The Journal of Systems & Software 177 (2021) 110962

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Runtime verification of train control systems with parameterized modal live sequence charts☆,☆☆

Ming Chai [a,b], Haifeng Wang [a,b], Tao Tang [c], Hongjie Liu [a,b,*]

[a] National Engineering Research Center of Rail Transportation Operation and Control System, Beijing Jiaotong University, Beijing 100044, China
[b] Beijing Laboratory of Urban Rail Transit, Beijing 100044, China
[c] State Key Laboratory of Rail Traffic Control and Safety, Beijing Jiaotong University, Beijing 100044, China

ARTICLE INFO

Article history:
Received 15 May 2020
Received in revised form 11 November 2020
Accepted 23 March 2021
Available online 29 March 2021

Keywords:
Runtime verification
Live sequence chart
Train control system

ABSTRACT

With the growing complexity of railway control systems, it is required to preform runtime safety checks of system executions that go beyond conventional runtime monitoring of pre-programmed safety conditions. Runtime verification is a lightweight and rigorous formal method that dynamically analyses execution traces against some formal specifications. A challenge in applying this method in railway systems is defining a suitable monitoring specification language, i.e., a language that is expressive, of reasonable complexity, and easy to understand. In this paper, we propose parameterized modal live sequence charts (PMLSCs) by introducing the alphabet of the specification into charts to distinguish between silent events and unexpected events. We further investigate the expressiveness and complexity theories of the language. In particular, we prove that PMLSCs are closed under negation and the complexity of a subclass of PMLSCs is linear, which allows the language to be used to monitor a system online. Finally, we use PMLSCs to monitor an RBC system in the Chinese high-speed railway and evaluate the performance. The experimental results show that the PMLSC has high monitoring efficiency, and can reduce false alarm rate by introducing alphabets of charts.

© 2021 Elsevier Inc. All rights reserved.

1. Introduction

Chinese train control system (CTCS) is a Chinese railway signalling system that ensures the safety of train operations. The CTCS Level 3 is closely related to the European Train Control System ETCS level 2 and 3, and is designed for high-speed passenger transportation. The CTCS is a typical safety critical system, whose failures may result in serious consequences of death to people and loss to property. *Fail-safety* is a key design feature of railway signalling systems. It means that whenever a failure occurs in a railway system, the system should transit into some safe-side states that lead the train to stop quickly. To achieve the fail-safety, runtime monitoring techniques are applied for detecting system failures by checking, e.g., memory integrity and pre-programmed safety conditions. However, with the growing complexity of CTCS software, it is required to monitor more complicated logical safety properties beyond conventional runtime monitors.

Runtime verification is a lightweight formal method that checks the execution of a system against some specifications. Runtime verification is performed by using a *monitor*, which is a device or piece of software that observes the system and generates a verdict (*true* or *false*) as a result. With scalable monitoring specifications, runtime verification has a potential to monitor desired safety properties of modern complex railway software. The quality of a runtime verification system depends on the quality of its monitoring specification. To make a monitor easier to understand and build, its specification is often formalized with some sort of temporal logic, such as linear temporal logic (LTL) and its extensions. Although these formalisms are expressive and technically sound for monitoring, they are inefficient for specifying the interleaving of subsystems. Moreover, railway engineers are not familiar with these textual formalisms and need extensive training to understand and use them efficiently.

We aim to address the above mentioned problems by writing monitoring specifications with a graphic formalism. In railways, a well adopted means to understand the control system of a train is via the specifications of its operational scenarios. Such typical scenarios include the start of mission, override, level transition, change of train orientation, RBC/RBC handover, etc. The formalism of sequence charts are attractive to specify interactive behaviours between subsystems. The UML sequence diagram is widely used

☆ This work was supported by the Beijing Natural Science Foundation, China (grant number L181005).
☆☆ Editor: Earl Barr.
* Corresponding author.
E-mail addresses: chaiming@bjtu.edu.cn (M. Chai), hfwang@bjtu.edu.cn (H. Wang), ttang@bjtu.edu.cn (T. Tang), hjliu2@bjtu.edu.cn (H. Liu).

https://doi.org/10.1016/j.jss.2021.110962
0164-1212/© 2021 Elsevier Inc. All rights reserved.

in railway industries for representing system specifications. However, being a semi-formal language, the semantics of the UML sequence diagram suffers from lack of expressiveness. One of the main drawbacks is that the UML sequence diagram cannot differentiate between the required and allowed behaviours of a system. Damm and Harel overcome this drawback by defining the language of live sequence charts (LSCs), which introduces the notations of the universal and existential charts.

When used for monitoring specifications, the language of LSCs suffers from an inherent shortcoming of expressing properties with necessary conditions. Additionally, the language of standard LSCs is not closed under complement, and it suffers from semantic confusions due to the introduction of negation operators (Harel and Maoz, 2008). In our previous work (Chai and Schlingloff, 2017), we defined the language of the parameterized extended LSC (PeLSC), which introduces the notations of the necessary and sufficient precharts, and a special symbol $\top$ for executing any one out of all possible events of charts. These extensions make PeLSCs capable for the expressiveness of regular languages under certain conditions. However, when the language of PeLSC is applied to construct monitoring specifications for real-world railway applications, sometime the symbol $\top$ is not used intuitively, resulting to highly incorrect monitoring rates in some cases. For example, consider the following property of the RBC/RBC handover procedure in CTCS level 3: *if and only if a train sends an "approaching" message to the handover RBC, then the RBC returns an "RBC/RBC handover command" message to the train. After that, if the train sends a "request movement authority (MA)" message to the handover RBC, it must return an "MA" message to the train*. This procedure is specified in two charts of PeLSCs. These two charts have different irrelevant messages, i.e., silent events. Since a silent event in PeLSC is defined based on a unique alphabet, the language of PeLSCs cannot express this property in an intuitive manner.

Contribution

In this paper, we first propose the language of the *parameterized modal live sequence chart* (PMLSC) by introducing the alphabet of charts into PeLSCs. With this extension, the language can accurately formulate silent events, which improves the safety protection ability of monitors. Second, we investigate the expressiveness and complexity theories of the novel language. In particular, we prove that PMLSC is closed under negation, which solves a long-standing problem in this area. The complexity of a subclass of PMLSC is linear with respect to the lengths of words, which enables a system to be monitored online. By contrast, full PMLSC is NP-complete or undecidable, depending on the feature of domains of variables. Third, we develop a PMLSC monitoring algorithm based on language translation and trace slicing. This algorithm substantially improves the monitoring efficiency of large-scale system executions. Finally, we apply our PMLSC to monitor a concrete example of Radio Block Center (RBC) executions of the CTCS level 3. The experimental results are promising: PMLSC monitors with the developed algorithm achieve high performance and are able to prevent known errors, as well as many other unknown possible defects, hidden in an RBC system from evolving into accidents.

The rest of this paper is organized as follows. Section 2 provides a brief survey of related work. Section 3 presents the framework and features of CTCS level 3. Section 4 defines the novel language of PMLSCs. Section 5 proves some theoretical results of the expressiveness and complexity of PMLSCs. Section 6 develops algorithms for monitoring large-scale system executions. Section 7 provides a concrete example of monitoring an RBC system of CTCS level 3. Section 8 contains conclusions and directions for future work.

2. Related work

Monitoring runtime status of railway control systems has a long tradition. For example, every Chinese high-speed railway system is equipped with a dynamic monitoring system (DMS) that checks memory integrity, communication between on-board and wayside subsystem and other preprogrammed safety conditions. Unfortunately, the growing complexity of railway systems requires runtime monitoring that go beyond conventional monitors. Runtime verification is a lightweight formal method that verifies runs of a software system in detecting faults (Delgado et al., 2004; Havelund and Goldberg, 2008; Sokolsky et al., 2012; Falcone et al., 2013; Bartocci et al., 2018). Linear temporal logic (LTL) is a widely adopted formalism in runtime verification. In order to allow the runtime verification to perform better over a finite length of executions, Bauer et al. propose a three valued LTL (LTL_3) (Bauer et al., 2011) and a four valued LTL (RV-LTL) (Bauer et al., 2010) for expressing uncertainties likely to be caused by unknown future executions. Runtime verification has been applied to monitor the interactions of agents in multi-agent systems. Ancona et al. propose a formalism for tracing expressions of specifications in such systems (Ancona et al., 2012). They further prove that the formalism is more expressive than LTL (Ancona et al., 2016). Aceto et al. prove the runtime monitorability of the properties specified by the Hennessy–Milner logic with recursion (Aceto et al., 2019a).

Boer et al. propose a textual formalism based approach for monitoring the protocol- and data-oriented properties of a communication process (de Boer and de Gouw, 2014). The language of live sequence charts, proposed as an extension of message sequence charts (Damm and Harel, 2001), can be used to verify event traces of systems at runtime (Maoz and Harel, 2011). Harel et al. propose a play-in/play-out approach for modelling with LSCs (Harel et al., 2002). Bontemps et al. prove that any LSC specification can be translated into LTL formulae (Bontemps and Schobbens, 2007). Kugler et al. develop an efficient algorithm for translating LSCs into LTL formulae (Kugler et al., 2005). The expressive power and complexity of LSCs are discussed in a previous survey (Harel et al., 2008). Sun et al. show that LSCs can be translated into CSP (Sun and Dong, 2005). Kumar et al. extend the LSC language with Kleene star, subcharts, and hierarchical charts (Kumar and Mercer, 2009). They further translate their LSCs into automata for verification. Taking silent actions into account, Aceto et al. study the issues related to the monitorability of properties (Aceto et al., 2017). Harel et al. propose *modal sequence diagrams* with "consider" and "ignore" operators to define relevant and irrelevant messages. However, those operations introduce confusions in negation of charts (Harel and Maoz, 2008). Translations from timed sequence diagrams into time Petri nets are then proposed (Andrade et al., 2009; Yang et al., 2012). Our parameterized extensions of LSCs are inspired by the treatment of time in LSCs proposed by Harel and Marelly (2002), in which a time constraint is defined by a combination of assignment and condition structures. By contrast, we provide general notation for arbitrary data parameters.

Diekert et al. prove that the complexity of deciding the liveness and monitorability for Büchi automata are both PSPACE-complete; and that of deciding the liveness of the LTL formulae is EXPSPACE-complete (Diekert et al., 2015). Bonakdarpour et al. study the complexity of the problem of model checking for HyperLTL (Bonakdarpour and Finkbeiner, 2018). Aceto et al. analyse the complexity bounds of various formalisms, which provides an excellent guide in choosing specification languages for monitoring overhead aspects (Aceto et al., 2020). As shown in Harel et al. (2008), the complexities of model checking reachability and centralized synthesis of LSCs are PSPACE-complete and EXPTIME-complete, respectively. Since the language of PMLSCs has more

expressiveness, solving the above problems of the formalism in general has no lower complexity. However, as shown in the present work, the word problem of a subclass of PMLSC is linear. Therefore, the language is feasible for specifying some kinds of monitoring properties.

Monitoring events that carry data are an important research area of runtime verification Havelund et al. (2018). Ahrendt et al. propose a runtime verification approach for monitoring those properties which combine features of both data and control (Ahrendt et al., 2017). Colombo et al. propose a well scaled tool, LARVA, for monitoring real-time properties of Java programmes (Colombo et al., 2009; Colombo et al., 2009). RuleR (Barringer et al., 2010) and LogicFire (Havelund, 2014b) are developed for monitoring data-relevant properties. Although these rule-based runtime verification systems have high performance, they are not attractive for practical applications. TraceMatches (Allan et al., 2005) is essentially a language of regular expressions that extends the language of AspectJ (Kiczales et al., 2001) by introducing free variables in the matching patterns. JavaMOP addresses parametric specification and monitoring using TraceMatches (Meredith et al., 2012). TraceMatches and JavaMOP are defined on the basis of trace slicing, which translates parameterized events into propositional events. However, these trace slicing algorithms can handle only traces where all events with the same name carry the same parameters.

Another important direction of monitoring parameterized properties is based on automata theory. Quantified event automata (Barringer et al., 2012) are an extension of the trace slicing methods mentioned above. Data automata have been proposed as a down-scaled version of Rete to an automaton-based formalism (Havelund, 2014a). Furthermore, various extensions of temporal logic have been proposed for monitoring parameterized/timed event traces, such as TLTL (Bauer et al., 2011), MTL (Basin et al., 2012), TPTL (Chai and Schlingloff, 2013), MDL (Basin et al., 2017), MFOTL (Basin and Klaedtke, 2015) and LTL^{FO} (Bauer et al., 2015). Basin et al. report the development and correctness proofs in Isabelle/HOL of monitoring for the metric first-order dynamic logic (Basin et al., 2020). These formalisms are expressive and technically sounds to perform runtime verification. However, the scenarios of railway systems specified with these textual-based languages are complicated and difficult to be understood by railway engineers.

3. CTCS level 3

CTCS level 3 is characterized by bi-directional communication and wayside subsystems via a radio block centre (RBC). It adopts features from the European train control system (ETCS) level 2. In CTCS level 3, continuous radio communication is used to exchange train control information between the train and the wayside equipment. Instead of using Global System for Mobile Communication (GSM), in the CTCS level 2 train operation related information are transmitted via track circuits. This information includes the speed, direction and location of the train, and the maximum speed and distance for the train.

3.1. Typical architecture of CTCS level 3

The typical architecture of CTCS level 3 is shown in Fig. 1.

The *centralized traffic control* (CTC) is responsible for directing train operations. Depending on the train schedule, the CTC generates route request commands and sends them to the interlocking. In CTCS level 3, a *route* is a path of railway (typically) between two signals in a railway station on which a train can move. The CTC also provides an interface for dispatchers to set up temporary

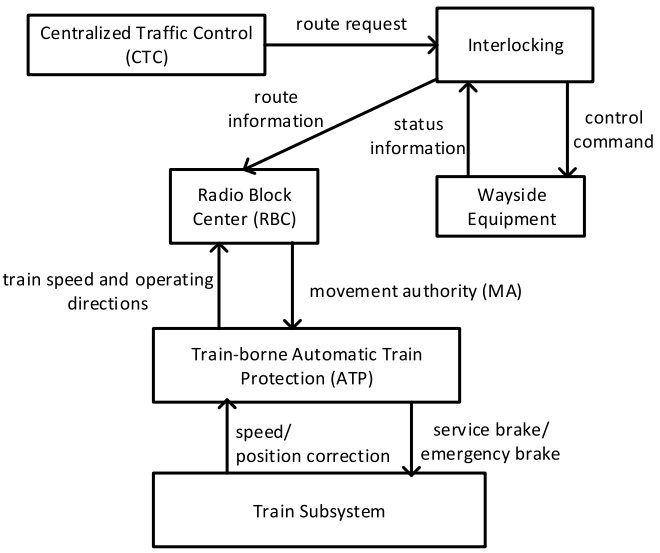

Fig. 1. Typical architecture of CTCS level 3.

speed restrictions and emergency brake commands and send them to the train via the RBC.

The *interlocking* is responsible for setting and granting requested routes. When a route command is received from the CTC, the interlocking guarantees that the route is safe to use by controlling the wayside infrastructure. The control is proceeded with respect to the status of the wayside infrastructure stipulated by the interlocking table. After the route is locked, the interlocking sends the route information to the RBC.

In an interlocking table, relevant wayside infrastructures include tracks, points and signals. When all tracks on the route are free and all points are locked in the required positions, the interlocking clears the signal and informs the RBC that the route is available. By contrast, if (some part of) the route is occupied by another train, the interlocking informs the RBC that the route is used.

The RBC is responsible for computing *movement authorities* (MAs) for a train. An MA is a succession of geographic railway areas that a train can safely operate in. The furthest point of an MA is indicated by the *end of authority* (EoA), which is given to a train by the RBC. When a train approaches its EoA, the train applies for a new MA by sending an "MA request" to the RBC. After receiving this request, the RBC generates an extension of the MA for the train according to the physical location and operation plan of the train. If the train is running through a railway station, the MA application procedure is performed based on the available routes supplied by the interlocking. After a route is locked, the RBC sends the train an updated MA to the end of the route.

The *automatic train protection* (ATP) is responsible for guaranteeing safe movement of a train. According to the EoA and railway line engineering data, the ATP computes a speed-distance curve of the MA area that consists of the maximum safe speed at each location between the train and the EoA. If the running speed of a train exceeds its speed-distance curve, the ATP initiates a service/emergency brake.

3.2. Monitoring specifications of CTCS level 3

In the context of CTCS level 3, the high-level safety requirements are collision and derailment free. Since these conditions are abstract, they cannot be used directly as monitoring specifications directly. To solve this problem, we formulate safety-critical

scenarios of CTCS level 3 as monitoring specifications of the system. With such properties, a monitor can detect execution errors in the system before the errors evolve into failures. The safety guarantee ability of the monitor is therefore increased. In this paper, we show how to specify procedures of MA extensions and RBC/RBC handover as properties to be monitored.

CTCS level 3 behaviour includes interactions between onboard and wayside subsystems with complicated logical relations. In addition, executions of CTCS level 3 system carry various data, including time, train position, train speed and other engineering data. Thus, the language should be able to formulate parameterized properties to specify the constraints of the data. Therefore, we choose to use LSCs and their extensions as the monitoring specification languages. The language of LSCs is defined by extending message sequence charts (MSCs), which can graphically specify the interleaving behaviour among the components of a system in an intuitive manner. Because the language of LSCs is proposed as a system modelling language, some shortcomings arise in expressiveness when specifying system properties. Consequently, the language should be extended to meet the expressiveness requirements of CTCS monitoring.

4. Parameterized modal live sequence charts

The language of LSC is defined by introducing universal and existential charts into MSCs. With this extension, the language can distinguish between necessary and possible behaviour of a system. We define parameterized extended live sequence charts (PMLSCs) by introducing the alphabet of charts to make the language more practical for expressing monitoring properties.

4.1. Basic charts

The language of PMLSCs is defined on the basis of *basic charts*, which are visually similar to MSCs. A basic chart specifies communication (i.e., the exchange of messages) among a set of lifelines. Intuitively, a lifeline is composed of a sequence of communication actions, i.e., sending and receiving of a message. A communication is made between two lifelines. It associates two events (i.e. actions) representing the sending and receiving actions of the message, respectively. A basic chart describes the following partial orders between two events.

R1: An event at a lower position of a lifeline follows an event at a higher position of the same lifeline; and
R2: an event of sending a message precedes the event of receiving the same message.

Formal definitions of basic charts are given as follows. Let M be a set of *messages* and $\Sigma \triangleq (M \times \{!, ?\})$ be an alphabet of *events*; that is, an event is either $m!$ or $m?$, indicating that the message m is sent or received, respectively. A *lifeline* $l \in \Sigma^*$ is a finite (possible empty) sequence of events. Given an n-tuple of lifelines $\mathbb{L} \triangleq (l_1, \ldots, l_n)$ with $l_i = (e_{i1}, \ldots, e_{im})$, we say that an event $e = e_{ij}$ occurs at the jth position of the ith lifeline in $\mathbb{L}$. An *event occurrence* $o \triangleq (e, i, j)$ is defined as a tuple consisting of an event $e \in \Sigma$ and the occurrence location (i, j) of e in $\mathbb{L}$. Given a message m, a *communication* $\langle o, o' \rangle \triangleq \langle (m!, i, j), (m?, i', j') \rangle$ is a pair of two event occurrences that represent sending and receiving m.

Definition 1 (*Basic Chart*)**.** A basic chart C is a three-tuple $C \triangleq \langle \mathbb{L}, \mathbb{C}, \Sigma \rangle$ of lifeline $\mathbb{L}$, a set $\mathbb{C}$ of communications and the alphabet Σ of the chart, where each event occurrence in $\mathbb{L}$ is contained in at most one communication.

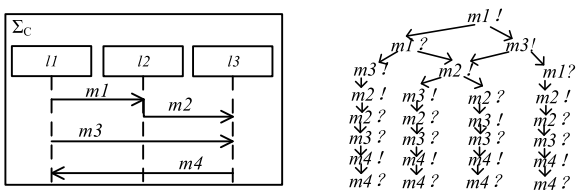

Fig. 2. Example: a basic chart (left) and the words defined by the chart (right).

Let $\mathsf{E}(C)$ and $\mathsf{EO}(C)$ be all *events* and *event occurrences* appearing in C, respectively. It holds that $\mathsf{E}(C) \subseteq \Sigma_C$. A basic chart is allowed to specify only the sending or the receiving of a message. In addition, an event is allowed to occur multiple times in a basic chart, e.g., a basic chart can express that a message is repeatedly sent or received; however, each event occurrence in a basic chart is unique. The semantics of basic charts are given as follows. Let $\prec$ be the smallest partial order relation between event occurrences induced by a basic chart $C = (\mathbb{L}, \mathbb{C})$. According to the above stated informal descriptions, the partial order of the occurrence of events satisfies the following two points.

1. $\forall j \in [1, |l_i|)$ with $l_i \in \mathbb{L}$: $(e, i, j) \prec (e', i, j+1)$ (R1), and
2. $\forall \langle o, o' \rangle \in \mathbb{C}$: $o \prec o'$ (R2).

For any event occurrence $o = (e, i, j)$, let $o[1] \triangleq e$ be the event of o. Let $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ be a (bijection) permutation. Let C be a basic chart with $\mathsf{EO}(C) = \{o_1, \ldots, o_n\}$. A sequence $o = (o_{f(1)}, \ldots, o_{f(n)})$ of event occurrences is called *consistent* with C, if $o_{f(i)} \prec o_{f(j)}$ implies $i < j$ for any $i, j \in [1, n]$.

The events in the alphabet of C but not appearing in the chart of *silent events* of C are denoted by $\mathcal{T}_C$. We define *silent events* of C as $\mathcal{T}_C \triangleq (\Sigma_C \setminus \mathsf{E}(C))$. Intuitively, the alphabet of C defines all relevant events of the property, whereas a silent event in the alphabet is not used in the chart. By $o \vdash C$ we denote o being consistent with C. The occurrence of silent events in an event occurrence does not impact its consistency of a basic chart.

Definition 2 (*Semantics of Basic Charts*)**.** Let $\tau_i^* \in \mathcal{T}_C^*$ be a (possible empty) sequence of silent events of a basic chart C; the language $\mathcal{L}(C)$ defined by the chart is given as follows.

$$\mathcal{L}(C) \triangleq \{(\tau_0, e_1, \tau_1, \ldots, e_n, \tau_n) | \exists (o_1, \ldots, o_n) \vdash C : e_i = o_i[1]\}$$

For example, Fig. 2 shows a basic chart C and the words $W(C)$ defined by the chart, with the alphabet Σ_C of C placed at the upper-left corner of the chart, where $\Sigma_C = (\{m1, m2, m3, m4\} \times \{!, ?\})$.

Definition 3 (*Empty Basic Chart*)**.** An empty basic chart $C_\emptyset$ is a basic chart with $\mathbb{L} = \emptyset$, where the language $\mathcal{L}(C_\emptyset)$ of $C_\emptyset$ is the set of empty words $\mathcal{L}(C_\emptyset) = \{\varepsilon\}$.

4.2. Parameterized modal LSCs

A PMLSC consists of two basic charts: a prechart (*Pch*, drawn within a hexagon) and a main chart (*Mch*, drawn within a solid rectangle). The language of a PMLSC U is defined over an alphabet Σ_U of events, where $\Sigma_U \triangleq (\Sigma_{Pch} \cup \Sigma_{Mch})$.

PMLSCs have modal precharts: *sufficient precharts* (drawn within a dashed hexagon) and *necessary precharts* (drawn within a solid hexagon). Intuitively, a PMLSC with a sufficient prechart specifies all the traces composed of two segments in a way that, if the first segment meets the prechart, then the second segment must meet the main chart. A necessary prechart is interpreted as a necessary condition. A PMLSC with a necessary prechart specifies all the traces composed of two segments in

a way that, the second segment cannot meet the main chart unless the first segment meets the prechart. The parameters of a property are specified by the *assignment* and *condition* structures in PMLSCs, which evaluate variables to parameter values and specify parameter constraints, respectively. In a PMLSC, the value of a parameter is first stored in an assignment structure and later on it is checked by a condition structure.

Formally, the syntax of PMLSCs is defined as follows. Let $\mathcal{S} \triangleq \{s_1, \ldots, s_n\}$ be a finite set of *nominals* and $\mathcal{D} \triangleq \{d_1, d_2, \ldots\}$ be a *domain* (e.g., integers or reals). A *parameter* is a pair $p \triangleq (s, d)$ from $(\mathcal{S} \times \mathcal{D})$, where s and d are the *name* and *value* of p, respectively. Let $\mathcal{X} \triangleq \{x_1, \ldots, x_n\}$ be a finite set of variables, assignment structures and condition structures are defined as follows.

Definition 4. An *assignment structure* is defined as a tuple $(x, s, \mathfrak{o})$, where $x \in \mathcal{X}$, $s \in \mathcal{S}$ and $\mathfrak{o} \in \mathsf{EO}(Uch)$.

Definition 5. A *condition structure* is defined as a pair $(\pi, \mathfrak{o})$, where π is a *proposition* with respect to $\mathcal{X}$ and $\mathcal{S}$.

Definition 6 (*Syntax of PMLSCs*)**.** A PMLSC U is a six-tuple $U \triangleq (\Sigma_U, P, M, Mod, \mathsf{ASSI}, \mathsf{COND})$, where Σ_U is the alphabet of the chart, P and M are basic charts, $Mod \in \{\mathfrak{S}, \mathfrak{N}\}$ is the modal (sufficient or necessary) of the prechart, and ASSI and COND are sets of assignment structures and condition structures, respectively.

Given sets $\mathcal{L}, \mathcal{L}' \in \Sigma^*$ of words, we denote $\mathcal{L} \cdot \mathcal{L}'$ to be the concatenation of $\mathcal{L}$ and $\mathcal{L}'$, and $\overline{\mathcal{L}} \triangleq (\Sigma^* \backslash \mathcal{L})$ to be the complement of $\mathcal{L}$. The set $W(U)$ of words defined by a PMLSC U is given as follows.

$$W(U) \triangleq \begin{cases} \overline{\mathcal{L}(P) \cdot \overline{\mathcal{L}(M)}} & \text{if } Mod = \mathfrak{S} \\ \overline{\overline{\mathcal{L}(P)} \cdot \mathcal{L}(M)} & \text{if } Mod = \mathfrak{N} \end{cases} \tag{1}$$

Intuitively, modals $\mathfrak{S}$ and $\mathfrak{N}$ indicate that the execution of the prechart is, respectively, the sufficient and necessary conditions for the execution of the main chart. For example, if a PMLSC is with a sufficient prechart (Mod $\mathfrak{S}$), then the chart specifies all the traces in a way that, if a trace contains a segment accepted by the prechart, then it must also contain a continuation segment (directly following the first segment) accepted by the main chart. On the other hand, if a PMLSC is with a necessary prechart (Mod $\mathfrak{N}$), then the chart specifies all the traces in a way that, if a trace contains a segment accepted by the main chart, then it must also contain a prefix segment (directly preceding the second segment) accepted by the prechart.

For runtime verification, we formalize an action of an SuM (System under Monitoring) by a *parameterized event* (i.e., an event carrying data). A parameterized event is a pair $\mathfrak{e} \triangleq (e, \mathcal{P})$, where $e \in \Sigma_E$ is an event and $\mathcal{P} \in 2^{\mathcal{S}\times\mathcal{D}}$ is a set of parameters. We require parameterized events to follow a *non-ambiguity* assumption: a parameterized event cannot carry two parameters with the same name but different values. Let $p[1]$ be the name of a parameter p. A parameterized event is *non-ambiguous* if and only if for any $p, p' \in \mathcal{P}$ it holds that $p[1] \neq p'[1]$. With the non-ambiguity assumption, a monitor can generate a true or false result from system executions. Otherwise, since a parameter carries uncertain values, the monitoring results may contain uncertainties.

An execution of a system (i.e., a sequence of actions) is formalized by a finite *parameterized word* $\mathfrak{w} \triangleq (\mathfrak{e}_1, \ldots, \mathfrak{e}_n)$, which is a finite sequence of parameterized events. We denote with $\mathfrak{w}[i]$ the i^{th} parameterized event $\mathfrak{e}_i$ of $\mathfrak{w}$, whereas with $\mathfrak{w}^{[i,n]}$, the suffix $(\mathfrak{e}_i, \mathfrak{e}_{i+1}, \ldots, \mathfrak{e}_n)$. Finite parameterized words over the alphabet $\mathcal{P}\Sigma_E \triangleq (\Sigma_E \times 2^{\mathcal{S}\times\mathcal{D}})$ are defined as elements of $\mathcal{P}\Sigma_E^*$.

A PMLSC is *well-formed* if and only if for any $(\pi(x), \mathfrak{o}) \in \mathsf{COND}$ and $(x, s, \mathfrak{o}') \in \mathsf{ASSI}$, it holds that $\mathfrak{o}' \prec \mathfrak{o}$. From a well-formed PMLSC, all free variables in a proposition of a condition structure is evaluated with an assignment structure. The value is obtained from the parameterized event associated with the assignment structure. To simplify the notation, we denote $\hat{a_n}$ as a sequence $(a_1, \ldots, a_n)$ of as, where a is an event, a nominal or a value of a parameter. Let $\mathbb{B}$ be boolean values and Π be propositions in condition structures. The function $[\_ \Vdash \_] : \mathcal{P}\Sigma_E^* \times \Pi \mapsto \mathbb{B}$ is defined as follows.

$$[\mathfrak{w} \Vdash \pi(\hat{s_n})] = \begin{cases} true & \pi(\hat{d_n}) \text{ is true with } \mathfrak{w}[1] \triangleq (e_1, \mathcal{P}_1) \text{ and} \\ & \{(s_1, d_1), \ldots, (s_n, d_n)\} \in \mathcal{P}_1 \\ false & otherwise \end{cases}$$

A parameterized word $\mathfrak{w} = ((e_1, \mathcal{P}_1), \ldots, (e_n, \mathcal{P}_n))$ is *consistent* with a PMLSC U if and only if

- $\hat{e_n} \in W(U)$, and
- for any $(\pi_i, \mathfrak{o}_i) \in \mathsf{COND}$, it holds that $[\![\mathfrak{w} \Vdash \pi]\!] = true$.

By $PW(U)$, we denote the set of all parameterized words consistent with U.

The alphabet Σ_U of PMLSC U represents all relevant events of the property specified by U. The language of U is thus defined on $\mathcal{P}\Sigma_U^*$. Given parameters $\mathcal{S}\times\mathcal{D}$, the set $\mathcal{PT}_C$ of silent parameterized events (SPEs) of basic chart C is $\mathcal{PT}_C \triangleq \mathcal{T}_C \times 2^{(\mathcal{S}\times\mathcal{D})}$. That is, an SPE is a silent event carrying an arbitrary set of parameters. The semantics of PMLSCs is given as follows.

Definition 7 (*Semantics of PMLSCs*)**.** Given a PMLSC U, the language $\mathcal{L}(U)$ is defined as follows.

$$\begin{aligned} \mathcal{L}(U) \triangleq \{&(\mathfrak{w}_{\tau 0}, \mathfrak{e}_1, \mathfrak{w}_{\tau 1}, \ldots, \mathfrak{e}_n, \ldots, \mathfrak{e}_m, \mathfrak{w}_{\tau m}) | \mathfrak{e}^{[1,n]} \in PW(P), \\ &\forall i \in [0, n] : \mathfrak{w}_{\tau i} \in \mathcal{PT}_P^*, \mathfrak{e}^{[n+1,m]} \in PW(M), \text{ and} \\ &\forall j \in [n+1, m] : \mathfrak{w}_{\tau j} \in \mathcal{PT}_M^*\} \end{aligned}$$

In our previous work (Chai and Schlingloff, 2017), we introduced the language of the parameterized extended LSC (PeLSC), which is defined in a similar manner as PMLSCs. In contrast, the syntax of PeLSC does not contain the alphabet of the chart, and any event not appearing in the chart is defined to be a silent event. Additionally, in order to enhance the expressiveness of PeLSCs, we introduced a special symbol $\top$ for specifying events in **U**. Let $\mathcal{E}(\mathbf{U})$ be the set of events appearing in **U**, the semantics of $\top$ is $\mathcal{L}(\top, U) \triangleq \bigvee_{e\in\mathcal{E}(\mathbf{U})} e$. From the symbol $\top$, we define $\top^*$ and $\top^+$ as the sequence of any events and the non-empty sequence of events in $\mathcal{E}(\mathbf{U})$, respectively.

However, when the language of PeLSC is applied to construct monitoring specifications for real-world applications, the symbol $\top$ is sometimes not used intuitively and monitors have high incorrect report rates for some cases. For example, consider the following property of a communication process: *for given messages a, b, c, d, if message 'a' is transmitted, then message 'b' must be transmitted immediately afterwards. After that, message 'd' cannot be transmitted unless message 'c' is sent*. The language L of this property is as follows.

$$L = \overline{\{c, d\}^* a \{c, d\}^* \overline{b\{c, d\}^*}} \cdot \overline{\overline{\{a, b\}^* c \{a, b\}^*} d \{a, b\}^*}$$

The concatenating PeLSC charts, as shown in the left part of Fig. 3, is not equal to L. Messages c and d are non-silent events of the former chart as they have already appeared in the latter chart. Consequently, a monitor built from this specification reports 'false' against some correct executions, while 'true' against some incorrect executions. For the PMLSCs as shown in the right part of Fig. 3, this property can be specified in an intuitive manner with alphabets: The alphabets of the two chart are defined to be

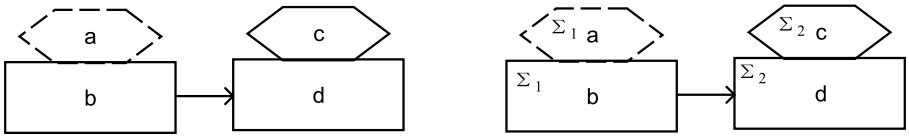

Fig. 3. Example PeLSCs (left) and PMLSCs (right).

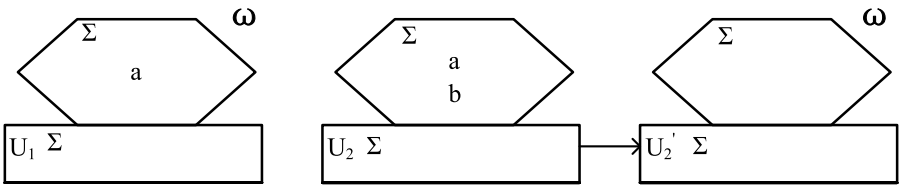

Fig. 4. An example of ∃PZ-monitorable Property.

$\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{c, d\}$, respectively. This makes c and d to be silent events in the former chart, and a and b in the latter chart.

Definition 8 (*PMLSC Specification*)**.** A PMLSC specification $\mathcal{U} \triangleq \{U_1, \ldots, U_n\}$ is a finite set of PMLSCs.

A PMLSC specification is a set of PMLSCs, each of which is expected to be satisfied by an execution of an SuM. The language $\mathcal{L}(\mathcal{U})$ defined by a PMLSC specification is

$$\mathcal{L}(\mathcal{U}) \triangleq \bigcap_{i\in[1,n]} \mathcal{L}(U_i).$$

With a PMLSC specification, a monitor can be generated to check whether an execution of an SuM (formalized by a parameterized word) satisfies or violates the properties of the specification.

Definition 9 (*PMLSC Monitoring*)**.** *PMLSC monitoring* is defined with the validation relation $[\![\_ \models \_]\!]$ between a parameterized word $\mathfrak{w} \in \mathcal{P}\Sigma^*$ and a monitor $\mathbf{M}(U)$ generated from $\mathcal{U}$, which is deductively given as follows.

$$[\![\mathfrak{w} \models \mathbf{M}(\mathcal{U})]\!] = \begin{cases} true & \mathfrak{w} \in \mathcal{L}(\mathcal{U}) \\ false & otherwise \end{cases}$$

A system **S** with an alphabet of $\mathcal{P}\Sigma$ satisfies a PMLSC specification *Spec* if and only if

$$\Big(\bigwedge_{1\le i\le n} [\![\mathfrak{w} \models \mathbf{M}(U_i)]\!]\Big) = true,$$

where $\mathfrak{w} \in \mathcal{P}\Sigma^*$ and $U_i \in Spec$.

As proposed in Aceto et al. (2019b) and Francalanza et al. (2017), the monitorability of the PMLSC properties is defined as follows. PMLSC property U is $\mathfrak{u}$-*monitorable* (here $\mathfrak{u} \in \mathcal{P}\Sigma_E^*$), if there exists some $\mathfrak{v} \in \mathcal{P}\Sigma_E^*$ such that whether U is satisfied or violated is determined by $\mathfrak{u}\mathfrak{v}$. Additionally, a property is ∀PZ-monitorable if it is $\mathfrak{u}$-monitorable for all finite words $\mathfrak{u}$, whereas a ∃PZ-monitorable if it is $\mathfrak{u}$-monitorable for some finite words $\mathfrak{u}$. According to the definitions, PMLSCs are defined over finite parameterized words. Therefore, every PMLSC property is ∀PZ-monitorable. Let $\mathcal{P}\Sigma^\omega$ be the language of infinite parameterized words. We define an *iteration* operator U^ω to specify the execution of chart U with infinite number of iterations. In the iteration operation, the language of PMLSC$^\omega$ is capable to specify properties over infinite words.

Proposition 1. *There exists some PMLSC$^\omega$ properties which are ∃PZ-monitorable.*

Proof. Consider the property "sequence c^*ab must occur first, and a occurs infinitely often", which defines the language $X \triangleq (\{b, c\}^*a\{b, c\}^*)^\omega) \cap (c^*ab\Sigma^\omega)$, where $\Sigma = \{a, b, c\}$. It is a ∃PZ-monitorable property. This is because of the fact that for a word c^*, there exists ab such that ab violates the property. The property is not ∀PZ-monitorable as for a word ac, there is no word $\mathfrak{v} \in \mathcal{P}\Sigma^\omega$ such that $ab\mathfrak{v}$ satisfies or violates the property. This property can be specified by the PMLSC$^\omega$ specification $\{U_1, U_2\}$ as shown in Fig. 4. The alphabets in the chart are as follows. $\Sigma_{1m} = \{a, b, c\}$, $\Sigma_{1p} = \{a, b, c\}$, $\Sigma_{2m} = \{a, b, c\}$ The language of chart U_1 is $\mathcal{L}(U_1) = \left(\overline{\{b, c\}^*a\{b, c\}^*\epsilon}\right)^\omega$, which equals $(\{b, c\}^*a\{b, c\}^*)^\omega$. The language of chart U_2 is $\mathcal{L}(U_2) = \overline{\overline{c^*ab\epsilon}} \cdot \Sigma^\omega$, which equals $c^*ab\Sigma^\omega$. Consequently, the specification $\{U_1, U_2\}$ defines the language X.

4.3. PMLSC properties

In this subsection, we illustrate features of PMLSCs by expressing some properties of a railroad crossing warning system. When a train approaches the railroad cross, the system closes a gate to prevent collisions between the train and road vehicles. In this subsection, we present patterns of interested monitoring properties of the system expressed by PMLSC specifications.

4.3.1. Reaction to trigger, absence of unsolicited reaction and strict trigger and reaction

Consider two behaviours:

- B1: Radar detects a train coming;
- B2: Radar alerts Controller about the coming train, and Controller closes the gate.

PMLSCs $U1$ and $U2$ in Fig. 5 specify B1 to be a sufficient and necessary condition, respectively.

In particular, PMLSC specification $\mathcal{U}^{iff} \triangleq \{U_1, U_2\}$ formulates B1 to be a necessary and sufficient condition of B2. By $\mathcal{P}$ and $\mathcal{M}$ we denote the languages of the prechart and the main chart of these two PMLSCs, respectively. The language defined by $\mathcal{U}^{iff}$ is

$$\mathcal{L}(\mathcal{U}^{iff}) \triangleq \left(\overline{\mathcal{P}\overline{\mathcal{M}}} \cap \overline{\overline{\mathcal{P}}\mathcal{M}}\right) \tag{2}$$

Eq. (2) is equal to

$$\left((\overline{\mathcal{P}\Sigma^*} \cup (\mathcal{P} \cdot \mathcal{M})) \cap (\mathcal{P}\Sigma^* \cup (\overline{\mathcal{P}} \cdot \overline{\mathcal{M}}))\right) = \left((\mathcal{P} \cdot \mathcal{M}) \cup (\overline{\mathcal{P}} \cdot \overline{\mathcal{M}})\right)$$

This language accepts all event words $w = uv$ such that $u \in \mathcal{P}$ if and only if $v \in \mathcal{M}$. More generally, let U and U' be two PMLSCs with $\Sigma = \Sigma'$, $P = P'$, $M = M'$, $\mathsf{ASSI} = \mathsf{ASSI}'$, $\mathsf{COND} = \mathsf{COND}'$, $Mod = \mathfrak{S}$ and $Mod' = \mathfrak{N}$. The PMLSC specification $\mathcal{U}^{iff} \triangleq \{U, U'\}$ expresses the property "the prechart is executed if and only if the main chart is executed afterwards". We use a double-lined

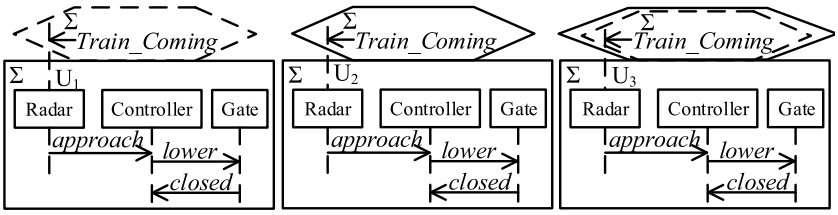

Fig. 5. PMLSCs for sufficient and necessary conditions.

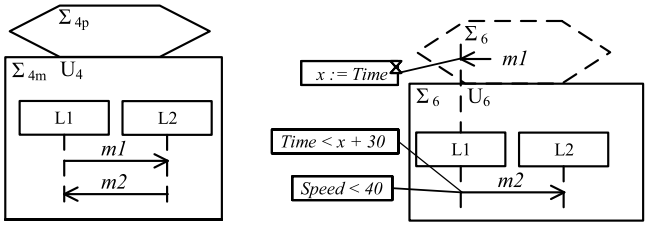

Fig. 6. PMLSCs for forbidden (left) and parameterized (right) behaviours.

notation for the abbreviation of this specification to express an *iff* prechart. For instance, the PMLSC U_3 in Fig. 5 expresses the specification $\mathcal{U}^{iff}$, as given above.

4.3.2. Forbidden behaviour

Consider the property of forbidden behaviour **FB** "The system cannot exhibit the behaviour of detect *approach* $(m1)$ first and then execute the command to *raise* $(m2)$ the gate afterwards". The language defined by **FB** is $\mathcal{L}(\mathbf{FB}) \triangleq \overline{\{(m1, m2)\}}$.

The property **FB** of forbidden $(m1, m2)$ is specified by PMLSC U_4 as shown on the left side of Fig. 6. This chart has an empty necessary prechart, whose alphabet is a singleton $\{m_\tau\}$ of a silent event. The intuitive interpretation of the chart is as follows: Chart U_4 accepts all event words $w = uv$ such that unless u is admitted by the prechart of U_4, sub-trace v cannot be admitted by the main chart of U_4. The reason is that any non-empty trace cannot be admitted by an empty chart, trace $(m1, m2)$, possibly interleaved with silent event sequence, is rejected by U_4. A strict proof of the property is given as follows.

Given U_4 with $\Sigma_{4p} = \{m_\tau\}$ and $\Sigma_{4m} = \{m1, m2\}$, we denote by $\mathcal{M}_4 \triangleq \{(m1, m2)\}$ the language of the main chart. The language defined by U_4 is $\mathcal{L}(U_4) \triangleq \overline{\varepsilon} \cdot \mathcal{M}_4$, which is equal to $(\{m_\tau\}^* \setminus \varepsilon) \cdot \mathcal{M}_4 = \{m_\tau\}^+ \cdot \mathcal{M}_4$. Because m_τ is a silent event of the main chart, the above language is equal to $\overline{\mathcal{M}_4}$, which is the language of property **FB**.

4.3.3. Parametric property

PMLSC U in Fig. 6 expresses the parametric property "If a message $m1$ is received, then a message $m2$ must be sent within 30 sec, and show that the train speed is under 40". By means of the assignment structure, the value of *Time* carried by $m1$ is stored in variable x. This value is used later by a condition structure to express that the time of receiving $m2$ must be less than $(x + 30)$ (i.e., within 30 sec after the time of receiving $m1$). The constraint of the "speed" feature is specified by the condition $(Speed < 40)$ combined with the sending of $m2$.

5. Expressiveness and complexity of PMLSCs

5.1. Expressiveness

In previous work (Chai and Schlingloff, 2017), we proposed an extension of LSCs (named PeLSCs) by introducing necessary precharts into standard LSCs. However, without explicating the alphabets of basic charts, the language of PeLSCs has some limitations in specifying monitoring properties in practice. One of the main shortcomings is that the language of PeLSCs cannot distinguish between silent events and unexpected events. Given a PeLSC specification, any event not appearing in any of the charts is directly interpreted to be a silent event. Consequently, by means of the intersection of charts, silent events of one chart can become non-silent events. This result is not expected in many cases and leads to false alarms in monitoring.

Theorem 1. *PMLSC is strictly more expressive than PeLSC.*

Proof. For any PeLSC U_e, there is a PMLSC U with $\Sigma_U = (\mathsf{E}(P) \cup \mathsf{E}(M))$ such that $\mathcal{L}(U_e) = \mathcal{L}(U)$. Therefore, PeLSCs do not have more expressiveness than PMLSCs.

We now show that the language $X = \overline{a \cdot \overline{b}}$ over $\Sigma = \{a, b, c\}$ is not definable by any PeLSC. According to the definitions, PeLSCs define languages of the form $\mathcal{L}(P) \cdot \overline{\mathcal{L}(M)}$. To define language X, the prechart and the main chart of a PeLSC U_e must define a and b, respectively. Event c is a silent event of U_e because the event does not appear in the chart. That is, a word $(accbc)$ is in the language $\mathcal{L}(U_e)$. However, because $(ccbc)$ is in $\overline{b}$ and $(accbc)$ is in $(a\overline{b})$, this word is not in X. Since X is defined by PMLSC U with $\Sigma_U = \{a, b, c\}$, the language of PMLSCs is strictly more expressive than the language of PeLSCs. □

As shown in Aceto et al. (2018), a language has more expressiveness for monitoring when it is closed under negation. The language of PeLSCs is closed under negation due to the introduction of the additional symbol $\top$. However, the symbol $\top$ is not used intuitively: if a language is not composed of Σ^*, a single PeLSC chart cannot contain the symbol $\top$. We now show that by introducing alphabets, without the symbol $\top$, the language of PMLSCs is still negation closure.

Theorem 2. *PMLSC is closed under negation.*

Proof. For a PMLSC U with a sufficient prechart, the language of U is $\mathcal{L}(U) \triangleq \mathcal{PM} \cup \overline{\mathcal{P}\Sigma^*}$. The complement of $\mathcal{L}(U)$ is $\overline{\mathcal{L}(U)} = (\overline{\mathcal{PM}} \cap \mathcal{P}\Sigma^*)$. Let the prechart of U_1 be an empty sufficient prechart, and let the main chart be the sequencing of message transmissions of P and M of U. The language of U_1 is $\mathcal{L}(U_1) = \varepsilon \cdot \tau^* \cdot \overline{\mathcal{PM}}$, which is equal to $\mathcal{PM}$. Let the prechart of $U2$ be a necessary prechart P and an empty main chart, and let the alphabet of the main chart be the same Σ as that of the prechart. The language of U_2 is $\mathcal{L}(U_2) = \overline{P} \cdot \tau^* \cdot \varepsilon$. Because all events in Σ are silent events, the sequence τ^* is actually Σ^*. The language of U_2 is thus $\mathcal{P}\Sigma^*$. Consequently, the PMLSC specification $\{U_1, U_2\}$ defines the complementation of U. □

5.2. Complexity

In this part, we investigate the efficiency of monitoring a system with PMLSC specifications, that is, the complexity of the parameterized word problem of PMLSCs with respect to the lengths

of the parameterized event traces. We introduce *simple numerical comparisons* (*SNCs*) that express only the comparisons of numerical values. For example, a simple numerical proposition can be $x \leq 5$, $s = 3$ and $s < (x + 10)$. By PMLSC-SNCs we denote a subclass of PMLSCs in which all propositions are SNC. We first prove the complexity of PMLSC-SNC.

The hybrid logic (Franceschet et al., 2003) (**HL**) is a well defined formal language that extends **LTL**. Let **HL**[↓] be a subclass of hybrid logic with "*down arrow*" (excluding the "*at*" operator). The *down arrow* operation $x \downarrow s$ is combined with a temporal formula. At the time of the local temporal context, it stores the value of parameter s in variable x. The formula of $x \downarrow s.\varphi(x)$ asserts about the parameterized word $\mathfrak{w} = (e_1, \mathcal{P}_1), \ldots, (e_n, \mathcal{P}_n)$ that $\mathfrak{w}$ satisfies $\varphi(d_1)$. The formula of $\varphi(d_1)$ is obtained by replacing all the free occurrences of variable x in φ with constant d_1, where $(s_1, d_1) \in \mathcal{P}_1$. For example, the formula

$$\mathbf{G}(x \downarrow time.(a \Rightarrow \mathbf{F}(x' \downarrow time.y \downarrow speed(b \wedge (y < 40) \wedge (x' < 30 + x))))).$$

asserts that whenever event 'a' occurs in a parameterized word, then event 'b' must eventually occur within 30 s, and the speed value carried by b must be smaller than 40. As the logic of **HL**[↓] captures the features of temporal logic properties and parameter constrains, we use it to analyse the expressiveness of PMLSCs. The language of **HL**[↓] is formally defined as follows.

Definition 10 (*Syntax of HL[↓]*)**.** Let $\mathcal{E}$ be a finite set of events, $\mathcal{S}$ be a finite set of nominals and $\mathcal{X}$ be a finite set of variables. The simple numerical terms θ and the formulae φ of **HL**[↓] is inductively formed according to the following grammar, where $x \in \mathcal{X}$, $r \in \mathbb{R}$, $e \in \mathcal{E}$, $s \in \mathcal{S}$ and $\sim \in \{<, \leq, =, >, \geq\}$:

$$\theta := x + r \mid r$$
$$\varphi := \top \mid e \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathbf{U} \varphi_2 \mid \theta_1 \sim \theta_2 \mid x \downarrow s.\varphi$$

The following shorthands are used in **HL**[↓] as in **LTL**: $\mathbf{F}\varphi$ stands for $\top\mathbf{U}\varphi$, $\mathbf{G}\varphi$ stands for $\neg\mathbf{F}\neg\varphi$ and $\mathbf{X}\varphi$ stands for $(\neg\top)\mathbf{U}\varphi$.

Given a set of variables $\mathcal{V}$ and their domains $\mathcal{D}$, an *assignment* g for $\mathcal{V}$ is a mapping $g : \mathcal{V} \mapsto \mathcal{D}$ such that $g(x + r) = g(x) + r$ and $g(r) = r$. Given an assignment g and a parameter (s, d), we define $g^x_{(s,d)}$ by $g^x_{(s,d)}(x) = d$ and $g^x_{(s,d)}(y) = g(y)$ for $y \neq x$. We define semantics of **HL**[↓] over finite linear models as follows.

Definition 11 (*HL[↓] Finite Linear Semantics*)**.** Let $\mathfrak{w}$ be a finite parameterized word with $i \in \mathbb{N}_{\geq 0}$ being a position, e be an event, $p \triangleq (s, d)$ be a parameter, and φ_1 and φ_2 are any **HL**[↓] formulae. Let $\mathfrak{w}[i][1]$ and $\mathfrak{w}[i][2]$ being the event and the parameter of the ith position of $\mathfrak{w}$, respectively. The satisfaction relation $(\mathfrak{w}, i, g) \models \varphi$ of φ over a linear model is defined inductively as follows:
$(\mathfrak{w}, i, g) \not\models \bot$;
$(\mathfrak{w}, i, g) \models e$ iff $e = \mathfrak{w}[i][1]$;
$(\mathfrak{w}, i, g) \models \varphi_1 \wedge \varphi_2$ iff $(\mathfrak{w}, i, g) \models \varphi_1$ and $(\mathfrak{w}, i, g) \models \varphi_2$;
$(\mathfrak{w}, i, g) \models \varphi_1 \mathbf{U} \varphi_2$ iff there exists $i < j < |\mathfrak{w}|$ with $(\mathfrak{w}, j, g) \models \varphi_2$ and for all $i < j' < j$ it holds that $(\mathfrak{w}, j', g) \models \varphi_1$;
$(\mathfrak{w}, i, g) \models \theta_1 \sim \theta_2$ iff $g(\theta_2) \sim g(\theta_1)$;
$(\mathfrak{w}, i, g) \models x \downarrow s.\varphi$ iff there exists a parameter $(s, d) \in \mathfrak{w}[i][2]$ such that $(\mathfrak{w}, i, g^x_{(s,d)}) \models \varphi$.

Theorem 3. *The language of PMLSC-SNC is at most as expressive as* ***HL***[↓].

Proof. We have to show that a PMLSC-SNC chart U can be translated into an equivalent **HL**[↓] formula. The resulting formula $\varphi(U)$ is the conjunction of two subformulae $\mathcal{X}(U)$ and $\mathcal{Y}(U)$, where $\mathcal{X}(U)$ specifies the temporal relations between events and $\mathcal{Y}(U)$ expresses the constraints of parameters carried by events.

For given alphabet $\Sigma = \{e_1, \ldots, e_n\}$ of events and $\mathcal{N}_\Sigma \triangleq \bigwedge_{e \in \Sigma} (\neg e)$, Kugler et al. (2005) proposed the following LTL fragment for defining a finite sequence of events $(e_1, \ldots, e_n)$

$$\psi = \mathcal{N}_\Sigma \mathbf{U}(e_1 \wedge (\mathbf{X}(\mathcal{N}_\Sigma \mathbf{U}(e_2 \wedge (\mathbf{X}(\mathcal{N}_\Sigma \mathbf{U} e_3 \ldots)\ldots)\ldots).$$

It indicates that nothing can occur before event e_1, after that nothing cannot occur before event e_2, and so on. Inspired by this LTL fragment, we develop the following **HL**[↓] formulae for specifying features of the languages defined by PMLSCs. The following formula $\mathcal{A}(\sigma, \Sigma)$ specifies a finite sequence of events $\sigma = e_1, \ldots, e_n$ over Σ^*:

$$\mathcal{A}(\sigma, \Sigma) \triangleq \mathcal{N}_\Sigma \mathbf{U} (e_1 \wedge \mathbf{X} (\mathcal{N}_\Sigma \mathbf{U} (e_2 \wedge \cdots \wedge \mathbf{X}(\mathcal{N}_\Sigma \mathbf{U} (e_n \wedge \mathbf{X}\, \mathbf{G}\, \mathcal{N}_\Sigma) \cdots).$$

Different from the Kugler's paradigm, we put the subformula of $\mathbf{X}\ \mathbf{G}\ \mathcal{N}_\Sigma$ at the end of the last event e_n of the sequence. This formula is used to specify that no event is allowed to occur after e_n. If events $\{e_1, \ldots, e_n\}$ appearing in σ is a subset of Σ, any sequence interleaved with the events in $\Sigma \setminus \{e_1, \ldots, e_n\}$ is rejected by the formula of $\mathcal{A}(\sigma, \Sigma)$. Given a superset $\Sigma_\mathbf{S}$ of Σ (i.e., $\Sigma_\mathbf{S} \supset \Sigma$), the formula specifies the word σ interleaving with any finite (possible empty) sequence of silent events, i.e., the language of

$$\{(\Sigma^*_\tau, e_1, \Sigma^*_\tau, \ldots, \Sigma^*_\tau, e_n, \Sigma^*_\tau) \mid (e_1, \ldots, e_n) = \sigma \text{ and } \Sigma_\tau = \Sigma_\mathbf{S} \setminus \Sigma\}.$$

According to the definitions of PMLSCs, a basic chart C specifies a finite set of finite sequences of words $W(C)$. Give the "*exclusive or*" operation $\oplus$, the language $W(C)$ can be specifies by a formula $\mathcal{B}(C, \Sigma)$ as follows

$$\mathcal{B}(C, \Sigma) \triangleq \bigoplus_{\sigma \in W(C)} \mathcal{A}(\sigma, \Sigma).$$

This formula expresses that exactly one trace of C is executed.

The formula $\mathcal{B}(C_1 \rightarrow C_2, \Sigma)$ for concatenation of charts can be defined in a similar manner.

Given a PMLSC-SNC U, the formula $\mathcal{R}(U)$ defines the temporal relations of U as follows

$$\mathcal{R}(U) \triangleq \begin{cases} \mathcal{B}(P, \Sigma_U) \Rightarrow \mathcal{B}(P \rightarrow M, \Sigma_U) & \text{if } Mod = \mathfrak{S} \\ \neg\mathcal{B}(P, \Sigma_U) \Rightarrow \neg\mathcal{B}(P \rightarrow M, \Sigma_U) & \text{if } Mod = \mathfrak{N} \end{cases}$$

Let $e(x) \in (\Sigma \times \mathbb{N}_{>0})$ be an event and the occurrence time of events in a chart. Let $\mathsf{E}(U) = \{e_1(x_1), \ldots, e_n(x_n)\}$ be the set of events appearing in PMLSC U, where $x_1, \ldots, x_n$ are the occurrence times of the respective events. The formula

$$\mathcal{C}(e(x)) \triangleq \underbrace{\mathbf{F}(e \wedge \mathbf{XF}(e \wedge \cdots \mathbf{XF}\ e) \cdots)}_{\text{x times of } \mathbf{F}}$$

expresses that event e has been executed x times. A sequence of events is accepted by the formulae if and only if an event e occurs at some position i and the suffix starting from position $i+1$ satisfies the formula of $\underbrace{\mathbf{F}(e \wedge \mathbf{XF}(e \wedge \cdots \mathbf{XF}\ e) \cdots)}_{\text{x}-1 \text{ times of } \mathbf{F}}$. The formula $\mathcal{D}(U, e) \triangleq (\neg \bigwedge_{e(x) \in \mathsf{E}(U)} \mathcal{C}(e(x)))\ \mathbf{U}\ e$ specifies that an event e cannot be executed unless all events in U have been executed. With the additional formula $\mathcal{D}(U, e)$, the concatenation of PMLSC-SNCs can be defined by formulae according to the definitions. Furthermore, a PMLSC-SNC specification $\{U_1, \ldots, U_n\}$ can be translated into the conjunction of $\mathcal{X}(U_i)$.

A simple numerical comparison π of a condition structure $\delta \in \mathsf{COND}$ in U can be either with or without variables.

For the case of $\delta = (\pi(s), \mathfrak{o})$ (i.e., π involves no variables) with $\mathfrak{o} = e$, we define the following HL formula.

$$\mathcal{Y}(\delta) \triangleq \mathbf{G}(x \downarrow s.e \Rightarrow \pi(x))$$

This formula specifies that if e occurs in a parameterized word, then it must carry a parameter (s, d) such that $\pi(d)$ is true.

For the case of $\delta = (\pi(s, x), \mathfrak{o})$ (i.e., π involves a variable x), there must be an assignment structure $(s', x, \mathfrak{o}')$ with $\mathfrak{o}' \prec \mathfrak{o}$. Let $\mathfrak{o}[1] = e$ and $\mathfrak{o}' = e'$, we define the following formula.

$$\mathcal{Y}(\delta) \triangleq \mathbf{G}((e' \wedge \mathbf{F}\, e) \Rightarrow x \downarrow s'.(e' \wedge \mathbf{F}\, x' \downarrow s.(e \wedge \pi(x', x))))$$

This formula specifies that if both e and e' occur in a parameterized word with $e' \prec e$, then the simple numerical proposition $\pi(s, x)$ must hold with the parameters carried by e. According to the assignment, the value of x in $\pi(s, x)$ is given by the parameter named s' carried by e'. The value of s is given by the parameter named s carried by e itself. If π contains more than one variable, the formula $\mathcal{Y}(\delta)$ can be built in the same manner. The formula $\mathcal{Y}(\delta)$ then specifies the constraints of the parameters in U.

According to these definitions, any parameterized word $\mathfrak{w}$ is admitted by a PMLSC-SNC U iff $\mathfrak{w} \models \varphi(U)$, where $\varphi(U) = \mathcal{X}(U) \wedge \mathcal{Y}(U)$. Therefore, any PMLSC can be translated into an equivalent **HL**[↓] formula. □

With the above theorem, the complexity of PMLSC-SNC for monitoring is equal to the complexity of the *parameterized word problem* of **HL**[↓]: Given a parameterized word $\mathfrak{w}$ and an **HL**[↓] formula φ, decide whether or not $\mathfrak{w} \models \varphi$ holds. Given $\mathfrak{w} = ((e_1, \mathcal{P}_1), \ldots, (e_n, \mathcal{P}_n))$, the satisfaction $\mathfrak{w} \models x \downarrow s.\varphi(\pi(x))$ can be translated into $(e_1, \ldots, e_n) \models \varphi(d)$, where $(s, d) \in \mathcal{P}_1$. Since $\pi(x)$ is an SNC, whether $\pi(d)$ is *true* or *false* can be decided directly; therefore, the parameterized word problem of **HL**[↓] is translated into the word problem of **LTL**. Since the complexity of the latter problem is linear with respect to lengths of the words (Roşu and Havelund, 2005), the following theorem holds.

Theorem 4. *The complexity of monitoring PMLSC-SNC properties is linear with respect to the lengths of the parameterized words.*

By introducing quantifiers into propositions of condition structures, another subclass of PMLSCs can be obtained, that has more expressive power than PMLSC-SNCs. For example, a property can be specified to state that the data carried by a message has some features of the first order logic. However, the complexity of the language increases with increasing expressiveness. By investigating the complexities of the language, we show below whether this subclass of PMLSCs is applicable for monitoring. Monitoring a system is essentially a problem of checking whether a linear model satisfies some properties. Although the scale of this problem is smaller than that of general model checking problems, the monitoring properties specified by this language have unacceptable efficiency.

Let $\mathbf{P}(\forall, \exists)$ and $\mathbf{P}(\forall, \exists, \infty)$ be first-order propositions with quantifiers over infinite and finite domains, respectively. By *PMLSC-FOL* and *PMLSC-FOL*$^\infty$, we denote the languages of super classes of PMLSC-SNCs involving $\mathbf{P}(\forall, \exists)$ and $\mathbf{P}(\forall, \exists, \infty)$, respectively. Intuitively, the complexity of checking PMLSC-FOL is related to the complexity of solving the conditions in the charts. Therefore, the hypothesis is that the complexities of PMLSC-FOL$^\infty$ and PMLSC-FOL are at least undecidable and NP-complete, respectively. In the following, we provide strict proof of the complexities of these languages through the *tiling problem* (Kari, 2007).

Theorem 5. *The complexity of monitoring PMLSC-FOL properties is NP-complete.*

Proof. We first prove that the complexity of the parameterized word problem of PMLSC-FOL is in NP. Given a PMLSC-FOL U with a proposition $\mathsf{P}(\exists, \forall)$, where $\mathsf{P}(\exists, \forall)$ contains a variable x and a domain $\mathcal{D}$ with $|\mathcal{D}| < \infty$. Whether a parameterized word $\mathfrak{w}$ is in the language $\mathcal{L}(U)$ of U can be determined through the following two steps:

1. Translate the PMLSC-FOL U into a PMLSC-SNC U' by guessing an assignment of x from the domain $\mathcal{D}$, and
2. Translate the chart U' into an equivalent HL[↓] formula $\varphi(U')$.

According to Theorem 4, the complexity of PMLSC-SNC is linear. Therefore, the problem of monitoring PMLSC-FOL is in NP.

We then prove that the problem is NP-complete. A tile is defined as a pair $tl = (t, c)$, where t and c represent the name and the four colours of the tile, respectively. We define c as a tuple $c \triangleq \langle col_{top}, col_{bot}, col_{lef}, col_{rig} \rangle$ representing the colours of the top, bottom, left and right edge of the tile. Given a finite set of tiles T, a finite $n \times n$ plane and a sequence $\widetilde{T} = (tl_1, \ldots, tl_n)$. For any tl_i and tl_{i+1} in $\widetilde{T}$, it holds that the right colour of tl_i is equal to the left colour of tl_{i+1}. Let T be a set of tiles, a *tiling* can be formalized with a function $f : \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \mapsto \mathsf{T}$. For $i, j \in \mathbb{N}_{\geq 0}$, the function $f(i, j) = tl$ indicates that the tile tl is placed at position (i, j) on the plane. By $f(i, j)_{top} \triangleq c[1]$, $f(i, j)_{bot} \triangleq c[2]$, $f(i, j)_{lef} \triangleq c[3]$, $f(i, j)_{rig} \triangleq c[4]$, we denote the colours of the tile $f(i, j)$ on the top, bottom, left and right edges, respectively. The *bounded tiling problem* requires to determine whether a tiling of the $n \times n$ plane exists in the given set T of tiles, where the first row is the given sequence $\widetilde{T}$. This problem can be formulated by using the proposition P_2 as follows.

$$(\forall i \in [1, n) \wedge \forall j \in [1, n) \wedge \exists tl \in \mathsf{T}) : ((f(i, j) = tl) \wedge (f(i, j)_{rig} = f(i + 1, j)_{lef}) \wedge (f(i, j)_{top} = f(i, j + 1)_{bot})). \quad (3)$$

The bounded tiling problem can be translated into U_2 as shown in Fig. 7. The prechart stores the tiles in the first row of the plane. After that, the main chart describes a tiling of the plane with the given tiles. Let $\mathfrak{w} = ((a, \mathsf{T}), (b, \mathsf{T}))$ be a word formed with the set of tiles representing the parameters carried by the events. Checking whether the word is admitted by U_2 is equivalent to the problem of deciding whether there exists a tiling for the plane. Since the above bounded tiling problem has been proved to be NP-complete, the complexity of solving the parameterized word problem of PMLSC-FOL also becomes NP-complete. □

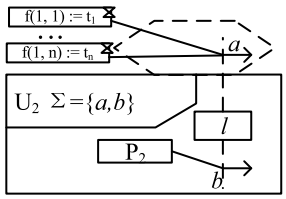

Fig. 7. PMLSC of tiling on a finite plane.

In the same manner, the following theorem can be proved by using the tiling problem of an infinite plane.

Theorem 6. *The problem of monitoring PMLSC-FOL*$^\infty$ *properties is undecidable.*

Proof. By allowing infinite domains, the language of PMLSC-FOL$^\infty$ can be used to specify tiling of an infinite plane. This property can be specified with the chart U_2 as shown in Fig. 7 by replacing P_2 with the following proposition.

$$(\forall i \in [1, \infty) \wedge \forall j \in [1, \infty) \wedge \exists tl \in \mathsf{T}) : ((f(i, j) = tl) \wedge (f(i, j)_{rig} = f(i + 1, j)_{lef}) \wedge (f(i, j)_{top} = f(i, j + 1)_{bot})). \quad (4)$$

Since unbounded tiling problem is undecidable, the word problem of PMLSC-FOL$^{\infty}$ is undecidable. □

Since the complexities of PMLSC-FOL and PMLSC-FO$^{\infty}$ are too high, these sub-classes of PMLSC are not suitable for performing online monitoring. Therefore, we focus on PMLSC-SNC in the following part of the paper.

6. Monitoring algorithms of PMLSC-SNCs

The translation of the standard LSCs into **LTL** was been introduced in Kugler et al. (2005). By assigning to variables parameters carried by events, a **HL**[↓] formula can be translated into an **LTL** formula, which has online verification algorithms. As shown in Section 5.3, PMLSC-SNC is not more expressive than **HL**[↓]. Therefore, a naive PMLSC-SNC monitoring algorithm can directly translate charts into **HL**[↓] formulae. The correctness of an SuM is then decided by checking whether the executions satisfy the resulting formulae. Unfortunately, this algorithm is not efficient enough for large-size (offline) monitoring system executions. Since the formulae have to specify all partial orders of events induced by charts, the sizes of the resulting formulae are very large. For example, the formula from the simple property U_3 in Fig. 5 has over 80 operations/propositions. The result cannot be obtained efficiently when checking the correctness of a large log file (i.e., a long parameterized event trace) directly with such a large formula. Furthermore, PMLSCs with iterations essentially introduce (infinite) Kleene star into the expressions: such a chart thus cannot be translated into an **HL**[↓] formula.

We develop an explicit algorithm to check PMLSC properties efficiently to solve these problems. For a given PMLSC U, a monitor $\mathfrak{M}(U)$ can be synthesized via the following steps.

1. Translating all basic charts (i.e., precharts and main charts) and parameter constraints of PMLSCs into **HL**[↓];
2. Filtering the word by projecting away all silent events (according to the alphabet of the PMLSC specification);
3. Splitting the word into a set of slices with respect to the precharts and main charts;
4. Checking the satisfaction relations between slices and basic charts to decide whether a PMLSC is "triggered" independently (according to the modality of its prechart); and
5. Checking whether the word satisfies all triggered PMLSCs in the specification.

The correctness of the monitors synthesis (Bonakdarpour and Finkbeiner, 2020; Basin et al., 2020) can be shown as follows.

Lemma 1. *For any PMLSC U, $\mathfrak{M}(U)$ is a correct monitor for U.*

As defined in Francalanza and Seychell (2015), the correctness of a monitor $\mathfrak{M}(U)$ is defined as follows: For any parameterized word $\mathfrak{w}$, if $\mathfrak{w}$ cannot be admitted by U then the monitor should detect it, and vice versa. Assume that the **HL**[↓] formula $\varphi(C)$ is a correct translation of the basic chart C. The formula is capable to decide whether a parameterized word is admitted by a basic chart. According to the definitions, the execution of the prechart precedes the execution of the main chart. With respect to a PMLSC, a word can be split into two slices $(\mathfrak{w}_1, \mathfrak{w}_2)$ in sequence. The former slice exhibits the execution of the prechart, whereas the latter slice exhibits that of the main chart. Whether the original word can be admitted by U with different modals (i.e., sufficient prechart and necessary prechart) of PMLSCs is decided by checking the satisfaction of $\mathfrak{w}_1$ and $\mathfrak{w}_2$, respectively.

6.1. Translation of PMLSC-SNCs into *HL*[↓]

By means of event occurrences, we can distinguish every event appearing in a chart. Inspired by the translation of LSC to **LTL** proposed by Kugler et al. (2005), we develop an efficient translation algorithm from PMLSC to **HL**[↓]. Let $\mathsf{PO}(C)$ and $\mathsf{E}(C)$ be sets of partial orders $(e \prec e')$ induced by a basic chart C and all events appearing in C, respectively. From C, an alphabet Σ_C of C and a set E of events, a formula $\varphi(C, \Sigma) \triangleq \varphi_1(C) \wedge \varphi_2(C) \wedge \varphi_3(C)$ can be defined, where

$$\varphi_1(C) \triangleq \bigwedge_{(e \prec e') \in \mathsf{PO}(C)} (\neg e' \ \mathbf{U} \ e) \tag{5}$$

$$\varphi_2(C) \triangleq \bigwedge_{e \in \mathsf{E}(C)} (\neg e \ \mathbf{W} \ (e \wedge \mathbf{X} \ \mathbf{G} \ \neg e)) \tag{6}$$

$$\varphi_3(C) \triangleq \bigwedge_{e \in \Sigma_C} \mathbf{F} \ e \wedge \bigwedge_{e \in (\mathsf{E} \setminus \Sigma_C)} \mathbf{G} \ \neg e \tag{7}$$

The correctness of the translations can be represented by the following lemma.

Lemma 2. *Given a PMLSC U, any word $\mathfrak{w}$ is admitted by the prechart P (resp. the main chart M) if and only if*
$\mathfrak{w} \models \varphi(P, \Sigma_U)$ *(resp.* $\mathfrak{w} \models \varphi(M, \Sigma_U)$*).*

The formula $\varphi_1(C)$ specifies that if $e \prec e'$, then e' cannot appear in a word before e; the formula $\varphi_2(C)$ specifies that each e can appear in a word at most one time; and the formula $\varphi_3(C)$ specifies that all the events in the alphabet occur eventually and any event not in the alphabet cannot occur in a word. According to the definitions, these formulae together represent the temporal requirements of event occurrences in a basic chart.

The simple numerical proposition π of a condition structure $\delta \in \mathsf{COND}$ in U can be either with or without variables. For the case of $\delta = (\pi(s), \mathfrak{o})$ (i.e., π involves no variables) with $\mathfrak{o}[1] = e$, we define the following $\mathsf{HL}[\downarrow]$ formula.

$$\mathcal{Y}(\delta) \triangleq \mathbf{G}(x \downarrow s.e \Rightarrow \pi(x))$$

This formula specifies that if e occurs in a parameterized word, then it must carry a parameter (s, d) such that $\pi(d)$ is true.

For the case of $\delta = (\pi(s, v), \mathfrak{o})$ (i.e., π involves a variable v), there must be an assignment structure $(s', v, \mathfrak{o}')$ with $\mathfrak{o}' \prec \mathfrak{o}$. Let $\mathfrak{o}[1] = e$ and $\mathfrak{o}'[1] = e'$: we define the following formula.

$$\mathcal{Y}(\delta) \triangleq \mathbf{G}((e' \wedge \mathbf{F} \ e) \Rightarrow v \downarrow s'.(e' \wedge \mathbf{F} \ v' \downarrow s.(e \wedge \pi(v', v))))$$

This formula specifies that if both e and e' occur in a parameterized word with $e' \prec e$, then the simple numerical proposition $\pi(s, v)$ must hold with the parameters carried by e. According to the assignment, the value of v in $\pi(s, v)$ is given by the parameter named s' carried by e'. The value of s is given by the parameter named s carried by e itself. The formula $\theta(U) = \bigwedge_{\delta \in \mathsf{COND}} \mathcal{Y}(\delta)$ specifies the parameter constraints of U.

6.2. Rewriting algorithms for *HL*[↓]

By means of formula rewriting, we develop algorithms for checking **HL**[↓] over parameterized words. Given a finite parameterized word $\mathfrak{w} = ((e_1, \mathcal{P}_1), \ldots, (e_n, \mathcal{P}_n))$ and an **HL**[↓] formula $\varphi = x_1 \downarrow s_1. \cdots x_m \downarrow s_m.\varphi(x_1, \ldots, x_m)$, the algorithm for checking whether $\mathfrak{w}$ satisfies φ is as shown in Algorithm 1. In this algorithm, the formula is rewritten by resuming the first parameterized event in $\mathfrak{w}$. No down arrow binder exists at the first depth for some intermediate formulae. We use the LTL rewriting algorithm proposed in Chen and Roşu (2005) to verify the resulting intermediate LTL formulae. This process is performed iteratively

Algorithm 1: Rewriting algorithm for **HL**[↓]

```
1  Function Check(𝔴, φ)
2      while 𝔴 ≠ ε do
3          sDb ← Binder (φ)                          // subformulae ↓
4          sND ← Nobinder (φ)                        // Non-binder
           subformulae of φ
5          if |sND|> 0 then                          // There exists ↓
6              for j ← 1 to |sND | do
7                  if ∃(s, d) ∈ P_i and s = s_j then  // if ε_i
                   carries s_j
8                      sND (x_j) ← sND (φ(d))
                       // assigning x_j to d
9                  else                              // if ε_i carries no s_j
10                     φ ← ⊥                         // rewrite φ to ⊥
11             φ ← sND // consume ↓ operator
12         φ ← LTLrw (φ)// LTL rewrite (Chen and Roşu,
               2005)
13         𝔴 ← 𝔴_1                                   // Consume 𝔴[1]
14     return [[φ]]                                  // assign φ to Boolean
```

Algorithm 2: Word slicing algorithm for PMLSC-SNC

```
1  Function Slice(𝔴, U)
2      B̂C ← (P_1, M_1, ..., P_n, M_n)
3      for i ← 1 to |𝔴| do
4          if ε_i ∈ PΣ_U then                        // ε_i is non-silent
5              for j ← 1 to |B̂C| do                  // |B̂C|= 2n
6                  if ε_i ∉ (Σ_U \ E(C)_j) then      // in C_j
7                      DW[j] ← (DW[j] · ε_i);
                                                     /* C_j execution */
8      for k1 ← 1 to 2n do
9          for k2 ← 1 to |DW[k1]| do
10             while Ê(C_k1) ≠ ε do
11                 if Ê(C_k1).have(DW[k1][k2]) = true then
                   // DW[k1][k2] in Ê(C_k1)
12                     sW(C_k1)[k3] ← sW(C_k1)[k3] · DW[k1][k2] ;
13                     Ê(C_k1).Remove(DW[k1][1]);
14                     DW[k1].Remove(DW[k1][k2]);
15                 else
16                     k2++;
17             WSet = WSet ∪{sW(C_k1)[k3++]}     // C_k1 admits
               sW(C_k1)[k3]
18     return WSet
```

until all the parameterized events are consumed. Since the length of $\mathfrak{w}$ is finite, the algorithm is terminated.

Example. The **HL**[↓] formula $\varphi = (x \downarrow t).((\neg a)\mathbf{U}((y \downarrow t).(b \wedge (x + y < 5))))$ represents the property "a cannot happen until b occurs within 5 sec". For the given parameterized word $\mathfrak{w} = (\neg a, (t, 1)), (\neg a, (t, 2)), (b, (t, 3))$, we denote $\mathfrak{w}_i$ as the suffix of $\mathfrak{w}$ starting from the ith position. Let $\lfloor \_ \models \_ \rfloor \in \{true, false\}$ be the satisfaction between a parameterized word and an **HL**[↓] formula, the formula rewriting process can be expressed as follows.

$$
\begin{aligned}
\lfloor \mathfrak{w} \models \varphi \rfloor &= \lfloor (\neg a, \mathfrak{w}_2) \models ((\neg a) \vee (y \downarrow t).(b \wedge (1 + y < 5)) \\
&\quad \wedge ((\neg a)\mathbf{U}((y \downarrow t).(b \wedge (1 + y < 5))))) \rfloor \ (line5 \sim 8) \\
&= \lfloor (\mathfrak{w}_2) \models (\top \vee \bot) \\
&\quad \wedge ((\neg a)\mathbf{U}((y \downarrow t).(b \wedge (1 + y < 5))))) \rfloor \ (line12 \sim 13) \\
&= \lfloor (\neg a, \mathfrak{w}_3) \models ((\neg a) \vee (b \wedge (1 + 2 < 5)) \\
&\quad \wedge ((\neg a)\mathbf{U}((y \downarrow t).(b \wedge (1 + y < 5))))) \rfloor \ (line5 \sim 8) \\
&= \lfloor ((b, (t, 3)) \models ((\neg a) \vee (y \downarrow t).(b \wedge (1 + y < 5)) \\
&\quad \wedge ((\neg a)\mathbf{U}((y \downarrow t).(b \wedge (1 + y < 5))))) \rfloor \ (line12 \sim 13) \\
&= \lfloor b \models ((\neg a) \vee (b \wedge (1 + 3 < 5)) \\
&\quad \wedge ((\neg a)\mathbf{U}((y \downarrow t).(b \wedge (1 + y < 5))))) \rfloor \ (line5 \sim 8) \\
&= true \ (line12 \sim 13)
\end{aligned}
$$

6.3. Slicing algorithm for PMLSC-SNC monitoring

Before checking the correctness of a parameterized word $\mathfrak{w}$, the word is preprocessed with Algorithm 2. The word is first filtered according to the alphabet of a given PMLSC-SNC specification. Then, the word is divided into a set of sub-traces, each of which represents an execution of a prechart or a main chart in the underlying PMLSC-SNC specification. The positions of all parameterized events in $\mathfrak{w}$ are recorded to restore the sub-traces into a word with the original execution order.

6.4. Monitoring algorithm for PMLSC-SNCs

Whether a parameterized word $\mathfrak{w}$ satisfies a PMLSC-SNC specification can be determined by checking the sliced sub-traces T (as shown in Algorithm 3). The algorithm first decides which PMLSC-SNCs in the specification are "triggered" according to the modalities (necessary or sufficient) of precharts. For a PMLSC-SNC with a sufficient prechart, the PMLSC-SNC is triggered when the prechart is satisfied; For a PMLSC-SNC with a necessary prechart, the PMLSC-SNC is triggered when the prechart is not satisfied. If a PMLSC-SNC is triggered, the parameter constraints should also be satisfied. A word satisfies a PMLSC specification iff the word satisfies all triggered charts with the correct execution orders. In addition, the word satisfies concatenations (as well as iterations) of triggered charts iff the execution orders are correct. Since each sub-trace can be checked by an independent CHECK procedure, this algorithm can verify large executions in parallel. As there is no backtracking, this algorithm is also suitable for online monitoring.

Example. Consider the parameterized word $\mathfrak{w} = (a, \mathcal{P}), (e, \mathcal{P}), \ldots, (c, \mathcal{P}), (e, \mathcal{P}), (b, \mathcal{P}), (e, \mathcal{P}), \ldots, (d, \mathcal{P})$ and the PMLSC specification $\mathcal{U} = \{U_1, U_2\}$, where the alphabets of U_1 and U_2 are $\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{c, d\}$, respectively. As shown in algorithm 2, the silent events $(e, \mathcal{P})$ are first filtered and then the trace is split into four slices according to the occurrences of events P_1, M_1, P_2 and M_2. The slices and the executions of the basic charts are stored in *DW* and *Wset*, respectively. The correctness of the execution, the partial order of executions, and the parameter constraints are then checked by algorithm 3.

7. A case study: monitoring RBCs of CTCS

In this section, we present an example of monitoring RBCs deployed on a main high-speed railway line of China. We choose this example because a concrete accident occurred in this system, see Fig. 8. Due to a sequence of internal errors, the CTCS level 3 system of train A fails to prevent the train from a collision accident with train B. By analysing the accident, the errors are found to arise from incorrect interactions between the RBC and interlocking as follows. According to the schedule, train A did not plan to stop at the station, and the interlocking locked a main line route *X-IG-121* for the train. With respect to this route, the RBC provided an MA to train A with the EoA at distance kilometre 121 (dk121). Then, the CTC decided that train A was still far

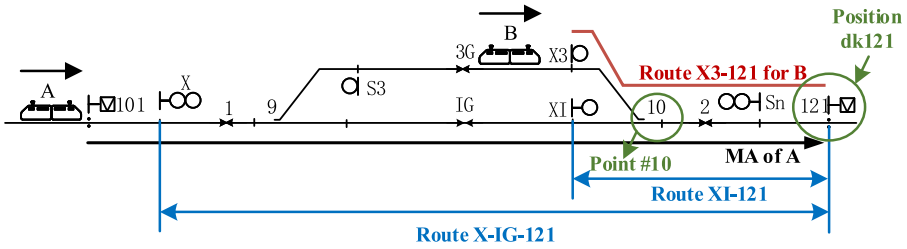

Fig. 8. An accident related to defects hidden in RBC.

Algorithm 3: Monitoring algorithm for PMLSC-SNCs

```
1 for i ← 1 to n do                                   // find triggered PMLSC
2   if P_i is a sufficient prechart then
3     for j ← 1 to |sW(P_i)| do
4       if Check(sW(P_i)[j], φ(P_i)) = true then
5         sW(U_i) ← Con(sW(P_i)[j], sW(M_i)[j]) ;
          /* Construct word by events in sW(P_i)[j]
             and sW(M_i)[j] with their original
             positions in w                          */
6         if Check(sW(U_i), θ(U_i)) ∧
          Check(sW(M_i)[j], φ(M)) = false then
7           return false;
            /* Not satisfy parameter
               constraints or main chart,
               report false                          */
8   else
      /* same manner as the above                    */
9 return true
```

away from the station and train B should leave the station first. Therefore, it temporarily cancelled, via the interlocking, the route *XI-121* and setted up a route *X3-121* for train B. However, since train A did not stop, the RBC decided that the route *X-IG-121* could not be cancelled. It still provides an MA with EoA at position dk121 to train A. Consequently, with the incorrect MA, the CTCS level 3 controlled train A would roll over at Point #10. In this accident, train B was controlled by the CTCS level 2, in which MAs are generated by track circuits. Since the RBC was not aware of the MA information provided by the track circuits, train A would have collided with train B.

This defect was corrected through a software update. Although a high price has been paid, it is difficult to guarantee that a similar accident will not occur again because such accidents are caused by multiple mistakes of different subsystem executions, and are hard to predict and eliminate during the development phases. To solve this problem, we propose to use runtime verification to protect RBC executions. By means of monitors, RBC software is allowed to carry defects, which are impossible to be fully eliminated, and the executions are checked at runtime. When a dangerous execution is exhibited, the monitor sets alarms and prevent the execution from evolving into an accident.

We obtain several properties of RBCs to monitor by translating the Chinese Train Control System Requirement Specifications. The abbreviations of messages, actions and parameters are shown in Table 1.

7.1. The MA extension process

To extend MAs in train stations, the interlocking first locks the route for the train and sends the route information (called signal authorization, SA) to the RBC. With the SA messages, the RBC computes the MA with respect to the engineering data and sends the MA to the train. The end of MA is at the end of the locked route. When the train enters the locked route, the interlocking sends an SA message indicating that the route is used, and the RBC sends a conditional emergency braking message (CEM) to the train. The braking target position of the CEM is at the starting position of the used route. If the front of the train has already passed the braking target position, the train ignores the CEM and keeps operating with its original speed-distance curve. If the route is cancelled, the interlocking will send an SA of route-cancellation and the RBC updates the MA with respect to the locked routes. This process is formulated with PMLSCs U_1, U_2 and U_3 as shown in Fig. 9, where

- $\Sigma_1 = \{SA, MA\} \times \{!, ?\}$
- $\Sigma_3 = \{SA, CEM\} \times \{!, ?\}$
- $\Sigma = \{RouteLocked, enterRoute, CancelRoute\} \times \{!, ?\}$

Table 1
Messages in the RBC/RBC Handover Process.

Abbreviation	Description
SA	Signal authorization
CEM	conditional emergency braking message
R_S (R_E)	start position (end position) of a route
RN	The RBC border
Hov_C	The RBC/RBC handover command
T_ID	Train ID
PA	Pre-announcement: information of an approaching train to the RBC-RBC border
MA	Movement authority
(R)RI	Related Route information
RRI_R	Related Route information request

7.2. The RBC/RBC handover process

In general, a long railway line is divided into several RBC supervision areas, each of which contains one or more interlocking train stations. When a train approaches the border of an RBC supervision area, the so-called "RBC/RBC handover process" occurs to ensure that the train can pass the RBC border without losing speed. The high-level RBC/RBC handover principles are described as follows. When a train approaches the border of two RBC areas, the current RBC (referred to as the handing over RBC, HOVRBC) sends a pre-announcement message to the adjacent RBC (referred to as the accepting RBC, ACCRBC). After receiving this message, the ACCRBC prepares to allow the train to enter its supervision area. At this time, the HOVRBC can only provide an MA with the EoA at the border of the two RBCs. To allow the train to pass the RBC border, the HOVRBC sends an *HOV_C* message to the train, and sends an *RRI* message to the accepting RBC. Upon receiving an RR, the ACCRBC sends route-related information message that includes the available route information in its supervision area. With the RI, the HOVRBC can extend its EoA to the ACCRBC area. PMLSC specifications U_4 and U_5 shown in Fig. 10 specify this process, where

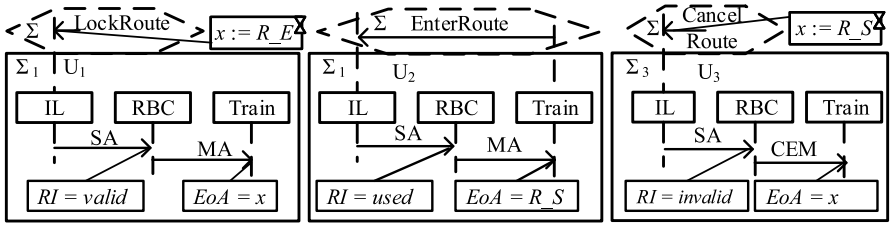

Fig. 9. PMLSCs for MA extensions.

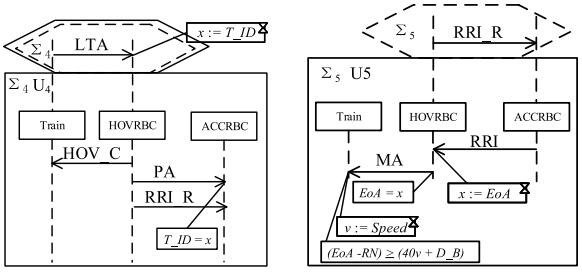

Fig. 10. PMLSCs for the RBC/RBC handover process.

Table 2
Evaluation of PMLSC Specifications.

		PMLSC	BFR/FAR	PeLSC	BFR/FAR
U1	TA	71	26.20%	58	21.40%
	FA	0	0%	0	0%
U2	TA	28	10.33%	15	5.54%
	FA	0	0%	4	5.97%
U3	TA	61	22.51%	43	15.87%
	FA	0	0%	6	8.96%
U4	TA	57	21.03%	57	21.03%
	FA	0	0%	0	0%
U5	TA	48	17.71%	48	17.71
	FA	0	0%	0	0%
ALL	TA	262	96.68%	23	41.70%
	FA	0	0%	113	34.33%

- $\Sigma_4 = \{LTA, HOV\_C, PA, RRI\_R\} \times \{!, ?\}$
- $\Sigma_5 = \{RRI\_R, RRI, MA\} \times \{!, ?\}$

7.3. Experimental results and discussion

Monitors were generated with the above algorithms, where the **Check** function is implemented as in Clavel et al. (2011). Maude provides a rewriting environment for various logics, including HL. We applied our monitors into a CTCS simulation platform with the structure described in Section 2, where the RBC software and engineering data are the same as of the above Chinese main railway line.

To analyse the protection abilities of PMLSC monitors, we do not simply inject the above errors that occurred but also apply a mutation technique to generate various incorrect executions. We apply four mutation operations for system executions: *Change Event Orders (CEO)*, *Delete Events (DEV)*, *Insert Events (IEV)* and *Change Parameter Values (CPV)*.

Let *CW* and *IW* be the numbers of correct and incorrect words being monitored, respectively. Let *TA* and *FA* be a true alarm and false alarm reported by a monitor, respectively. A monitor is evaluated with respect to the false alarm rate (FAR) and bug found rate (BFR), where $FAR \triangleq (FA/CW)$ and $BFR \triangleq (TA/IW)$. We constructed 338 parameterized words from a CTCS-3 simulation system, where 67 are correct executions and 271 are incorrect executions, by applying the above four mutation operators. For comparison, we constructed monitoring specifications with PeLSCs, where the alphabets are not separately defined in charts.[1] Unfortunately, if one directly copy the charts of PMLSCs, the FAR of monitoring with the resulting PeLSC specification becomes to 100%. This is because by means of the intersection of charts, some silent events are automatically transformed into unexpected non-silent events. This changes the meaning of the specifications. For solving this problem, we put a Σ^* at the end of each chart to define its alphabet. This modelling strategy reduces the *FAR* of the PeLSC specifications. The experimental results are shown in Table 2. The rows U_1 to U_5 indicate monitoring with only PMLSCs/PeLSCs U_1 to U_5, respectively. The row "All" shows the monitoring results obtained by defining all the charts to be the monitoring specification.

The experimental results show that monitoring a system with PMLSC specifications can eliminate all false alarms of the PeLSC specifications. These false alarms are caused by the strict definition of silent events. For example, *CancelRoute* (appearing in U_3) is supposed to be a silent event when monitoring property U_2; however, this event cannot be defined as a silent event with PeLSCs. For example, consider the following execution trace

(*EnterRoute*!, *EnterRoute*? , *SA*!, *SA*? , *MA*!, *Cancel Route*? ,
MA? , *SA*!*SA*? , *CEM*!, *CEM*?).

It shows that when a train attempts to enter a route, the route is cancelled and the RBC sends a CEM to the train. This is a correct execution. Due to the incorrect definition of the non-silent event *Cancel Route*, PeLSC monitors generates false alarm. By contrast, by introducing alphabets, the above event is allowed to be both a non-silent event of U_3 and a silent event of U_2. False alarms caused by this problem are therefore eliminated with PMLSCs. In addition, PMLSCs have greater capability to catch bugs than PeLSCs. For example, consider the following illegal execution of U_5:

(*RRI_R*!, *LTA*? , *RRI_R*? , *RRI*? , *RRI*!, *MA*!, *MA*?).

The trace indicates that with an RRI requirement, the HOVRBC sends an old RRI (receive before send) information to the train. The PMLSC monitor can catch this fault and report it as false. However, as the prechart of U_5 does not accept any prefix of the execution, the whole chart U_5 accepts the entire execution. Therefore, the monitors of PeLSCs will report it as *true*. For a single specification such that the alphabet is equal to the events appearing in the charts, PMLSCs and PeLSCs have the same bug finding capability. However, when the alphabet has more events (e.g., U_1 to U_3), PMLSCs have a lower FAR and a higher BFR. A single PMLSC specification has a very low BFR because some features of (incorrect) executions are not described by the specification. Since the above five specifications do not formulate all

[1] In PeLSCs, the alphabet is all the events appearing in the charts: any event not in the alphabet is a silent event.

Table 3
Log of RBC executions.

Event	Count
SA	11,812,579
MA	151,517,320
CancelRoute	125,574
RouteLocked	7,471,581
enterRoute	111,800
LTA	742,281
Hov_C	790,588
PA	775,149
RRI_R	92,628,696
RRI	157,513,118

Table 4
PMLSC monitor performance.

Property	Rewriting Steps (per slice, median)	Runtime (overall, min)	Rewriting Steps (cumulative)
U1	2.39*10^9	–	4.92*10^14
U2	6.41*10^5	2.68	1.32*10^11
U3	8.53*10^5	3.98	1.75*10^10
U4	5.12*10^6	21.95	1.05*10^11
U5	1.92*10^6	4.32	3.95*10^11

the properties of RBC, some injected errors are not found in our experiments. This problem can be solved by introducing more charts of properties into the specifications.

We further investigate the monitoring efficiency of PMLSCs. We analyse approximately 200GB of log data generated from running the CTCS simulation platform for 30 days. Table 3 presents the numbers of the events with respect to the chart alphabets (after mutation). We use rewriting steps to measure the monitor performance, which is shown in Table 4. The log file is divided into approximately 20,580 sub-files. For each property, we use 10 computers for slicing and monitoring. Maude is expected to perform 1×10^6 rewrites per second. However, in practice, additional time for compiling traces grows exponentially with respect to the sizes of traces. Consequently, if the size of the trace under verifying is too large, then Maude will not be able to provide the verification results. For the property U_1, the log file containing the relevant events has a considerably large size (approximate 1.7×10^8 events). In our experiments, affected by the time required to compile such a long trace, Maude failed to give results within a reasonable computational time. For the properties $U2$ and $U3$, the log files contain relatively smaller numbers of relevant events, and the monitor can give results within a few minutes. Since the size of **HL**[↓] formula $\varphi(U_4)$ is bigger than that of $\varphi(U_5)$, the monitor of property U_4 spends more time to give results in comparison to those required by the monitor of the properties U_5. The experimental results show that the algorithms with trace slicing reduce the computational overhead significantly.

8. Conclusion

In this paper, we have proposed PMLSC for monitoring the parameterized properties of CTCS level 3. The language of PMLSC is defined by introducing alphabets of charts. We proved that PMLSC is closed under negation and that the complexity of a subclass of the language is linear with respect to the lengths of words. Moreover, the complexity increases significantly if the condition structures involve quantifiers. For generating monitors, we showed how to translate PMLSC-SNCs into HL[↓] and developed an efficient monitoring algorithm based on trace slicing. Finally, we presented a concrete example of monitoring RBCs of CTCS level 3. The results showed that by defining alphabets of charts, the properties to be monitored can be specified accurately; therefore, monitors achieve better BFR and FAR.

The quality of a monitor depends on the quality of the monitoring specification (Legunsen et al., 2016). We have developed an expressive graphic language for describing the (parameterized) properties of CTCS that is similar to the train control system requirement specifications. However, how to write a complete and accurate specification to monitor railway systems remains an open question. We plan to investigate the methods for (automatically) generating PMLSC properties via specification mining techniques.

CRediT authorship contribution statement

Ming Chai: Conceptualization, Methodology, Investigation, Writing - original draft, Funding acquisition. **Haifeng Wang:** Conceptualization, Methodology, Writing - review & editing. **Tao Tang:** Supervision, Writing - review & editing. **Hongjie Liu:** Software, Investigation, Validation, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., 2017. Monitoring for silent actions. In: 37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2017), pp. 7:1–7:14.

Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., 2018. A framework for parameterized monitorability. In: Baier, C., Dal Lago, U. (Eds.), Foundations of Software Science and Computation Structures. Springer International Publishing, Cham, pp. 203–220.

Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Kjartansson, S.O., 2020. Determinizing monitors for HML with recursion. J. Log. Algebraic Methods Program. 111, 100515. https://doi.org/10.1016/j.jlamp.2019.100515, URL: http://www.sciencedirect.com/science/article/pii/S2352220819301609.

Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K., 2019a. Adventures in monitorability: From branching to linear time and back again. Proc. ACM Program. Lang. 3 (POPL), https://doi.org/10.1145/3290365.

Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K., 2019b. An operational guide to monitorability. In: Ölveczky, P.C., Salaün, G. (Eds.), Software Engineering and Formal Methods. Springer International Publishing, Cham, pp. 433–453.

Ahrendt, W., Chimento, J.M., Pace, G.J., Schneider, G., 2017. Verifying data- and control-oriented properties combining static and runtime verification: Theory and tools. Form. Methods Syst. Des. 51 (1), 200–265.

Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., De Moor, O., Sereni, D., Sittampalam, G., Tibble, J., 2005. Adding trace matching with free variables to aspectj. ACM SIGPLAN Notices 40 (10), 345–364.

Ancona, D., Drossopoulou, S., Mascardi, V., 2012. Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In: International Workshop on Declarative Agent Languages and Technologies. Springer, pp. 76–95.

Ancona, D., Ferrando, A., Mascardi, V., 2016. Comparing trace expressions and linear temporal logic for runtime verification. In: Theory and Practice of Formal Methods. Springer, pp. 47–64.

Andrade, E., Maciel, P., Callou, G., Nogueira, B., 2009. Mapping UML sequence diagram to time petri net for requirement validation of embedded real-time systems with energy constraints, in: ACM Symposium on Applied Computing, pp. 377–381.

Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D., 2012. Quantified event automata: Towards expressive and efficient runtime monitors. In: FM 2012: Formal Methods. Springer, pp. 68–84.

Barringer, H., Rydeheard, D., Havelund, K., 2010. Rule systems for run-time monitoring: from eagle to RuleR. J. Log. Comput. 20 (3), 675–706.

Bartocci, E., Falcone, Y., Francalanza, A., Reger, G., 2018. Introduction to runtime verification. In: Lectures on Runtime Verification - Introductory and Advanced Topics. pp. 1–33.

Basin, D., Dardinier, T., Heimes, L., Krstić, S., Raszyk, M., Schneider, J., Traytel, D., 2020. A formally verified, optimized monitor for metric first-order dynamic logic. In: International Joint Conference on Automated Reasoning. Springer, pp. 432–453.

Basin, D., Klaedtke, F., 2015. Monitoring metric first-order temporal properties. J. ACM 62 (2), 1–45.

Basin, D., Klaedtke, F., Zălinescu, E., 2012. Algorithms for monitoring real-time properties. In: Runtime Verification. Springer, pp. 260–275.

Basin, D., Krstić, S., Traytel, D., 2017. Almost event-rate independent monitoring of metric dynamic logic. In: Lahiri, S., Reger, G. (Eds.), Runtime Verification: 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings. Springer International Publishing, Cham, pp. 85–102. https://doi.org/10.1007/978-3-319-67531-2_6.

Bauer, A., Küster, J.-C., Vegliach, G., 2015. The ins and outs of first-order runtime verification. Form. Methods Syst. Des. 46 (3), 286–316.

Bauer, A., Leucker, M., Schallhart, C., 2010. Comparing LTL semantics for runtime verification. J. Log. Comput. 20 (3), 651–674.

Bauer, A., Leucker, M., Schallhart, C., 2011. Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. (TOSEM) 20 (4), 14.

Bonakdarpour, B., Finkbeiner, B., 2018. The complexity of monitoring hyperproperties. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 162–174. https://doi.org/10.1109/CSF.2018.00019.

Bonakdarpour, B., Finkbeiner, B., 2020. Controller synthesis for hyperproperties. In: 2020 IEEE 33rd Computer Security Foundations Symposium (CSF). pp. 366–379. https://doi.org/10.1109/CSF49147.2020.00033.

Bontemps, Y., Schobbens, P.-Y., 2007. The computational complexity of scenario-based agent verification and design. J. Appl. Log. 5 (2), 252–276.

Chai, M., Schlingloff, H., 2013. A rewriting based monitoring algorithm for TPTL. In: CS&P 2013. Citeseer, pp. 61–72.

Chai, M., Schlingloff, B.-H., 2017. Monitoring with parametrized extended life sequence charts. Fund. Inform. 153 (3), 173–198.

Chen, F., Roşu, G., 2005. Java-MOP: A monitoring oriented programming environment for java. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 546–550.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Talcott, C., 2011. Maude manual (version 2.6). Univ. Illinois, Urbana-Champaign 1 (3), 4–6.

Colombo, C., Pace, G.J., Schneider, G., 2009. Dynamic event-based runtime monitoring of real-time and contextual properties. In: Cofer, D., Fantechi, A. (Eds.), Formal Methods for Industrial Critical Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 135–149.

Colombo, C., Pace, G.J., Schneider, G., 2009. LARVA — SAfer monitoring of real-time Java programs (tool paper). In: 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods. pp. 33–37. https://doi.org/10.1109/SEFM.2009.13.

Damm, W., Harel, D., 2001. LSCs: Breathing life into message sequence charts. Form. Methods Syst. Des. 19 (1), 45–80.

de Boer, F.S., de Gouw, S., 2014. Combining monitoring with run-time assertion checking. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (Eds.), Formal Methods for Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures. Springer International Publishing, Cham, pp. 217–262. https://doi.org/10.1007/978-3-319-07317-0_6.

Delgado, N., Gates, A.Q., Roach, S., 2004. A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. 30 (12), 859–872. https://doi.org/10.1109/TSE.2004.91.

Diekert, V., Muscholl, A., Walukiewicz, I., 2015. A note on monitors and Büchi automata. In: Leucker, M., Rueda, C., Valencia, F.D. (Eds.), Theoretical Aspects of Computing - ICTAC 2015. Springer International Publishing, Cham, pp. 39–57.

Falcone, Y., Havelund, K., Reger, G., 2013. A tutorial on runtime verification. In: Engineering Dependable Software Systems. pp. 141–175. https://doi.org/10.3233/978-1-61499-207-3-141.

Francalanza, A., Aceto, L., Achilleos, A., Attard, D.P., Cassar, I., Della Monica, D., Ingólfsdóttir, A., 2017. A foundation for runtime monitoring. In: Lahiri, S., Reger, G. (Eds.), Runtime Verification. Springer International Publishing, Cham, pp. 8–29.

Francalanza, A., Seychell, A., 2015. Synthesising correct concurrent runtime monitors. Form. Methods Syst. Des. 46 (3), 226–261.

Franceschet, M., de Rijke, M., Schlingloff, B.-H., 2003. 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, pp. 166–173.

Harel, D., Kugler, H., Marelly, R., Pnueli, A., 2002. Smart play-out of behavioral requirements. In: Formal Methods in Computer-Aided Design. Springer, pp. 378–398.

Harel, D., Maoz, S., 2008. Assert and negate revisited: Modal semantics for UML sequence diagrams. Softw. Syst. Model. 7 (2), 237–252.

Harel, D., Maoz, S., Segall, I., 2008. Some results on the expressive power and complexity of LSCs. In: Pillars of Computer Science. Springer, pp. 351–366.

Harel, D., Marelly, R., 2002. 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, pp. 193–202.

Havelund, K., 2014a. Monitoring with data automata. In: Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications. Springer, pp. 254–273.

Havelund, K., 2014b. Rule-based runtime verification revisited. Int. J. Softw. Tools Technol. Transf. 17 (2), 143–170.

Havelund, K., Goldberg, A., 2008. Verify your runs. In: Meyer, B., Woodcock, J. (Eds.), Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 374–383. https://doi.org/10.1007/978-3-540-69149-5_40.

Havelund, K., Reger, G., Thoma, D., Zalinescu, E., 2018. Monitoring events that carry data. In: Lectures on Runtime Verification - Introductory and Advanced Topics. pp. 61–102. https://doi.org/10.1007/978-3-319-75632-5_3.

Kari, J., 2007. The tiling problem revisited. In: International Conference on Machines, Computations, and Universality. Springer, pp. 72–79.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., 2001. An overview of aspectj. In: ECOOP 2001—Object-Oriented Programming. Springer, pp. 327–354.

Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y., 2005. Temporal logic for scenario-based specifications. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 445–460.

Kumar, R., Mercer, E.G., 2009. Verifying communication protocols using live sequence chart specifications. Electron. Notes Theor. Comput. Sci. 250 (2), 33–48.

Legunsen, O., Hassan, W.U., Xu, X., Roşu, G., Marinov, D., 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In: Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on. IEEE, pp. 602–613.

Maoz, S., Harel, D., 2011. On tracing reactive systems. Softw. Syst. Model. 10 (4), 447–468.

Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G., 2012. An overview of the MOP runtime verification framework. Int. J. Softw. Tools Technol. Transf. 14 (3), 249–289.

Roşu, G., Havelund, K., 2005. Rewriting-based techniques for runtime verification. Autom. Softw. Eng. 12 (2), 151–197.

Sokolsky, O., Havelund, K., Lee, I., 2012. Introduction to the Special Section on Runtime Verification. 14 (3), 243–247. https://doi.org/10.1007/s10009-011-0218-6.

Sun, J., Dong, J.S., 2005. Model checking live sequence charts. In: IEEE International Conference on Engineering of Complex Computer Systems, pp. 529–538.

Yang, N., Yu, H., Sun, H., Qian, Z., 2012. Modeling UML sequence diagrams using extended Petri nets. Telecommun. Syst. 51 (2–3), 147–158.