. For monitoring LTL^{FO} an algorithm using a spawning automaton has been developed [32]. A domain-specific language for monitoring the exchange of XML messages of a web service is LTL^{FO+} [33]. This language has a lower complexity than full first order temporal logic. However, its expressiveness is limited by only allowing to express equivalence of variables. Metric First-order Temporal Logic (MFOTL) adds quantifiers to MTL [34], and has been used for monitoring data applications [35]. An MFOTL monitoring system has been built based on a trace decomposing technique, which may introduce additional errors/mistakes. Similar to the languages of automata, all these temporal logics have difficulties in specifying concurrency properties. The language of PeLSCs can avoid these shortcom-

ings. PeLSCs have a richer expressiveness than LTL^{FO+} by allowing to express general comparisons of terms. Nonetheless, the word problem for certain PeLSCs is still linear with respect to the size of traces.

3. Parametrized extended live sequence charts (PeLSC)

Message sequence charts (MSCs) represent communication protocols between actors. They have been standardized by the ITU in the SDL language, and adapted as sequence diagrams in UML. Both SDL and UML are widely used in industry for the modelling of computational systems.

To differentiate between necessary and possible behaviours of a system, Damm et al. extended the MSC language to live sequence charts (LSCs) [1]. LSCs have been used to model systems in various domains, e.g., railway [36], telecommunication [37], and biology [38].

3.1. Basic charts

The LSC language can be defined on the basis of *basic charts*. A basic chart is visually similar to an MSC. It specifies the exchange of messages among a set of actors. Each actor is represented by its lifeline (drawn as a vertical dashed line), and each message exchange is represented by a solid arrow between two lifelines. With each message exchange, there are two actions associated: the action of sending the message and the action of receiving it. Each action occurs at a unique position in a lifeline. Intuitively, a basic chart describes a partial order between these actions as follows.

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

Formally, we define basic charts as follows. Given a set AP of *atomic propositions*, let $M \triangleq 2^{AP}$ be a set of *messages*, and $\Sigma \triangleq (M \times \{!, ?\})$ an alphabet of *events*. That is, an event e is either $m!$ (indicating that message m is sent) or $m?$ (indicating that m is received). A *lifeline* l is a finite (possibly empty) sequence of events $l \triangleq (e_1, e_2, ..., e_n)$, i.e., a word over Σ^*. Consider a k-tuple of lifelines $\mathbb{L} \triangleq \langle l_1, ..., l_k \rangle$ with $l_i = (e_{i1}, ..., e_{in'})$. We say that the event e *occurs* at the *location* (i, j) in $\mathbb{L}$ if $e = e_{ij}$. An *event occurrence* $\mathfrak{o} \triangleq (e, i, j)$ is a tuple consisting of an event $e \in \Sigma$ and the location (i, j) of e in $\mathbb{L}$.

A *communication* $\langle (m!, i, j), (m?, i', j') \rangle$ is a pair of two event occurrences representing sending and receiving of the same message m. A basic chart C is an n-tuple of lifelines $\mathbb{L}$ together with a set $\mathbb{C}$ of *communications*, such that each event occurrence is contained in at most one communication. Given a basic chart C and a sending event occurrence $\mathfrak{o} = (m!, i, j)$ in a communication of $\mathbb{C}$, we define $match(\mathfrak{o}) \triangleq (m?, i', j')$ to be the matching receiving event occurrence. An event occurrence in a basic chart does not have to be part of a communication; it is possible that only the sending or the receiving of a message occurs in the chart. In addition, an event is allowed to occur multiple times in a basic chart, i.e., a basic chart can express that a message is repeatedly exchanged. However, each event occurrence is unique.

In accordance to the above informal description, we define the partial order relation $\prec$ induced by a chart C on its event occurrences to be the smallest relation satisfying

1. For any $1 \leq j < |l_i|$ with l_i being a lifeline in C, it holds that $(e, i, j) \prec (e', i, j+1)$, and
2. for any sending event occurrence $\mathfrak{o}$ in a communication of C, it holds that $\mathfrak{o} \prec match(\mathfrak{o})$.

We admit the *non-degeneracy* assumption proposed by Alur et. al. [39]: a basic chart cannot reverse the receiving order of two identical messages sent by some lifeline. Formally, a basic chart is *non-degenerate* if and only if for any two sending event occurrences $\mathfrak{o}_1 = (m!, i_1, j_1)$ and $\mathfrak{o}_2 = (m!, i_2, j_2)$ (with the same message m), if $\mathfrak{o}_1 \prec \mathfrak{o}_2$ then $match(\mathfrak{o}_1) \prec match(\mathfrak{o}_2)$. Henceforth, we consider non-degenerate charts only.

For any event occurrence $\mathfrak{o} = (e, i, j)$, let *evt*$(\mathfrak{o}) \triangleq e$ be the event of $\mathfrak{o}$. We denote the set of events appearing in a chart C by $\mathsf{Evt}(C)$ (where $\mathsf{Evt}(C) \subseteq \Sigma$), and the set of event occurrences in C with $\mathsf{EOcc}(C)$. Given $\mathsf{EOcc}(C) = \{\mathfrak{o}_1, \mathfrak{o}_2, ..., \mathfrak{o}_n\}$, and let $f : \{1, ..., n\} \rightarrow \{1, ..., n\}$ be a (bijection) permutation. A sequence $(\mathfrak{o}_{f(1)}, \mathfrak{o}_{f(2)}, ..., \mathfrak{o}_{f(n)})$ of event occurrences is called *consistent with* C, if $\mathfrak{o}_{f(i)} \prec \mathfrak{o}_{f(j)}$ implies $i < j$ for all $1 \leq i, j \leq n$. That is, the total order of event occurrences in the sequence must respect the partial order induced by the chart. An *event trace* σ is just a finite sequence of events, i.e., a word over Σ^*. We say that σ is *defined* by the basic chart C, if σ is the projection of some event occurrence sequence consistent with C to its events. Formally, the set of traces *Traces*(C) defined by C is given as follows:

$$\textit{Traces}(C) \triangleq \{(e_1, ..., e_n) \mid \exists(\mathfrak{o}_1, ..., \mathfrak{o}_n)\ s.t\ e_i = evt(\mathfrak{o}_i)\ and\ (\mathfrak{o}_1, ..., \mathfrak{o}_n)\ consistent\ with\ C\}.$$

Let Σ_τ be a set of *silent events*. For each basic chart C, the language $\mathcal{L}(C, \Sigma_\tau)$ consists of all traces defined by C, interleaved with sequences of silent events from Σ_τ:

$$\mathcal{L}(C, \Sigma_\tau) \triangleq \{(\tau_0^*, e_1, \tau_1^*, e_2, ..., \tau_{n-1}^*, e_n, \tau_n^*) \mid (e_1, e_2, ..., e_n) \in \textit{Traces}(C), \tau_i^* \in \Sigma_\tau^*\}.$$

The language $\mathcal{L}(C)$ of a basic chart is defined to be $\mathcal{L}(C, \Sigma \backslash \mathsf{Evt}(C))$. A trace σ is *admitted* by a basic chart C (denoted by $\sigma \Vdash C$), if $\sigma \in \mathcal{L}(C)$.

3.2. Extended LSCs

A universal chart consists of two basic charts: a *prechart* (Pch, drawn as a hexagon) and a *main chart* (Mch, drawn within a solid rectangle). We define extended LSCs (eLSCs) by introducing two categories of precharts: *sufficient precharts* (drawn within surrounding hashed hexagons) and *necessary precharts* (drawn within surrounding solid hexagons).

Definition 3.1. An *extended Live Sequence Chart* (eLSC) is a tuple $U \triangleq (Pch, Mch, Cate)$, where Pch and Mch are basic charts, and $Cate \in \{Suff, Nec\}$ is the *category* (sufficient or necessary) of the prechart.

Intuitively, an eLSC with a sufficient prechart $(Pch, Mch, Suff)$ specifies all traces composed of two segments such that, if the first segment is admitted by the prechart, then the second must be admitted by the main chart; and an eLSC with a necessary prechart (Pch, Mch, Nec) specifies all

traces composed of two segments such that, the first segment cannot be admitted by the prechart, unless the second is admitted by the main chart.

Given an eLSC $U = (Pch, Mch, Cate)$, and the set $\Sigma_\tau = (\Sigma \backslash (Evt(Pch) \cup Evt(Mch)))$ of silent events of U, the languages $\mathcal{L}(Pch)$ and $\mathcal{L}(Mch)$ are defined to be $\mathcal{L}(Pch, \Sigma_\tau)$ and $\mathcal{L}(Mch, \Sigma_\tau)$, respectively. For languages $\mathcal{L}$ and $\mathcal{L}'$, let $\overline{\mathcal{L}}$ be the complement of $\mathcal{L}$ and $(\mathcal{L} \circ \mathcal{L}')$ be the concatenation of $\mathcal{L}$ and $\mathcal{L}'$. The semantics of eLSCs is defined as follows.

Definition 3.2. The language of an eLSC $U \triangleq (Pch, Mch, Cate)$ is
$\mathcal{L}(U) \triangleq \overline{\mathcal{L}(Pch) \circ \overline{\mathcal{L}(Mch)}}$, if $Cate = \mathit{Suff}$; and
$\mathcal{L}(U) \triangleq \overline{\overline{\mathcal{L}(Pch)} \circ \mathcal{L}(Mch)}$, if $Cate = Nec$.

The concatenation of two eLSCs U and U' represents the sequencing of the charts. It essentially introduces additional sequencing restrictions on executions of the charts. The concatenation $(U \to U')$ is intuitively understood as "an execution of U is followed by an execution of U'". A trace σ composed of two segments υ and υ' (i.e., $\sigma \triangleq \upsilon\upsilon'$) is in the language of $(U \to U')$ if and only if υ and υ' are admitted by U and U', respectively. The set of event traces defined by $(U \to U')$ is

$$Traces(U \to U') \triangleq Traces(U) \circ Traces(U'),$$

Given $\Sigma^\tau_{(U \to U')} = \Sigma \backslash (\mathsf{Evt}(U) \cup \mathsf{Evt}(U'))$, the language of $(U \to U')$ is then defined by

$$\mathcal{L}(U \to U', \Sigma^\tau_{(U \to U')}) \triangleq \{(\tau^*_{(U \to U')}, e_1, \tau^*_{(U \to U')}, e_2, ..., \tau^*_{(U \to U')}, e_n, \tau^*_{(U \to U')})\},$$

where $(e_1, ..., e_n) \in Traces(U \to U')$ and $\tau^*_{(U \to U')} \in (\Sigma^\tau_{(U \to U')})^*$. As above, the language $\mathcal{L}(U \to U')$ of the concatenation of U and U' is defined to be $\mathcal{L}(U \to U', \Sigma^\tau_{(U \to U')})$.

Remark 3.3. Given an empty chart $U_\emptyset \triangleq \emptyset$ (i.e., a chart with no communications) and any chart U over an alphabet $\Sigma = \mathsf{Evt}(U)$, it holds that $\mathcal{L}(U \to U_\emptyset) = \mathcal{L}(U_\emptyset \to U) = \mathcal{L}(U)$.

Similarly, the language $\mathcal{L}(U^*)$ of the iteration of universal charts can be defined as follows.

$$\mathcal{L}(U^0) \triangleq \emptyset,$$
$$\mathcal{L}(U^n) \triangleq (\mathcal{L}(U) \to \mathcal{L}(U^{n-1})), \text{ and}$$
$$\mathcal{L}(U^*) \triangleq \textstyle\bigcup_{i \in \mathbb{N}} \mathcal{L}(U^i).$$

An *eLSC specification* $\mathfrak{U}$ is a finite set of (possible concatenated) eLSCs. The language of $\mathfrak{U}$ is $\mathcal{L}(\mathfrak{U}) \triangleq (\bigcap_{U \in \mathfrak{U}} \mathcal{L}(U))$ with the silent events $\tau_\mathfrak{U} \in \Sigma \backslash (\bigcup_{U \in \mathfrak{U}} \mathsf{Evt}(U))$.

3.3. Parametrized eLSCs

We define parametrized eLSCs (PeLSCs) by introducing *assignment structures* and *condition structures* into eLSCs. An assignment structure is given by $v := s$ with v being a variable and s being the name of a parameter. Intuitively, this means that the variable v is evaluated to the value of a parameter named s. In the graphical representation, an assignment structure is surrounded by a rectangle with a

Figure 2. Notation for assignment structure (left) and condition structure (right)

sandglass icon at the top right corner. A condition structure is a simple numerical proposition π with variables, parameters and values; the formal definition is given below. In a PeLSC, it is surrounded by a rectangle. The graphical notations for assignment- and condition-structures are shown in Fig. 2.

In a PeLSC, assignment- and condition-structures combine naturally with event occurrences. Intuitively, an assignment structure stores the value of a parameter carried by the associated event occurrence; and a condition structure expresses the restrictions of parameters carried by the event occurrence. Formally, let $\mathcal{N} \triangleq \{s_1, s_2, ...\}$ be a countable set of *nominals* and $\mathcal{D} \triangleq \{d_1, d_2, ...\}$ be a *domain* (e.g., integers or reals). A *parameter* is a pair $p \triangleq \langle s, d\rangle$ from $\mathcal{N} \times \mathcal{D}$, where s is the *name* of p and d is the *value* of p.

To formally define the syntax of PeLSCs, assume we are given a countable set of variables $\mathcal{V} \triangleq \{v_1, v_2, ...\}$. An assignment structure is the defined to be a tuple $\mathsf{assi} \triangleq \langle v, s, \mathfrak{o}\rangle$, where v is a variable, s is the name of a parameter and $\mathfrak{o}$ is an event occurrence in C. A condition structure is a pair $\mathsf{cond} \triangleq \langle \pi, \mathfrak{o}\rangle$, where π is a *simple numerical proposition*. This notion is defined as follows: Each variable, parameter name and domain element is a simple numerical term. If t is a simple numerical term and c is a domain element, than $(t + c)$ is a simple numerical term. If t_1 and t_2 are simple numerical terms, then $t_1 < t_2$ and $t_1 = t_2$ are simple numerical propositions. Finally, if π_1 and π_2 are simple numerical propositions, then so are $(\pi_1 \wedge \pi_2)$, $(\pi_1 \vee \pi_2)$ and $\neg\pi_1$. Examples are $v \leq 5$, $s = 3$ and $s < (v + 10)$.

Assume that Σ is an alphabet of events and $\mathcal{N}$ and $\mathcal{D}$ are nominals and domain, as above. A *parametrized event* is a pair $\mathfrak{w} \triangleq \langle e, \mathcal{P}\rangle$, where $e \in \Sigma$ is an event and $\mathcal{P} \in 2^{\mathcal{N}\times\mathcal{D}}$ is a set of parameters. For example, in the chart P3 above, a parametrized event could be $(\mathit{login}?, \{(\mathit{id}, 1101), (\mathit{time}, 25)\})$. We assume that each parameter name in $\mathcal{P}$ is unique, i.e., for all $p, p' \in \mathcal{P}$ with $p \neq p'$ it holds that the name of p is different from the name of p'. Parametrized events can be used as basic constituents of an eLSC as above, resulting in *parametrized* eLSCs. Formally, a PeLSC is defined as follows.

Definition 3.4. A PeLSC is a tuple $PU \triangleq (U, \mathsf{COND}, \mathsf{ASSI})$, where U is an eLSC on parametrized events, and COND and ASSI are sets of condition structures and assignment structures, respectively.

For the semantics of PeLSCs, a parametrized event trace $t \triangleq (\mathfrak{w}_1, ..., \mathfrak{w}_n)$ is a (finite) sequence of parametrized events. A PeLSC $(U, \mathsf{COND}, \mathsf{ASSI})$ defines a parametrized language (i.e., a set of parametrized event traces). Intuitively, a parametrized event trace $t = (\langle e_1, \mathcal{P}_1\rangle, ..., \langle e_n, \mathcal{P}_n\rangle)$ is admitted by a PeLSC if $(e_1, ..., e_n)$ is admitted by U, and all propositions in COND are evaluated to *true* with respect to the parameter values in $(\mathcal{P}_1, ..., \mathcal{P}_n)$. To formally define the latter notion, we have to define the evaluation of a simple numerical proposition π in a parametrized event trace t at a particular position $\mathfrak{w}_i$.

If π does not contain any variables, then the value of π can simply be obtained by replacing all parameter names by their respective values and applying basic arithmetic. If π contains a variable v, then it can be evaluated to a boolean value (*true* or *false*) only if v has been assigned a certain value

earlier. Therefore, we require our PeLSCs to satisfy an additional *non-ambiguity* assumption. We say a PeLSC is non-ambiguous, if for any condition structure $\mathsf{cond} = (\pi(s', v), \mathfrak{o}')$ in COND, there exists exactly one assignment structure $\mathsf{assi}(v, s, \mathfrak{o}) \in \mathsf{ASSI}$ such that $\mathfrak{o} \prec \mathfrak{o}'$. Under this assumption, a value of a variable at a given event occurrence refers to a specific value of a certain parameter in a precisely defined "earlier" event occurrence. The non-ambiguity assumption can be checked statically on the structure of the PeLSC. Given a parametrized event trace of a non-ambiguous PeLSC and a particular event, each simple numerical proposition can be evaluated to a boolean value by replacing each variable by its value in the associated event.

A PeLSC PU defines all parametrized event traces $(\langle e_1, \mathcal{P}_1\rangle, ..., \langle e_n, \mathcal{P}_n\rangle)$ such that

- for any $1 \leq i \leq n$, it holds that $e_i = evt(\mathfrak{o}_i)$ with $(\mathfrak{o}_1, ..., \mathfrak{o}_n)$ being consistent with U; and
- all simple numerical propositions in PU are evaluated to *true*.

By *PTraces*(PU) we denote the set of parametrized event traces defined by PU. A parametrized event $w_\tau \triangleq \langle \tau, \mathcal{P}_\tau\rangle$ is a *silent* event of PU, if $\tau \in (\Sigma \backslash \mathsf{Evt}(U))$ is a silent event of U.

Definition 3.5. The parametrized language defined by a PeLSC PU is

$$\mathcal{PL}(PU) \triangleq \{(w_\tau^*, w_1, w_\tau^*, ..., w_\tau^*, w_n, w_\tau^*)\},$$

where $(w_1, ..., w_n) \in PTraces(PU)$, and w_τ^* are finite (possible empty) sequences of silent events.

4. Expressiveness of eLSCs

We assume that an eLSC specification $\mathfrak{U}$ is defined over a given alphabet $\Sigma_{\mathfrak{U}}$. If all events observed from the system under monitoring are in $\Sigma_{\mathfrak{U}}$, the language defined by the specification has no silent events. We call such a specification a *silence-free eLSC specification*. We now investigate the expressive power of silence-free eLSC specifications.

Theorem 4.1. eLSCs are strictly more expressive than standard (negation-free) LSCs.

Proof:
Clearly, standard LSCs are included in eLSCs: Any LSC (Pch, Mch) defines the same language as the eLSC $(Pch, Mch, \mathit{Suff})$.

For the strict inclusion, consider, for instance, the language $X \triangleq (\Sigma^*(aa)\Sigma^*)$ over the alphabet $\Sigma = \{a, b\}$. This language can be defined by the silence-free eLSC $(U_1 \rightarrow U_2)$, see Fig. 3. The language of U_1 is $\mathcal{L}(U_1) = \overline{\overline{aa} \circ b} = (\overline{\Sigma^* b} \cup aab) = (\Sigma^* a \cup \varepsilon \cup aab)$, and the language of U_2 is $\mathcal{L}(U_2) = \overline{b \circ \overline{aa}} = (\overline{b\Sigma^*} \cup baa) = (a\Sigma^* \cup \varepsilon \cup baa)$. It can be seen that $\mathcal{L}(U_1) \circ \mathcal{L}(U_2) = \Sigma^*(aa)\Sigma^*$. Therefore, the language of $(U_1 \rightarrow U_2)$ is X. As proved by Bontemps [40], this language cannot be defined by an LSC. □

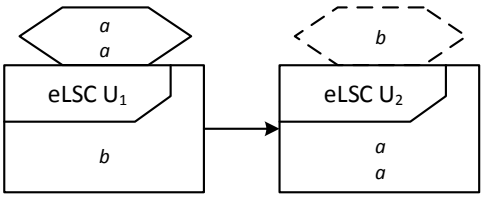


Figure 3. eLSCs expressing $\Sigma^*(aa)\Sigma^*$ with $\Sigma = \{a, b\}$

In our previous work [4], we proved the slightly more general statement that given a prechart Pch with $\mathcal{L}(Pch) = \{(abc, acb)\}$ and a main chart Mch with $\mathcal{L}(Mch) = \{cd, dc\}$, the eLSC language $\overline{\overline{\mathcal{L}(Pch)} \circ \mathcal{L}(Mch)}$ of (Pch, Mch, Nec) cannot be expressed by a finite set of negation-free LSCs. This proof does not rely on the assumption that eLSCs are silence-free.

To generate monitors, in the following section we translate eLSCs to linear temporal logic (LTL) [1] over finite traces. We first show the expressiveness of *iteration-free* eLSCs (i.e., the subclass of eLSCs with no iteration operators) as follows.

Theorem 4.2. The language of iteration-free eLSCs is at most as expressive as LTL.

Proof:
We have to show that any eLSC specification can be translated into an equivalent LTL formula. The propositions from which the formula is built are the events appearing in the eLSC. We first prove that any universal eLSC without iteration can be translated into an equivalent LTL formula. Consider a basic chart C, the set $\mathsf{Evt}(C) \triangleq \{e_1, e_2, ..., e_m\}$ of events appearing in C, and a trace $\sigma \triangleq (e_1, e_2, ..., e_n)$ in the set $Traces(C)$. The propositional formula

$$\nu_{\mathsf{Evt}(C)} \triangleq (\neg e_1 \wedge \neg e_2 \wedge \cdots \wedge \neg e_m)$$

states that none of the events in $\mathsf{Evt}(C)$ is executed. We then define an LTL formula

$$\alpha(\sigma, \mathsf{Evt}(C)) \triangleq \nu_{\mathsf{Evt}(C)}\ \mathbf{U}\ (e_1 \wedge (\mathbf{X}(\mathfrak{n}_{\mathsf{Evt}(C)}\ \mathbf{U}\ (e_2 \wedge \cdots \wedge (\nu_{\mathsf{Evt}(C)}\ \mathbf{U}\ (e_n\ \wedge\ \Gamma_{\mathsf{Evt}(C)}) \cdots)$$

with $\Gamma_{\mathsf{Evt}(C)} \triangleq \mathbf{X}\ \mathbf{G}\ \nu_{\mathsf{Evt}(C)}$. The formula $\alpha(\sigma, \mathsf{Evt}(C))$ states that the trace σ is executed, where $\Gamma_{\mathsf{Evt}(C)}$ states that no event from $\mathsf{Evt}(C)$ can occur after the execution of σ.

Let **XOR** be the boolean *exclusive-or* connective. The formula

$$\mathcal{F}(C_1, \mathsf{Evt}(C_2)) \triangleq \underset{\sigma \in Traces(C_1)}{\mathbf{XOR}} (\alpha(\sigma, \mathsf{Evt}(C_2)))$$

states that exactly one of the traces in C_1 is executed, with no silent events from $\mathsf{Evt}(C_2)$. For a basic chart C, in [13] it is shown that the formula $\mathcal{F}(C, \mathsf{Evt}(C))$ defines exactly the same language as $\mathcal{L}(C)$.

[1] We use symbols **G**, **F**, **U**, **W** and **X** for the LTL operators *always*, *sometimes*, *until*, *weak until* and *next*, respectively.

In addition, we define

$$\mathcal{F}((C_1 \to C_2), \mathsf{Evt}(C)) \triangleq \underset{\sigma \in Traces(C_1) \circ Traces(C_2)}{\mathbf{XOR}} (\alpha(\mathsf{Evt}(C), \sigma))$$

for concatenation of charts. Note that this disjunction is finite, since $Traces(C_1)$ and $Traces(C_2)$ are both finite; with iteration this would not work.

Given a universal chart $U \triangleq (Pch, Mch, Cate)$ with $\mathsf{Evt}(U) = (\mathsf{Evt}(Pch) \cup \mathsf{Evt}(Mch))$, the formulae

$$\Phi_{Suff}(U, \mathsf{Evt}(U)) \triangleq (\mathcal{F}(Pch, \mathsf{Evt}(U)) \Rightarrow (\mathcal{F}((Pch \to Mch), \mathsf{Evt}(U)))), \text{ and}$$
$$\Phi_{Nec}(U, \mathsf{Evt}(U)) \triangleq (\neg(\mathcal{F}(Pch, \mathsf{Evt}(U)) \Rightarrow \neg(\mathcal{F}((Pch \to Mch), \mathsf{Evt}(U))))$$

define exactly the same languages as $\mathcal{L}(Pch, U, Suff)$ and $\mathcal{L}(Pch, U, Nec)$, respectively. □

Remark that this proof does not require the eLSC specification to be silence-free. For theoretical reasons, we now introduce a further slight extension of eLSCs. Let $\top$ be a special symbol in an eLSC U to specify any event in the alphabet Σ_U of U. In the graphical notation, the symbol $\top$ is surrounded by a rectangle. The language defined by $\top$ in an eLSC U is $\mathcal{L}(\top, U) \triangleq \bigvee_{e \in \Sigma_U} e$. From the symbol $\top$, we define $\top^*$ and $\top^+$ to be any event sequence and any non-empty event sequence in Σ_U, respectively. Formally, the languages defined by $\top^*$ and $\top^+$ are $\mathcal{L}(\top^*, U) \triangleq \Sigma_U^*$ and $\mathcal{L}(\top^+, U) \triangleq \Sigma_U^+$, respectively. In the graphical representation, the symbol $\top^*$ (resp. $\top^+$) can be placed at either the bottom of a prechart or the top of a main chart. The language defined by the former chart is $(\mathcal{L}(Pch) \circ \Sigma^*)$ (resp. $(\mathcal{L}(Pch) \circ \Sigma^+)$)); and the language defined by the latter one is $(\Sigma^* \circ \mathcal{L}(Mch))$ (resp. $(\Sigma^+ \circ \mathcal{L}(Mch))$). Using the $\top$ symbol, we subsequently show that eLSCs are closed under finite union and negation. As a remark, we note that we could have used a finite union operator (or "choice" operator as in UML) $\bigcup$ instead of $\top$ as an additional language element in our language; given a PeLSC over an alphabet Σ, the symbol $\top$ is defined as $\bigcup_{e \in \Sigma} e$. Omitting both $\top$ and $\bigcup$, however, does not allow us to prove the subsequent expressivity results.

Theorem 4.3. Given a set of eLSCs $\mathfrak{U} = \{U_1, ..., U_n\}$ without silent events (but possibly using $\top^*$), the complement of $\mathcal{L}(\mathfrak{U})$ can be defined by a set of eLSCs (again, possibly using $\top^*$).

Proof:
Given a set $\mathfrak{U} = \{U_1, ..., U_n\}$ of eLSCs, the complement of $\mathfrak{U}$ can be constructed as follows. In the following part of this section, we denote the languages $\mathcal{L}(Pch_i)$ and $\mathcal{L}(Mch_i)$ by P_i and M_i, respectively.

CASE I: Let all eLSCs in $\mathfrak{U}$ be with sufficient precharts, and no eLSC in $\mathfrak{U}$ have a $\top^*$ in it. The language of $\mathfrak{U}$ is as follows.

$$\begin{aligned}\mathcal{L}(\mathfrak{U}) &= \bigcap_{1 \le i \le n} (\overline{P_i \circ \Sigma^*} \cup (P_i \circ M_i)) \\ &= \Big(\bigcap_{1 \le i \le n} \overline{P_i \circ \Sigma^*} \Big) \cup \Big(\bigcap_{1 \le i \le n} (P_i \circ M_i) \Big) \cup \bigcup_{1 \le k < n} \Big(\bigcap_{1 \le i \le k} \overline{P_i \circ \Sigma^*} \cap \bigcap_{k+1 \le j \le n} (P_j \circ M_j) \Big)\end{aligned} \tag{1}$$

Given a basic chart C, we denote the size of C by $|C|$, i.e., the number of events appearing in a prechart or a main chart (excluding $\top^*$). Given $x_i = |Pch_i|$ and $x = \max\limits_{1\le i\le n} |x_i|$, the subformula $(\bigcap_{1\le i\le n} \overline{P_i \circ \Sigma^*})$ of the above (1) equals to

$$\overline{\bigcup_{1\le i\le n} (P_i \circ \Sigma^*)} = \overline{(\bigcup_{1\le i\le n} P_i) \circ \Sigma^*} = \Big(\Big(\Sigma^x \backslash (\bigcup_{1\le i\le n} (P_i \circ \Sigma^{(x-x_i)}))\Big) \circ \Sigma^*\Big) \cup \varepsilon.$$

In the following part of the proof, we use the abbreviations

$$A(i,k) \triangleq \Sigma^x \backslash (\bigcup_{i\le j\le k} (P_j \circ \Sigma^{(x-x_j)})),$$

$$B(i,k) \triangleq \bigcap_{i\le j\le k} (P_j \circ M_j),$$

and

$$D(k) \triangleq (\bigcap_{1\le i\le k} \overline{P_i \circ \Sigma^*} \cap \bigcap_{k+1\le j\le n} (P_j \circ M_j)).$$

Thus, $\mathcal{L}(\mathfrak{U}) = (A(1,n) \circ \Sigma^*) \cup \varepsilon \cup B(1,n) \cup \bigcup_{1\le k\le n} D(k)$. As a remark, all traces in $A(i,k)$ have length x. Moreover, all traces in $B(i,k)$ have the same length. This holds because no chart has $\top^*$ in it. Let $z_{(i,k)}$ be the length of a trace in $B(i,k)$. For any $1 \le k < n$, it holds that

$$D(k) = \begin{cases} (A(1,k) \circ \Sigma^{(z_{(k+1,n)}-x)}) \cap B(k+1,n) & \text{if } z_{(k+1,n)} \ge x \\ \emptyset & \text{otherwise} \end{cases}$$

It can be seen that $A(1,n)$, $B(1,n)$ and all $D(k)$ have a finite number of traces. This holds because $A(1,n)$ defines a subset of Σ^x; $B(1,n)$ defines a finite number of intersections of $(P_j \circ M_j)$, where P_j and M_j are both finite; and $D(k)$ is defined by the intersection of some $A(i,k)$ and $B(i,k)$. Therefore, the language of $\mathfrak{U}$ can be written as a union of traces as follows.

$$\mathcal{L}(\mathfrak{U}) = \Big(\bigcup_{\tau_a\in A(1,n)} (\tau_a \circ \Sigma^*)\Big) \cup \varepsilon \cup \Big(\bigcup_{\tau_b\in B(1,n)} \tau_b\Big) \cup \Big(\bigcup_{1\le k<n} \bigcup_{\tau_d\in D(k)} \tau_d\Big).$$

The complement of $\mathfrak{U}$ is

$$\overline{\mathcal{L}(\mathfrak{U})} = \Big(\bigcap_{\tau_a\in A(1,n)} (\overline{\tau_a \circ \Sigma^*})\Big) \cap \Sigma^+ \cap \Big(\bigcap_{\tau_b\in B(1,n)} \overline{\tau_b}\Big) \cap \Big(\bigcap_{1\le k<n} \bigcap_{\tau_d\in D(k)} \overline{\tau_d}\Big). \tag{2}$$

For any single trace τ, the complement $\overline{\tau}$ of τ can be defined by an eLSC $U_{neg}(\tau)$ as shown in Fig. 4(a). The language of $U_{neg}(\tau)$ is $\mathcal{L}(U_{neg}(\tau)) = \overline{\tau \circ \overline{\Sigma^+}} = \overline{\tau}$. The eLSC $U(\Sigma^+)$ in Fig. 4(b) defines Σ^+. Let $E = (A(1,n) \cup B(1,n) \cup \bigcup_{1\le k<n} D(k))$, the language of $\overline{\mathcal{L}(\mathfrak{U})}$ (i.e., the above (2)) can then be defined by the set $\Big(\bigcup_{\tau\in E}\{U_{neg}(\tau)\} \cup \{U(\Sigma^+)\}\Big)$ of eLSCs.

CASE II: If at least one eLSC in $\mathfrak{U}$ has a $\top^*$ in its prechart and/or main chart, the language of $D(k)$ introduces some traces with the structure $(\tau_1 \circ \Sigma^* \circ \tau_2)$. Moreover, if all eLSCs in $\mathfrak{U}$ have $\top^*$ in their prechart and/or main chart, the language of $B(1,n)$ is defined by the conjunction of traces with the structure $(\tau_1 \circ \Sigma^* \circ \tau_2)$. In both cases, the language $\overline{\mathcal{L}(\mathfrak{U})}$ is then defined by the intersection of traces with the structures $\overline{\tau \circ \Sigma^*}$, $\overline{\tau}$ and $\overline{\tau_1 \circ \Sigma^* \circ \tau_2}$. For any (nonempty) traces τ_1 and τ_2, the language $\overline{\tau_1 \circ \Sigma^* \circ \tau_2}$ can be defined by a set $\mathfrak{U}' = \{U(\tau_1), U(\tau_2)\}$ of eLSCs as shown in Fig. 4(d). The language of $\mathfrak{U}'$ is

$$\mathcal{L}(\mathfrak{U}') = \overline{\tau_1\Sigma^* \circ \overline{\Sigma^+}} \cap \overline{\overline{\Sigma^+} \circ \Sigma^*\tau_2} = \overline{\tau_1\Sigma^*} \cap \overline{\Sigma^*\tau_2} = \overline{\tau_1\Sigma^*\tau_2}$$

CASE III: Let all eLSCs in the set $\mathfrak{U}$ be with necessary precharts, and no eLSC have a $\top^*$. This case is similar to CASE I. The language of $\mathfrak{U}$ is

$$\mathcal{L}(\mathfrak{U}) = \Big(\bigcap_{1\le i\le n} \overline{\Sigma^* \circ M_i} \Big) \cup \Big(\bigcap_{1\le i\le n} (P_i \circ M_i) \Big) \cup \bigcup_{1\le k<n} \Big(\bigcap_{1\le i\le k} \overline{\Sigma^* \circ M_i} \cap \bigcap_{k+1\le j\le n} (P_j \circ M_j) \Big).$$

Given $r_j \triangleq |Mch_j|$ and $r \triangleq \max\limits_{1\le j\le n} r_i$, we define

$$H(i,k) \triangleq \Sigma^r \backslash (\bigcup_{i\le j\le k} (\Sigma^{(r-r_j)} \circ M_j))$$

and

$$G(k) \triangleq \Big(\bigcap_{1\le i\le k} \overline{\Sigma^* \circ M_i} \cap \bigcap_{k+1\le j\le n} (P_j \circ M_j) \Big).$$

For any $1 \le k < n$, it holds that

$$G(k) = \begin{cases} (\Sigma^{(z_{(k+1,n)}-r)} \circ H(1,k)) \cap B(k+1,n) & \text{if } z_{(k+1,n)} \ge r \\ \emptyset & otherwise \end{cases}$$

(where $B(i,k)$ is defined as in CASE I). According to the above, the language of $\mathfrak{U}$ is

$$\mathcal{L}(\mathfrak{U}) = \Big(\bigcup_{\tau_h \in H(1,n)} (\Sigma^* \circ \tau_h) \Big) \cup \varepsilon \cup \Big(\bigcup_{\tau_b \in B(1,n)} \tau_b \Big) \cup \Big(\bigcup_{1\le k<n} \bigcup_{\tau_g \in G(k)} \tau_g \Big).$$

the complement of $\mathfrak{U}$ is

$$\overline{\mathcal{L}(\mathfrak{U})} = \Big(\bigcap_{\tau_h \in H(1,n)} (\overline{\Sigma^* \circ \tau_h}) \Big) \cap \Sigma^+ \cap \Big(\bigcap_{\tau_b \in B(1,n)} \overline{\tau_b} \Big) \cap \Big(\bigcap_{1\le k<n} \bigcap_{\tau_g \in G(k)} \overline{\tau_g} \Big).$$

Similar as the above, for any τ_h, the eLSC shown in Fig. 4(c)) defines $\overline{\Sigma^* \circ \tau_h}$.

CASE IV: Similar to the CASE II, if at least one eLSC in $\mathfrak{U}$ has one or more $\top^*$, the language $\overline{\mathcal{L}(\mathfrak{U})}$ involves the intersection of traces with the structure $\overline{\tau_1\Sigma^*\tau_2}$.

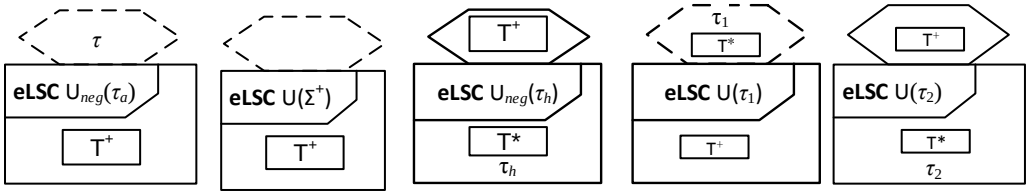
CASE V: Let eLSCs in $\mathfrak{U}$ involve both sufficient precharts and necessary precharts. The set can be simply divided into two groups $\mathfrak{U}_{suff} = \{U_1, ..., U_m\}$ and $\mathfrak{U}_{nec} = \{U_{m+1}, ..., U_n\}$, where all eLSCs in $\mathfrak{U}_{suff}$ are with sufficient precharts and all eLSCs in $\mathfrak{U}_{suff}$ are with necessary precharts. The language of $\mathfrak{U}$ is

$$\mathcal{L}(\mathfrak{U}) = \Big(\bigcap_{1\le i\le m} \overline{P_i \circ \Sigma^*} \cap \bigcap_{m+1\le j\le n} \overline{\Sigma^* \circ M_j} \Big) \cup \Big(\bigcap_{1\le i\le n} (P_i \circ M_i) \Big) \cup \varepsilon$$
$$\bigcup_{1\le k<m, m+1\le k'<n} \Big(\bigcap_{1\le i\le k} \overline{P_i \circ \Sigma^*} \cap \bigcap_{k+1\le j\le m} (P_j \circ M_j) \cap \bigcap_{1\le i'\le k'} \overline{\Sigma^* \circ M_{i'}} \cap \bigcap_{k'+1\le j'\le n} (P_{j'} \circ M_{j'}) \Big)$$

The subformula $\bigcap_{1\le i\le m} \overline{P_i \circ \Sigma^*} \cap \bigcap_{m+1\le j\le n} \overline{\Sigma^* \circ M_j}$ is equal to

$$\bigcup_{\tau_a \in A(1,m)} (\tau_a \circ \Sigma^*) \cap \bigcup_{\tau_h \in H(m+1,n)} (\Sigma^* \circ \tau_h) = \bigcup_{\tau_a \in A(1,m), \tau_h \in H(m+1,n)} (\tau_a \Sigma^* \tau_h).$$

Other subformulae can be reformulated by finite unions of traces. For any τ_a and τ_h, the language $\overline{\tau_a \Sigma^* \tau_h}$ is defined by a set of eLSCs with the same structure as Fig. 4(d)).



(a) eLSC for $\overline{\tau}$ (b) eLSC for Σ^+ (c) eLSC for $\overline{\Sigma^* \circ \tau_h}$ (d) eLSC for $\overline{\tau_1 \circ \Sigma^* \circ \tau_2}$

Figure 4. eLSCs expressing the complement of sets

□

As an example, consider the set $\mathfrak{U} = \{U1, U2\}$ of eLSCs for the client/server system in Fig. 1. Given $\Sigma = \{login, logout\}$, the complement of $\mathfrak{U}$ can be obtained as follows. In the silence-free interpretation, the language $L1$ of Fig. 1(a) is

$$L1 = \overline{\overline{login} \circ \overline{logout}} = (login \circ logout) \cup (logout \circ \Sigma^*) \cup \varepsilon$$

Similarly, the language $L2$ of Fig. 1(b) is

$$L2 = \overline{\overline{\overline{login} \circ logout}} = (login \circ logout) \cup (\Sigma^* \circ login) \cup \varepsilon.$$

Therefore, the language of $\mathfrak{U}$ is $L1 \cap L2 = (login \circ logout) \cup (logout \circ \Sigma^* \circ login) \cup \varepsilon$.

To complement the set $\mathfrak{U}$, it is first divided into two groups $\mathfrak{U}_s = \{U1\}$ and $\mathfrak{U}_n = \{U2\}$. Let $P_1 = \{login\}$ and $P_2 = \{login\}$ (resp. $M_1 = \{logout\}$ and $M_2 = \{logout\}$) be the languages of pre-

charts (resp. main charts) of $U1$ and $U2$, respectively. The language of $\mathfrak{U}$ is

$$\mathcal{L}(\mathfrak{U}) = (\overline{P_1 \circ \Sigma^*} \cup (P_1 \circ M_1)) \cap (\overline{\Sigma^* \circ M_2} \cup (P_2 \circ M_2)) =$$

$$(\overline{P_1 \circ \Sigma^*} \cap \overline{\Sigma^* \circ M_2}) \cup (\overline{P_1 \circ \Sigma^*} \cap P_2 \circ M_2) \cup ((P_1 \circ M_1) \cap \overline{\Sigma^* \circ M_2}) \cup ((P_1 \circ M_1) \cap (P_2 \circ M_2)) \cup \varepsilon. \quad (3)$$

Since $P_1 = P_2$ and $M_1 = M_2$, the second and third subformula of the above (3) are empty. Therefore

$$\mathcal{L}(\mathfrak{U}) = (\overline{P_1 \circ \Sigma^*} \cap \overline{\Sigma^* \circ M_2}) \cup ((P_1 \circ M_1) \cap (P_2 \circ M_2)) \cup \varepsilon.$$

Since the size of the prechart of U_1 is 1, the subformula $\overline{P_1 \circ \Sigma^*}$ equals to $\bigcup_{\tau_a \in A(1,1)}(\tau_a \circ \Sigma^*)$, where $A(1,1) = \Sigma^1 \backslash P_1 = \mathit{logout}$. Similarly, since the size of the main chart of U_2 is 1, the subformula $\overline{\Sigma^* \circ M_2} = \bigcup_{\tau_h \in H(2,2)}(\Sigma^* \circ \tau_h)$, where $H(2,2) = \Sigma^1 \backslash M_2 = \mathit{login}$. The subformula $(\overline{P_1 \circ \Sigma^*} \cap \overline{\Sigma^* \circ M_2})$ then equals to $(\mathit{logout} \circ \Sigma^* \circ \mathit{login})$. In addition, the language $(P_1 \circ M_1) = (P_2 \circ M_2) = (\mathit{login} \circ \mathit{logout})$. Therefore, the language $\overline{\mathcal{L}(\mathfrak{U})}$ equals to $(\overline{\mathit{logout} \circ \Sigma^* \circ \mathit{login}} \cap \overline{\mathit{login} \circ \mathit{logout}} \cap \Sigma^+)$. The subformula $\overline{\mathit{logout} \circ \Sigma^* \circ \mathit{login}}$ has the same structure as $\overline{\tau_1 \Sigma^* \tau_2}$ introduced by CASE II in the proof of Theorem 4.3. It can be defined by Fig. 4(d). The subformula $\overline{\mathit{login} \circ \mathit{logout}}$ has the same structure as $\overline{\tau}$ introduced by CASE I. It can be defined by Fig. 4(a). The subformula Σ^+ can be defined by Fig. 4(b). Therefore, the language $\overline{\mathcal{L}(\mathfrak{U})}$ can be defined by the eLSCs depicted in Fig. 5.

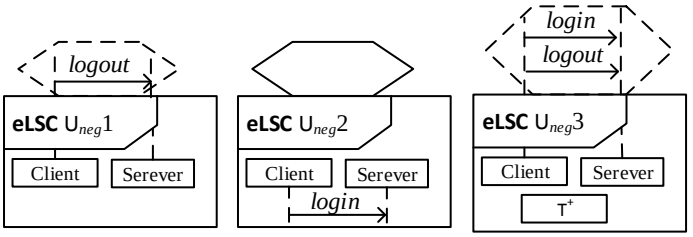


Figure 5. Negation of the client/server system

Theorem 4.4. For any eLSCs U_1 and U_2, the complement of $\mathcal{L}(U_1 \to U_2)$ can be defined by a set of eLSCs with concatenations.

Proof:
Given the concatenation $U_1 \to U_2$ of eLSCs, the complement of $U_1 \to U_2$ can be constructed as follows. Similar to the proof of Theorem 4.3, we distinguish several cases.

Case I: Both U_1 and U_2 are with sufficient precharts. The language of $U_1 \to U_2$ is

$$\begin{aligned}\mathcal{L}(U_1 \to U_2) &= (\overline{P_1\Sigma^*} \cup (P_1M_1)) \circ (\overline{P_2\Sigma^*} \cup (P_2M_2)) \\ &= (\overline{P_1\Sigma^*} \circ \overline{P_2\Sigma^*}) \cup (\overline{P_1\Sigma^*} \circ (P_2M_2)) \cup ((P_1M_1) \circ \overline{P_2\Sigma^*}) \cup (P_1M_1P_2M_2).\end{aligned}$$

The complement of $\mathcal{L}(U_1 \to U_2)$ is

$$\overline{\mathcal{L}(U_1 \to U_2)} = \overline{\overline{P_1\Sigma^*} \circ \overline{P_2\Sigma^*}} \cap \overline{\overline{P_1\Sigma^*} \circ (P_2M_2)} \cap \overline{(P_1M_1) \circ \overline{P_2\Sigma^*}} \cap \overline{P_1M_1P_2M_2}. \quad (4)$$

We now show that this language can defined by a set of eLSCs.

We first prove that for any languages A and B, it holds that $\overline{AB} = ((A\overline{B}) \cup \overline{A\Sigma^*})$. From $\overline{A\overline{B}} = (\overline{A\Sigma^*} \cup AB)$, we have $A\overline{B} = (A\Sigma^* \cap \overline{AB})$. Therefore, $(A\overline{B} \cup \overline{A\Sigma^*}) = ((A\Sigma^* \cap \overline{AB}) \cup \overline{A\Sigma^*}) = ((A\Sigma^* \cup \overline{A\Sigma^*}) \cap (\overline{AB} \cup \overline{A\Sigma^*}))$. Since $AB \subseteq A\Sigma^*$, it holds that $\overline{AB} \supseteq \overline{A\Sigma^*}$. Therefore, $(\overline{AB} \cup \overline{A\Sigma^*}) = \overline{AB}$. Thus $(A\overline{B} \cup \overline{A\Sigma^*}) = \overline{AB}$. Similarly, it also holds that $\overline{AB} = (\overline{A}B \cup \overline{\Sigma^* B})$.

Let x_1 and x_2 be the sizes of P_1 and P_2 (excluding $\top^*$), respectively. The subformula $\overline{\overline{P_1\Sigma^*} \circ \overline{P_2\Sigma^*}}$ of the above (4) can be reformulated as follows.

$$\begin{aligned}
\overline{\overline{P_1\Sigma^*} \circ \overline{P_2\Sigma^*}} &= \overline{(((\Sigma^{x_1} \backslash P_1) \circ \Sigma^*) \cup \varepsilon) \circ (((\Sigma^{x_2} \backslash P_2) \circ \Sigma^*) \cup \varepsilon)} \\
&= \overline{((\Sigma^{x_1} \backslash P_1)\Sigma^*) \circ ((\Sigma^{x_2} \backslash P_2)\Sigma^*)} \cap \overline{(\Sigma^{x_1} \backslash P_1)\Sigma^*} \cap \overline{(\Sigma^{x_2} \backslash P_2)\Sigma^*)} \cap \overline{\varepsilon} \\
&= \Big(\bigcap_{\tau_1 \in \Sigma^{x_1} \backslash P_1, \tau_2 \in \Sigma^{x_2} \backslash P_2} \overline{\tau_1 \Sigma^* \tau_2 \Sigma^*} \Big) \cap \Big(\bigcap_{\tau \in \Sigma^{x_1} \backslash P_1} \overline{\tau \Sigma^*} \Big) \cap \Big(\bigcap_{\tau \in \Sigma^{x_2} \backslash P_2} \overline{\tau \Sigma^*} \Big)
\end{aligned}$$

The language $\overline{\tau_1\Sigma^*\tau_2\Sigma^*} = (\overline{\tau_1\Sigma^*\tau_2}\Sigma^* \cup \overline{\Sigma^*\Sigma^*})$ equals to $\overline{\tau_1\Sigma^*\tau_2}\Sigma^*$. Based on Fig. 4 (d), the language $\overline{\tau_1\Sigma^*\tau_2}\Sigma^*$ can be defined by eLSCs with concatenations in Fig. 6.

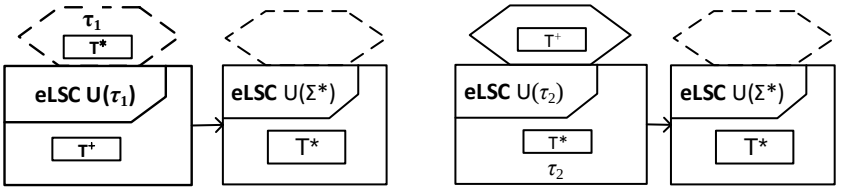


Figure 6. Concatenated eLSCs for $\overline{\tau_1\Sigma^*\tau_2}\Sigma^*$

The language $\overline{\tau\Sigma^*}$ can be defined by an eLSC structured as shown in Fig. 4(a). Therefore, the language $\overline{\overline{P_1\Sigma^*} \circ \overline{P_2\Sigma^*}}$ can be defined by a set of (concatenated) eLSCs. Depending on whether the main charts being with $\top^*$, the subformulae $\overline{\overline{P_1\Sigma^*} \circ (P_2M_2)}$, $\overline{(P_1M_1) \circ \overline{P_2\Sigma^*}}$ and $\overline{P_1M_1P_2M_2}$ can be reformed to the intersection of languages with the structures $\overline{\tau_1\Sigma^*\tau_2\Sigma^*}$, $\overline{\tau_1\Sigma^*\tau_2}$ and/or $\overline{\tau\Sigma^*}$. All of these languages can be defined by some eLSCs as given above. Therefore, $\overline{\mathcal{L}(A \to B)}$ can be defined by a set of eLSCs.

Case II: Both U_1 and U_2 are with necessary precharts. The language of the complement of $U_1 \to U_2$ is $\overline{\mathcal{L}(U_1 \to U_2)} = \overline{\overline{\Sigma^*M_1} \circ \overline{\Sigma^*M_2}} \cap \overline{\overline{\Sigma^*M_1} \circ P_2M_2} \cap \overline{P_1M_1 \circ \overline{\Sigma^*M_2}} \cap \overline{P_1M_1P_2M_2}$. Given some traces τ_1 and τ_2, the language can be defined by the intersection of languages with the structures $\overline{\Sigma^*\tau_1\Sigma^*\tau_2}$, $\overline{\Sigma^*\tau}$, $\overline{\tau_1\Sigma^*\tau_2\Sigma^*}$, $\overline{\tau_1\Sigma^*\tau_2}$ and $\overline{\Sigma^*\tau\Sigma^*}$. The language $\overline{\Sigma^*\tau_1\Sigma^*\tau_2} = (\Sigma^*\overline{\tau_1\Sigma^*\tau_2}) \cup \overline{\Sigma^*\Sigma^*}$. The language $\overline{\Sigma^*\tau\Sigma^*} = (\Sigma^*\overline{\tau\Sigma^*}) \cup (\overline{\Sigma^*\Sigma^*}) = \Sigma^*\overline{\tau\Sigma^*}$.

Case III: U_1 is with a sufficient prechart and U_2 is with a necessary prechart. The language of the complement of $U_1 \to U_2$ is $\overline{\mathcal{L}(U_1 \to U_2)} = \overline{\overline{P_1\Sigma^*} \circ \overline{\Sigma^*M_2}} \cap \overline{P_1\Sigma^*P_2M_2} \cap \overline{P_1M_1\overline{\Sigma^*M_2}} \cap \overline{P_1M_1P_2M_2}$.

Case IV: U_1 is with a necessary prechart and U_2 is with a sufficient prechart. The language of the complement of $U_1 \to U_2$ is $\overline{\mathcal{L}(U_1 \to U_2)} = \overline{\overline{\Sigma^*M_1} \circ \overline{P_2\Sigma^*}} \cap \overline{\Sigma^*M_1P_2M_2} \cap \overline{P_1M_1\overline{P_2\Sigma^*}} \cap \overline{P_1M_1P_2M_2}$.

It can been seen that all these languages can be defined by eLSCs as given in Fig. 4. □

Given a set $\mathfrak{U} = \{Uch_1 \rightarrow Uch'_1, ..., Uch_n \rightarrow Uch'_n\}$ of concatenated eLSCs and languages X_1 and X_2, the language $\mathcal{L}(\mathfrak{U})$ of $\mathfrak{U}$ is defined by some languages with the structures $\overline{X_1} \circ \overline{X_2}$, $\overline{X_1} \circ X_2$, $X_1\overline{X_2}$ and $X_1 \circ X_2$. According to the categories of the precharts, the intersections of the four languages can be reformed by traces with the structures $\tau_1\Sigma^*\tau_2\Sigma^*$, $\tau\Sigma^*$, $\tau_1\Sigma^*\tau_2$, $\Sigma^*\tau\Sigma^*$ and/or $\Sigma^*\tau_1\Sigma^*\tau_2$. The complement of $\mathcal{L}(\mathfrak{U})$ is then defined by the intersection of the complement of the above traces. As shown in the proofs of Theorem 4.3 and 4.4, all of these traces can be defined by (concatenated) eLSCs. Therefore, the complement of $\mathcal{L}(\mathfrak{U})$ can be defined by a set of concatenated eLSCs. That is, the language of $\bigcup_{1\leq i\leq n}(Uch_i \rightarrow Uch'_i)$ can be defined by concatenated eLSCs. The complement of the language of $(U_1 \rightarrow ... \rightarrow U_n)$ equals to $((\mathcal{L}(Uch_1) \circ \overline{\mathcal{L}(Uch_2) \circ ... \circ \mathcal{L}(Uch_n)}) \cup \overline{\mathcal{L}(Uch_1)\Sigma^*})$. Since the language $\overline{\mathcal{L}(Uch_2) \circ ... \circ \mathcal{L}(Uch_n)}$ can be iteratively reformed, it can be reduced to some languages of $\overline{\mathcal{L}(Uch_i) \rightarrow \mathcal{L}(Uch_j)}$ and $\overline{\mathcal{L}(Uch_i)\Sigma^*}$. Therefore, the complement of $\overline{\mathcal{L}(Uch_2) \circ ... \circ \mathcal{L}(Uch_n)}$ can be defined by a set of concatenated eLSCs. By iteratively rewriting the sublanguages, the complement of a set of concatenated eLSCs can be defined by traces with the same structures as the above. Therefore, the language of silence-free eLSC specifications is closed under negation.

Theorem 4.5. Silence-free eLSC specifications without iteration have the same expressive power as star-free regular expressions.

Proof:
Since LTL defines the star-free regular languages, and the iteration-free eLSC language is at most as expressive as LTL, the language is also at most as expressive as star-free regular expressions. We have to prove that any star-free language can be defined by a set of eLSCs. That is, for every star-free regular expression σ there exists an eLSC specification $\mathfrak{U}_\sigma$ such that $\mathcal{L}(\mathfrak{U}_\sigma) = \mathcal{L}(\sigma)$.

The empty language $\emptyset$ can be defined by a set $\mathfrak{U}_\emptyset = \{U_\varepsilon, U_a\}$ of eLSCs as shown in Fig. 7. The language of U_ε is $\mathcal{L}(U_\varepsilon) = \overline{\varepsilon \circ \overline{\varepsilon}} = \overline{\varepsilon \circ \Sigma^+} = \varepsilon$, and the language of U_a is $\mathcal{L}(U_a) = \overline{\overline{a} \circ \varepsilon} = a$. The language of $\mathfrak{U}_\emptyset$ is $\mathcal{L}(\mathfrak{U}_\emptyset) = \{\varepsilon\} \cap \{a\} = \emptyset$. According to theorems above, the complement $\overline{\sigma}$ of an expression σ can be defined by an eLSC. Union of expressions $\{\sigma_1, ..., \sigma_n\}$ can be defined by $\overline{\overline{\sigma_1} \cap ... \cap \overline{\sigma_n}}$. The concatenation of single traces can be defined by the concatenation of eLSCs directly. The complement of concatenations can be expressed by an eLSC specification. Therefore, any star free language can be defined by a set of eLSCs.

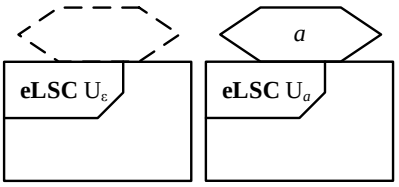


Figure 7. eLSCs expressing regular language

□

It is well known that LTL over finite traces has exactly the same expressiveness as star-free regular languages. Therefore, the following corollary holds.

Corollary 4.6. Silence-free eLSC specifications without iteration have the same expressive power as LTL over finite traces.

As an example, given an (finite) alphabet Σ and $a, b \in \Sigma$ with $a \neq b$, the LTL formula $(a \mathbf{U} b)$ defines the language $a^*b\Sigma^*$. (Note that this language can also be defined by a star-free expression). It can be defined by the eLSC specification $\mathfrak{U} \triangleq \{(U(\Sigma^*) \rightarrow U(e) \rightarrow U(b))|e \in \Sigma \setminus a\}$ as shown in Fig. 8. That is, for the eLSC specification has one concatenated chart for every $e \in \Sigma \setminus a$. The language of $U(e)$ is $\mathcal{L}(U(e)) = \overline{e\Sigma^*}$, and the language of $U(b)$ is $\mathcal{L}(U(b)) = \overline{\overline{b\Sigma^*} \circ \varepsilon} = b\Sigma^*$. Therefore, the language of $\mathfrak{U}$ is

$$\mathcal{L}(\mathfrak{U}) \triangleq \bigcap_{e \in \Sigma \setminus a} (\Sigma^* \circ \overline{e\Sigma^*} \circ b\Sigma^*).$$

As shown above, $\Sigma^*\overline{e\Sigma^*} = \overline{\Sigma^* e \Sigma^*}$. The language $\mathcal{L}(\mathfrak{U})$ defines all traces $\sigma = \sigma_1\sigma_2$ such that $\sigma_2 = b\Sigma^*$ and $\sigma_1 \in \bigcap_{e \in \Sigma \setminus a} (\overline{\Sigma^* e \Sigma^*})$. Since $\bigcap_{e \in \Sigma \setminus a} (\overline{\Sigma^* e \Sigma^*}) = a^*$, the language $\mathcal{L}(\mathfrak{U})$ is $a^*b\Sigma^*$.

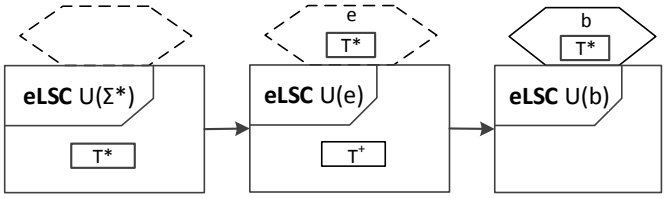


Figure 8. eLSCs expressing $(a \mathbf{U} b)$

Remark 4.7. It is easy to see that with iteration, eLSCs are equal in expressivity to regular expressions.

5. A translation of PeLSCs into HL formulae

For specifying and monitoring actual systems, PeLSCs are more appropriate than eLSCs. In this section, we translate PeLSCs into formulae of hybrid logic (HL) [41]. This will allow us to automatically generate monitors from PeLSC specifications. Given a PeLSC PU, the resulting formula $\varphi(PU)$ is the conjunction of two sub-formulae $\varphi(U)$ and $\mathfrak{d}(\mathsf{cond})$:

- $\varphi(U)$ is an LTL formula representing the temporal requirements of PU; and
- $\mathfrak{d}(\mathsf{cond})$ is an HL formula specifying the parameter constraints of PU.

5.1. Translation of eLSCs into LTL

We first present a translation from an eLSC into LTL. In contrast to the general translation given in section 4, Theorem 4.2, here we use event occurrences (rather than events) as propositions for the temporal formula. Each event occurrence in a chart is unique; this allows a more efficient translation.

Let $\mathsf{EOcc}(C)$ be the set of event occurrences in a basic chart C. From C, we define the LTL formula $\xi_C \triangleq \psi_C \wedge \gamma_C \wedge \mathfrak{y}_C$, where

$\psi_C \triangleq \bigwedge_{e \prec e'} (\neg e' \mathbf{W} e)$

$\gamma_C \triangleq \bigwedge_{e \in \mathsf{EOcc}(C)} (\neg e \mathbf{W} (e \wedge \mathbf{XG} \neg e))$

$\mathfrak{y}_C \triangleq \bigwedge_{e \in \mathsf{EOcc}(C)} \mathbf{F} e$

The formula ψ_C specifies that if $e \prec e'$ then e' cannot occur before e in a trace . The formula γ_C specifies that each e can only occur at most once in the trace. The formula $\mathfrak{y}_C$ specifies that every event appearing in the chart will eventually occur in the trace.

With these formulae, we can then obtain LTL formulae from eLSCs with necessary and sufficient precharts. From an eLSC $U \triangleq (Pch, Mch, \mathit{Cate})$, we define the following formulae.

$\xi_{Pch} \triangleq \psi_{Pch} \wedge \gamma_{Pch} \wedge \mathfrak{y}_{Pch}$, and $\xi_{Mch} \triangleq \psi_{Mch} \wedge \gamma_{Mch} \wedge \mathfrak{y}_{Mch}$,

$\chi \triangleq \bigwedge_{e' \in \mathsf{EOcc}(Pch)} ((\bigwedge_{e \in \mathsf{EOcc}(Mch)} (\neg e)) \mathbf{W} e')$

$\varphi(U) \triangleq ((\xi_{Pch} \wedge \chi) \Rightarrow \xi_{Mch})$ if *Cate* = *Suff*; and

$\varphi(U) \triangleq (\neg(\xi_{Pch} \wedge \chi) \Rightarrow \neg \xi_{Mch})$ if *Cate* = *Nec*.

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. From this definition it directly follows that the formula $\varphi(U)$ defines the language of U.

5.2. Translation of PeLSCs into HL

In this subsection, we present a translation of PeLSCs into a subclass of hybrid logic [41]. With the translation, it can then be checked by the resulting formula whether an observation is admitted by a PeLSC.

The formalism of HL has a type of symbols called *nominals*. A nominal represents the *name* of a parameter. Let s be a nominal and x be a free variable. An HL formula may contain the *down-narrow* binder "$x \downarrow s$.". When evaluating an HL formula over a parametrized event trace, the down-narrow binder assigns all variables x in the formula to the value of the parameter s of the "current" parametrized event. For instance, an HL formula $(x \downarrow s.\varphi(x))$ is satisfied by a parametrized event trace $t \triangleq (\langle e_1, \mathcal{P}_1 \rangle, ..., \langle e_n, \mathcal{P}_n \rangle)$ if and only if $\varphi(d)$ is satisfied by $(e_1, ..., e_n)$ with $\langle s, d \rangle \in \mathcal{P}_1$.

A condition structure in a PeLSC can be either without or with variables, the following formulae are defined.

- Given a condition structure $\mathsf{cond} = (\mathsf{prop}(s), \mathfrak{o})$ involving no variables and $\mathit{evt}(\mathfrak{o}) = e$, we define the following HL formula

$$\mathfrak{d}(\mathsf{cond}) \triangleq \Box\, (x \downarrow s.(e \Rightarrow \mathsf{prop}(x))).$$

  The formula specifies that whenever the event e occurs, it must carry a parameter $\langle s, d \rangle$ such that the proposition $\mathsf{prop}(d)$ is evaluated to *true*.

- Let $\mathsf{prop}(s, v)$ be a proposition containing (without loss of generality) one variable v. According to the non-ambiguity assumption, for a condition structure $\mathsf{cond} = (\mathsf{prop}(s, v), \mathfrak{o})$ with $evt(\mathfrak{o}) = e$, there is an assignment structure $\mathsf{assi}(s', v, \mathfrak{o}')$ with $evt(\mathfrak{o}') = e'$ and $\mathfrak{o}' \prec \mathfrak{o}$. We define the following HL formula

$$\mathfrak{d}(\mathsf{cond}) \triangleq \Box((e' \wedge \Diamond e) \Rightarrow (x \downarrow v.(e' \wedge \Diamond y \downarrow s'.(e \wedge (\mathsf{prop}(x, y)))))).$$

  Given a condition structure with variable v and the associated assignment structure with the same variable v, let e and e' be the events combined with these two structures, respectively. This formula expresses that if both of the events occur, then the proposition must be evaluated to *true* with the values of the parameters carried by the two events. That is, under the non-ambiguity assumption, if events combined with the assignment- and the condition structure occur at positions i and j of the trace, respectively, the proposition $\mathsf{prop}(d', d)$ is true with $\langle s, d\rangle \in \mathcal{P}_i$ and $\langle s', d'\rangle \in \mathcal{P}_j$.

  If cond contains more than one variable, the formula $\mathfrak{d}(\mathsf{cond})$ is similar.

From a PeLSC $PU \triangleq (U, \mathsf{COND}, \mathsf{ASSI})$ we define the HL formula

$$\varphi(PU) \triangleq \left(\varphi(U) \wedge \bigwedge_{\mathsf{cond} \in \mathsf{COND}} \mathfrak{d}(\mathsf{cond})\right).$$

The formula expresses that a parametrized event trace $t = (\langle e_1, \mathcal{P}_1\rangle, ..., \langle e_n, \mathcal{P}_n\rangle)$ satisfies the formula $\varphi(PU)$ if and only if $(e_1, ..., e_n)$ is in the language defined by U, and the parameters carried by the events meet the constraints of the condition structures. From these definitions, the following theorem is immediate.

Theorem 5.1. Any parametrized event trace t is admitted by a PeLSC PU iff $t \models \varphi(PU)$.

Proof:
The proof follows directly from the definitions. □

Note that in this translation we use only a restricted subset **HL'** of full hybrid logic, which does not involve quantification and the @-operator.

Theorem 5.2. The complexity of the translation of PeLSC specifications into **HL'** is polynomial in the number of events occurring in the chart, with a constant nesting depth of temporal operators.

Proof:
Given a basic chart C, we define the number of events occurring in the chart as $|C|$. We consider the worst case of the translation, where any two events in a prechart or in a main chart are ordered by $\prec$. In this case, the formula ψ_C (the first formula of Section 5.1) consists of a conjunction of all 2-combinations of events. Given a PeLSC PU with $|Pch| = p$ and $|Mch| = m$, the formulae ψ_{Pch} and ψ_{Mch} thus have size $4\binom{p}{2} = \frac{4p!}{2!(p-2)!} = 2p(p-1)$ and $2m(m-1)$, respectively. The size of all

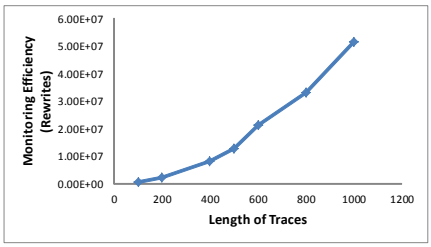
other sub-formulae of $\varphi(PU)$ are linear with respect to p and m. Therefore, any PeLSC PU with $|PU| = n$ can be translated into an **HL'** formula $\varphi(PU)$ with length $\mathcal{O}(n^2)$.

The maximal nesting of temporal operators appears in γ_C with a depth of 3; the formula $\mathfrak{d}(\mathsf{cond})$ involves a nesting of temporal and hybrid operators of depth 4. □

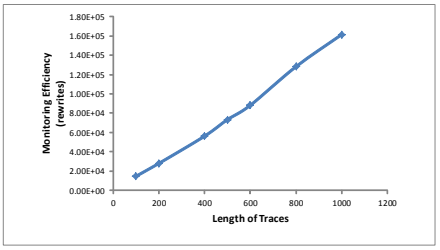
We now prove that the translation can be used to obtain an efficient monitoring algorithm for PeLSC specifications of parametrized event traces.

Theorem 5.3. The complexity of monitoring PeLSC specifications on parametrized event traces is linear with respect to the length of the trace.

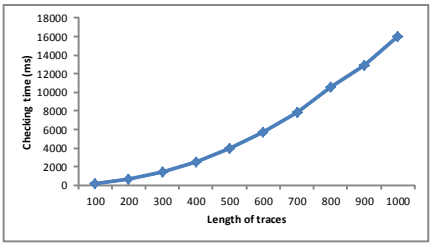
Proof:
We first show that the complexity of the model checking problem of **HL'** on finite sequences is linear with respect to the size of the model. Let $\varphi = x \downarrow s.\psi(x)$ be an **HL'** formula, where we assume without loss of generality that ψ is an LTL formula (without further binding operators). Furthermore, let t be a parametrized event trace with $\sigma \triangleq (e_1, ..., e_n)$ and $\rho \triangleq (\mathcal{P}_1, ..., \mathcal{P}_n)$. The trace t satisfies φ if and only if $\sigma \models \psi(d)$ with $(s, d) \in \mathcal{P}_1$. The sub-formulae of φ are comparisons of terms and can be directly evaluated to a boolean value $true$ or $false$, given the integer value of d. Therefore, the problem of checking φ over t can be reduced to the problem of checking ψ over σ. The complexity of checking whether a trace σ satisfies an LTL formula ψ is linear with respect to the length of σ. □



(a) Parametric requirements ($\mathfrak{d}(\mathsf{cond})$)



(b) Temporal requirements ($\varphi(U)$)



(c) Checking time by Maude (ms)

Figure 9. Monitoring efficiency for **P3**

According to this theorem, the language of PeLSCs is feasible for runtime verification. The monitoring algorithm proceeds by rewriting formulae along the trace, according to the semantics of the logic. We implemented such an algorithm in the rewriting environment Maude [42]. As experimental results, in Fig. 9 we show the number of rewrites for the property **P3** with respect to the length of the trace. The property **P3** is comprised of an eLSC and two condition structures with assignments (i.e., with variables). Fig. 9(a) shows the monitoring efficiency for the condition structures, Fig. 9(b) shows the monitoring efficiency for the eLSC, and Fig. 9(c) shows the checking time of the formula by Maude. In this monitoring implementation, most of the execution time is spent on checking the subformula $\mathfrak{d}(\mathsf{cond})$. We suspect that due to its internal organization of data, the execution time of Maude is quadratic in the number of clauses. The subformula $\varphi(U)$ takes about $100ms$ to check 1000 parametrized events.

6. Conclusion

In this paper, we defined and investigated the visual formalism PeLSC for monitoring specifications. This formalism extends classical LSCs by introducing modal pre-charts, regular operators, and assignment- and condition structures. The language can intuitively express sufficient- and necessary conditions of properties, and constraints of data (e.g., time values) carried by events. We proved that the formalism is strictly more expressive than the language of LSCs. For generating monitors, we developed a translation of PeLSCs into HL and proved that the complexity of monitoring PeLSC specifications is polynomial with respect to the length of the trace. This allowed us to develop an effective monitoring prototype for our formalism.

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 PeLSC based monitoring is feasible. Since the size of resulting formulae is often large, translating eLSC into LTL formulae is not an efficient way for monitoring. 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. Even though we do not think that the full LSC language poses additional fundamental problems, this needs to be worked out. Last but not least, it remains open to define an automaton concept which has exactly the same expressiveness as our PeLSCs.

Ackowledgment

This work was supported by the Project Sponsored by the Scientific Research Foundation for talents, Beijing Jiaotong University (NO. 2016RC006); the National Natural Science Foundation of China (U1434209); the National Basic Research Program of China (NO.2014CB340703); the Foundation of State Key Laboratory of Rail Traffic Control and Safety (NO.RCS2015ZT002); and the Fundamental Research Funds for the Central Universities (NO. 2016JBZ004, NO. 2016JBM007).

References

[1] Damm W, Harel D. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 2001;19(1):45–80. doi:10.1023/A:1011227529550.

[2] Havelund K. Monitoring with Data Automata. In: Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications, pp. 254–273. Springer, 2014. doi:10.1007/978-3-662-45231-8_18.

[3] Ben-abdallah H, Leue S. Timing Constraints in Message Sequence Chart Specifications. In: In IFIP. Chapman. Hall, 1997. doi:10.1007/978-0-387-35271-8_6.

[4] Chai M, Schlingloff BH. Monitoring Systems with Extended Live Sequence Charts. In: Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. 2014 pp. 48–63. doi:10.1007/978-3-319-11164-3_5. URL `http://dx.doi.org/10.1007/978-3-319-11164-3\_5`.

[5] Chai M, Schlingloff BH. Monitoring with Parametrized Extended Life Sequence Charts. In: Concurrency, Specification & Programming. 2015 pp. 88–102.

[6] Blackburn P, Seligman J. Hybrid Languages. *Journal of Logic, Language and Information*, 1995;4(3):251–272. doi:10.1007/BF01049415. URL `http://dx.doi.org/10.1007/BF01049415`.

[7] Alur R, Yannakakis M. Model Checking of Message Sequence Charts. In: CONCUR' 99 Concurrency Theory, pp. 114–129. Springer, 1999. doi:10.1007/3-540-48320-9_10.

[8] Simmonds J, Chechik M, Nejati S, Litani E, O??Farrell B. Property Patterns for Runtime Monitoring of Web Service Conversations. In: Runtime Verification. Springer, 2008 pp. 137–157. doi:10.1007/978-3-540-89247-2_9.

[9] Ciraci S, Malakuti S, Katz S, Aksit M. Checking the Correspondence between UML Models and Implementation. In: Runtime Verification. Springer, 2010 pp. 198–213. URL `http://doc.utwente.nl/74133/`.

[10] Harel D, Kugler H, Marelly R, Pnueli A. Smart Play-out of Behavioral Requirements. In: Formal Methods in Computer-aided Design. Springer, 2002 pp. 378–398. doi:10.1007/3-540-36126-X_23.

[11] Bontemps Y, Schobbens PY. The Computational Complexity of Scenario-based Agent Verification and Design. *Journal of Applied Logic*, 2007;5(2):252–276. URL `https://doi.org/10.1016/j.jal.2005.12.013`.

[12] Kugler H, Harel D, Pnueli A, Lu Y, Bontemps Y. Temporal Logic for Scenario-based Specifications. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 445–460. Springer, 2005. doi:10.1007/978-3-540-31980-1_29.

[13] Harel D, Maoz S, Segall I. Some Results on the Expressive Power and Complexity of LSCs. In: Pillars of computer science, pp. 351–366. Springer, 2008. doi:10.1007/978-3-540-78127-1_19.

[14] Kumar R, Mercer EG. Verifying Communication Protocols Using Live Sequence Chart Specifications. *Electronic Notes in Theoretical Computer Science*, 2009;250(2):33–48. URL `https://doi.org/10.1016/j.entcs.2009.08.016`.

[15] Harel D, Marelly R. Playing with Time: On the Specification and Execution of Time-enriched LSCs. In: Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on. IEEE, 2002 pp. 193–202. URL `http://dl.acm.org/citation.cfm?id=882460.882611`.

[16] Barringer H, Goldberg A, Havelund K, Sen K. Rule-based Runtime Verification. In: Verification, Model Checking, and Abstract Interpretation. Springer, 2004 pp. 44–57. doi:10.1007/978-3-540-24622-0_5.

[17] Barringer H, Rydeheard D, Havelund K. Rule Systems for Run-time Monitoring: from Eagle to RuleR. *Journal of Logic and Computation*, 2010;20(3):675–706. URL `https://doi.org/10.1093/logcom/exn076`.

[18] Barringer H, Fisher M, Gabbay D, Gough G, Owens R. MetateM: A Framework for Programming in Temporal Logic. In: de Bakker J, de Roever WP, Rozenberg G (eds.), Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness, volume 430 of *Lecture Notes in Computer Science*, pp. 94–129. Springer Berlin Heidelberg. ISBN 978-3-540-52559-2, 1990. doi:10.1007/3-540-52559-9_62. URL `http://dx.doi.org/10.1007/3-540-52559-9_62`.

[19] Forgy CL. Rete: A Fast Algorithm for the Many Pattern/many Object Pattern Match Problem. *Artificial intelligence*, 1982;19(1):17–37.

[20] Havelund K. Rule-based Runtime Verification Revisited. *International Journal on Software Tools for Technology Transfer*, 2015;17(2):143–170. doi:10.1007/s10009-014-0309-2.

[21] Allan C, Avgustinov P, Christensen AS, Hendren L, Kuzins S, Lhoták O, De Moor O, Sereni D, Sittampalam G, Tibble J. Adding Trace Matching with Free Variables to AspectJ. In: ACM SIGPLAN Notices, volume 40. ACM, 2005 pp. 345–364.

[22] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An Overview of AspectJ. In: ECOOP 2001Object-Oriented Programming, pp. 327–354. Springer, 2001. ISBN:3-540-42206-4.

[23] Barringer H, Havelund K. TraceContract: A Scala DSL for Trace Analysis. In: Butler M, Schulte W (eds.), FM 2011: Formal Methods, volume 6664 of *Lecture Notes in Computer Science*, pp. 57–72. Springer Berlin Heidelberg. ISBN 978-3-642-21436-3, 2011. doi:10.1007/978-3-642-21437-0_7. URL `http://dx.doi.org/10.1007/978-3-642-21437-0_7`.

[24] Chen F, Roşu G. Mop: an Efficient and Generic Runtime Verification Framework. In: ACM SIGPLAN Notices, volume 42. ACM, 2007 pp. 569–588. doi:10.1145/1297105.1297069.

[25] Meredith PO, Jin D, Griffith D, Chen F, Roşu G. An Overview of the MOP Runtime Verification Framework. *International Journal on Software Tools for Technology Transfer*, 2012;14(3):249–289. doi:10.1007/s10009-011-0198-6.

[26] Barringer H, Falcone Y, Havelund K, Reger G, Rydeheard D. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: FM 2012: Formal Methods, pp. 68–84. Springer, 2012. doi:10.1007/978-3-642-32759-9_9.

[27] Bauer A, Leucker M, Schallhart C. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2011;20(4):14. doi:10.1145/2000799.2000800.

[28] Basin D, Klaedtke F, Zălinescu E. Algorithms for Monitoring Real-time Properties. In: Runtime Verification. Springer, 2012 pp. 260–275. doi:10.1007/978-3-642-29860-8_20.

[29] Chai M, Schlingloff BH. A Rewriting Based Monitoring Algorithm for TPTL. In: Concurrency, Specification & Programming. 2013 pp. 61–72.

[30] Stolz V. Temporal Assertions with Parametrized Propositions. *Journal of Logic and Computation*, 2010;20(3):743–757. doi:10.1093/logcom/exn078.

[31] Merz S. Decidability and Incompleteness Results for First-order Temporal Logics of Linear Time. *Journal of Applied Non-Classical Logics*, 1992;2(2):139–156.

[32] Bauer A, Küster JC, Vegliach G. From Propositional to First-order Monitoring. In: Runtime Verification. Springer, 2013 pp. 59–75.

[33] Halle S, Villemaire R. Runtime Monitoring of Message-based Workflows with Data. In: Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE. IEEE, 2008 pp. 63–72. doi:10. 1109/EDOC.2008.32.

[34] Basin D, Klaedtke F, Müller S. Policy Monitoring in First-order Temporal Logic. In: Computer Aided Verification. Springer, 2010 pp. 1–18. doi:10.1007/978-3-642-14295-6_1.

[35] Basin D, Caronni G, Ereth S, Harvan M, Klaedtke F, Mantel H. Scalable Offline Monitoring. In: Bonakdarpour B, Smolka S (eds.), Runtime Verification, volume 8734 of *Lecture Notes in Computer Science*, pp. 31–47. Springer International Publishing. ISBN 978-3-319-11163-6, 2014. doi: 10.1007/978-3-319-11164-3_4. URL http://dx.doi.org/10.1007/978-3-319-11164-3_4.

[36] Bohn J, Damm W, Klose J, Moik A, Wittke H, Ehrig H, Kramer B, Ertas A. Modeling and Validating Train System Applications Using Statemate and Live Sequence Charts. In: Proc. IDPT. Citeseer, 2002.

[37] Combes P, Harel D, Kugler H. Modeling and Verification of a Telecommunication Application Using Live Sequence Charts and the Play-engine Tool. *Software & Systems Modeling*, 2008;7(2):157–175. doi:10.1007/s10270-007-0069-5.

[38] Fisher J, Harel D, Hubbard EJA, Piterman N, Stern MJ, Swerdlin N. Combining State-based and Scenario-based Approaches in Modeling Biological Systems. In: Computational Methods in Systems Biology. Springer, 2005 pp. 236–241.

[39] Alur R, Etessami K, Yannakakis M. Inference of Message Sequence Charts. *Software Engineering, IEEE Transactions on*, 2003;29(7):623–633. doi:10.1109/TSE.2003.1214326.

[40] Bontemps Y. Relating Inter-Agent and Intra-Agent Specifications. Ph.D. thesis, PhD thesis, University of Namur (Belgium), 2005.

[41] Franceschet M, de Rijke M, Schlingloff BH. Hybrid Logics on Linear Structures: Expressivity and Complexity. In: Temporal Representation and Reasoning, 2003 and Fourth International Conference on Temporal Logic. Proceedings. 10th International Symposium on. IEEE, 2003 pp. 166–173. doi:10.1109/TIME.2003.1214893.

[42] Ölveczky PC. Real-Time Maude 2.3 Manual. *Research report http://urn. nb. no/URN: NBN: no-35645*, 2004. URL http://urn.nb.no/URN:NBN:no-18692Delav.