ulm university universität
uulm

Rule-based Monitoring Framework for Business Process Compliance

**Ping Gong, David Knuplesch,
and Manfred Reichert**

Ulmer Informatik-Berichte

Nr. 2016-03
März 2016

Rule-based Monitoring Framework for Business Process Compliance

Ping Gong[1]*, David Knuplesch[2], and Manfred Reichert[2]

[1] Department of Computer Science
Fujian Normal University
Fuzhou 350007, P. R. China
tnfair@126.com
[2] Institute of Databases and Information Systems,
Ulm University, Germany
{david.knuplesch,manfred.reichert}@uni-ulm.de

Abstract. Business processes compliance monitoring can be viewed as the task of detecting and reacting to the compliance of running business processes with compliance rules, which are the semantic constraints originated from norms, standards, and laws, etc. Normally, compliance rules not only refer to normal process perspectives, like control flow, data flow, and time, but also perspectives of data aggregation as well as their mixtures. Such characteristics as well as potentially high number of concurrently running process instances, post challenges for processes compliance monitoring from the aspects of specification and monitoring efficiency. In this work, we address these challenges by proposing a compliance monitoring framework (*bpCMon*), which includes an event-based compliance language (ECL) and event reaction system (ERS). More specifically, ECL is a formal language enabling specifying compliance rules of multi-perspective. ERS is a powerful rule-based system enriched with events indexing structure, and fully supports the monitoring for compliance rules in ECL. Experiments on a real life datasets indicate the applicability of bpCMon; and the comparisons with three related works over benchmarks demonstrate the efficiency of bpCMon.

Keywords: Business process compliance monitoring, runtime verification

1 Introduction

Business process compliance (BPC) essentially means that business processes are executed in conformance with prescribed and agreed sets of compliance rules [1]. BPC can be ensured and verified at different phases of process life cycle, e.g., a priori at design time or a posteriori based on logs of completed processes. However, in realistic setting, the deviation of actual running from the process definition and the potential implicitness of process definition, highlight the necessary of *compliance monitoring* for BPC.

Compliance monitoring is the task of detecting and reacting to the compliance violations of running business processes based on the monitoring mechanism, which is generated from prescribed compliance rule. Lion's share of compliance monitoring research has been focusing on compliance rule language and monitoring mechanism. As

* The work was finished during the author visited Ulm univeristy.

Running Examples

Fraud Prevention: Following compliance rules address the prevention of frauds in the banking domain and source from [10, 14]:
B1. Every executed transferring transaction of customer, who has within the last 30 days been involved in a suspicious transaction (transferring with amount greater than 10,000€), must be reported suspicious within 2 days.
B2. The sum of withdraws of each user over the last 30 days does not exceed the limit of 10,000€.
B3. For each user, the number of withdrawing peaks over the last 30 days does not exceed a threshold of 5, where a peak is a value at least twice the average over some time window(30 days).

BPIC 2011: The Business Process Intelligence 2011 Contest (BPIC 2011) logs[3] are real life datasets stemmed from a Dutch hospital and are provided in XES format [22]. In [20], 16 rules referring to various process perspectives were mined from these logs by using Declare Miner tool. Out of these 16 rules, following R9 and R11 are listed as examples:
R9. If "administratief tarief -eerste pol" occurs in a trace, it is always preceded by "vervolgconsult poliklinisch" and it occurs at most 1030 days before "administratief tarief -eerste pol".
R11. If "natrium vlamfortometrisch" occurs in a trace and the condition "(Age$\geq$71 && Treatment code $\geq$ 803&& Diagnosis Treatment Combination ID$\leq$ 394,725) $\|$ (Treatment code==703 $\|$ Treatment code ==803))" holds, then "natrium valmfotometrisch" is not followed eventually by "calcium".

opposed to *a priori* compliance checking, compliance monitoring does not require to explore the whole state space of process model, and also enable handling the running data of process in real-time manner. These characteristics make compliance monitoring to be promising techniques which enable providing business practitioners with meaningful and timely insights into their running processes.

Usually, the compliance requirements are sourced from norms, guidelines, and standards. Additinal effort is needed to refine them operable through relevance methods, e.g., semantic parameterization [3]. In this work, the term of compliance rule refers to the constraints which are yielded after compliance requirement refinement. Normally, compliance rules refer to different process *perspectives*, including activity, control flow, data flow, time, and resource as well as data aggregation. These perspectives are not simply co-existed, but related each orther to associate with single activity or correlated among activities. This characteristic poses further requirements for compliance monitoring:

More dedicated language is needed. Existing compliance languages [4–9] are designed base on the notions of *activity* and *control flow*. However, regarding to above mentioned characteristic, more dedicated constructs of compliance language are needed to specify the structure of activity as well as the correlations among activities. Taking *R11* as an example, besides the control flow between two involved activities, the first activity also includes complicated data constraints, which implies the need for proper

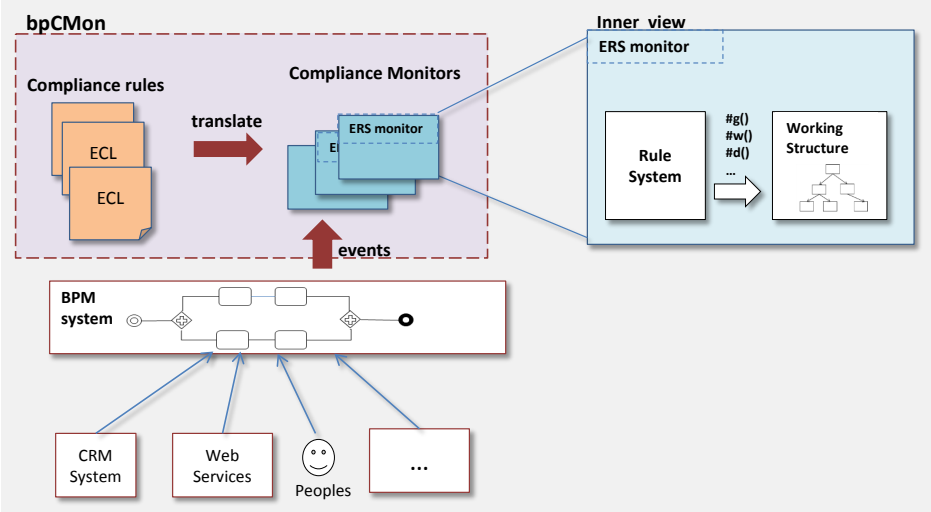

Fig. 1. The applying environment for bpCMon and the inner strucutre of ERS monitor

construct to describe desired activity. Also considering *B1*, in addition to the apparent control flow and time constraints among *transferring* and *report*, there also exist two data correlations among these activities, i.e., one *transferring* is correlated by the same user to another suspicious *transferring* happened before, and also such *transferring* is correlated to *report* by given attribute, e.g., *transactionID*. Note that, without specifying these correlations, the complance rules would not be correctly specified, which would further result in false reports during compliance monitoring. Furthermore, *B2* and *B3*, referring to the aggregations of *withdraw* amount over time periods, e.g., sum, max, and average, highlight the need for additional constructs to support data aggregation.

Powerful and efficient monitoring mechanism is required, to cope with complex compliance rules and potentially high volume of running process instances. Existing monitoring techniques, e.g., temporal logic based [12–14], event calculus based [15], state-machine/marking based [16, 8, 9], rule sytem based [21], etc, have their own pros and cons. Considering the potential power and extensibility of underlying formalisms, as stated in [21], rule system based techniques have the potential of by the uniform formalism enabling to support compliance rules of full perspectives, especially data aggregation. However, as implied in the experiments [21], current rule-based systems, e.g., LOGFIRE [21], Drools [18], etc, are not efficient enough since lack of effective indexing structure as in MOP, which is known as fastest monitor for parametric properties, but has limited support for data-aware properties [39]. Thereby, to make rule system based technique really applicable in the compliance monitoring, besides the powerful reaction rules, effective indexing structure is needed to support efficiently manuplating running data during compliance monitoring.

To address these requirements, in this work we propose a business processes compliance monitoring framework (bpCMon), as Fig. 1, which includes *event-based compliance language* (ECL) and *event reaction system* for compliance monitoring (ERS).

The ECL is a formal compliance language, which is designed based on the central notions of *event* and *event-relation pattern*, and enables specifying data constraints, correlations among events, and aggregation. The *event* in ECL is an abstraction over a set of interesting event instances, and its defintion includes sets of constraints. For example, the *withdraw* event, defined as $(1,'\ withdraw', [amount > 10,000])$, describes all *withdraw* instances with the amount greater than 10,000 (€), where 1 is the unique identifier of *withdraw* event. Note that, the difference exists between event and event instance in this work. The *event-relation pattern*, also the atomic formula in ECL, is originated from typical temporal orders as in classical temporal logic (e.g., LTL), but includes and highlines in pattern form the correlations among events. As for the correlation, it is specified by the involved events and their correlating condition. Considering *before* pattern, $\texttt{before}([0, 10d), ta, tr, econ)$, given two events ta and tr, it means,"when tr instance happens, then correlated ta instance must have happened *before* within 10 days", where *econ* is the *correlating condition* for ta and tr. Note that, the involved events, tr and ta, play distinct roles within the correlation, i.e., the trigger for the relation and the target needed to happen. Currently, ECL supports 6 event-relation patterns: *before*, *after*, *when*, *beforeSince*, and *afterUntil* and *aggregate...with*, where *aggregate...with* is introduced to enable ECL to specify data aggregation. Based on these event-relation patterns, by using logic connectives, ECL is capable of specifying complex compliance rules.

To efficiently monitor compliance rules in ECL, we introduce event reaction system (ERS). Essentially, ERS is a rule system attached with a *working structure*, which is the working memory of ERS and characterized by tree-like indexing structure. At first glance, ERS seems to be similar to RETE algorithm [17] based rule systems like Drools [18] and Jess [19]. However, ERS is a light-weight rule system, which differs from these systems not only in the aspect of rule form, but also in the structure of working memory. Within the ERS working memory, there is one essential component, instances indexed structure (IIS), which stores event instance as fact, and also makes use of tree-like indexing to speed up the assessing of desired instances. The reaction rule of ERS is defined based on dedicated operations, which operate upon the working structure. By these operations, ERS is able to provide meaningful feedback for violated compliance rules. Moreover, by introducing B-tree variant, the data aggregating operators is implemented effciently in ERS.

To investigate the applicability of bpCMon, we evaluted bpCMon in several ways. First, we discussed the fulfillment of bpCMon to meet the compliance monitoring functionalities (CMF) proposed in [20]. Second we showed that ECL is able to cover most compliance patterns [20][6] in order to evaluate the expressive of ECL. Furthermore, a real life logs from Dutch academic hospital, which stems from the Business Processes Intelligence Contest 2011 and consisted of 1143 traces and 150291 events, was applied as case study for the bpCMon. The logs is run by bpCMon monitor to analysis its compliance with 16 rules, which are mined from the logs. After seconds run, bpCMon monitor discovered 10 rules out of 16 are incompliant, and also reported 4937 violations in total and their root causes as well.

To evaluate the efficiency of bpCMon, the benchmark from [21] is adopted to compare with three related works: MOP, known as the fastest monitor for parametric property in runtime verification community, Drools, the state-of-art rule-based inference engine, and the MonPoly which is a powerful facility supporting monitoring the metric linear temporal logic(MLTL) as well as its aggregation extension. The second

test case is generated based on [10] and used to compare bpCMon to the MonPoly. The comparing data demonstrate that, thanks to the indexing structure and statistic tree, the bpCMon monitor is efficient no matter for compliance rules with or without data aggregation.

Regarding to the contributions made in this work, they consist of three aspects:

- ECL: an event based compliance language and has rich expressive ability.
- ERS: a rules system with events indexing working structure for efficient monitoring.
- bpCMon: a business process compliance monitoring framework which is implemented in Java and also evaluated through real life logs and others test cases.

The remainder of this paper is structured as follows: section 2 is devoted to define the ECL as well as its extension to the data aggregation; section 3 is for definition of ERS, including rule system and working structure; section 4 is the translating from ECL to ERS and also includes translation soundness assurance theorem; The implementation of bpCMon is included in section 5; and the evaluation of bpCMon is presented in section 6 which includes the language and monitor evaluations; And then, section 7 is the comparing related works concerning with the facts which are not included in [20][21], and also includes the discussion concerning with the specific relations between the bpCMon and existed works; the last section is the conclusion and future work.

2 Event-pattern based Compliance Language(ECL)

As mentioned above, the ECL is designed based on the notions of *event-pattern* and *events relation pattern*. *event-pattern* is an abstraction over a set of interested event instances, which correspond to the events in BPM system or messages in SOA-enabled processes system. *events relation pattern* describes frequently occurred relation among events. In this section, event pattern and the ECL will be defined firstly and then the ECL will be extended to include aggregation.

2.1 Event pattern

Before the formal definition, some notations are needed:
Let `VAR` be the variables set; `VALUE` be the values domain, including all the basic data type, and `ENAME` $\subseteq$ `VALUE` is the event names set.

Definition 1. *Event pattern e is 4-ary tuple* $(id, ename, attrs_{cstr}, ts)$*, where id is the unique identifier for e, ename* $\in$ `ENAME` *is the name of e, ts is the timestamp of e, and* $attrs_{cstr}$ *is attributes constraints defined as :*

$$
\begin{aligned}
attrs_{cstr} ::= &\ [\ items\ ] \\
items ::= &\ item \mid items\ ,\ item \\
item ::= &\ attr \mid constr \\
constr ::= &\ attr \sim c \mid attr_1 \sim attr_2 \mid !\ constr_1 \mid constr_1 \&\& constr_2,
\end{aligned}
$$

where $attr \in$ `VAR`*,* $c \in$ `VALUE`*, and* $\sim \in \{ =, \neq, <, \leq, >, \geq \}$

From the definition, the event pattern is a structure which consists of three compulsory attributes, *id*, *ename*, and *ts*, and a set of attributes patterns which are used for events matching. Normally, *ts* attribute is just a placeholder requiring the time stamp needed for each matched instance. Usually, it is ignored in the event patterns definitions if there is not time constraint for the event pattern. Considering the event pattern of money withdrawing with the amount greater than 1000, $e = (1,' withdraw', [amount > 1000])$, where event pattern e contains, 1 as *id*, 'withdraw' as its name, and attribute *amount* with related constraint. Note that, for the attribute values or constants, they will be enclosed by ' ' if their values are string. During the business processes executing, there are two special types of events, *start* and *end*, which represent respectively the starting and end for the process instance execution.

For *item*, $attrs_{cstr}$, and e as defined above, let $\texttt{attr}()$ be the attributes variables getting function and defined as:

- $\texttt{attr}(item) = \{attr\}$, if *attr* is the attribute variable of *item*;
- $\texttt{attr}(attrs_{cstr}) = \bigcup_{item \in attrs_{cstr}} \texttt{attr}(item)$.
- $\texttt{attr}(e) = \{id, ename, ts\} \cup \texttt{attr}(attrs_{cstr}\, T) \subseteq \texttt{VAR}$ is the attributes set for e ;

We use $e.attr$ to denote the attribute *attr* of e, and notation EVENT to denote all the events patterns, and also ATTR to denote all the attributes occurred in the events in EVENT.

The *event instance* is occurred instaneously and carrying relevant information. Hence, the instance *inst* is defined as a partial mapping $\texttt{ATTR} \rightharpoonup \texttt{VALUE}$, with $\{ename, ts\} \subseteq \texttt{attr}(inst)$, where $\texttt{attr}(inst) = \{\ attr \mid inst(attr) \neq \perp, for\ attr \in \texttt{ATTR}\ \}$. Let INS denote event instances set, then the trace τ is a finite sequence in $\texttt{SEQ}(\texttt{INS})$[4]. Then we have *event matching* definition.

Definition 2. *For event $e \in$ EVENT and instance $inst \in$ INS, e is matched by inst, denoted as $inst \models_{em} e$, iff the followings are hold:*

- $\texttt{attr}(e) \subseteq \texttt{attr}(inst)$;
- $e.ename{=}inst(ename)$;
- $inst \models_c constr$ [5], *for each constr in e.*

In the definition, the first condition assures that the instance should have enough information and the others requires the instance should satisfy all the constraints in the event.

For given even pattern e, the *other* event related to e, denoted as $\texttt{ors}(e)$, is also an event and e is the *base* of $\texttt{ors}(e)$. In syntax, it defined as: $\texttt{ors}(e) \in \texttt{EVENT}$ satisfying that, $\texttt{ors}(e).id \neq e.id$ and $\texttt{attr}(\texttt{ors}(e)) = \texttt{attr}(e)$. Semantically $inst \models_{em} \texttt{ors}(e)$ iff $inst \not\models_{em} e$. The *other* event of e is used to refer to all instances unmatched to e and is necessary in dealing with the case of " something should no happened".

2.2 Definition of ECL

The event-pattern based compliance language(ECL) is a kind of pattern based logic language. Syntactically, the ECL has a more abstract signature than classical logic's ,

[4] SEQ represents the mapping to get the set of all the finite sequences from related set.
[5] The notation $\models_c$ means the semantical explanation as in classical propositional logic.

e.g., liner temporal logic, which is built on the signature including variables, functions, and predicates. However, the signature of ECL includes event patterns and events relation patterns, which are of higher level. For the relation patterns, the ECL so far includes five types relation patterns. Their atomic forms as well as intuitive semantics are listed as follows:

- *before*-type: when event e is occurred, then event e' with condition *econ* must be happened *before* e with time constraint tc, denoted as $\texttt{before}(tc, e', e, econ)$;
- *after*-type: when event e is occurred, then event e' with condition *econ* must be happened *after* e with time constraint tc, denoted as $\texttt{after}(tc, e, e', econ)$;
- *when*-type: when event e is occurred, then data constraint *constr* must be satisfied, denoted as *constr* `when` e;
- *beforeSince*-type: when event e is occurred, then event e_2 must be happened *before* e with time constraint tc, and *since* e_2 *before* e , event e_1 must always be happened, and meanwhile condition *econ* must be satisfied, denoted as $\texttt{beforeSince}(tc, e_2, e_1, e, econ)$;
- *afterUntil*-type: when event e is occurred, then event e_2 must be happened *after* e with time constraint tc, and event e_1 must always be happened *after* e *until* e_2, and meanwhile condition *econ* must be satisfied, denoted as $\texttt{afterUntil}(tc, e_2, e_1, e, econ)$;

Note that, within each relation pattern, it may include, the *trigger*, for activating formula, the *target*, as a fact needed to be assessed by the trigger, the correlation *econ* for target and trigger, and time interval as time constraint.

Normally, the time-point semantics for instances/messages is adopted, which means that these instances are ordered based on linar time point with some time scale, e.g., day, second, or millisecond, etc, but it is possible for more than one instances occurred for the same time point. In this work, we assume that all the instances are fully ordered by some order, e.g., the arriving order of messages in the inBound queue of ESB, which means for each point there is only one instance occurred. To this end, in the definition of ECL as Listing 1.1, single event is defined as the trigger for event matching formula(EMF) *emf*, and the target part is recursively extended with EMF-formula.

Listing 1.1. The definition of ECL

$$
\begin{aligned}
ecl &::= [\ event\ ]^+\ \ [\ tmf\ ]^+ \\
tmf &::= \texttt{always}\ emf \mid \texttt{exists}\ emf \\
&\mid !\ tmf \mid tmf_1\ \&\&\ tmf_2 \\
emf &::= f \mid !\ emf \mid emf_1\ \&\ emf_2 \\
f &::= e \mid \texttt{ors}(e) \mid constr\ \texttt{when}\ f \\
&\mid \texttt{before}(tc, f, e, econ) \\
&\mid \texttt{after}(tc, e, f, econ) \\
&\mid \texttt{beforeSince}(tc, f_2, f_1, e, econ) \\
&\mid \texttt{afterUntil}(tc, e, f_1, f_2, econ) \\
econ &::= e_1.attr_1 = e_2.attr_2 \mid econ_1\ \&\&\ econ_2 \\
tc &::= [\ t,\ right\ ) \\
right &::= t\ [\ \texttt{d} \mid \texttt{h} \mid \texttt{m} \mid \texttt{s}\ ]
\end{aligned}
$$

From the synax definition, the structure of ECL is consisted of two parts, events part and rules part. Events part is for event patterns definitions which form the events alphabet for ECL formula, whereas within the rules part, rules are specified in *trace matching formula tmf*, which concerns with the properties of traces; TMF-formula *tmf* is defined by extending *event matching formula emf* from event point scope to the trace scope by `always` or `exists` qualifiers as well as negation and conjunction operators. Semantically, TMF-formula *tmf* corresponds to a set of traces which satisfy *tmf*; EMF-formula *emf* is built on a set of atomic formulas, which correspond to the event relations patterns as listed above, but with the structurally recursive extending which enable ECL to describe more complicate rules, e.g., *chain* pattern [6] as listed in the evaluation section. Semantically, EMF-formula *emf* is a set of event points/instances which satisfy *emf* within a given trace.

Note that, in ECL definition, two negations ! are placed both in TMF and EMF for the clearity, although one of them can be deleted since the axiom of ! `always`(*emf*) = `exists` (!*emf*). Also, for the conjunction operators for EMF-formula and TMF-formula, $'\&'$ plays the role of connecting two *emf* operands with the same trigger point and hence requires two operands should have the same trigger, whereas, $'\&\&'$ has not such restriction. Furthermore, as classic formal logic, the disjunction"|" and implication"$\rightarrow$" can be introduced for EMF formula based on negation and conjunction operators as usual way; so does the "||" and "=>" for TMF-formula.

Correlating condition *econ* is used to correlate target and trigger instances. In this work, *econ* is defined as equal-based conjunction and also it can be extended if needed. For the time constraint, similar to [14], the time interval $[t, right)$ form is adopted where the left t is a integer and *right* is another integer followed by a time measurement from day to millisecond, for instance, $[2, 31d)$. Note that, usually, the time interval is used to represent a time period before given time point and when representing some time period after given time point, then the contrary of time interval is needed. Let $tc = [t_1, t_2\circ)$, where $\circ$ represent the time measurement, then the *contrary* of tc is $\overline{tc}$ defined as $[t'_2, t'_1\circ)$, where $t'_2 = -(t_2 - 1)$ and $t'_1 = -(t_1 - 1)$.

Example 1. From [14]: every executed transaction of a customer c, who has within the last 30 days been involved in a suspicious transaction(with amount greater than 10000), must be reported suspicious within 2 days.

```
//events part
e1 = (1, 'transfer', [ customer, amount, tId ]) ;
e2 = (2, 'transfer', [ customer, amount > 10000 ]) ;
e3 = (3, 'report', [customer, tId]) ;
// policy part
rule1 = always( before([0,31d),e2,e1,e1.customer=e2.customer)
         -> after([0,3d),e1,e3,
                  e1.customer=e3.customer && e1.tId=e3.tId ) )
```

Within the ECL formula, events in different positions play different roles, e.g., *triggering*, *deciding*, or both, for the compliant monitoring. Here, the *triggering* is played by the *trigger* event in the formula as mentioned above; whereas, *deciding* event represents, when such event instance arrived, it is the time to make the decision regarding to the satisfiability of its activated formula. For example, within the formula of rule1 in the above example, for its sub-formula, `before`$([0, 31d), e_2, e_1, \cdots)$, e_1 is its *triggering* event as well as *deciding* event, and for another sub-formula, `after`$([0, 3d), e_1, e_3, \cdots)$, e_1 is its trigger and e_3 is its *deciding* event.

For a given EMF-formula *emf*, it is *overlapped* if it is a *before*-type or *beforeSince*-type formula and meanwhile one of its sub-formula is of *after*-type or *afterUntil*-type. The *overlapped* is the relation between triggering and deciding events of given formula and its sub-formula. For instance, the *overlapped* formula *emf*:

$$\texttt{before}(\_, \texttt{after}(\_, e_1, e_2, \_), e, \_),$$

event e is its trigger and also deciding event which requires desired e_1 instance must occur before, but for its sub-formula, $\texttt{after}(\_, e_1, e_2, \_)$, e_1 is the trigger and e_2 is the deciding event which decides whether desired e_1 instance is occurred. The point, which *overlapped* targets at, is that e_2 instance could be occurred after e instance for all valid traces of this formula. Then during monitoring, when event e instance was occurred, it would be impossible for the monitor to make decision about whether desired e_1 instance was occurred before or not, if meanwhile e_2 instance was not occurred yet but might occurred after e instance. For the *overlapped* formula, the monitor would have to delay to make its decision when deciding instance occurred. However, for the `after`-type formula, even if there is a `before`-type sub-formula inside, it still belongs to *non-overlapped*, since the deciding instances of its sub-formula are not allowed to occur after its deciding instances.

Definition 3. *For event matching formula f, the triggering events set, deciding events set of f , denoted as* $\texttt{tr}(f)$ *and* $\texttt{de}(f)$*, are defined recursively based on formula type as follows:*

- $\texttt{tr}(f) = \texttt{de}(f) = \{e\}$*, if* $f = e$*, or* $f =$ *constr* `when` e*.*
- $\texttt{tr}(f) = \texttt{de}(f) = \{\texttt{ors}(e)\}$*, if* $f = \texttt{ors}(e)$*.*
- $\texttt{tr}(f) = \texttt{tr}(f')$*, and* $\texttt{de}(f) = \texttt{de}(f')$*, if* $f =$ *constr* `when` f'*.*
- $\texttt{tr}(f) = \texttt{de}(f) = \{e\}$*, if* $f = \texttt{before}(tc, f_1, e, econ)$ *or* $\texttt{beforeSince}(tc, f_1, f_2, e, econ)$*, and f is non-overlapped.*
- $\texttt{tr}(f) = \{e\}$ *and* $\texttt{de}(f) = \texttt{de}(f_1) \cup \{e\}$*, if* $f = \texttt{before}(tc, f_1, e, econ)$*, or* $\texttt{beforeSince}(tc, f_1, f_2, e, econ)$ *and f is overlapped.*
- $\texttt{tr}(f) = \{e\}$*, and* $\texttt{de}(f) = \texttt{de}(f_2)$*, if* $f = \texttt{after}(tc, e, f_2, econ)$ *or* $\texttt{afterUntil}(tc, e, f_1, f_2, econ)$*.*
- $\texttt{tr}(f) = \texttt{tr}(f_1)$ *and* $\texttt{de}(f) = \texttt{de}(f_1)$*, if* $f = !\, f_1$ *and* $f_1 \neq e$ *and* $f_1 \neq \texttt{ors}(e)$*.*
- $\texttt{tr}(f) = \texttt{tr}(f_1) \cup \texttt{tr}(f_2)$ *and* $\texttt{de}(f) = \texttt{de}(f_1) \cup \texttt{de}(f_2)$*, if* $f = f_1 \,\&\, f_2$*.*

Within EMF-formula, there are two special atomic formulas, i.e.,e and $\texttt{ors}(e)$, since their unsatisfiable are not triggered by themselves but others, for instance, for e, the trigger for its unsat is not e instance but other instance. Then for f, the set of *triggering events* for its unsatisfiable is, denoted as $\texttt{tr\_un}(f)$, defined as

$$\texttt{tr\_un}(f) = \begin{cases} \texttt{ors}(\texttt{tr}(f)), & \text{if } f{=}e \text{ or } \texttt{ors}(e); \\ \texttt{tr}(f), & \text{o.w.} \end{cases}$$

Based on $\texttt{tr}()$ and $\texttt{de}()$, a **restriction** for EMF formula f is made as:

$$\textit{if } f = f_1 \textit{ op } f_2, \textit{ then } \texttt{tr}(f_1) = \texttt{tr}(f_2),$$

where $op \in \{\ |,\ \&,\ \rightarrow\ \}$. Note that, the intuition behind the restriction is that, two EMFs should have the same triggering event as connecting points before they was connected each other by relevant operators.

An EMF formula is *well-formed*, if it satisfies the restriction. Note that, for each well formed EMF formula f, there is only one *triggering event* for such formula, i.e., $|\texttt{tr}(f)| = 1$. In the following, only well-formed EMF formula is considered.

2.3 Extending ECL for aggregation

For the compliance rules with data aggregation, e.g., "the sum of withdraws of each user over the last 30 days does not exceed the limit of €10,000", the data aggregation includes these elements:

- *aggregation operator* is the statistic function over related data, e.g., *sum* in the example;
- *target data/events* is the data set over which the aggregation operator is applied, e.g., withdrawing events of last 30 days in the example;
- *grouping attributes* are the attributes of involved data/events by which the grouping data are gotten from the data set, e.g., the *user* in the example;
- *aggregating attributes* are also the attributes of related data set and the aggregation value for each group is gotten by applying operator over the selected data based on *aggregating attributes* from the grouping data, e.g., the *amount* attribute of withdraw in the example;
- *aggregating constraint* is the condition which should be satisfied by each aggregation value for each group, e,g., does not exceed €10,000.

Then, the example can be specified with `aggregate with` pattern as follow:

$$\texttt{aggregate}(\ \texttt{sum}(\ [0, 31d), e,\ e.user,\ e.amount, \_)\ \texttt{to}\ s)\ \texttt{with}\ s <= 10,000.$$

where event e could be $('withdraw', [user, amount])$ and it is the trigger for the formula. Structurally, from the inside, the aggregation value of *aggregating expression*, $\texttt{sum}(\ [0, 31d), e,\ e.user,\ e.amount, \_)$, is assigned to the *aggregating variable* s, and then the aggregating expression is encapsulated by `aggregate` and `with`, where `aggregate` can include more than one aggregate expressions and `with` is followed by the constraints for the inside aggregating expressions.

Intuitively, the aggregating formula, `aggregate` *aggexp* `with` *aggcon*, can be understood as " when the trigger of *aggexp* occurred, the aggregating values of *aggexp* within the aggregate must satisfy the condition of *aggcon*". Listing 1.2 is the definition for ECL aggregating formula, which belongs to the sub-formula of EMF.

Listing 1.2. Aggregating formula for ECL

$$\begin{aligned}
f &::= \texttt{aggregate}(\ aggexp\ )\ \texttt{with}\ aggcon \\
aggexp &::= aggop(tc, emf, g_attrs, a_attrs, aggcon)\ \texttt{to}\ aggvar \\
&\quad \mid aggexp_1,\ aggexp_2 \\
g_attrs, a_attrs &::= <\ [\ attr\ ]^{+}\ > \\
aggcon &::= aggvar \sim attr \mid aggvar \sim c \mid aggvar_1 \sim aggvar_2 \\
&\quad \mid\ !\ aggcon \mid aggcon_1\ \&\&\ aggcon_2 \\
aggop &::= \texttt{sum} \mid \texttt{count} \mid \texttt{avg} \mid \texttt{min} \mid \texttt{max}
\end{aligned}$$

Within the aggregating expression *aggexp*, it can be single expression as well as expressions sequence. For single example, it includes the aggregation elements as described

above, where: *g_attrs*, *a_attrs* $\in$ SEQ(ATTR) are the attributes sequences for the grouping and aggregating; EMF-formula *emf* specifies the desired properties for the trigger event, of which the target data set is created from the matched instance; *aggvar* is a variable name in syntax but semantically it is a mapping from group values to aggregating values.

For aggregating expression *aggexp*, it is *well-formed* if (1) *g_attrs* and *a_attrs* are disjointed; (2) (*g_attrs* $\cup$ *a_attrs*) $\subseteq$ attr(*e*), where e = tr(*emf*). In addition, the trigger and deciding event for *aggexp* are defined as the same to tr(*emf*). Then for the aggregating formula, it is *well-formed*, if (1) each of its aggregating expression is *well-formed*; (2) each of its aggregating expressions has the *same* trigger; (3) each of its aggregating expressions has the *same* grouping attributes.

Example 2. From [10]: for each user, the number of withdrawing peaks over the last 30 days does not exceed a threshold of 5, where a peak is a value at least twice the average over some time window(30 days).

```
//events part
e1 = ('withdraw', [ user, amount ]) ;
// policy part
policy =  always ( aggregate(
             avg ([0,31d), e1, <e1.user>, <e1.amount>, _) to ave,
             cnt ([0,31d), e1, <e1.user>, <e1.amount>,
                       e1.amount >= 2 * ave ) to c
          ) with c < 5 )
```

In the example, two aggregating expressions are included in the aggregate and these two expressions have the same time interval, grouping and aggregating attributes. Within the aggregating constraint of *cnt* expression, the aggregating variable *ave* of *avg* expression is referred to compare with the value of regular attribute. Semantically, the comparing is finished by comparing each withdrawing amounts with the average value for the same user, through one user to another. In fact, such nest structure not only complicates the implementation but also results in expensive running costs which shall be seen from evaluation section.

2.4 Formal Semantics for ECL

In this section, the semantics of ECL will be defined based on the trace.

Notation: Let INS denote event instances set, then trace τ is a finite sequence in SEQ(INS). For trace τ, $\tau(i)$ denotes $(i+1)$-th instance in the trace for integer $i \geq 0$; $|\tau|$ is the length of trace; trace τ is *well-formed*, if it satisfies, $\tau(i).ts \leq \tau(j).ts$, for each two non-negative integers i and j with $i < j$. It means there is a full order over well-formed trace based on the time stamp attribute, denoted as $\preceq$. $\tau_{\leq i}$ denotes the sub trace of τ with the index of its instance not greater than i. We use TRACE to denote all the well-formed traces of the discourse.

For two traces τ_1 and τ_2, τ_1 is a *sub-trace* of τ_2, denoted as $\tau_1 \sqsubseteq \tau_2$, iff, (1) $\tau_1 \subseteq \tau_2$; (2) for instances $ins_1, ins_2 \in \tau_1$, if $ins_1 \preceq ins_2$ in τ_1, then also $ins_1 \preceq ins_2$ in τ_2.

$\tau \sqcup \langle ins \rangle$ denotes the trace after appending *ins* to the its tail; $\tau \backslash (i)$ denotes the trace after deleting first $(i+1)$ instances from τ, where $i > 0$; $\tau \backslash \{ins\}$ denotes the trace

after deleting the first instance *ins* nearest to the tail; For time interval $tc = [t_1, t_2\circ]$, the bound of tc is non-negative integer $b = t_2 -^\circ t_1$[6].

$\tau \uplus_b \langle ins \rangle$ denotes *bounded appending ins* to τ, and formally defined as, $(\tau \sqcup \langle ins \rangle)\backslash(k)$, where k is maximal number of $\{j|\ ins.ts -^\circ \tau(j).ts > b\}$, i.e., all the out-of-date instances w.r.t. new added *ins*;

$\tau \Uparrow_b \langle ins \rangle$ denotes *bounded updating* τ by *ins* and defined as $\tau\backslash(k)$, where k is maximal number of $(\tau \sqcup \langle ins \rangle)\backslash(k)$. Note that, both operators need to update the τ by new instance, but with the difference regrading to whether the new *ins* needs to be added.

For instance *ins* and its attributes $attrs \subseteq \texttt{attr}(ins)$, $ins \rightharpoonup_{\langle attrs \rangle}$ denotes the value restriction from *ins* with respect to *attrs*;

$\tau \restriction_{\langle attrs \rangle}$ denote the value restrictions set satisfying that, for each value $\bar{d}$ in the set, there exists at least one instance $ins \in \tau$ with $\bar{d} = ins|_{\langle attrs \rangle}$.

For instances set *Ins* and attributes set *attrs*, $Ins|_{\langle attrs \rangle}$ is a multi-set mt with $\texttt{VALUE}_{\langle attrs \rangle} \rightharpoonup \mathbb{N}$, where $\texttt{VALUE}_{\langle attrs \rangle}$ represents the value restrictions domain for attributes sequence $\langle attrs \rangle$.

The *event valuation* υ_e is a *binding* of $\texttt{EVENT} \rightharpoonup \texttt{INS}$; event e is satisfied by υ_e, denoted as $\upsilon_e \vDash_{em} e$, iff e is matched by $\upsilon_e(e)$. For valuation υ_e, event e, and its instance *ins*, notation $\upsilon_e[e \mapsto ins]$ denotes the updated from υ_e by e and *ins*, and $[\,]$ denotes empty valuation. For two *valuation* υ_1, υ_2, υ_1 is a *sub-binding* of υ_2, denoted as $\upsilon_1 \sqsubseteq \upsilon_2$, if $\texttt{var}(\upsilon_1) \subseteq \texttt{var}(\upsilon_2)$ and $\upsilon_1(x) = \upsilon_2(x)$;

Based on event matching notion, following we provide semantics for two event operators `ors`, representing *others*, and `clone`, as follows:

Definition 4. *For event e, event set E, and event valuation* υ_e, $\texttt{ors}(e)$, $\texttt{ors}(E)$, *and* $\texttt{clone}(e)$ *are events in* EVENT, *where* $\texttt{clone}(e).id \neq e.id$.
$\upsilon_e \vDash_{em} \texttt{ors}(e)$, *iff* $\upsilon_e \nvDash_{em} e$; $\upsilon_e \vDash_{em} \texttt{ors}(E)$, *iff* $\upsilon_e \nvDash_{em} e'$, *for each* $e' \in E$.
$\upsilon_e \vDash_{em} \texttt{clone}(e)$, *iff* $\upsilon_e \vDash_{em} e$.

Note that, `ors` operator could be considered as some way of events abstraction, and the use of operator `clone` is to distinguish one from another among the copies of same event, which will be useful in the creating correlating relations. In fact, based on the event matching semantics, it is possible to define other composite events patterns if necessary.

Definition 5. *For events correlating constraint econ, and event valuation* υ_e, *the relation* $\upsilon_e \vDash_{ec}$ *econ is defined as follows :*

- $\upsilon_e \vDash_{ec} e_1.attr_1 = e_2.attr_2 \iff \upsilon_e \vDash_{em} e_1$, $\upsilon_e \vDash_{em} e_2$, *and* $\upsilon_e(e_1)(attr_1) = \upsilon_e(e_2)(attr_2)$.
- $\upsilon_e \vDash_{ec} econ_1 \,\&\&\, econ_2 \iff \upsilon_e \vDash_{ec} econ_1$ *and* $\upsilon_e \vDash_{ec} econ_2$.

For time interval $[t_1, t_2\circ)$, *where* t_1, t_2 *are non-negative integers with* $t_1 < t_2$, *and* $\circ \in \{d, h, m, s\}$, $[t_1, t_2\circ)$ *is satisfied by the instances sequence* $\langle ins_1, ins_2 \rangle$ *formed by two instances* ins_1 *and* ins_2, *denoted as* $\langle ins_1, ins_2 \rangle \vDash_t [t_1, t_2\circ)$, *iff,* $t_1 \leq ins_2(ts) -^\circ ins_1(ts) < t_2$.

Semantics for ECL Formula

[6] $-^\circ$ is the regular operator but calculated based on $\circ$ scale, hereafter, the notation would be ignored.

Definition 6. *For event matching formula f, trace τ, and integer $i \geq 0$, the relation $\tau(i) \models f$ is defined recursively as follows, where $e' = \texttt{tr}(f_1)$ and $e'' = \texttt{tr}(f_2)$:*

- $\tau(i) \models e$ `iff` $[e \mapsto \tau(i)] \models_{em} e$.
- $\tau(i) \models \texttt{ors}(e)$ `iff` $\tau(i) \not\models e$.
- $\tau(i) \models constr$ `when` f_1 `iff` *if* $\tau(i) \models f_1$ *and* $\tau(i) \models e'$, *then* $[e' \mapsto \tau(i)] \models_c constr$.
- $\tau(i) \models \texttt{before}(tc, f_1, e, econ)$ `iff` *if* $\tau(i) \models e$, *then there exists non-negative integer* $j < i$, *s.t.,* $\tau(j) \models f_1$, $\tau(j) \models e'$, $\langle ins_1, ins_2 \rangle \models_t tc$, *and* $[e' \mapsto \tau(j), e \mapsto \tau(i)] \models_{ec} econ$.
- $\tau(i) \models \texttt{after}(tc, e, f_1, econ)$ `iff` *if* $\tau(i) \models e$, *then there exists a non-negative integer* $j > i$, *s.t.,* $\tau(j) \models f$, $\tau(j) \models e'$, $\langle ins_1, ins_2 \rangle \models_t tc$, *and* $[e \mapsto \tau(i), e' \mapsto \tau(j)] \models_{ec} econ$.
- $\tau(i) \models \texttt{beforeSince}(tc, f_1, f_2, e, econ)$ `iff` *if* $\tau(i) \models e$, *then there exists a non-negative integer* $j < i$, *s.t., for every integer* $j < k < i$, $\tau(j) \models f_1$, $\tau(j) \models e'$, $\tau(k) \models f_2$, $\tau(k) \models e''$, $\langle ins_1, ins_2 \rangle \models_t tc$, *and* $[e' \mapsto \tau(j), e'' \mapsto \tau(k), e \mapsto \tau(i)] \models_{ec} econ$.
- $\tau(i) \models \texttt{afterUntil}(tc, e, f_1, f_2, econ)$ `iff` *if* $\tau(i) \models e$, *then there exists a non-negative integer* $j > i$, *s.t., for every integer* $i < k < j$, $\tau(j) \models f_2$, $\tau(j) \models e''$, $\tau(k) \models f_1$, $\tau(k) \models e'$, $\langle ins_1, ins_2 \rangle \models_t tc$, *and* $[e \mapsto \tau(i), e' \mapsto \tau(k), e'' \mapsto \tau(j)] \models_{ec} econ$.
- $\tau(i) \models !\, f_1$ `iff` $\tau(i) \not\models f_1$.
- $\tau(i) \models f_1 \,\&\, f_2$ `iff` $\tau(i) \models f_1$ *and* $\tau(i) \models f_2$.

For the TMF formula, its semantics is defined over trace as follows.

Definition 7. *For TMF-formula f, f_1, f_2, EMF-formula emf, and trace τ, the relation $\tau \models f$ is defined based on two types of f :*
(1) $\tau \models$ `always` *emf* `iff` *for every* $i \geq 0$, $\tau(i) \models emf$.
(2) $\tau \models$ `exists` *emf* `iff` *there exists* $i \geq 0$, $\tau(i) \models emf$.
(3) $\tau \models !\, f$, `iff` $\tau \not\models f$.
(4) $\tau \models f_1 \,\&\&\, f_2$ `iff` $\tau \models f_1$ *and* $\tau \models f_2$.

Semantics for Aggregation

The semantics of aggregation refers to *sliding window model* with respect to time interval.

For trace $\tau \in \texttt{TRACE}$ and time interval tc, the *sliding window model* with time interval tc within τ up to i, denoted as $M_{\tau_{\leq i}}(tc)$, is the sub-trace of $\tau_{\leq i}$ with $\langle ins, \tau(i) \rangle \models_t tc$ for each its instance ins.

For EMF-formula f, $M_{\tau \leq i}(tc, f)$ is a sub-trace of $M_{\tau_{\leq i}}(tc)$ with its instance satisfying $ins \models f$ and $ins \models_{em} e$, where $e \in \texttt{tr}(f)$. Formally, it is defined as follows:

$$M_{\tau \leq i}(tc, f) = \begin{cases} M_{\tau_{\leq i-1}}(tc, f) \uplus_b \langle \tau(i) \rangle, & \text{if } \tau(i) \models f \\ & \text{and } ins \models_{em} e \text{ with } e = \texttt{tr}(f); \\ M_{\tau_{\leq i-1}}(tc, f) \Uparrow_b \langle \tau(i) \rangle, & \text{o.w.} \end{cases} \quad (1)$$

, for $i \geq 0$, and $M_{\tau_{\leq -1}}(tc, f) = \emptyset$.

Aggregating variable: for aggregating variable *aggvar*, formally, *aggvar* is a 3-ary (*aname*, *event*, *g_attrs*), where *aname*, *event*, and *g_attrs* are the name, involved event, and *grouping attributes* of *aggvar*. Aggregating variable is *well-formed*, if $aggvar.g\_attrs \subseteq \texttt{attr}(aggvar.event)$. Semantically, it corresponds to the partial mapping of $\texttt{VALUE} \rightharpoonup \mathbb{Q}$,

which assign a statistic value (e.g., sum, average,etc.) to *grouping* value in VALUE. We use agVAR and agVALUE to denote all the aggregating variables and all aggregating values respectively, meanwhile, agOPS to the aggregation operators set $\{\texttt{sum}, \texttt{avg}, \texttt{cnt}, \texttt{min}, \texttt{max}\}$, where ω is used to refer the aggregation operator.

Definition 8. *For instances sequence θ, two attributes sets g_attrs and a_attrs, and aggregating operator ω, the aggregating map λ_{ag} : $\texttt{SEQ}(\texttt{INS}) \times \texttt{SEQ}(\texttt{ATTR}) \times \texttt{SEQ}(\texttt{ATTR}) \times \texttt{agOPS} \rightarrow (\texttt{VALUE} \rightarrow \mathbb{Q})$ is defined as:*

$$\lambda_{ag}(\theta, g\_attrs, a\_attrs, \omega) = \chi, \tag{2}$$

where

- $\texttt{Dom}(\chi) = \theta \restriction_{\langle g\_attrs \rangle}$;
- $\chi(\bar{d}) = \omega(Ins_{\bar{d}}|_{\langle a\_attrs \rangle})$, *for each* $\bar{d} \in Dom(\chi)$;
- $Ins_{\bar{d}} = \{ins \mid \bar{d} = ins|_{\langle g\_attrs \rangle}, ins \in \theta\}$

Within the definition, for the instances sequences, the first equation is to group instances based on grouping attributes g_attrs; the second one is do the aggregation over grouped instances set for given grouping value $\bar{d}$; the last one is define related instances group for given grouping value.

aggregating valuation υ_{ag} is a binding of $\texttt{agVAR} \rightharpoonup \texttt{agVALUE}$. then the satisfiable for aggregating constraints could be provided as follow:

Definition 9. *For aggregating constraint aggcon, event e, attribute attr, event valuation υ_e, and aggregating valuation υ_{ag}, aggcon is satisfied by valuations υ_e and υ_{ag}, denoted as $(\upsilon_e, \upsilon_{ag}) \vDash_{ag} aggcon$, is defined as follows:*

- $(\upsilon_e, \upsilon_{ag}) \vDash_{ag} aggvar \sim e.attr$, iff, $\upsilon_{ag}(aggvar)(\bar{d}) \sim \upsilon_e(e)(attr)$ *for each* $\bar{d} \in \texttt{Dom}(\upsilon_{ag}(aggvar))$.
- $(\upsilon_e, \upsilon_{ag}) \vDash_{ag} aggvar \sim c$, iff, $\upsilon_{ag}(aggvar)(\bar{d}) \sim c$ *for each* $\bar{d} \in \texttt{Dom}(\upsilon_{ag}(aggvar))$.
- $(\upsilon_e, \upsilon_{ag}) \vDash_{ag} aggvar_1 \sim aggvar_2$, iff, $aggvar_1.g\_attrs = aggvar_2.g\_attrs$, $\texttt{Dom}(\upsilon_{ag}(aggvar_1)) = \texttt{Dom}(\upsilon_{ag}(aggvar_2))$, *and* $\upsilon_{ag}(aggvar_1)(\bar{d}) \sim \upsilon_{ag}(aggvar_2)(\bar{d})$ *for each* $\bar{d} \in \texttt{Dom}(\upsilon_{ag}(aggvar_1))$.
- $(\upsilon_e, \upsilon_{ag}) \vDash_{ag} !aggcon, aggcon_1 \;\&\&\; aggvar_2$, *can be defined in standard way.*

Then the semantics of aggregating expression can be defined as:

Definition 10. *For aggregating expression aggexp, trace τ, and two aggregating valuations υ_{ag} and υ'_{ag}, the relation $(\tau(i), \upsilon_{ag}, \upsilon'_{ag}) \vDash aggexp$ is defined as follows:*

$(\tau(i), \upsilon_{ag}, \upsilon'_{ag}) \vDash aggop(f, g\_attrs, a\_attrs, aggcon)\texttt{to}\ aggvar$, iff, $\tau(i) \vDash f$, $([e \mapsto \tau(i)], \upsilon_{ag}) \vDash_{ag} aggcon$, *and* $\bar{\upsilon}_{ag} = \upsilon_{ag}[aggvar \mapsto av]$, *where* $e \in \texttt{tr}(f)$ *and* $av = \lambda_{ag}(M_{\tau_{\leq i}}(tc, f), g\_attrs, a\_attrs, aggop)$.

Definition 11. *For aggregation event matching formula f and trace τ, $\tau(i) \vDash f$ is defined as follows:*

$\tau(i) \vDash \texttt{aggregate}(aggexp_1, aggexp_2)\ \texttt{with}\ aggcon$, iff, *there exist two aggregating valuations υ_{ag} and υ'_{ag}, s.t., $(\tau(i), [\,], \upsilon_{ag}) \vDash aggexp_1$, $(\tau(i), \upsilon_{ag}, \upsilon'_{ag}) \vDash aggexp_2$, and $([e \mapsto \tau(i)], \upsilon'_{ag}) \vDash_{ag} aggcon$, where $e \in \texttt{tr}(f)$.*

3 Events indexed reaction system(ERS)

To monitor the full featured ECL, it is necessary to have a uniform and powerful analysis theory. In this section, *Events Indexed Reaction System*(ERS) is proposed, which in fact is the *rules system* plus *working structure*. More specifically, ERS is a 2-ary(rs, ws), where:
(1)rs is the *rules system* with *reaction* rules, where *reaction* is sequence of operations over *working structure*;
(2) rs is the *working structure* in charge of organizing instances for their efficiently storing, consuming, and assessing.

Different to the *net* form of working memory in RETE algorithm [17] or other rule engine, ERS working structure is of tree structure including indexing, bounded queue, and also statistics tree for aggregation computing.

3.1 The rules system of ERS

To define the definition of rules system, let notations of $\texttt{ATTR}$, $\texttt{ECON}$, $\texttt{TC}$, $\texttt{agOPS}$, $\texttt{agVAR} \subseteq \texttt{VAR}$, $\texttt{CONSTR}$, and $\texttt{agCON}$, to denote the events attributes set, events correlating constraints set, time constraints set, aggregating operators set, aggregating variables set, constraints set, and aggregating constraints, respectively; and notation $\texttt{agVALUE} \subseteq \texttt{VALUE} \times \texttt{VALUE}$ is for aggregating values set; finally, notation $\texttt{BOOL} \subseteq \texttt{VALUE}$ denotes boolean set, and $\texttt{VIOD}$ is a special symbol denoting the fact of without returning in operation definitions.

Definition 12. *The rules system rs of ERS is a 3-ary* (E, OPS, R)*, where:*
(1) E *is the events set with* $E \subseteq \texttt{EVENT}$*;*
(2) OPS *is the operations set, where each operation is of one of following types:*

- *delete-operation* $\#d$*:* $\{0,1\} \times \texttt{EVENT} \times \texttt{EVENT} \longrightarrow \texttt{VIOD}$ *;*
- *get-operation* $\#g$*:* $\{0,1\} \times \texttt{EVENT} \times \texttt{EVENT} \times \texttt{TC} \longrightarrow \texttt{SEQ}(\texttt{INS})$*;*
- *write-operation* $\#w$*:* $\{0,1\} \times \texttt{EVENT} \times \texttt{EVENT} \longrightarrow \texttt{VIOD}$*;*
- *failure/success-operation* $\#fail$*,* $\#succ$*:* $\mathbb{N} \times \texttt{EVENT} \times \texttt{EVENT} \times \texttt{EVENT} \longrightarrow \texttt{VIOD}$*;*
- *next-operation* $\#next$*:* $\longrightarrow \texttt{VIOD}$*;*
- *existence-operation* $\#ge$*:* $\{0,1\} \times \texttt{EVENT} \times \texttt{EVENT} \times \texttt{TC} \longrightarrow \texttt{BOOL}$*;*
- *empty-operation* $\#empty$*:* $\{0,1\} \times \texttt{EVENT} \times \texttt{EVENT} \longrightarrow \texttt{BOOL}$*;*
- *evaluation-operation* $\#eval$*:* $\texttt{CONSTR} \longrightarrow \texttt{BOOL}$*;*
- *time-comparing-operation* $\#tcm$*:* $\texttt{EVENT} \times \texttt{EVENT} \longrightarrow \{-1, 0, 1\}$*;*
- *aggop-operation* $\#\omega$*:*
  $\{0,1\} \times \texttt{EVENT} \times \texttt{SEQ}(\texttt{ATTR}) \times \texttt{SEQ}(\texttt{ATTR}) \times \texttt{agCON} \times \texttt{agVAR} \longrightarrow \texttt{agVALUE}$*;*
  where $\omega \in \{\texttt{sum}, \texttt{avg}, \texttt{cnt}, \texttt{min}, \texttt{max}\}$*;*
- *aggregate-operation* $\#ag$*:* $\texttt{SEQ}(\texttt{agOPS}) \times \texttt{agCON} \longrightarrow \texttt{BOOL}$*.*

(3)R *is the rules set and each rule* $r \in R$ *is of the form* $r : lhs \rightarrow rhs$*, where* lhs *is the event from* E *and* rhs *is defined as follows:*

$$
\begin{aligned}
rhs ::= \;& c\_reaction \\
c\_reaction ::= \;& ops \mid c\_reaction \;;\; ops \\
ops ::= \;& op \mid op \cdot ops \mid cond \;?\; ops_1 : ops_2 \\
op ::= \;& \#d \mid \#g \mid \#w \mid \#fail \mid \#succ \mid \#\omega \\
cond ::= \;& \#ag \mid \#empty \mid \#ge \mid \#tcm \sim 0 \mid \#eval \\
& \mid \;!\, cond \mid cond_1 \;\&\&\; cond_2
\end{aligned}
$$

In the definition, event set E is the alphabet for the operation as well as the right hand of rule. For each operations in OPS, most of them share common features: $\{0, 1\}$, representing the related structure on which operations take effect on, i.e., *beforeIIS* or *afterIIS* instances indexed structure which is the main storing mechanism of working structure; pair of events $\texttt{EVENT} \times \texttt{EVENT}$ with the first one representing *target* event and the second for *trigger* event.

For events $e, ta, tr \in \texttt{EVENT}$, integer $i \in \{0, 1\}$, and integer $t \in \mathbb{N}$, the intuitive semantics of each operations are briefed listed as follows:

- $\#d(i, ta, tr)$: delete all the correlated ta instances from the related value structure of i instances indexed structure, when tr instance was occurred;
- $\#g(i, ta, tr, tc)$: get all the correlated ta instances within time interval tc from the related value structure of i instances indexed structure, when tr instance was occurred;
- $\#w(i, ta, tr)$: write the matched ta instances as a new fact into the related value structure of i instances indexed structure for the future occurrence of tr instance;
- $\#fail/succ(t, ta, e, tr)$: create a new t-type *failure*/*success* information and add it into the related container of working structure, when tr instance was occurred;
- $\#next()$: terminate current operations executing and go to the next operations if there is, otherwise read the next instance;
- $\#ge(i, ta, tr, tc)$: check whether there exists correlated ta instances within time interval tc in the related value structure of i instances indexed structure, when tr instance was occurred;
- $\#empty(i, ta, tr)$: check whether it is empty for the value structures determined by ta and tr in i instances indexed structure.
- $\#eval(constr)$: check whether the $constr$ is satisfiable in current moment.
- $\#tcm(ta, tr)$: comparing the time values of current matched ta instance and tr instance. if ta instance is occurred after tr instance, then return 1; else if their time is equal, then return 0; otherwise return -1;
- $\omega(i, e, g\_attrs, a\_attrs, aggcon, aggvar)$: aggregate the values, with g_attrs as grouping attributes and a_attrs as aggregating attributes, for all the currently matched e instances which also satisfy $aggcon$ constraints, and use $aggvar$ to refer to such aggregating values.
- $\#ag(agops, aggcon)$: sequentially execute aggregation operators as specified in $agops$, and then evaluate the $aggcon$ by the aggregating values.

For each rule in rs, it is of the form, $event \rightarrow c\_reaction$. Semantically, it means, when the trigger $event$ is matched, then reaction $c\_reaction$ is invoked and started to execute the operations as specified in $c\_reaction$. The rule system rs is *deterministic* if there do not exist any two rules in R with same right hand, and ERS is *deterministic* if its rule system is deterministic. In this work, only deterministic rule system is considered.

For the rule right hand, $c\_reaction$, it could be operations expression ops as well as the compositional reaction, which composites ops with sequential operator ";". For the ops, it could be simple operation, conditional operations, or their connection by the chain operator "$\cdot$". Note that, the sequential operator is different to chain operators. The sequential operator is used in the merging of rule systems by connecting related reactions to form composite reactions, and during the running, each of these reactions would be executed sequentially; whereas, chain operators is used to chain operations

into composite operation and when rule triggered, these operations might not be executed all, since if the final operation, $\#succ/fail$, or next operation is included, the followed operations would not be executed.

Comparing to classic rule form, e.g., *event-condition* $\rightarrow$ *action* [21], ERS rule has distinct characteristics:(1) the left hand is only one event as rule *trigger*, by which finding triggered rule would be speeded up; (2) the conditional operation adopts the form of concise conditional expression, "?_:_", which could speed up conditions evaluation by avoiding repeated conditions evaluation; (3) two distinct connecting operators are used with different purposes.

Example 3. For TMF-formula `always` (`after`(_, e6, e7, e6.caseID = e7.caseID)), then its rules system can be specified as follows:

```
ERS :
%% wrting e6 instance into the related value structure of e6 for e7
e6 ->  #w(1,e6,e7)

%% if there is related e6 instance in the structure for e7 instance,
%% delete all such e6 instances and create success insances of type2
%% for such e6 instances and e7 instance; otherwise, read the next.
e7 ->  #ge(1, e6, e7) ? #d(1,e6,e7).#succ(2,e6,e7 ):#next()

%% when end event occurred, if the structure of e6 for e7 is
%% not empty, then create violation instances of type 3 for
%% each such e6 instance; otherwise, read the next.
end -> !#empty(1,e6,e7) ? #fail(3,e6,e7):#next()
```

3.2 Instances Indexed Structure(IIS)

From the above, it should be implied that, the value structure, as the mechanism for storing instances, is the essential part for explaining and understanding operations executions. Furthermore, to efficiently assess the stored instances, the value structure needs to be equiped with indexing structure. In this work, we term the instances value structure with indexing as instances indexed structure(IIS) and its skeleton structure is a tree as Fig. 2, which is of four layers and with storing mechanism as leaf nodes.

Definition 13. *Instances Indexed Structure iis is a 4-ary (Target, Trigger, VS, δ), where :*

- *Target, Trigger* $\subseteq$ `EVENT` *are events sets ;*
- *VS is the value structure set, and value structure vs is defined as*

$$vs{=}(corr, b, \theta),$$

  where:
  - $corr \in$ `SEQ`(`ATTR`) *is correlating attributes;*
  - $b \in \mathbb{N} \cup \{\infty\}$ *is the bound of storing mechanism;*
  - θ : `VTuple` $\rightarrow$ `SEQ`(`INS`) *is a partial one to one mapping assigning to vtuple with instances storing mechanism.*
- δ : *Target*$\rightarrow$ (*Trigger*$\rightarrow$*VS*) *is a mapping assign an value structure to target and trigger.*

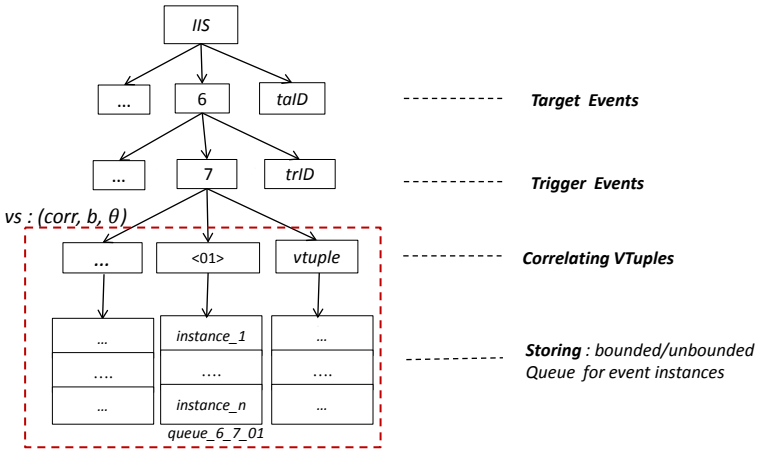

Fig. 2. Instances indexed structure

Within the IIS, the basis element is value structure, where: $corr \in \texttt{SEQ}(\texttt{ATTR})$ is the *correlating sequence* and its value is a tuple represented as *vtuple*. Notation $\texttt{VTuple} \subseteq \texttt{VALUE}$ denotes all the value of *vtuple*. By the *vtuple* and mapping θ, the target instances are classified and stored in relevance place. As depicted with dashed rectangle in Fig.2, value structure $(corr, b, \theta)$ semantically corresponds to a set of the pair of *vtuple* and storing mechanism. Note that, in this work the storing mechanism is the bounded/unbounded queue. For the correlated target and trigger events, the mapping δ not only decides the queue for target instances, but also defines the indexing, i.e., *corr*, for storing and assessing target instances. Specifically, storing target instance is done by first selecting relevance queue though its *vtuple* and θ; on the other hand, assessing target instance is also finished by first finding relevance queue based on the *vtuple* of trigger instance. To this end, it is clear that, *corr* plays the indexing role for trigger instance finding target one.

As depicted in Fig.2, the queue finding is exactly like finding a path from the top to the related leaf. Structurally, the IIS in fact is determined by the set of *correlating tuples*, i.e.,(*target*, *trigger*, *corr*, *b*), which corresponds set of paths from target and trigger to queue. For instance in the figure, the path from root, node 6, node 7, $< 01 >$, to queue_6_7_01 corresponds to correlating tuple $(e_6, e_7, caseID, b)$, which means, fo each e_7 instances, if their caseID is $< 01 >$, then the storing and assesing for related e_6 instances are operated over queue_6_7_01.

Intuitively, the IIS is built based on following two frequently and basis operations during monitoring:

- GET: when *trigger* instance arrived, it needs to assess the *target instance* (or *fact*) occurred before with *correlating condition* and also might under *time bound*.
- WRITE: when a *target* instance arrived, it needs to be saved for behind *trigger* instances to assess it with *correlating condition* and also might under *time bound*.

However, as for WRITE, after target instances stored, there are two subcases with subtle differences regarding to whether the stored instance requires the desired trigger

to be occurred after. Then, IIS is divided into two types: *beforeIIS*, wherein the stored target instance does not require some trigger instance must occur after, and *afterIIS*, where each stored target instance requires desired trigger instance must occur after. For example, for the operation $\#w(1, e_6, e_7)$, it writes occurred e_6 instance into afterIIS, and for each stored e_6 instance, it requires desired e_7 to be occurred; on the contrary, if changed 1 to 0 in the operation, e_6 instance is stored in beforeIIS for e_7 since e_7 instance requires desired e_6 instance needed to occur before.

Definition 14. *For an indexed events reaction system, its working structure ws is a 4-ary (beforeIIS, afterIIS, fCon,sCon), where:*

- *beforeIIS and afterIIS are instances indexed structures;*
- *fCon* $\in \mathtt{SET}(Failure)$ *is the failures container, where the failures domain* $Failure \subseteq \mathbb{N} \times \mathtt{INS} \times \mathtt{INS} \times \mathtt{INS}$.
- *sCon* $\in \mathtt{SET}(Success)$ *is the success container, where the success domain* $Success \subseteq \mathbb{N} \times \mathtt{INS} \times \mathtt{INS} \times \mathtt{INS}$.

In the definition, working structure includes not only two types IIS, but also failure and success containers. For convenience, we use $ws(0)$ and $ws(1)$ to denote beforeIIS and afterIIS of *ws*. Currently, 8 types of failures and successes are considered, which corresponds to the current set of events relation patterns in ECL. They are created respectively by *#fail* and *#succ* operators as listed in Table 1, where: type-1 is the basis; type-2 and type-3 are for *before* and *after* order; type-4 is for *beforeSince* and *afterUntil*; type-5 and type-6 are for *aggregation* and *when* respectively; type-7 and type-8 are for emf-composition and exists-tmf formula. For instance, the expression $\#succ(1, \_, \_, e_1)$ would create a type-1 success instance for e_1 instance happened, and then add it into *sCon* container. Note that, the success and failure instances, created by *#succ* and *#fail*, contain useful debugging information, especially for the failure.

Table 1. The success&failure models

No	Success	Failure
1	$\#succ(1, \_, \_, e_1)$: e_1 instance happened	$\#fail(1, \_, e_1, e_2)$: e_1 should happen but e_2 instance happened
2	$\#succ(2, \_, e_1, e_2)$: e_1 instance happened before e_2 instance	$\#fail(2, \_, e_1, e_2)$: no e_1 happened before e_2 instance
3	$\#succ(3, \_, e_1, e_2)$: e_2 instance happened after e_1 instance	$\#fail(3, \_, e_1, e_2)$: no e_2 happened after e_1 instance
4	$\#succ(4, e_0, e_1, e_2)$: e_1 always happens after e_0 instance and before e_2 instance	$\#fail(4, e_0, e_1, e_2)$: e_1 instance happened after e_0 instance and before e_2 instance
5	$\#succ(5, \_, \_, e_1)$: the aggcon is satisfied for e_1 instance occurring.	$\#fail(5, \_, \_, e_1)$: the aggcon is unsatisfied for e_1 instance occurring.
6	$\#succ(6, \_, \_, e_1)$: when-constr is satisfied for e_1 instance occurring.	$\#fail(6, \_, \_, e_1)$: when-constr is unsatisfied for e_1 instance occurring.
7	$\#succ(7, \_, \_, e_1)$: emf-composition is satisfied for e_1 instance occurring.	$\#fail(7, \_, \_, e_1)$: emf-composition is unsatisfied for e_1 instance occurring.
8	$\#succ(8, \_, \_, e_1)$:exists-tmf is satisfied for e_1 instance occurring.	$\#fail(8, \_, \_, e_1)$: exists-tmf is unsatisfied for e_1 instance occurring.

Within the ERS, reaction rules and working structure are closed related since the assessing and storing operations are executed based on the working structure. For instance, when e_2 instance occurred, to execute the operation $\#g(0, e_1, e_2, \_)$, finding the queue, which stores e_1 instances, is needed to be done first in such way: get the

value structure $(corr, b)$ by $\delta(e_1)(e_2)$ from beforeIIS; and then get the related queue by $\theta(vtuple)$, where *vtuple* is the correlating value of *corr* from e_2 instance. Such relation is depicted as the *coverable* property of working structure for rule system as follow:

Definition 15. *For working structure ws and rules system rs, ws is coverable for rs, iff for each op ∈ OPS, if the i, ta, and tr are the iis symbol, target, and trigger of op respectively, then $\delta(ta)(tr) \neq \perp$, where δ is the mapping of $ws(i)$.*

Based on the *coverable* notion, then for ERS $ers = (rs, ws)$, it is *well-formed*, iff *ws* is coverable for *rs*. In this work, only well-formed ERS is considered.

3.3 Operational semantics of ERS

In this section, the operational semantics of ERS will be defined in SOS-style. In fact, the core of semantics is how to interpret the rule in the system, i.e., how to activate the rule and react on the read. Regarding to such reaction, it is essentially the state change of working structure of ERS, i.e., the working structure is the abstract state of ERS.

Definition 16. *For event reaction system ERS ers and working structure ws, the configuration of ERS is a 2-ary $\langle S, \upsilon_e \rangle$, where $S = (S_{iis}, S_{fCon}, S_{sCon})$, $S_{iis} : \{0,1\} \times \texttt{EVENT} \times \texttt{EVENT} \times \texttt{VTuple} \rightharpoonup \texttt{SEQ}(\texttt{INS})$ is the state of ws representing the instantaneous contents of queues in the indexed instances structures in ws, S_{fCon} represents the content of fContainer at state S, and υ_e is an events valuation.*

For simplicity, the subscript of S_{iis} will be ignored when the context is clear. For instance sequence $queue \in \texttt{SEQ}(\texttt{INS})$, we use $\texttt{size}(queue)$ to denote the size of *queue*. For working structure *ws*, its state *S*, two events *ta* and *tr*, attributes sequence *corr*, and event instance *ins*, we introduce some operators for easy exhibition:

- *vs-getting* operator: $\texttt{vs}(i, ta, tr) = ws(i).\delta(ta)(tr)$;
- *queue-selecting-for-getting* operator: $\texttt{S}_{\texttt{qg}}(S, 0, ta, tr, ins) = queue$, where $queue = S(i, ta, tr, \bar{d})$ is a bounded queue with bound of *vs.bound* , where $\bar{d} = ins|_{vs.corr}$, $vs = \texttt{vs}(i, ta, tr)$ and *tr* is matched by *ins*;
- *queue-selecting-for-writing* operator: $\texttt{S}_{\texttt{qw}}(S, i, ta, ins, tr) = queue$, where $queue = S(i, ta, tr, \bar{d})$ is a bounded queue with bound of *vs.bound* , where $\bar{d} = ins|_{vs.corr}$, $vs = \texttt{vs}(i, ta, tr)$ and *ta* is matched by *ins*;
- *queues-getting* operator: $\texttt{S}_{\texttt{Q}}(S, i, ta, tr) = \{q \mid \exists \bar{d} \in \texttt{VTuple}, q = S(i, ta, tr, \bar{d}), \texttt{size}(q) > 0, \text{ and } vs = \texttt{vs}(i, ta, tr)\}$;
- *queues-updating* operator: $\texttt{UP}_{\texttt{Q}}(S, i, ta, tr, ins)$ is a queue set, $\texttt{S}_{\texttt{Q}}(S, i, ta, tr)$ in *S*, updated by *ins*, where *tr* is matched by *ins*. Formally, for each $q \in \texttt{UP}_{\texttt{Q}}(S, i, ta, tr, ins)$, let $S_Q = \texttt{S}_{\texttt{Q}}(S, i, ta, tr)$, then *q* is defined based on cases:

$$q = \begin{cases} q' \uplus_b \langle ins \rangle, & \text{if } \exists q' \in S_Q, q' = \texttt{S}_{\texttt{qg}}(S, i, ta, tr, ins); \\ \langle ins \rangle, & \text{if } \texttt{S}_{\texttt{qg}}(S, i, ta, tr, ins) = \perp; \\ q' \Uparrow_b \langle ins \rangle, & \text{for all other } q' \in S_Q. \end{cases} \tag{3}$$

- *new* failure/success operators:
  $\texttt{new}_{\texttt{f}}$: $\mathbb{N} \times \texttt{EVENT} \times \texttt{EVENT} \times \texttt{EVENT} \times (\texttt{EVENT} \rightarrow \texttt{INS}) \longrightarrow \texttt{SET}(\text{Failure})$.
  $\texttt{new}_{\texttt{s}}$: $\mathbb{N} \times \texttt{EVENT} \times \texttt{EVENT} \times \texttt{EVENT} \times (\texttt{EVENT} \rightarrow \texttt{INS}) \longrightarrow \texttt{SET}(\text{Success})$.

Based on the *configuration* of ERS, its dynamics is defined by the reaction of its rules system on working structure as follows.

Definition 17. *For event reaction system ERS ers, trace τ, and rule r: lhs $\rightarrow$ rhs, rule r can make ers transit from S to S′ for the instance $\tau(i)$, denoted as $S \vdash\ lhs \rightarrow rhs \Rightarrow^{\tau(i)} S'$, $\texttt{iff}$, $\tau(i) \vDash_{em} lhs$ and for valuation $\upsilon_e = [\ lhs \mapsto \langle \tau(i) \rangle\ ]$, it is satisfied for $\langle S,\ \upsilon_e \rangle \vdash rhs \Rightarrow \langle S', \upsilon'_e \rangle$, where the relation $\langle S,\ \upsilon_e \rangle \vdash rhs \Rightarrow \langle S', \upsilon'_e \rangle$ is defined recursively based on the structure of rhs as follows:*

1. *(Basic Operators) if rhs is the basic operator, then*
   - *(Get) rhs= $\#g(j, e_1, e_2, tc)$, $j \in \{0, 1\}$, then for each $ins \in \upsilon_e(e_2)$,*

$$(\texttt{GET}_0) \quad \frac{ins \vDash_{em} e_2; \langle ins', ins \rangle\ \vDash_t\ tc}{\langle S, \upsilon_e \rangle \vdash\ \#g(0, e_1, e_2, tc) \Rightarrow \langle S, \upsilon_e[e_1 \mapsto \langle ins \rangle] \rangle}$$

     , where $ins' \in$ queue and queue $= \texttt{S}_{\texttt{qg}}(S, j, e_1, e_2, \upsilon_e(e_2))$.

$$(\texttt{GET}_1) \quad \frac{ins \vDash_{em} e_2; \langle ins', ins \rangle\ \vDash_t\ tc \text{ for each } ins' \in \overline{ins}}{\langle S, \upsilon_e \rangle \vdash\ \#g(1, e_1, e_2, tc) \Rightarrow \langle S, \upsilon_e[e_1 \mapsto \overline{ins}] \rangle}$$

     , where $\overline{ins} \preceq$ queue and queue $= \texttt{S}_{\texttt{qg}}(S, j, e_1, e_2, \upsilon_e(e_2))$.
   - *(Delete) rhs= $\#d(j, e_1, e_2, tc)$, then,*

$$(\texttt{DELETE}) \quad \frac{\langle S, \upsilon_e \rangle \vdash\ \#g(j, e_1, e_2, tc) \Rightarrow \langle S, \upsilon_e[e_1 \mapsto \overline{ins}] \rangle}{\langle S, \upsilon_e \rangle \vdash\ rhs \Rightarrow \langle S', \upsilon_e[e_1 \mapsto \overline{ins}] \rangle}$$

     , where $\texttt{S}_{\texttt{q}}(S, j, e_1, \upsilon_e(e_2)) = \texttt{S}_{\texttt{q}}(S', j, e_1, \upsilon_e(e_2)) \backslash \overline{ins}$.
   - *(Write) rhs= $\#w(j, e_1, e_2)$, then for each $ins \in \upsilon_e(e_1)$,*

$$(\texttt{WRITE}) \quad \frac{ins \vDash_{em} e_1}{\langle S, \upsilon_e \rangle \vdash\ rhs \Rightarrow \langle S', \upsilon_e \rangle}$$

     , where let queue $= \texttt{S}_{\texttt{qw}}(S, j, e_1, \upsilon_e(e_1), e_2)$, then

$$\texttt{S}_{\texttt{qw}}(S', j, e_1, \upsilon_e(e_1), e_2) = \begin{cases} queue \uplus_b \langle ins \rangle, & \text{if queue} \neq \perp; \\ \langle \upsilon_e(e_1) \rangle, & o.w. \end{cases} \tag{4}$$

   - *(Next) rhs= $\#next()$, then $\langle S, \upsilon_e \rangle \vdash\ rhs \Rightarrow \langle S, \upsilon_e \rangle$.*
   - *(Failure/Success) rhs= $\#fail(k, e, e_1, e_2)$ or $\#succ(k, e, e_1, e_2)$, then for each $ins \in \upsilon_e(e_2)$:*

$$(\texttt{FAIL/SUCC}) \quad \frac{ins \vDash_{em} e_2}{\langle S, \upsilon_e \rangle \vdash\ rhs \Rightarrow \langle S', \upsilon_e \rangle},$$

     where $S'.fCon = S.fCon \cup \texttt{new}_{\texttt{f}}(k, e, e_1, ins, \upsilon_e)$ for failure operator, and $S'.sCon = S.sCon \cup \texttt{new}_{\texttt{f}}(k, e, e_1, ins, \upsilon_e)$ for success operator;
2. *(OPSChain) rhs= $op \cdot ops$, then*
   - *if $op \neq \#next$, $\#fail$, or $\#succ$, then*

$$(\texttt{OPSChain}_1) \quad \frac{\langle S, \upsilon_e \rangle \vdash\ op \Rightarrow \langle S', \upsilon'_e \rangle; \langle S', \upsilon'_e \rangle \vdash\ ops \Rightarrow \langle S'', \upsilon''_e \rangle}{\langle S, \upsilon_e \rangle \vdash\ rhs \Rightarrow \langle S'', \upsilon''_e \rangle},$$

   - *if $op = \#next$, $\#fail$, or $\#succ$, then*

$$(\texttt{OPSChain}_2) \quad \frac{\langle S, \upsilon_e \rangle \vdash\ op \Rightarrow \langle S', \upsilon_e \rangle}{\langle S, \upsilon_e \rangle \vdash\ rhs \Rightarrow \langle S', \upsilon_e \rangle},$$

3. *(Cond_ops) if rhs = cond ? ops_1 : ops_2 is the conditional operations, then*

$$(\texttt{IF} - \texttt{TRUE}) \quad \frac{\langle S, \upsilon_e \rangle \vdash\ cond \Rightarrow_c \texttt{true}; \langle S, \upsilon_e \rangle \vdash\ ops_1 \Rightarrow \langle S', \upsilon'_e \rangle}{\langle S, \upsilon_e \rangle \vdash\ rhs \Rightarrow \langle S', \upsilon'_e \rangle},$$

$$(\texttt{IF}-\texttt{FALSE}) \quad \frac{\langle S, v_e\rangle \vdash \ cond \Rightarrow_c \texttt{false}; \langle S, v_e\rangle \vdash \ ops_2 \Rightarrow \langle S', v'_e\rangle}{\langle S, v_e\rangle \vdash \ rhs \Rightarrow \langle S, v_e\rangle},$$

4. *(OPSEQ) rhs = c_reaction ; ops, then*

$$(\texttt{OPSEQ}) \quad \frac{\langle S, v_e\rangle \vdash \ c\_reaction \Rightarrow \langle S', v'_e\rangle; \langle S', v_e\rangle \vdash \ rhs_2 \Rightarrow \langle S'', v''_e\rangle}{\langle S, v_e\rangle \vdash \ rhs \Rightarrow \langle S'', v''_e\rangle},$$

Following is the definition of conditional expressions valuation.

Definition 18. *For ERS configuration $\langle S, v_e\rangle$ and conditional expression cond, the valuation cond under configuration, $\langle S, v_e\rangle \vdash cond \Rightarrow_c$, is defined as follows:*

- *(Get-Ex) if $cond = \#ge(j, e_1, e_2, tc)$, then*

$$(\texttt{GETEX}-\texttt{True}) \quad \frac{\langle S, v_e\rangle \vdash \ \#g(j, e_1, e_2, tc) \Rightarrow \langle S, v'_e\rangle; v'_e(e_1) \neq \perp}{\langle S, v_e\rangle \vdash \ cond \Rightarrow_c \texttt{true}};$$

$$(\texttt{GETEX}-\texttt{False}) \quad \frac{\langle S, v_e\rangle \vdash \ \#g(j, e_1, e_2, tc) \not\Rightarrow \langle S, v'_e\rangle; v'_e(e_1) = \perp}{\langle S, v_e\rangle \vdash \ cond \Rightarrow_c \texttt{false}};$$

- *(Empty) if $cond = \#empty(j, e_1, e_2)$, then*

$$(\texttt{EMPTY}-\texttt{True}) \quad \frac{queue = \mathtt{S_q}(S, j, e_1, v_e(e_2)); \texttt{size}(queue) = 0}{\langle S, v_e\rangle \vdash \ cond \Rightarrow_c \texttt{true}};$$

$$(\texttt{EMPTY}-\texttt{False}) \quad \frac{queue = \mathtt{S_q}(S, j, e_1, v_e(e_2)), \texttt{size}(queue) > 0}{\langle S, v_e\rangle \vdash \ cond \Rightarrow_c \texttt{false}};$$

- *(TCM) if $cond = \#tcm(e_1, e_2) \sim 0$, then*

$$(\texttt{TCM}-\texttt{True}) \quad \frac{v_e \vDash_{em} e_1, v_e \vDash_{em} e_2, v_e(e_1).ts \sim v_e(e_2).ts}{\langle S, v_e\rangle \vdash \ cond \Rightarrow_c \texttt{true}}$$

$$(\texttt{TCM}-\texttt{FALSE}) \quad \frac{v_e \vDash_{em} e_1, v_e \vDash_{em} e_2, v_e(e_1).ts \nsim v_e(e_2).ts}{\langle S, v_e\rangle \vdash \ cond \Rightarrow_c \texttt{false}};$$

- *(EVAL) if $cond = \#eval(constr)$, then for $ins \in v_e(e)$,*

$$(\texttt{EVAL}-\texttt{True}) \quad \frac{ins \vDash_{em} e, ins \vDash_c constr}{\langle S, v_e\rangle \vdash \ cond \Rightarrow_c \texttt{true}}$$

$$(\texttt{EVAL}-\texttt{FALSE}) \quad \frac{ins \vDash_{em} e, ins \nvDash_c constr}{\langle S, v_e\rangle \vdash \ cond \Rightarrow_c \texttt{false}}$$

  , where e is the involved event in the constr.

Following is devoted to define the evaluations of aggregating operators and aggregating expression. To distinguish event and aggvar valuations, ERS configuration is extended to $\langle S, v_e, v_{ag}\rangle$;

For given state S, event ta, and instance ins, similar to Equation 2, the aggregating mapping for ERS is defined as

$$\bar{\lambda}_{ag}(S, ta, ins, a\_attrs, \omega) = \bar{\chi}, \tag{5}$$

where

- $\texttt{Dom}(\bar{\chi}) = \bigcup_{q \in S_Q} \{q \restriction_{vs.corr}\}$;
- $\chi(\bar{d}) = \omega(q|_{\langle a\_attrs \rangle})$, where $\bar{d} = q \restriction_{vs.corr}$ and $q \in S_Q$;
- $S_Q = \texttt{UP}_\texttt{Q}(S, 0, ta, tr, ins)$, where $tr = ta$.

Comparting to the aggregating mapping λ in Equation 2 where all the related instances are scattered in the trace, $\bar{\lambda}$ is basically the same except dealing with the related instances in working structure which are classified and stored in the queue based on correlating values(or called grouping values in λ case). Also, note that, aggregating computing only takes place over beforeIIS.

Definition 19. *For ERS configuration $\langle S, \upsilon_e, \upsilon_{ag} \rangle$ and aggregating operator $\#\omega(0, e_1, g\_attrs, a\_attrs, aggvar)$, the valuation of aggregating operator under configuration, denoted as*
$\langle S, \upsilon_e, \upsilon_{ag} \rangle \vdash \#\omega(0, e_1, g\_attrs, a\_attrs, aggvar) \Rightarrow \upsilon_{ag}[aggvar \mapsto av]$*, is defined as follows:*
*(*AGGOP*)*

$$\frac{ins \vDash_{em} e_1; ([e_1 \mapsto ins], \upsilon_{ag}) \vDash_{ag} aggcon; av = \bar{\lambda}(S, e_1, ins, a\_attrs, \omega)}{\langle S, \upsilon_e, \upsilon_{ag} \rangle \vdash \#\omega(e_1, g\_attrs, a\_attrs, aggcon, aggvar) \Rightarrow \upsilon_{ag}[aggvar \mapsto av]}$$

, where $ins \in \upsilon_e e_1$.

Definition 20. *For ERS configuration $\langle S, \upsilon_e, \upsilon_{ag} \rangle$ and aggregation operator $\#ag(aggops, aggcon)$, where $aggops = aggop; aggops'$, the valuation of aggregation expression is defined as follows:*

$$\frac{(\texttt{AGGCOND} - \texttt{TRUE})}{\langle S, \upsilon_e, \upsilon_{ag} \rangle \vdash aggop \Rightarrow \upsilon'_{ag}; \langle S, \upsilon_e, \upsilon'_{ag} \rangle \vdash aggops' \Rightarrow \upsilon''_{ag}; ([e_1 \mapsto ins], \upsilon''_{ag}) \vDash_{ag} aggcon}{\langle S, \upsilon_e, \upsilon_{ag} \rangle \vdash \ \#ag(aggops, aggcon) \Rightarrow_c \texttt{true}};$$

$$\frac{(\texttt{AGGCOND} - \texttt{FALSE})}{\langle S, \upsilon_e, \upsilon_{ag} \rangle \vdash aggop \Rightarrow \upsilon'_{ag}; \langle S, \upsilon_e, \upsilon'_{ag} \rangle \vdash aggops' \Rightarrow \upsilon''_{ag}; ([e_1 \mapsto ins], \upsilon''_{ag}) \nvDash_{ag} aggcon}{\langle S, \upsilon_e, \upsilon_{ag} \rangle \vdash \ \#ag(aggops, aggcon) \Rightarrow_c \texttt{false}};$$

For ERS, its *configuration* is the basis notion for its execution semantics.

Definition 21. *For events indexed reaction system ers and working structure ws, the configuration of ers is a 2-ary $\langle S, \upsilon_e \rangle$, where $S = (S_{iis}, S_{fCon}, S_{sCon})$, $S_{iis} : \{0,1\} \times \texttt{EVENT} \times \texttt{EVENT} \times \texttt{VTuple} \rightharpoonup \texttt{SEQ}(\texttt{INS})$ is the state of ws representing the instantaneous contents of queues in the indexed instances structures in ws, S_{fCon}/S_{sCon} represents the content of failure/success container at state S, and $\upsilon_e : \texttt{EVENT} \rightharpoonup \texttt{SEQ}(\texttt{INS})$ is the events valuation at state S.*

For given ERS *ers*, the *initial configuration* cg_0 is a configuration with the *initial state* S_0 and *intial event valuation* υ_{e_0}, where $\upsilon_{e_0}(e) = \perp$ for each event e and $S_0(i, ta, tr, vt) = \perp$, for each $i \in \{0,1\}, ta, tr \in \texttt{EVENT}$ and $vt \in \texttt{VTuple}$, $S_0.fCon = \emptyset$, and $S_0.sCon = \emptyset$; its *violated* configurations are the configurations with the *violated state*, which is the state S with $\texttt{size}(S.fCon) > 0$; its *partially compliant* configurations are with the states satisfying $\texttt{size}(S.sCon) > 0$ or $\texttt{size}(S.fCon) = 0$.

Definition 22. *For given ERS ers, its initial configuration cg_0, and trace τ:*

- *(fully compliant)* τ *is fully compliant with respect to ers up to n, denoted as* $\tau_{\leq n} \vdash_f ers$, `iff`, *there exists an executing sequence* $cg_0\tau(0)cg_1\tau(1)cg_2, \cdots, \tau(n-1)cg_n$ *satisfying:*
  - *for each* $0 \leq i \leq n$, *if there exists a rule r, such that, if* $\tau(i) \vDash_{em} r.lhs$, *then* $cg_i \vdash r.rhs \Rightarrow cg_{i+1}$; *otherwise* $cg_i = cg_{i+1}$;
  - *for each* $0 \leq i \leq n$, cg_i *is not a violated configuration.*
- *(partially compliant)* τ *is partially compliant with ers up to n, denoted as* $\tau_{\leq n} \vdash_p ers$, `iff`, *it satisfies the conditions of fully compliant, except for (2),* cg_n *is partially compliant.*
- τ *is fully(partially) compliant with respect to ers, denoted as* $\tau \vdash_f ers$ $(\tau \vdash_p ers)$, `iff`, $\tau_{\leq n} \vdash_f ers$ $(\tau_{\leq n} \vdash_p ers)$ *and* $n = |\tau|$.

From the definition, it is known that, the violated/compliant of the trace is decided by the failure/success container. For better shaping the compliant situation of the trace, it is capable for *ers* to introduce some relevance metric, e.g., compliance ratio $\texttt{size}(S.sCon)/(\texttt{size}(S.sCon) + \texttt{size}(S.fCon))$, however, it is not the target issues of this work and we leave it as future work.

4 Translating from ECL to ERS

In this section, the translation from ECL formula to ERS is determined by two mappings: $\|\cdot\| : \texttt{EMF} \rightarrow \texttt{ERS}$ and $\|\cdot\|_{tmf} : \texttt{TMF} \rightarrow \texttt{ERS}$. However, no matter which types of formula, constructing working structure is essential for their translating.

4.1 Construct working structure

As mentioned above, the structure of IIS is determined by the its *correlating tuples* and so is working structure, where the *correlating tuple* should be extended to include iis symbol i as $(i, target, trigger, corr, b)$, where $i = 0$ or 1. In the following, constructing set of correlating tuples is presented first from given ECL formula, and then the working structure is created from the correlating tuples.

For event $e_1, e_2 \in \texttt{EVENT}$ and constraint $econ \in \texttt{ECON}$, the *correlating sequence* for e_1 and e_2 in *econ*, denoted as $\texttt{corr}(e_1, e_2, econ)$, is a sequence of the attributes of e_1 and e_2 which occur in the *econ*. For example, let *econ* be

$$e_1.user = e_2.user \;\&\&\; e_1.tID = e_2.tID$$

, then $\texttt{corr}(e_1, e_2, econ) =< user, tID >$. Furthermore, if *econ* is empty, then $\texttt{corr}(e_1, e_2, \_) =< attrs >$, where $attrs = \texttt{attr}(e_1) \cap \texttt{attr}(e_2)$. Here, we assume that the correlated attributes would have the same name if two events were correlated. For event $e \in \texttt{EVENT}$, its clone, $\texttt{clone}(e)$, is a new event same to e except its unique identity: $\texttt{clone}(e) \in \texttt{EVENT}$ and $\texttt{clone}(e).id \neq e.id$. Semantically, for instance $ins \in \texttt{INS}$, $ins \vDash_{em} \texttt{clone}(e)$ iff $ins \vDash_{em} e$. The use of $\texttt{clone}()$ operator is in keeping the independency of queues with the same target.

Then following definition is presented for creating correlating tuples from the event matching formula. For the conciseness of exhibition, two frequently used tuples are presented in advance. For correlating constraint *econ*, events e, e_1, and $tc = [t_1, t_2\circ)$, let two correlating tuples t', t'' be:

$$t' = (0, e_1, e, \texttt{corr}(e, e_1, econ), b),$$
$$t'' = (1, e, e_1, \texttt{corr}(e, e_1, econ), b),$$

where $b = t_2 - t_1$ and e, e_1 will be specified in the following definition.

Definition 23. *For given EMF-formula f, the correlating tuples set of f, denoted as $\texttt{crt}(f)$, is defined recursively as follows, where $e = \texttt{tr}(f)$:*

- *if $f = e$ or $\texttt{ors}(e)$, then $\texttt{crt}(f) = \varnothing$;*
- *if $f =$ constr $\texttt{when}(f_1)$, then $\texttt{crt}(f) = \texttt{crt}(f_1)$;*
- *if $f = \texttt{before}(tc, f_1, e, econ)$, then $\texttt{crt}(f) = \texttt{crt}(f_1) \cup T$, where*

$$T = \begin{cases} \{t'\}, & \text{if } f \text{ is not over-lapped;} \\ \{t'\} \cup \{t''\}, & \text{o.w.} \end{cases} \tag{6}$$

  ,where $e_1 = \texttt{tr}(f_1)$;
- *if $f = \texttt{after}(tc, e, f_1, econ)$, then $\texttt{crt}(f) = \texttt{crt}(f_1) \cup \{t''\}$, where $e_1 = \texttt{tr}(f_1)$.*
- *if $f = \texttt{beforeSince}(tc, f_2, f_1, e, econ)$, then*

$$\texttt{crt}(f) = \texttt{crt}(\texttt{before}(tc, f_2, e, econ)) \cup \{t'\} \tag{7}$$

  , where $e_1 = \texttt{tr\_un}(f_1)$;
- *if $f = \texttt{afterUntil}(tc, e, f_1, f_2, econ)$, then $\texttt{crt}(f) = \texttt{crt}(\texttt{after}(tc, e, f_2, econ)) \cup \{t'\} \cup \{t''\}$,where $e_1 = \texttt{tr\_un}(f_1)$;*
- *if f is an aggregation formula, let $f = \texttt{aggregate}(aggexp_1, aggexp_2)\texttt{with}\ aggcon$, and*

$$aggexp_1 = \omega_1(tc_1, f_1, g\_attrs_1, a\_attrs_1, aggcon_1)\ \texttt{to}\ aggvar_1,$$
$$aggexp_2 = \omega_2(tc_2, f_2, g\_attrs_2, a\_attrs_2, aggcon_2)\ \texttt{to}\ aggvar_2,$$

  then $\texttt{crt}(f) = \texttt{crt}(f_1\ \&\ f_2) \cup \{t_1, t_2\}$, where

$$t_1 = (0, e_{a_1}, e', g\_attrs_1, b_1),$$
$$t_2 = (0, e_{a_2}, e', g\_attrs_2, b_2) \tag{8}$$

  , where $e' = \texttt{tr}(f_1)$, b_i the bound of tc_i, and e_{a_i} is new event with $e_{a_i} = \texttt{clone}(e')$ for $i \in \{1, 2\}$.
- *if $f = !\ f_1$, then $\texttt{crt}(f) = \texttt{crt}(f_1)$.*
- *if $f = f_1$ op f_2, then $\texttt{crt}(f) = \texttt{crt}(f_1) \cup \texttt{crt}(f_2) \cup \{t\}$, where*

$$t = (1, e, e_f, attrs, \_) \tag{9}$$

  ,where e_f is a new event with $e_f = \texttt{clone}(e)$, $attrs = \texttt{attr}(e)$, and op $\in \{\&, |, \rightarrow\}$.

In the definition, creating *correlating tuple* means that, a new indexed queue will be added into working structure to store target instances for the correlated trigger. For the base cases, e or $\texttt{ors}(e)$, it is not necessary to store any instances. For the *before*-type formula, as specified in Eq.(6), if f is not overlapped, then f_1 is its *before*-type subformula and if f_1 was satisfied then the target of f, also the trigger and the only decider of f_1, needed to be stored for the trigger of f, which is exactly covered by t'; otherwise, whether the desired target of f, also the trigger of f_1, was occurred or not would be decided by the decider of f_1 which possibly occurs before or after the trigger

of f, then both t' and t'' are needed to cover these two cases. For the *beforeSince*-type formula, the Eq.(7) describes, the correlating tuples of f include the tuples from its *before*-type formula and also the new tuple t' which corresponds to the unmatched instances w.r.t. trigger of f_1. For the *aggregation*-type fomrula, with two *aggexp* as in the definition, the $\texttt{crt}(f)$ should firstly include $\texttt{crt}(f_1 \& f_2)$ to enable the generated working structure to cover the rule systems for f_1 and f_2. In addition, for supporting the aggregating operators ω_1 and ω_2, t_1 and t_2 are needed to describe the independent queues for storing the triggers of f_1 and f_2 respectively, where the grouping attributes are used as correlating sequences as specified in Eq.(8). For the compositional EMF-formula, besides including the correlating sequences set for its subformulas, the $\texttt{crt}(f)$ also contains the new tuple t as Eq.(9), which characterizes an independent queue in afterIIS for storing the trigger of f and also for communicating the satisfiable situations for the subformulas.

Based on the correlating tuples, the working structure for event matching formula can be constructed as follows.

Definition 24. *For correlating tuples set crt, the working structure from crt, denoted as* $\texttt{ws}(crt)$*, is constructed as:*

- *(fContainers/sContainters) sCon=fCon=∅ .*
- *(beforeIIS/afterIIS) for the IIS of* $\texttt{ws}(crt)$*, where* $i \in \{0,1\}$*:*
  - *(Target)* $Target = \{e \mid for\ each\ t = (i, e, e', corr, b) \in crt\}$.
  - *(Trigger)* $Trigger = \{e' \mid for\ each\ t = (i, e, e', corr, b) \in crt\}$.
  - *(VStructure)* $VS = \{(corr, b, \_) \mid for\ each\ t = (i, e, e', corr, b) \in crt)\}$*, where '_' represents the empty mapping for* θ.
  - *(δ)* $\delta(e)(e') = (corr, b, \_)$*, if there exists a tuple* $(i, e, e', corr, b) \in crt\}$.

Furthermore, for two working structure ws_1 *and* ws_2*, then the merging of* ws_1 *and* ws_2*, denoted as* $ws_1 \uplus_{ws} ws_2$*, is* $ws = (beforeIIS, afterIIS, fCon, sCon)$ *where:*

- $fCon = ws_1.fCon$ *and* $fCon = ws_1.fCon$.
- *(IIS) let* $iis \in \{beforeIIS, afterIIS\}$ *and it is created as follows*$(i \in \{0,1\})$*:*
  - $Target = iis_1.Target \cup iis_2.Target$.
  - $Trigger = iis_1.Trigger \cup iis_2.Trigger$.
  - $VS = iis_1.VS \cup iis_2.VS$.
  - $\delta(e)(e') = (corr, b)$*, if* $(corr, b) = iis_1.\delta(e)(e')$ *or* $(corr, b) = iis_2.\delta(e)(e')$ *for each* $e \in iis.Target$ *and* $e' \in iis.Trigger$.

For given EMF-formula f, its working structure $\texttt{ws}(\texttt{crt}(f))$ will be abbreviated as $\texttt{ws}(f)$ for simplicity. Then based on above definitions, the working structure for TMF-formula could be created as following definition.

Definition 25. *For given TMF-formula f, its working structure, denoted as* $\texttt{ws}_{\texttt{tmf}}(f)$*, is constructed based on the types of f as follows:*

- $\texttt{ws}_{\texttt{tmf}}(f) = \texttt{ws}(f_1)$*, if* $f = \texttt{always}(f_1)$.
- $\texttt{ws}_{\texttt{tmf}}(f) = \texttt{ws}(f_1) \uplus_{ws} \texttt{ws}(\{t\})$*, if* $f = \texttt{exists}(f_1)$*, where*

$$t = (1, e_1, end, \texttt{corr}(e_1, end, \_), 1) \tag{10}$$

, and $e_1 = \texttt{tr}(f_1)$.

- $\texttt{ws}_{\texttt{tmf}}(f) = \texttt{ws}(f_1)$, *if* $f = !f_1$.
- $\texttt{ws}_{\texttt{tmf}}(f) = \texttt{ws}(f_1) \uplus_{ws} \texttt{ws}(f_2)$, *if* $f = f_1$ *op* f_2, *where* $op \in \{\ \&\&, \|\ \}$.

In the definition, correlating tuple t as Eq.(10) is used to specify the related queue for storing the trigger instances which will be assessed to check whether f_1 was satisfied before the end of trace.

Note that, in the above when adding new correlating tuple, there might be the case of two correlating tuples *intersecting* with each other, which corresponds to two tuples t_1 and t_2,

$$t_1 = (i, e_{11}, e_{12}, < attr_1 >, b_1)$$
$$t_2 = (i, e_{21}, e_{22}, < attr_2 >, b_2),$$

with $e_{11} = e_{21}$ and $e_{12} = e_{22}$. For such *intersecting* case, there would result in sharing memory for different rules in the rule system, however, if there exists some conflicted actions from different reactions over such sharing memory, e.g., writing and deleting operators, then different execution orders of these actions would result in inconsistency outcomes. For such issue, it would be solvable by extending current δ mapping and/or using `clone` operator to replicate the target event.

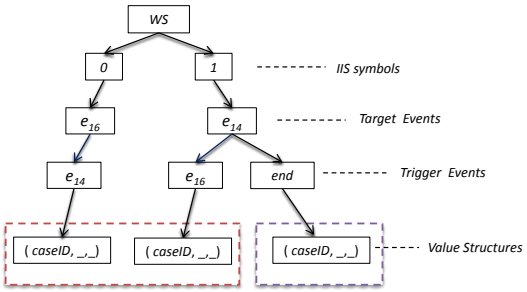

Fig. 3. The working structure for formula f in Eq.(11).

Example 4. Considering the TMF-formula

$$\begin{aligned} f_1 = \texttt{always}(\ \texttt{afterUntil}(\ \_,\ e_{14},\ \texttt{ors}(e_{16}),\ end, \\ e_{14}.caseID = \texttt{ors}(e_{16}).caseID\ \&\& \\ e_{14}.caseID = end.caseID\ )\ ) \end{aligned} \tag{11}$$

, where events defintions are presented in following evaluation section, it specifies that for each running instances, after each e_{14} instances, e_{16} instances are not allowed to occur. Its working structure is depicted as Fig. 3, where leaf nodes in left dashed part are created based on the Def.23 for *afterUntil*-type formula, and the right dashed part follows the defintion for *after*-type formula. Note that, in the tree, one path from root to the leaf node corresponds to one correlating tuple, and for the leaf node $(caseID, \_, \_)$, *caseID* is the correlating attribute and the first "_" represents no boundness for the queue.

4.2 Translating to ERS

In this section, the rules system for the ECL formula will be presented recursively based on the structure of formula.

Before diving into the translation, some operator over rules are needed. For rules set R, a rule r, an event e, events set E, operations ops, and operation op, op_1, the needed rule operators are defined as follows:

- (*rule selection*) $R \dagger (e)$: select from R the rule with e as its left hand.
  $R \dagger (e) = \begin{cases} r, & \text{if } \exists\ r \in R, \text{ s.t., } r.lhs = e, \\ \varnothing, & \text{o.w.} \end{cases}$
- (*rule deleting*) $R \backslash_r (e)$: delete from R the rule with e as its left hand.
  $R \backslash_r (e) = \begin{cases} R \backslash \{r\}, & \text{if } \exists\ r \in R, \text{ s.t., } r.lhs = e, \\ \varnothing, & \text{o.w.} \end{cases}$
- (*rules merging in chain*) $R \uplus_c (r)$: add rule r into set R, if there was the other rule r' in R sharing the same left hand of r, then right hand of r was chained to right hand of r'.
  $R \uplus_c (r) = \begin{cases} (R \backslash \{r'\}) \cup \{r''\}, & \text{if } r' = R \dagger (r.lhs), \\ & \text{where } r''.lhs = r'.lhs \text{ and} \\ & \ r''.rhs = r'.rhs \ \cdot \ r.rhs \\ R \cup \{r\}, & \text{o.w.} \end{cases}$
- (*rules merging in sequence*) $R \uplus_s \{r\}$: add rule r into set R, if there was the other rule r' in R sharing the same left hand of r, then right hand of r was sequentially appended to right hand of r'.
  $R \uplus_s \{r\} = \begin{cases} (R \backslash \{r'\}) \cup \{r''\}, & \text{if } r' = R \dagger (r.lhs), \\ & \text{where } r''.lhs = r'.lhs \text{ and} \\ & \ r''.rhs = r'.rhs \ ; \ r.rhs \\ R \cup \{r\}, & \text{o.w.} \end{cases}$
- (*rule renaming*) $R[e : op_1 \twoheadrightarrow ops]$: replace in R with ops for all the op_1 occurred in the right hand of e rule. Also, it is possibly extended to the events set case $R[E : op_1 \twoheadrightarrow ops]$.
- (*operation deleting*) $R \backslash_o \{op\}$: delete operation op occurred in rules of R.
  $R \backslash_o \{op\} = R[E : op \twoheadrightarrow \#next]$, where $E = \{\ r.lhs \mid r.rhs \cap\ op \neq \emptyset, r \in R\}$.
- (*rule concreting for ors*) $R \Downarrow_c E$: for all the rules in R with ors event as left hand, replace the left hand of these rules for each events from E which are not the base for the ors event.
  $R \Downarrow_c E = \{r \mid r.lhs \in E \backslash \{e\}\ and\ r.rhs = r'.rhs,\ for\ each\ r' \in R\ with\ r'.lhs = \texttt{ors}(e)\}$.

Note that, for rules concerting operator, if there was no rules in R with `ors` event as left hand, then the result would be empty set.

Based on above rule operators, the translating for given EMF formula is presented in the following.

Definition 26. *For given events set E and event matching formula f, the mapping $\|\_\| : EMF \rightarrow ERS$ is the translation from EMF formula to ERS, where the working structure of $\|f\|$ is $\texttt{ws}(f)$ and the rules system (E, OPS, R) with the rules set $R = (R' \uplus_c (R' \Downarrow_c E)) \backslash \{ors\}$, where R' is constructed recursively. Let $e_1 \in \texttt{tr}(f_1)$, $e_2 \in \texttt{tr}(f_2)$, and $e' \in \texttt{tr\_un}(f_1)$, then:*

– *if* $f = e$, *then*

$$R' = \{\ e \rightarrow \#succ(1, \_, \_, e),\ \texttt{ors}(e) \rightarrow \#fail(1, \_, e, \texttt{ors}(e))\}.$$

– *if* $f = \texttt{ors}(e)$, *then*

$$R' = \{\ e \rightarrow \#fail(1, \_, \texttt{ors}(e), e),\ \texttt{ors}(e) \rightarrow \#succ(1, \_, \_, \texttt{ors}(e))\}.$$

– *if* $f = constr\ \texttt{when} f_1$, *then*

$$\begin{aligned} R' &= (\|f_1\|.R)\backslash_o\{\#fail\}[\texttt{de}(f_1) : \#succ \twoheadrightarrow ops)]; \\ ops &= \#eval(constr)?\#succ(6, \_, \_, e_1) : \#fail(6, \_, \_, e_1). \end{aligned}$$

– *if* $f = \texttt{before}(tc, f_1, e, econ)$, *then*

$$R' = R^+ \uplus_c R^*,$$

where R^+ *and* R^* *are defined based on following cases and time interval* $\overline{tc}$ *is the contrary of* tc*:*

- f *is non-overlapped,*

$$\begin{aligned} R^+ &= \{e \rightarrow \#ge(0, e_1, e, tc)?\#succ(2, \_, e_1, e) : \#fail(2, \_, e_1, e)\} \\ R^* &= (\|f_1\|.R)[\texttt{de}(f_1) : \#succ \twoheadrightarrow \#w(0, e_1, e))]\backslash_o\{\#fail\}; \end{aligned} \tag{12}$$

- f *is overlapped,*

$$\begin{aligned} R^+ &= \{\ e \rightarrow \#ge(0, e_1, e, tc)?\#succ(2, \_, e_1, e) : \#w(1, e, e_1), \\ &\qquad end \rightarrow !\#empty(1, e, e_1)?fail(2, \_, e_1, e)\ \} \\ R^* &= (\|f_1\|.R)[\texttt{de}(f_1) : \#succ \twoheadrightarrow ops]\backslash_o\{\#fail\}; \\ ops &= \#ge(1, e, e_1, \overline{tc})?\#d(1, e, e_1) \cdot \#succ(2, \_, e_1, e) : \#w(0, e_1, e) \end{aligned} \tag{13}$$

– *if* $f = \texttt{after}(tc, e, f_1, econ)$, *then*

$$R' = R^+ \uplus_c R^*,$$

,where

$$\begin{aligned} R^+ &= \{\ e \rightarrow \#w(1, e, e_1), \\ &\qquad end \rightarrow !\#empty(1, e, e_1)?\#fail(3, \_, e, e_1)\ \} \\ R^* &= (\|f_1\|.R)[\texttt{de}(f_1) : \#succ \twoheadrightarrow ops)\backslash_o\{\#fail\}. \\ ops &= \#ge(1, e, e_1, tc)?\#d(1, e, e_1) \cdot \#succ(3, \_, e, e_1). \end{aligned}$$

– *if* $f = \texttt{beforeSince}(tc, f_2, f_1, e, econ)$ *and* f_1 *is non-overlapped, then*

$$R' = R^+ \uplus_c (R^*\backslash_o\{\#succ\})$$

, where

$$\begin{aligned} R^+ &= (\|\texttt{before}(tc, f_2, e, econ)\|.R)[e : \#succ \twoheadrightarrow ops]; \\ R^* &= (\|f_1\|.R)[e' : \#fail \twoheadrightarrow \#w(0, e', e)] \\ ops &= con\ ?\ \#fail(4, e_2, e', e) : \#succ(4, e_2, e_1, e) \\ con &= \#ge(0, e', e, tc)\&\&\#tcm(e', e_2) > 0 \end{aligned}$$

– *if* $f = \texttt{afterUntil}(tc, e, f_1, f_2, econ)$ *and* f_1 *is non-overlapped, then*

$$R' = \bar{R} \uplus_c R^+ \uplus_c (R^*\backslash_o\{\#succ\})$$

, where

$$\begin{aligned} R^+ &= \{\ e \rightarrow \#w(1,e,e')\} \\ R^* &= (\|f_1\|.R)[e' : \#fail \twoheadrightarrow ops_1] \\ \bar{R} &= (\|\texttt{after}(tc,e,f_2,econ)\|.R)[\texttt{de}(f_2) : \#succ \twoheadrightarrow ops_2] \\ ops_1 &= \#ge(1,e,e',tc)?\#d(1,e,e') \cdot \#w(0,e',e) \\ ops_2 &= \#ge(0,e',e,\overline{tc})\ ?\ \#fail(4,e,e',e_2) : \#succ(4,e,e_1,e_2) \end{aligned} \tag{14}$$

– *if* $f = !f_1$*, then*

$$R' = \|f_1\|.R[\texttt{de}(f_1) : \#succ \twoheadrightarrow \#fail, \#fail \twoheadrightarrow \#succ].$$

– *if* $f = f_1\ \&\ f_2$*, then*

$$R' = R_1^* \uplus_c R_2^*$$

, where

$$\begin{aligned} R_1^* &= (\|f_1\|)[\texttt{de}(f_1) : \#succ \twoheadrightarrow ops] \\ R_2^* &= (\|f_2\|)[\texttt{de}(f_2) : \#succ \twoheadrightarrow ops] \\ ops &= \#ge(1,e_f,e,\_)?\#d(1,e_f,e) \cdot \#succ(7,\_,\_,e) : \#w(1,e_f,e). \end{aligned}$$

– *if* $f = f_1 \mid f_2$*, then*

$$R' = R^+ \uplus_c R_1^* \uplus_c R_2^*$$

, where

$$\begin{aligned} R^+ &= \{\ e \rightarrow \#w(1,e_f,e) \\ &\qquad end \rightarrow\ !\#empty(1,e_f,e)?\#fail(7,\_,\_,e)\} \\ R_1^* &= (\|f_1\|)\backslash_o\{\#fail\}[\texttt{de}(f_1) : \#succ \twoheadrightarrow ops] \\ R_2^* &= (\|f_2\|)\backslash_o\{\#fail\}[\texttt{de}(f_2) : \#succ \twoheadrightarrow ops] \\ ops &= \#ge(1,e_f,e,\_)?\#d(1,e_f,e) \cdot \#succ(7,\_,\_,e) \end{aligned} \tag{15}$$

From above definition, it shall be seen that, the integral part of translation is creating rule system, which should follow the semantics of ECL formula. Basically, for given formula f, creating its rule system is to specify, how to decide at decisional time points the compliant/violation state of the rule system, after instances of triggers from $\texttt{tr}(f)$ and $\texttt{tr\_un}(f)$ occur, where decisional time points refer to the arrivals of its deciders instances.

For *before*-type formula f, the translation is devided into two cases based on whether f is of overlapped or not. If f is not overlapped, then f_1 would be non-overlapped *before* subformula of f, which means the decider of f_1, also its trigger, is requried to occur before e. Thereby, as specified by e rule in R^+ of Eq.(12), at time point of e instance arriving, it is decidable for e regarding to the compliant/violation of f by checking whether desired e_1 instance was occurred before. However, if f is overlapped, its rule system is more complicated, since the decider e can not make full decision when the decider instance of f_1 occurs after certain e instance. Thereby, as specified in Eq.(13), e rule copes with the case of desired e_1 instance as occurred before, *end* rule is to the case of no desired e_1 instances occurred, and operations *ops* in rules of f_1 deciders deal with the case of desired e_1 instance occurred but decided at time points of f_1 deciders, whose instances might occur after certain e instance. For *afterUntil* formula, firstly, the trigger e instance should be writted into the afterIIS to require desired outcomes

happened after. $\bar{R}$ is used to decide whether e instance is followed by desired e_2 at time points of f_2 deciders arriving. If the desired e_2 occurred for e instance, operations ops_2 would decide for such instance the compliant/violation of f, by checking whether after the e instance there is e' instance occurrence, which is corresponding to violation of f_1 and determined by R^*. For the disjunction of two EMF formulas, one of rule systems for these formulas needs to be in compliant state for trigger instances. Thereby, as specified in Eq.(15): firstly, the trigger e instance is written into afterIIS to require one of rule systems for subformulas to be compliant; then two R^* devote to decide the compliant situation for each subformulas, and if one of them is compliant, the rule system for the disjunction is compliant and otherwise, *end* rule would be triggered later to invoke the fail operation.

Following definition is the translation for aggregation formula.

Definition 27. *For aggregation formula* $f = \texttt{aggregate}(aggexp_1, aggexp_2)$ `with` *aggcon and let*

$$\begin{aligned} aggexp_1 &= \omega_1(f_1, g\_attrs_1, a\_attrs_1, aggcon_1) \texttt{ to } aggvar_1, \\ aggexp_2 &= \omega_2(f_2, g\_attrs_2, a\_attrs_2, aggcon_2) \texttt{ to } aggvar_2, \end{aligned}$$

where f_1 *and* f_2 *are of non-overlapped before type formulas, then the translation* $\|\_\| : EMF \to ERS$ *for aggregation formula* f *is* $\|f\|$*, where* $\texttt{ws}(f)$ *is its working structure and its the rules system is* (E, OPS, R) *with*

$$\begin{aligned} E &= \|f_1\|.E \cup \|f_2\|.E, \\ R &= (\|f_1\ \&f_2\ \|.R)[e : \#succ \twoheadrightarrow c\_ops]\backslash_o\{\#fail\} \end{aligned}$$

, *where* $e = \texttt{tr}(f_1)$ *and*

$$\begin{aligned} c\_ops &= w\_ops \cdot \#ag(op_1.op_2, aggcon)\ ?\ \#succ(5, \_, \_, e) : \#fail(5, \_, \_, e) \\ w\_ops &= \#w(0, e_{a_1}, e) \cdot \#w(0, e_{a_2}, e) \\ op_1 &= \#\omega_1(0, e_{a_1}, g\_attrs_1, a\_attrs_1, aggcon_1, aggvar_1) \\ op_2 &= \#\omega_2(0, e_{a_2}, g\_attrs_2, a\_attrs_2, aggcon_2, aggvar_2) \end{aligned} \tag{16}$$

From the definition, it shall be seen that, before invoking aggregation operations, rule system for the conjunction of f_1 and f_2 should be in compliant state which is decided by e instance, i.e. the trigger and also decider for f_1 and f_2; then as specified in Eq.(16), the instance is written sequentially into two queues and also two aggregation operations are invoked to do the aggregation over related data queues; finally, the success/fail operation is invoked based on the *aggcon* result evaluated by aggregation data.

Based on the translation for EMF formula, following is devoted to the TMF formula.

Definition 28. *For given events set* E *and TMF* f*, the mapping* $\|\cdot\|_{tmf} : TMF \to ERS$ *is the translation from TMF formula to ERS, where the working structure of* $\|f\|_{tmf}$ *is* $\texttt{ws}_{\texttt{tmf}}(f)$ *and its rules system is* (E, OPS, R)*. Let* $e_1 \in \texttt{tr}(f_1)$*,* $e_2 \in \texttt{tr}(f_2)$*, and* $e' \in \texttt{tr\_un}(f_1)$ *), then its rule set* R *is constructed as follows:*

- *if* $f = \texttt{always}f_1$*, then* $R = \|f_1\|.R$*;*
- *if* $f = \texttt{exists}f_1$*, then*

$$R = R^* \uplus_c R^+,$$

where

$$R^* = (\|f_1\|.R)[\texttt{de}(f_1) : \#succ \twoheadrightarrow \#w(0, e_1, end)]\backslash_o\{\#fail\}$$
$$R^+ = \{\ end \rightarrow \#ge(1, e_1, end, \_)\ ?\ \#succ(8, \_, \_, e_1)\ :\ \#fail(8, \_, \_, e_1)\}$$

- *if* $f =$ $\texttt{!exists}(f_1)$, *then* $R = \|\texttt{always}(\texttt{!}f_1)\|_{tmf}.R$;
- *if* $f =$ $\texttt{!always}(f_1)$, *then* $R = \|\texttt{exists}(\texttt{!}f_1)\|_{tmf}.R$;
- *if* $f = f_1 \&\& f_2$, *then* $R = \|f_1\|_{tmf}.R \uplus_s \|f_2\|_{tmf}.R$;
- *if* $f = \texttt{exists}(f_1) \parallel \texttt{exists}(f_2)$, *then*

$$R = R_1^* \uplus_s R_2^*$$

, *where*

$$\begin{aligned} R_1^* &= (\|\texttt{exists}(f_1)\|_{tmf}.R)[end : \#fail \twoheadrightarrow ops] \\ R_2^* &= (\|\texttt{exists}(f_2)\|_{tmf}.R)\backslash_r\{end\} \\ ops &= (\|\texttt{exists}(f_2)\|_{tmf}.R) \dagger \{end\}. \end{aligned} \tag{17}$$

For the *always*-type formula, its rule system is the same to the rule system of its EMF counterpart, whereas, *end* rule is freqently used in the rule system for *exists*-type formula, since the eventuality of *exists*. For the disjunction of two *exists* formulas, as specified in Eq.(17), R_1^* catchs that, if rule system for first *exists* formula is in compliant, so is the whole rule system, otherwise, the result of whole rule sytem is determined by the rule system of second *exists* formula.

Example 5. Considering following two formulas,

$$\begin{aligned} f_2 = \texttt{always}(\ \texttt{afterUntil}(\ \_,\ e_{15},\ \texttt{ors}(e_{16}),\ end, \\ e_{15}.caseID = \texttt{ors}(e_{16}).caseID\ \&\& \\ e_{15}.caseID = end.caseID\ )\ ) \\ f_{11} = f_1\ \&\&\ f_2 \end{aligned} \tag{18}$$

, where f_{11} is ECL formula for rule R11 presented in use case and f_1 is the formula in Eq.(11), then following Def.28, $\|f_{11}\|_{tmf}$ is the merging of $\|f_1\|_{tmf}$ and $\|f_2\|_{tmf}$, which are generated respectively as Eq.(14) in the Def.26. And the rules set of $\|f_{11}\|_{tmf}$ is presented as follows:

```
ERS :

  e14 -> #w(1, e14, e16) . #w(1, e14, end)
  e15 -> #w(1, e15, e16) . #w(1, e15, end)
  e16 -> ( #ge(1, e15, e16) ? #d(1, e15, e16). #w(0, e16, e15) )
       ; ( #ge(1, e14, e16) ? #d(1, e14, e16). #w(0, e16, e14) )

 end -> ( #ge(1, e15, end)  ? ( !#ge(0, e16, e15)  ?
       #succ(4, e15, ors(e16), end) : #failure(4, e15, e16, end) ))
      ; ( #ge(1, e14, end) ? ( !#ge(0, e16, e14 ) ?
       #succ(4, e14, ors(e16), end) : #failure(4, e14, e16, end) ))
```

For formula f_{11}, its triggers include e_{14} and e_{15}, and *end* is its only decider. When e_{14} instance occurs, it is written as a fact into two places: one is for e_{16} and the other is for *end*. When e_{16} instance occurs, rule system will access e_{14} to check whether related e_{14} instance has already occurred or not. If occurred, e_{16} instance will be saved. And then, when *end* instance arrives, rule system will check the existances of e_{14} instance and related e_{16} instance. Then based on the checking result, the compliant/violation of rule system can be determined through success and fail operation respectively. For e_{15}, the operation process is similar.

4.3 The soundness of translation

Now, it is the time to consider the soundness of the above translation. As known from above, the working structure plays essentially fundamental roles in the interpretation and execution of ERS. Then it is necessary to make sure firstly the *coverable* relation between working structure and rules system.

Lemma 1. *For given TMF-formula f, the rule system of* $\|f\|_{tmf}$ *is covered by the working structure* $\mathtt{ws}_{\mathtt{tmf}}(f)$.

The proof can be completed by comparing the definitions for creating working structure and rule systems based on the type of formula.

Semantically, TMF formula is to specify the properties of running trace, while the EMF formula is for the properties of certain time-point. Based on Lemma 1, following lemma is presented for the relation between EMF formula and its corresponded rule system generated by $\|\_\|$. Also, without loss of generality, we assume that each running traces are ended with *end* instance.

Lemma 2. *Let* $\|\cdot\| : EMF \rightarrow ERS$ *be the translating mapping, then for EMF formula f and trace* τ*, and integer* $i > 0$*, the follows hold:*
(1) *if* $\tau(i) \not\models f$ *, then:*

- $\tau_{\leq i} \not\vdash_f \|f\|$*, if f is non-overlapped before-type formula.*
- $\tau \not\vdash_f \|f\|$*, o.w.*

(2) $if \tau \not\vdash_f \|f\|$*, then* $\tau(i) \not\models f$ *for some* $i < |\tau|$.

Proof. proof for statement(1) by the induction on formula structure.

- (base)$f = e$. if $\tau(i) \not\models f$, then $\tau(i)$ does not match e, then within the rule system of $\|f\|$, the rule for $\mathtt{ors}(e)$ would be triggered and make $\|f\|$ transit to *violated state*, then $\tau_{\leq i} \not\vdash_f \|f\|$. It also true for $f = \mathtt{ors}(e)$ by similar proof.
- $f =$ *constr* $\mathtt{when} f_1$. if $\tau(i) \not\models f$, then by the semantics of *when*-type formula, $\tau(i) \models f_1$, $\tau(i) \models e'$, and $[e' \mapsto \tau(i)] \not\models_c$ *constr*. Then $\tau \vdash_p \|f_1\|$ and then based on the rules system of $\|f\|$, $\tau \not\vdash_f \|f\|$.
- $f = \mathtt{before}(tc, f_1, e, econ)$. if $\tau(i) \not\models f$, then $\tau(i) \models e$, and let

$$Trig = \{\ \tau(j) \mid \langle \tau(j), \tau(i) \rangle \models_t tc,\ and\ [e' \mapsto \tau_j, e \mapsto \tau_i] \models_{ec} econ\}, \tag{19}$$

  , where $e' = \mathtt{tr}(f_1)$, then for each $\tau(j) \in Trig$, $\tau(j) \not\models f_1$. Following, we should prove that, for all such $\tau(j)$, it can lead $\|f_1\|$ to *viloated state* based on the type of f_1.
  - f_1 is *non-overlapped* before-type formula. By induction, $\tau_{\leq j} \not\vdash_f \|f_1\|$. Suppose that, τ_j does not lead $\|f_1\|$ to the violated state. Then, since $\tau_j \models e'$, then it would fire and make the $\|f_1\|$ reach the *success* state, based on the rules in $\|f_1\|$. It means, there would be desired event instances occurred before. Hence, $\tau(j) \models f_1$. It is a contradiction. Then all such $\tau(j)$ would lead $\|f_1\|$ to violated states. Following the rules of $\|f\|$, it can be inferred that, the $\tau_{\leq i}$ could lead $\|f\|$ to violated state. Then $\tau_{\leq i} \not\vdash_f \|f\|$.

- f_1 is an *after*-type formula. By induction, $\tau \nvDash f_1$ is true. Suppose that, there exists one $\tau(j) \in Trig$ which does not lead $\|f_1\|$ to the violated state. Since $\tau(j) \vDash e'$, then based on rules in R^+ for *after* type formula, $\tau(j)$ would be stored and when *end* instance occurred, the $\tau(j)$ would already be deleted by the related rules in R^*. It means that, after $\tau(j)$, there would be would be some desired instance occurred and then for such $\tau(j)$ $\tau(j) \vDash f_1$. It is a contraction. Then for each such $\tau(j)$, there do not exist desired instances occurred after. Based on rules of $\|f\|$, τ would lead $\|f\|$ to violated state, i.e., $\tau \nvDash_f \|f\|$.
- f_1 is an *overlapped* before-type formula. By induction, $\tau \nvDash f_1$ is true. Similar to above cases, it is true that, for each $\tau(j) \in Trig$ there is a sub-trace τ' with $\tau(j) \in \tau'$ which leads $\|f_1\|$ to violated state. Considering the rules of $\|f\|$, it can be inferred that, the sub-trace after combing event e's instance would violate $\|f\|$, i.e., $\tau \nvDash_f \|f\|$.

– $f = \texttt{after}(tc, e, f_1, econ)$. if $\tau(i) \nvDash f$, then $\tau(i) \vDash e$, and let

$$Resp = \{\ \tau(j) \mid \langle \tau(i), \tau(j) \rangle \vDash_t tc,\ and\ [e' \mapsto \tau_j, e \mapsto \tau_i] \vDash_{ec} econ\} \tag{20}$$

, then for all $\tau(j) \in Resp$, $\tau(j) \nvDash f_1$ and then $\tau(j) \vDash e'$. By induction, $\tau(j) \nvDash_f f_1$ is true. From the fact of $\tau(j) \vDash e'$, then $\tau(j)$ fire $\|f_1\|$ and then sub-trace $\tau_{\geq j}$ would lead it to violated state, otherwise there would be a contradiction as shown above. Based on rules of $\|f\|$, then sub-trace $\tau_{\geq i}$ would violate $\|f\|$.

– Case 4. $f = \texttt{beforeSince}(tc, f_2, f_1, e, econ)$. if $\tau(i) \nvDash f$, then $\tau(i) \vDash e$, and let

$$Trig = \{\ \tau(j) \mid \langle \tau(j), \tau(i) \rangle \vDash_t tc,\ and\ \tau(j) \vDash f_2\}.$$

Then, for all the $\tau(j) \in Trig$, there exists at least one $t(k)$ with $j < k < i$ and $[e' \mapsto \tau(j), \bar{e} \mapsto \tau(k), e \mapsto \tau(i)] \vDash_{ec} econ$, but $\tau(k) \nvDash f_1$.

- if $Trig = \emptyset$. Then for all $\tau(j)$ with $\langle \tau(j), \tau(i) \rangle \vDash_t tc$, $\tau(j) \nvDash f_2$ would be true. then by induction, $\tau_{\leq j} \nvDash f_2$. From Case 2, $\tau_{\leq i} \nvDash \texttt{before}(tc, f_2, e, econ')$, where $econ'$ is the part of $econ$ only related to e' and e. Then based on the rules of $\|f\|$, $\tau_{\leq i} \nvDash \|f\|$.
- if $Trig \neq \emptyset$. By the induction on $\tau(k) \nvDash f_1$, $\tau_{\leq k} \nvDash \|f_1\|$, then based on the rules of $\|f\|$. the writing operator would be executed and then following the rule of $r(e)$ in $\|f\|$, the condition would be true and then the violated state is reached, then $\tau_{\leq i} \nvDash \|f\|$.

– Case 5. $f = \texttt{afterUntil}(tc, ef_1, f_2, e, econ)$. if $\tau(i) \nvDash f$, then $\tau(i) \vDash e$, and let

$$Resp = \{\ \tau(j) \mid \langle \tau(i), \tau(j) \rangle \vDash_t tc,\ and\ \tau(j) \vDash f_2\}.$$

Then, for all the $\tau(j) \in Resp$, there exists at least one $t(k)$ with $i < k < j$ and $[e' \mapsto \tau(j), \bar{e} \mapsto \tau(k), e \mapsto \tau(i)] \vDash_{ec} econ$, but $\tau(k) \nvDash f_1$.

- if $Resp = \emptyset$. Then for all $\tau(j)$ with $\langle \tau(j), \tau(i) \rangle \vDash_t tc$, $\tau(j) \nvDash f_2$ would be true. then by induction, $\tau_{\leq j} \nvDash f_2$. From Case 3, $\tau_{\leq i} \nvDash \texttt{after}(tc, e, f_2, econ')$, where $econ'$ is the part of $econ$ only related to e' and e. Then based on the rules of $\|f\|$, $\tau_{\geq i} \nvDash \|f\|$.
- if $Resp \neq \emptyset$. By the induction on $\tau(k) \nvDash f_1$, $\tau_{\geq i} \nvDash \|f_1\|$, then based on the rules of $\|f\|$. the deleting and writing operator would both be executed and then following the rule of $r(e')$ in $\|f_2\|$, the condition would be false and then the violated state is reached, then $\tau_{\geq i} \nvDash_f \|f\|$.

– Case 6. $f = !f_1$. if $\tau(i) \nvDash f$, then $\tau(i) \vDash f_1$, then $\tau \vdash_p f_1$. And then $\tau \nvDash_f !f_1$.

– Case 7 $f = f_1 \&\& f_2$. if $\tau(i) \not\models f$, then $\tau(i) \not\models f_1$ or $\tau(i) \not\models f_1$. Then by induction, $\tau(i) \not\vdash_f f_1$ or $\tau(i) \not\vdash_f f_1$. No matter which cases, based on rules of $\|f\|$, it is correct for $\tau \not\vdash_f \|f\|$.

Proof for statement(2) by induction on formula structure. If $\tau \not\vdash_f \|f\|$, based on the rule system for $\|f\|$:

– (*base*) $f = e$. There exists integer $i < |\tau|$, $\tau(i) \vdash \mathtt{tr}_{\mathtt{un}}(f)$. Then $\tau(i) \not\models f$. Similar proof for $f = \mathtt{ors}(e)$.
– (*when*) $f = constr$ `when` f_1. Then there exist integer i and trace τ', s.t., $\tau(i) \models e_1$, $\tau(i) \in \tau'$, $\tau' \vdash_f \|f_1\|$, and $\tau(i) \not\models f_1$. Based on statement(1), $\tau(i) \models f_1$. Thereby, following the semantics of *when* formula, then $\tau(i) \not\models f$.
– (*before*) $f = \mathtt{before}(tc, f_1, e, econ)$. Then there would be integer $i < \|\tau\|$, s.t., $\tau(i)$ makes $\|f\|$ transit to *violated* state. For each $\tau(j) \in Trig$ as Eq.19, we should prove $\tau(j) \not\models f_1$ by types of f.
  - f is *non-overlapped* formula. Then f_1 would be of *before* type. Based on the rule for event e in $\|f\|$, it can be inferred that, for each $\tau(j) \in Trig$, there exist partial trace of τ, τ', such that, $\tau(j) \in \tau'$, $\tau(j) \models e'$, and $\tau' \not\vdash_f f_1$. Then by induction, $\tau(i) \not\models f_1$. Then for $\tau(i)$, each $\tau(j) \in Trig$ satisfies $\tau(j) \not\models f_1$, thereby, $\tau(i) \not\models f$.
  - f_1 is *after* type formula. Based on the rule for *end* event, there would at least exist one e instance, say $\tau(i)$, which does not have desired instance in $Trig$. Then for each $\tau(j) \in Trig$, based on the rules of $\|f_1\|$, there exists τ partial trace τ', s.t., $\tau(j) \models e'$, $\tau(j) \in \tau'$, and $\tau' \not\vdash_f f_1$, then by induction, $\tau(j) \not\models f_1$. Then $\tau(i) \not\models f$.
  - f_1 is overlapped before formula. similar to after type.
– (*after*)$f = \mathtt{after}(tc, e, f_1, econ)$. Then from the rule for *end* event, it is inferred that, there exists integer $i \leq |\tau|$, s.t., $\tau(i) \models e$, and also the set *Resp* as Eq.20, such that, for each $\tau(j) \in Resp$, it can not trigger $\|f_1\|$ to transit to the *success* state in τ. If $Resp = \emptyset$, then for $\tau(i)$, it does not have desired $\tau(j)$ for f_1. By the semantics of *after* type formula, $\tau(i) \not\models f$. On the other hand, if $Resp \neq \emptyset$, then for each $\tau(j) \in Resp$, there would exist τ's partial sub-trace τ', s.t., $\tau(j) \in \tau'$ and $\tau' \not\vdash_f f_1$. By induction, it can be inferred that, $\tau(j) \not\models f_1$. Also by the semantics of formula, $\tau(i) \not\models f$.
– (*beforeSince*)$f = \mathtt{beforeSince}(tc, f_2, f_1, e, econ)$, where f_1 is non-overlapped *before* emf formula and let $f' = \mathtt{before}(tc, f_2, e, econ')$. Since $\tau \not\vdash f$. Then the *fail* operation would be invoked to execute at least one time. Based on rule system of $\|f\|$, it could be known, there are two places enable invoking *fail* operations.
  - Case 1. *fail* in $\|f'\|$. Then $\tau \vdash_f \|f'\|$. By the induction, there exists $\tau(i)$, s.t., $\tau(i) \not\models f'$. Then $\tau(i) \not\models f$.
  - Case 2. *fail* in *ops* operations. Then it infers that, there exists event e instance $\tau(i)$ triggering $\|f'\|$ to *success* state, and also event e' instance $\tau(k)$ with $k < i$ which triggers $\|f_1\|$ transit to *violated* state. Then for $\tau(i)$, since it makes $\|f'\|$ transit to success state, it means, there exists τ partial sub-trace τ' and $\tau' \vdash_f f'$. By statement 1, then there exists event e_2 instance $\tau(j)$ with $\tau(j) \models f'$ and also $j < k$ from the *con* condition in rule system. Furthermore, for $\tau(k)$, there would exist τ''s partial sub-trace τ'', such that, $\tau'' \not\vdash f_1$. By induction, $\tau(k) \not\models f_1$ is true. Combing all these, $\tau(i) \not\models f$ is true.

- (*afterUnitl*) f = `afterUnitl`$(tc, e, f_1, f_2, econ)$. where f_1 is non-overlapped *after* emf formula and let f' = `after`$(tc, e, f_2, econ')$. Since $\tau \nvdash f$. Then the *fail* operation would be invoked to execute at least one time. Similar to *sinceBefore*, it is also true that, there exists integer $i < |\tau|$, s.t., $\tau(i) \nvDash f$.
- (!)f =!f_1.if $\tau \nvdash_f \|f\|$, then based on the rule system of $\|f\|$, by exchanging the *succ* and *fail* operations in rule system of $\|f_1\|$, it infers that, there would exists τ's partial sub-trace τ', s.t., $\tau' \vdash \|_f f_1\|$. Then by statement 1, there would be integer $i < |\tau'|$, such that $\tau'(i) \vDash f_1$. Then $\tau'(i) \nvDash f$ in τ'. Thereby $\tau'(i) \nvDash f$ in trace τ is true since τ' is the partial sub trace of τ.
- (&) $f = f_1 \& f_2$. If $\tau \nvdash_f f$, then based on the rule system of $\|f\|$, there would exist *tau*'s partial sub-trace τ', such that, event e instance, say $\tau(i)$, with $\tau(i) \in \tau'$ and for $\tau(i)$, $\tau' \nvdash_f f_1$ or $\tau' \nvdash_f f_2$. Let $\tau' \nvdash_f f_1$, then by induction, $\tau(i) \nvDash f_1$. Then $\tau(i) \nvDash f$ is true.
- (||) $f = f_1 || f_2$. If $\tau \nvdash_f f$, then also based on rule system of $\|f\|$, there exists event e_f instance $\tau(i)$, the clone of e, which triggers the *fail* operation. Then there would exist τ partial sub-trace τ', s.t., $\tau(i) \in \tau'$, $\tau' \nvdash_f \|f_1\|$ and $\tau' \nvdash_f \|f_2\|$. By the induction on τ', $\tau(i) \nvDash f_1$ and $\tau(i) \nvDash f_2$ would both be true. Then $\tau(i) \nvDash f$ for τ, since τ' is the partial sub trace of τ.
- (*aggregation*)f = `aggregate`$(aggexp_1, aggexp_2)$ `with` *aggcon*. If $\tau \nvdash_f \|f\|$, then from the rule system, there exists event e instance $\tau(i)$ and τ's partial sub-trace τ', s.t., $\tau(i) \in \tau'$ and $\tau' \vdash_f \|f_1 \& f_2\|$ for $\tau(i)$, which infers $\tau(i) \vDash f_1$ and $\tau(i) \vDash f_2$. Then after $\tau(i)$ was written into the working structure and two aggregating operations were executed, there would be aggregating valuation v_{ag}, such that, $v_{ag} \nvDash_{ag} aggcon$. Then combing these, it infers that, $\tau(i) \nvDash f$.

Within the lemma, statement(1) specifies the soundness for rule system and its EMF formula. For non-overlapped before formula, its violation could be timely reported by the rule system; however for overlapped formula, it would be delayed to end time point for the violation report, if there is not time constraint for the future. Furthermore, statement(2) assures the completeness for EMF formula and its rule system. I

Based on Lemma 2, it is assured for the relation between TMF formula and its corresponded rule system generated by $\|\_\|_{tmf}$ by following theorem.

Theorem 1. *Let* $\|\cdot\|_{tmf}$: *TMF* $\rightarrow$ *ERS be the translating mapping, then for TMF-formula f and trace* τ*, it holds that*
(1) *if f is of* `always` *type, then* $\tau \vDash f$ iff $\tau \vdash_f \|f\|$;
(2) *if f is of* `exists` *type, then* $\tau \vDash f$ iff $\tau \vdash_p \|f\|$.

Proof. By induction on the TMF formula structure.

- for *always*-type formula.
  - (base) f = `always`f_1, where f_1 is EMF formula.
    (=>) if $\tau \vDash f$, then for each integer $0 \leq i < |\tau|$, $\tau(i) \vDash f_1$. Then by Lemma 2, $\tau \vdash_f \|f_1\|$. Based on the definition for creating $\|f\|_{tmf}$, it is true for $\|f\|_{tmf} = \|f_1\|$, then $\tau \vdash_f \|f_1\|_{tmf}$.
    (<=) if $\tau \vdash_f \|f\|_{tmf}$, then $\tau \vdash_f \|f_1\|$. By Lemma 2, then for each integer $i < |\tau|$, $\tau(i) \vDash f_1$. Then $\tau \vDash f$ is true.
  - (&&) $f = f_1 \&\& f_2$, where f_1, f_2 are the *always*-type formula.
    By the induction, it is true for that, $\tau \vDash f_i$ iff $\tau \vdash_f \|f_i\|_{tmf}$ for $i \in \{1, 2\}$. Then if $\tau \vDash f$, then $\tau \vdash_f \|f_1\|_{tmf}$ and $\tau \vdash_f \|f_2\|_{tmf}$. Then based on the

assumption of no *intersecting* between $\texttt{ws}(f_1)$ and $\texttt{ws}(f_2)$, it is true for $\|f\|_{tmf} = \|f_1\|_{tmf} \uplus_r \|f_2\|_{tmf}$. Thereby, it is true,$\tau \vDash f$ iff $\tau \vdash_f \|f\|_{tmf}$.

- (!) $f = !\texttt{exists}(f_1)$,
  if $\tau \vDash f$, then based on the definition, for each integer $0 \leq i < |\tau|$, $\tau(i) \vDash !f_1$, then $\tau \vDash \texttt{always}(!f_1)$. Then based on above prove, it is true for that, $\tau \vDash \texttt{always}(!f_1)$ iff $\tau \vdash_f \|\texttt{always}(!f_1)\|_{emf}$.

– for *exists*-type formula.

- (base)$f = \texttt{exists} f_1$.
  (=>) for trace τ, if $\tau \vDash f$, then there exists integer $i < |\tau|$, s.t., $\tau(i) \vDash f_1$, then based on Lemma 2, $\tau \vdash_p \|f_1\|$. It means that, there is at least one time of $\#succ$ is invoked. Based on the rule specified in R^* and R^+ of $\|f\|_{tmf}$, the $succ(8, \_, \_, e_1)$ operation would be invoked and executed, which means τ could lead $\|f\|_{tmf}$ to partial compliant state, i.e., $\tau \vdash_p \|f\|_{tmf}$.
  (<=) if $\tau \vdash_p \|f\|_{tmf}$, then from the rule system, $\tau \vdash_p \|f_1\|_{tmf}$. then there exists i, s.t., $\tau(i) \vDash f_1$. Then obviously, $\tau \vDash f$.
- $(||)f = \texttt{exists}(f_1)||\texttt{exists}(f_2)$.
  (=>) if $\tau \vDash f$, then by the definition, $\tau \vDash \texttt{exists}(f_1)$ or $\tau \vDash \texttt{exists}(f_2)$. By the induction, $\tau \vdash_p \|(f_i)\|_{tmf}$, for $i \in \{1, 2\}$. If $\tau \vdash_p \|(f_1)\|_{tmf}$, Then by the rules in R_1^* of $\|f\|_{tmf}$, $\tau \vdash_p \|(f)\|_{tmf}$. Else if $\tau \vdash_p \|(f_2)\|_{tmf}$, then based on the rule for *end* in R_1^* of $\|f\|_{tmf}$, when *end* instance occurred, if *ops* operation would be invoked, then based on the reactions of $\|f_2\|_{tmf}$, it would be true for $\tau \vdash_p \|f\|_{tmf}$.
- (&&) $f = f_1 \&\& f_2$, where f_1, f_2 are the *exists*-type formula.
  By the induction, it is true for that, $\tau \vDash f_i$ iff $\tau \vdash_f \|f_i\|_{tmf}$ for $i \in \{1, 2\}$. Then if $\tau \vDash f$, then $\tau \vdash_p \|f_1\|_{tmf}$ and $\tau \vdash_p \|f_2\|_{tmf}$. Then based on the assumption of no *intersecting* between $\texttt{ws}(f_1)$ and $\texttt{ws}(f_2)$, it is true for $\|f\|_{tmf} = \|f_1\|_{tmf} \uplus_r \|f_2\|_{tmf}$. Thereby, it is true,$\tau \vDash f$ iff $\tau \vdash_p \|f\|_{tmf}$.
- (!) $f = !\texttt{always}(f_1)$,
  if $\tau \vDash f$, then based on the definition, for there exists an integer $0 \leq i < |\tau|$, $\tau(i) \vDash !f_1$, then $\tau \vDash \texttt{exists}(!f_1)$. Then based on above prove, it is true for that, $\tau \vDash \texttt{exists}(!f_1)$ iff $\tau \vdash_p \|\texttt{exists}(!f_1)\|_{emf}$.

Note that, to make the theorem conciseness, *always* type is used to refers to *always* formulas and their composition by connectives, as well as the negation of *exists* formula; *exists* type is similar and include the negation of *always* formula. In the theorem, statement(1) assures the soundness and completeness for *always* type formula and its rule system from the translation; and statement(2) presents such relations for *exists* type formula and its rules system based on partial compliant sense.

5 Implementation

The bpCMon was implemented in Java. Here, the structure of bpCMon-monitor is presented as well as basic monitoring procedure. Also, two algorithms are introduced for the aggregation: reuse-based and statistic B+tree based.

5.1 The implementation of bpCMon-monitor

The basic structure of bpCMon-monitor consists of three parts, *working interface*, *working structure*, and *rules system*:

- *working interface*: acts as data reader, in charge of reading, filtering and matching the outside data into the inner system. It is implemented as Java interface `WorkingInterface`, including the essential methods:
  - `Instance next()`: read the instance from the data source;
  - `Event matchToEvent(Instance inst)`: match the instance to event;
  - `void init()` and `void close()`: initialize and close the working interface.
- *working structure*: it is implemented as Java class, which include two IIS and containers as well as methods to deal with container. For the IIS, it is also implemented as Java class including an essential map and the methods for operations:
  - `deltaMap : Map < Integer, Map < Integer, ValueStructure >>`: corresponds to the δ in IIS definition.
  - `void add(Instance inst, Integer trID)`: add the instance into the right place of value structure by vtuple and related event ids;
  - `Instance get(Integer taID, Instance trigger, Constraint constr, TimeConstraint tc)`: get desired taID instance for trigger.
  - `List < Instance > getAll(Integer taID, Instance trigger, Constraint constr, TimeConstraint tc)`: get all the desired taID instances for trigger.
  - `void delete(Integer taID, Instance trigger, Constraint constr, TimeConstraint tc)`: delete the desired taID instance for trigger.
  - `boolean isEmpty(Integer taID, Instance trigger)`: check whether the queue of taID for trigger is empty or not.
  - `void aggregate(String op, Instance trigger, AttrSEQ group, AttrSEQ aggre, Constraint aggcon, String aggVar, Assigment assign)`: add trigger into the related queue and then do the `op` aggregation over all the `aggre` values for each group, and finally save the new value referred by `aggvar` in the assignment for further using, where `Assignment` is a class corresponding to events evaluation in ERS configuration.
- *rules system*: it is also implemented as Java class including methods for manipulating rules:
  - `Reactions getReaction(Integer trID)`: get the reaction for trID, where `Reactions` is a interface including method `void doReaction(WorkingStructure ws, Assignment assign)` to execute the reaction.
  - `void replace(Integer trID, int sign, Operation op)`: replace the final operator, i.e., *#succ/#fail* in the reaction of trID with op, where final operator is specified by sign;
  - `RuleMap rulesJoin(RuleMap other)`: merging two rule systems, wherein the sequential operator is used to connect reactions with the same trigger.

Obviously, the real implementation is far complicate than the above. But it is enough for presenting following basic monitoring procedure of bpCMon-monitor.

Listing 1.3. Basic monitoring procedure

```
1  PROCEDURE ers_monitoring( ers, workingInterface)
2  Input: ers, workingInterface
3  begin
4      var inst, event, assign;
5      var reaction ;
6      while (true) {
```

```
7           inst = workingInterface.next() ;
8           event = workingInerface.matchToEvent(inst) ;
9           if (read == null) continue ;
10          reaction = ers.rs.getReaction(event) ;
11          if ( rhs != null ) {
12              assign = new Assignment() ;
13              assign.add(inst) ;
14              reaction.doReaction(ers.workingStructure, assign) ;
15              if (!ers.workingStructure.fCon.isEmpty()) {
16                  workingStructure.responseF() ;
17              }
18              if (!ers.workingStructure.sCon.isEmpty()) {
19                  workingStructure.responseS() ;
20              }
21          }
22          if ( event == endOfRunning ) break ;
23      }
24
25  end
```

As described in the Listing 1.3, bpCMon-monitor do the reactions for each matched instance until the event of end running is read. For each time of reaction, an assignment is created and also added the just matched instance for further use. The main use of assignment is temporally storing the intermediate results generated from operation executions, e.g. $\#ge$ operation, during each time of reaction. After the reaction, in line 15-19, ers monitor responses the compliance situations by checking two containers in the working structure. Note that, if the bpCMon monitor was used to analysis the logs, the codes of line 15-19 would be moved out of while-body, that means, the compliance result would be delayed to be reported for the convenience and also time efficiency.

5.2 The implementation for aggregation

To implement the aggregation for bpCMon, one core issue needs to be considered, how to efficiently get the statistics over time-bounded queue. Note that, the target instances have already been grouped into related queues based on the their grouping values. The prominent property of time-bounded queue is dynamics, as new data would be added as well as out-of-date data be deleted. As for such issue, the straightforward algorithm is intuitive and of linear cost with the number of data in the queue, but it is still too expensive since there might exist tons of rounds for computing the statistics in aggregation monitoring. Therefore, two different methods are proposed to address the issue: reusing-based and tree-based.

reusing-based. The method makes use of the fact: the data would stay in the queue until it was out of date and during its stay, it would continually contribute to the statistics for each rounds. Based on the fact, to get current statistic value, it is not necessary to recompute statistics over the queue but update previous value by considering only the data to be out of date and just new added. The method is abstractly presented as Listing 1.4, where: for the first **if**, line13-15, the trigger is checked whether it is relevant for the statistics of current queue, if it is, the previous statistic value *stValue* would be updated by this new added instance; otherwise, the trigger would be just used to evaluate whether the existed data is out of date or not; the **while** part is used

to update the *stValue* for the old data. Comparing to recompute the *stValue* for each time, reusing based one would be more efficient.

Listing 1.4. Reusing-based aggregating computing

```
1  PROCEDURE aggregate_reusing()
2  Input: op ,      // aggregation operator
3         trigger,  // new added instance
4         group,    // grouping attributes
5         aggre,    // aggregating attributes
6         aggcon,   // constraint for data selecting
7         stValue.  // previous statistics value
8
9  Output: stValue
10 begin
11     var inst, value, it;
12
13     if (queue.isRelated(group, trigger) && aggcon.valueOf(trigger)) {
14         value = inst.getValue(aggre) ;
15         updateByNew(op, stValue, value) ;
16     }
17     inst = queue.getLast() ;
18     while (queue.isOutOfDate(inst, trigger)) {
19         if ( aggcon.valueOf(inst) ) {
20             value = inst.getValue(aggre) ;
21             aggop.updateByOld(op, stValue, value);
22         }
23         queue.removeLast() ;
24         inst = queue.getLast() ;
25     }
26     return stValue ;
27 end
```

However, the precondition for using reusing-based algorithm is that, the relation between the instances and the constraint *aggcon* is *unchanged*, which means, for the instance in the queue, if it satisfied *aggcon* in current round, then it would keep satisfying the constraint until it was out of date. For example, the constraint, $e_1.amount \geq 1000$, is of such sort. However, such precondition is not always true. Taking the constraint $e_1.amount \geq 2 * av$ as example, where av is the average amount for given period, considering that, now a new instance with big *amount* is added into the queue, obviously the av would become greater, then in the queue some instances, which used to satisfy the constraint, would not be able to satisfy it again. Such *changeability* would be possibly occurred if the constraint *aggcon* includes aggregating variables. For such case, the reusing-based method would not suitable.

tree-based. To address above changeability, we link the queue with a variant of B+tree, named as *statistics annotated* B+tree, abbreviated as *st-tree*. The node type of st-tree is defined as listing 1.5.

Listing 1.5. node type of st-tree

```
1  Node
2  {
3      keys: K[] ;      //  keys in the node
```

```
4       left: Node ;      //  left sibling
5       right : Node ;    //  right sibling
6       parent : Node ;   //  parent of node
7       statistics : V ;  // statistics value
8       keyAmount : int ;  // the amount of keys
9       size : int ;      // size of node
10  }
11  InnerNode extends Node
12  {
13      children : Node[] ; // the children of the node
14  }
15  LeafNode extends Node
16  {
17      values : V[] ; // the occurring numbers for keys
18  }
```

where, K is aggregating values domain for the instances in the queue; V is the st-value domain. Within the queue, all the aggregating values of the instances are orderly stored as *key* in the leaf nodes of the st-tree as well as their occurring numbers as *values* since aggregating values might equal for different instances. For the inner node, it stores the ordered keys which act as the index for searching the key in the leaf, and also includes statistics value for all the keys stored its descendant leaves. When new instance is added, its aggregating value is obtained and then as new key inserted into tree. During the traverse for finding the right place to insert, the st-values of traversed node are updated by this new key. On the other hand, when out of date instance is to be deleted from the queue, the aggregating value as key needs also to be deleted from the tree. Similar to insert key, the st-values for all traversed nodes are also updated by the key. Besides, the overflow and underflow of nodes need to be deal with for keys adding and deleting. The basic operators of st-tree are listed as follows:

- `insert(key)`: when new instance is added into the queue. the value of aggregating property of new instance, as `key`, is insert into the linked st-tree. Then the method traverses the tree from root to leaf through ordered nodes, meanwhile, updates the statistics of each traversed nodes by `key`, and then inserts the `key` to right position of right leaf node and also updates the occurring value of this key.
- `delete(key)`: when some instance in the queue is to be deleted since out of date, the `key` w.r.t the instance should also be deleted at same time. The method first traverse the tree to find the right leaf node, meanwhile, updates the statistics of each traversed nodes by `key`, and then deletes the `key` in the leaf node and reduce the occurring value of `key`.
- `dealwithOverflow(node)`: after key is inserted, the size of the node might be equal to `SIZE`, then the method will split the node to two nodes and add the mid key up to the parent as standardly processing in B+tree. However, for st-tree, the statistics of splitted node should also be updated and splitted.
- `dealwithUnderflow(node)`: after key is deleted, the size of the node might be less than the half of `SIZE`, then the method will borrow to the node a key and child from its sibling, if one of its sibling has enough keys, i.e., greater than the half of `SIZE`. If the node surrounds with poor siblings, the method would merge the node with one of its sibling. During borrowing or merging, the statistics of related nodes should also be updated.

Note that, for given number of instances in queue n, the time complexity of all above methods are $\mathcal{O}(\log n)$. Then by these methods, the contents of queue and the linked st-tree are synchronized with reasonable cost.

Based on the st-tree, computing the st-value for desired instances in the queue, is referred to selecting and combing st-values from related nodes in the tree. Here, we assume that, the aggregating constraint can be specified as key interval form, [k1, k2), similar to time interval. The *tree-based* aggregating algorithm is presented as List 1.6, which consists of two steps: (1) the recursive step, line 17, is to compute the st-value for the keys with greater than k1; (2) the **for** part, line 18-25, is to computing the st-value for inner nodes with keys less than k2. For the base case of the algorithm, it is corresponding to the leaf node of tree and the st-value is obtained based on the keys with greater than k1 as well as their occurring number, i.e., *statisticsUpdateForLeaf*() in the algorithm. Note that, it is the only place for computing st-value from the keys and also the time cost for such computing is a constant with greater than the size of node. For the inner nodes, the computing is updating current st-value by its child value, which does not need to compute but just get.

Listing 1.6. st-tree based aggregating computing

```
1  PROCEDURE aggregate_tree(node, [k1, k2))
2  Input: node,     // get the stValue for the node
3         [k1, k2). // key interval
4  Output: stValue.
5  begin
6      var stValue=0, value, index;
7
8      if (node.getNodeType() == Node.LEAFNODE) {
9          for (int i=0; i<node.keyAmount; i++) {
10             if (node.getKey(i)>=k1 && node.getKey(i)< k2)
11                 statisticsUpdateForLeaf( stValue,
12                        node.getValue(i), node.getKey(i)) ;
13         }
14     } else {
15         for (index=0; index<node.keyAmount; index++)
16             if (node.getKey(index)>= k1) break ;
17         stValue = aggregate_tree(node.getChild(index), [k1,k2)) ;
18         for (int i=index+1; i<node.keyAmount+1; i++) {
19             if (node.getKey(i) >= k2){
20                  break ;
21             } else {
22                 value = node.getChild(i).getStatsitics() ;
23                 statisticsUpdateForInner(stValue, value) ;
24             }
25         }
26     }
27     return stValue ;
28 end
```

Therefore, the process for the computing can be described briefly as: based on the key interval, traverse from root to the desired leaf node, and then in the leaf node, compute the initial st-value from desired keys and their occurring number; then recursively update current st-value by the child nodes and return updated st-value to parent node

until root node is reached. Hence, the cost for computing st-value is determined by $2 \times h \times size \times update$, where h is the height of tree, *size* is the size of node, and *update* is the cost for given values updating. Let the number of instances in the queue is n, then the time complexity for aggregating computing by *tree-based* algorithm is $\mathcal{O}(\log n)$.

6 Evaluations

The functionals of bpCMon framework will be firstly evaluated by 10 CMFs [20] which are the qualitative and suggestive requirements for the compliance monitoring framework. And then, to evaluate the expressiveness of language, ECL is used to specify various perspectives of compliance rule patterns. For the applicability of bpCMon, the real hospital logs is adopted and analysed by bpCMon. To evaluate the performance of bpCMon monitor, it is compared with three other facilities over the benchmark from [21]. Finally, bpCMon monitor is also evaluated for its aggregation functional by the synthesized test case[7]. Note that, if without specifying, the tests are performed on Luna version of Eclipse IDE with jdk-1.8.0_40 in laptop with win7 64-bit OS, Intel(R) i5 CPU 2.4G, and 8G RAM.

6.1 The functionals of bpCMon

The compliance monitoring functionals(CMFs) are the suggestive requirements for the compliance monitor framework, and they are divided into three categories: modeling requirements, execution requirements, and user requirements. Regarding to these requirements, the situation of bpCMon is presented as follows:

- CMF 1. *constraints referring to time*. It requires that the modeling language of monitoring framework should support qualitative and quantitative temporal orders. In bpCMon, ECL satisfies such requirement thanks to the events relation patterns and time interval defined in ECL.
- CMF 2. *constraints referring to data*. In this CMF, the data is distinguished into unary data conditions and extended conditions, where unary data conditions refers to constraints involving just one data object and extended conditions relating multiple data objects at same time. In ECL, there are several places embodying its capability for supporting the unary/extended data conditions, i.e., event constraints in event definition, events correlating constraints, and aggregating constraints.
- CMF 3. *constraints referring to resources*. Although resources constraints could be considered as a special case of data constraints, this prospective is still promoted as an independent standard since its important and prevailed in BPM. It requires that the modeling language should associate resources(e.g., agent, role, organization)with activities, events or something. Also, ECL supports this requirement thanks to its data capability.
- CMF 4. *supporting non-atomic activites*. The non-atomic activity here means the activity has time duration. Basically, to support such activities, there are two ways, explicit or implicit, wherein, the explicit way makes use of atomic life-cycles events of activities, e.g., start, ready, suspended, aborted, and completed events of activity;

[7] The testing data, including logs, benchmarks, and rules specifying, are available at: https://github.com/PingFair/bpCMon

while implicit one associate activities with their time duration. Obviously, this CMF can be implemented in explicit way in bpCMon.

- CMF 5. *supporting lifecycles activites*. This CMF requires that the monitor framework should has correlating mechanism to link multi atomic events to one same activity. Also, this requirement can be supported by events correlation constraints in ECL.
- CMF 6. *supporting multiple-instances constraints*. It requires that the framework should support the case of multiple activations for compliance rules in one trace. In bpCMon, for each ECL formula, once its trigger was occurred, it would be activated immediately.
- CMF 7. *reactively detect and manage violations*. For bpCMon monitor, it is designed in reactive way and also capable of continuous monitoring even violation occurred.
- CMF 8. *pro-actively detect and manage violations*. Comparing to CMF 7, this CMF requires totally different monitoring mechanism. Obviously, bpCMon now only support CMF 7.
- CMF 9 and 10. *ablity for root cause of violation* and *quantifying compliance*. In bpCMon, once the success/violation was occurred, its information, including root causes for the results, would be created based on success/failure table and then stored in the related containers. Thereby, the way is paved for further root cause analysis and quantifying the compliance situation.

Table 2. Compliance patterns in ECL

Categories	Patterns	ECL description
Occurrence	Existence of A	$\texttt{exists}\ A$
	Absence of A	$\texttt{!exists}\ A$
	Limit A to N	$\texttt{always}(\texttt{before}(\_, \cdots, \texttt{before}(\_, A, A, \_), \cdots, A) \rightarrow \texttt{afterUntil}(\_, \texttt{A}, \texttt{ors}(\texttt{A}), \texttt{end}, \_))$
	A requires B	$\texttt{exists}\ A \rightarrow \texttt{exists}\ B$
	A coexists B	$(\texttt{exists}\ A \rightarrow \texttt{exists}\ B)\&\&(\texttt{exists}\ B \rightarrow \texttt{exists}\ A)$
	A mutex B	$(\texttt{exists}\ A \rightarrow \texttt{!exists}\ B)\&\&(\texttt{exists}\ B \rightarrow \texttt{!exists}\ A)$
	Choose A or B	$\texttt{exists}\ A \ \texttt{||}\ \texttt{exists}\ B$
Order	A followed by B	$\texttt{after}(\_, A, B, \_)$
	A precedes B	$\texttt{before}(\_, B, A, \_)$
	A block B	$\texttt{afterUntil}(\_, A, \texttt{ors}(B), \texttt{end}, \_)$
	A block B until C	$\texttt{afterUntil}(\_, A, \texttt{ors}(B), C, \_)$
	(A_1,$\cdots$, A_n) chainLeadsTo (B_1,$\cdots$, B_m)	$\texttt{before}(\_, (\cdots \texttt{before}(\_, A_1, A_2, \_) \cdots), A_n, \_) \rightarrow \texttt{after}(\_, A_n, \texttt{after}(\_, B_1, (\cdots \texttt{after}(\_, B_{m-1}, B_m, \_) \cdots), \_), \_)$
	(A_1,$\cdots$, A_n) chainPrecedes (B_1,$\cdots$, B_m)	$\texttt{after}(\_, B_1, (\cdots (\texttt{after}(\_, B_{m-1}, B_m, \_) \cdots), \_)) \rightarrow \texttt{before}(\_, (\cdots \texttt{before}(\_, A_1, A_2, \_) \cdots, A_n), B_1, \_))$
Resource	P PerformedBy r	$P.role = r \texttt{when}\ P$
	P SegregatedFrom Q	$(P.role \neq Q.role)\texttt{when}(\texttt{before}(\_, Q, P, \_) \mid \texttt{after}(\_, P, Q, \_))$
	P BondedWit Q	$(P.role = Q.role)\texttt{when}(\texttt{before}(\_, Q, P, \_) \mid \texttt{after}(\_, P, Q, \_))$
Time	P leasTo Q within k	$\texttt{after}([0, k), P, Q, \_)$
	P leadsTo Q AtleastAfter k	$\texttt{after}([k, \infty), P, Q, \_)$
	P precedes Q AtleastAfter k	$\texttt{before}([k, \infty), P, Q, \_)$

In addition, to evaluate the expressiveness of ECL, in Table2, we use ECL to specify the compliance patterns which cover most part of patterns list proposed in [20][23] except some advanced but rare occurred patterns. Note that, from the definition of ECL, its expressive capability is far beyond the expressing of above patterns, since other powerful properties has not yet used, e.g., event constraints, event correlating, aggregating

operators, and the nesting of operators, and also the events relations patterns set is extensible if needed.

6.2 Running on BPIC 2011 logs

To evaluate the applicability of bpCMon, as mentioned before, the test case is adopted which includes the real hospital logs as well as 16 compliance rules of various perspectives. Following is the fragment of ECL specification for R11:

```
//events part

e14 = ( 14, 'natrium vlamfotometrisch',
            [ caseID, 'Age'>=71, 'Treatment code'>=803,
             'Diagnosis Treatment Combination ID'<=394725 ] ) ;
e15 = ( 15,'natrium vlamfotometrisch',
           [ caseID, 'Treatment code'=803 || =703] ) ;
e16 = ( 16, 'calcium', [caseID] ) ;
// policy part
R11 = R11_1 && R11_2 ;
R11_1 = always( afterUntil(_, e14 ,ors(e16), endOfCase,
                  e14.caseID=ors(e16).caseID
                  && e14.caseID =endOfCase.caseID ));
R11_2 = always( afterUntil(_, e15 ,ors(e16), endOfCase,
                    e15.caseID=ors(e16).caseID
                    && e15.caseID =endOfCase.caseID ));
```

Where, R11 is specified as the composition of two formula `R11_1` and `R11_2` by TMF operator &&. Note that, the "`caseID`" in the above in fact is "`concept:name`", the trace-scope identity property of traces in the logs. `endOfCase` is the event representing the end of one trace.

To read the data from the logs, a class `XESWorkingInterface`, which implements the interface `WorkingInterface`, is developed based on the OpenXES libary[8]. The `XESWorkingInterface` is in charge of generating interested event instances by parsing, selecting, and merging related event scope and trace-scope attributes values. These instances also include instances of `endOfCase` and `endOfLogs` for the end of trace and logs.

After the rules are specified in ECL formula, based on the type of formula, relevant translator creates the ERS for each formula and then these ERS are merged into one by merging operators for working structure and rules system, since among them there is no *intersecting* issues, i.e., *intersecting* of different working structures. Note that, although it seems that the *intersecting* is different to the *conflictness* among policies, there might exist some relation between them and we leave it as future work.

The evaluation is consists of two phrases: at first phrase, running the ERS monitor over the logs 100 times, where for each running, the whole ERS monitor is regenerated by merging each formula's ERS in randomly order; at second phrase, running the ERS monitor for another five times with fixed order of merging. In the first phrase test, the compliance results as well as #viloation are the same for 100 times running, which exactly prove the fact of independent for the working structures of these formulas. From

[8] http://code.deckfour.org/xes/

Table 3, it is known that, among 16 rules, the logs is compliant with five rules, R4, R7, R13, R15, R16, and for other rules, there are various violations. For the violation, the ERS monitor provides useful feedbacks as following two samples:

BEFORE-VIOLATE-TYPE for policy 1: no event 5 is happened before event instance(caseid=00000388, eID= 1, name= administratief tarief - eerste pol, pId= null, at=01:00:00 09/05/2005) in given time with given condition !

AFTER-VIOLATE-TYPE for policy 3: no event 13 is happened after event instance(caseid=00000830, eID= 4, name= aanname laboratoriumonderzoek, at=00:00:00 03/03/2008) in given time with given condition !

For the running cost, it mainly consists of two parts: for ERS monitor running and for OpenXES caching all the event instances. From Table 4, the number of involved event patterns is 21 and the total of violations is 4937, and for the running cost, ERS is of practical efficient. Note that, the symbol "/" in the table is used to delimit the cost for ERS monitor (at the front) and OpenXES file caching (at the behind).

As for the performances of ERS monitor, the event features and reaction rules length would be the main influencing factors. Event features here refer to the event structure property and the *sub-event* relation among events. If the event consists of complex attributes constraints or there exist couples of events with sub-event relation, then the cost of event matching as well as reaction would be increased. In this test case, there are three pairs of events with sub-event relation. As for the memory cost of ERS monitor, it might be related to the working manner of OpenXES: loading all the data from logs file into the memory and then the data available for use. After data loaded, there is the overhead of memory for ERS ranged from 8% to 20%.

Table 3. Violations for incompliant rules

rule	#violation
R1	805
R2	1
R3	30
R5	593
R6	1833
R8	1
R9	798
R11	8
R12	45
R14	823

Table 4. Performance of bpCMon monitor for the BPIC2011 hospital logs

rules	#event	#viol.	time(sec)	memory(mb)
R1-R16	21	4937	5.39/5.49	378.3/350.20
			4.936/5.492	326.8/294.59
			5.143/5.406	377.53/335.19
			5.361/5.648	293.69/240.18
			5.370/5.471	383.51/328.45

6.3 The efficiency of bpCMon-Monitor

In this subsection, the efficiency of bpCMon monitor will be evaluated by two test cases: one is adopted from [21], wherein, the compliance rules and logs are concerning with resources granting and releasing for the tasks of planetary rover; the other is from banking setting and created based on [10].

TestCase1. (*Release*) A resource granted to a task should eventually be released by that task ;
(*NoRelease*) A resource can only be released by a task, if it has been granted to that task, and not yet released;

(*NoGrant*) As long as a resource is granted to a task, it can not be granted, neither to that task nor to any other task.

Within these rules, there are mainly two events, $grant(s, t, r)$ and $release(s, t, r)$, which represent "task t is granted/released resource r at timestamp s" respectively. The testing logs is created exactly following the same procedure in [21]. The logs creating is controlled by three parameters (G, L, R): the head and tail part are the *grant* instances with the same number controlled by G, and the middle part is consisted of L groups of R pairs of *release* and *grant* instances. Furthermore, the created logs has instances number determined by $2 \times (G + L \times R)$ and also satisfies that, if $G \geq R$, the logs would be compliant, otherwise, the violation number is determined by $2 \times (R - G)$.

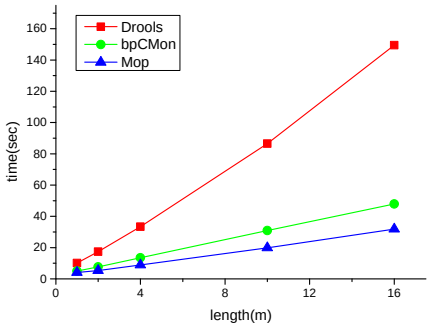

Fig. 4. The time costs for Drools, MOP, and bpCMon running over TestCase 1.

To compare bpCMon monitor, three related works [9] are adopted: the first is MOP, which is known as fastest monitor for Java programs in runtime verification community, and the second is Drools, which is the state-of-art rule-based inference engine, and the last one is MonPoly, which supports monitoring the metric linar temporal logic and the aggregation extension as well. The comparing is separated into two groups since their different running requirements: (MOP, Drools, bpCMon) and (MonPoly, bpCMon).

For the first group, the comparing is performed over six logs with instances number ranged from 1 millon to 16 millon. Before starting the comparison, for MOP, the monitor should be generated by running *rv-monitor* over the *.rvm* file which specifies above rules as a finite state machine; for Drools, a *kie-session* should be gotten first from the *.drl* file which describe the above rules as drools rules; and for bpCMon, the monitor is generated from the ECL specification of above rules. After these preparations, for each logs, each subject sequently runs three times over the logs by reacting on the event instance one by one through the related logs reader, meanwhile, for each running the maximal memory is recorded as well as its spent time. Note that, the time costs for logs reader initializing and closing are excluded from the spent time. The running data are listed in Table 5 and also the spent time comparing w.r.t. the length of logs is depicted as Figure4.

[9] MOP with version 4.0.0, Drools with 6.3.0, and MonPoly with 1.1.6.

Table 5. The performances of Drools, MOP, and bpCMon for running over TestCase 1.

NO	1	2	3	4	5	6
Type	(5k, 5k, 100)	(30, 100k, 10)	(1, 1m, 1)	(2, 1m, 2)	(5, 1m, 5)	(10, 1m, 8)
Length	1,010,000	2,000,060	2,000,002	4,000,004	10,000,010	16,000,020
Drools (time sec)	10.492	16.845	17.081	33.226	85.285	145.201
	10.18	17.451	17.27	33.406	87.521	149.49
	10.274	18.271	17.06	34.69	86.578	152.249
MOP (time sec)	4.07	5.137	4.977	8.91	20.718	31.173
	4.093	5.299	4.77	8.95	19.858	31.85
	4.341	5.308	4.84	8.714	19.676	32.467
bpCMon (time sec)	4.916	6.919	7.641	13.851	30.607	47.947
	5.014	7.679	7.944	13.482	30.948	47.877
	5.205	7.802	7.976	13.52	31.167	48.611
Drools (memory MB)	132.13	678.95	741.02	864.35	1155.7	1435.5
	420.53	740.88	629.25	800.52	1135.6	1420.9
	396.28	723.38	583.11	746.60	1097.8	1464.5
MOP (memory MB)	310.47	274.84	402.77	169.14	529.39	620.6
	392.52	425.8	441.25	194.50	542.9	617.0
	456.14	531.67	482.47	274.94	555.3	604.3
bpCMon (memory MB)	284.12	215.81	355.42	297.74	483.94	575.32
	419.67	406.07	503.31	517.47	515.07	612.8
	467.07	557.74	576.52	534.9	500.2	623.6

For the second group, five logs are used with lengths ranged from 1 million to 8 millon. The testing is conducted over the same laptop but on its VMware workstation with Fedora Linux 20 setting maximal memory to 3.8G, and bpCMon runs on the Eclipse of Mars version 4.5.0 with JDK1.8. The comparing is performed similarly to the first group, except the recorded running time including additional costs for file intializing and closing. The running data are listed as Table 6 and their time costs are drawed as Figure 5.

From Tables 5, 6 and Figures 4, 5, it shall be seen that, in this test case, both of the MOP and bpCMon outperform the Drools; the bpCMon peforms better than the MonPoly based on the consumed time and memories; and also the bpCMon is comparable to MOP, although there is some unstable in the memory consuption [10] might because of the garbage collecting of JVM.

Drools has its strength in finding fireable rules among large numbers of rules by using of net-based working memory. However, in monitoring setting, the solution for the issue, i.e., how to quickly find some *target* when some *trigger* occurs, is one of essential factors influencing the efficiency of monitor. For MOP, the solution is by parameterized indexing and the target is related state machinethe for the trigger; and for the bpCMon, thanks to the working structure which speeds up finding the stored target by the indexing relation between trigger and target. For the Drools, although there are also indexes in different types of node memories, additional time and memory might be needed because of its manner for dealing with the matched event instance: every matched instance needs to be stored firstly in related Alpha memory and then processing its influence on other related nodes. Considering the *release* rule, when a release instance occurs, to check whether there is related grant instance occurred before, it would be enough to simply check the state of related state machines for MOP and the related queue for bpCMon by parameters of release instance. However, in Drools, the release instance would be stored first and then deleted after processed,

[10] In the testing, the VisualVM 1.3.8 plugin is used to monitor and measure the memory consuming: https://visualvm.java.net/.

which might undermine the efficiency no matter in time or memory consuming. For the MonPoly, its analysis is based on formula rewritting rules and its running performance is very sensitive to the time points range, however, it is unclear whether there are some indexing structures for organizing related data or memory optimizing techniques.

Table 6. The performances of MonPoly and bpCMon for running over TestCase 1.

NO	1	2	3	4	5
Type	(5k, 5k, 100)	(1, 1m, 1)	(2, 1m, 2)	(2, 1m, 3)	(2, 1m, 4)
Length	1,010,000	2,000,002	4,000,004	6,000,004	8,000,004
MonPoly (time sec)	9.245	14.936	29.723	44.481	58.644
	8.865	14.557	28.828	43.541	58.540
	8.898	14.285	28.490	44.481	58.458
bpCMon (time sec)	5.308	8.084	15.426	21.744	27.778
	5.234	8.014	15.394	21.349	27.716
	4.945	7.759	14.713	21.744	27.338
MonPoly (memory MB)	433.072	442.260	826.980	904.480	1568.368
bpCMon (memory MB)	301.470	200.508	300.423	311.377	283.947
	303.764	266.052	303.355	321.671	302.657
	317.462	290.365	321.767	324.052	330.859

For the bpCMon, it is the indexing based working structure which enables bpCMon attain time efficiency; and for saving the memory consuption, each storing queues in bpCMon just keep the reference to the matched instance object, which is designed based on the fact: for each occurred instance as a fact, it can be assessed but unchangable. Although in the test case, MOP is more efficient then bpCMon, however, it can not yet support the compliance rules with metric time and data aggregation, e.g., the following test case 2.

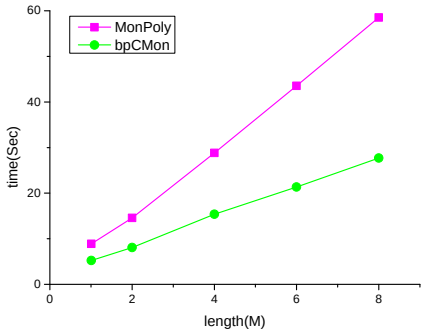

Fig. 5. The time costs for MonPoly and bpCMon running over TestCase 1.

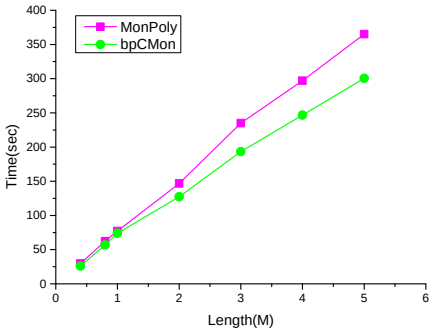

Fig. 6. The time costs for MonPoly and bpCMon running over TestCase 2.

TestCase2. For each user, the number of peaks over the last 30 days does not exceed a threshold of 5, where a peak is a value at least twice the average over some time window(here 30 days).

The test logs is generated based on parameters, $\#U$, the number of user, $\#EP$, the number of events for each user per time point, and $\#R$, the time point range. The

syntheising process is similar to [10]: setting $\#U$ to 100, $\#EP$ to 5, and $\#R$ from 800 to 10000, then in each time point, for each user generate 5 *withdraw* event instances of the form: (ts,$'$ *withdraw*$'$, *userID*, *amount*), where ts is the time stamp for the instance, *userID* orderly selected from [1, 100], and *amount* is set as a constant[11], and then, write all of these instances into the given file with their generated order. Note that, the time window(30 days) in the test case is set as 30 seconds. In this way, 7 test logs are generated with the length ranged from 400 thousand to 5 million.

The testing consists of 7 groups and within each group, the MonPoly and bpCMon run sequencely for the same logs. For each running, the monitors react on the instances parsed by relevant reader from the logs, and during the running the spent time was recorded and also the result was checked. For each group testing, the MonPloy does not response counter-example and also bpCMon reports compliant results. Their spent times are drawed as Figure 6. From the figure, it shall be seen that, such compliance rule is of expensive time consuming for both monitors, and their performances are not distinct too much for the logs with length within 1 million, but after that, since the st+tree structure, bpCMon monitor attains minor better scalability than the MonPoly which has the additional time overhead from 15% to 21% as the logs lengths increased from 2 to 5 million.

7 Related works

Basically, no matter what kind of monitoring, it includes the contents of three aspects: the *subject* to be monitored, the *objective* for monitoring, and the *technique* used by monitoring for making decision. Compliance monitoring aims at understanding the relation between behaviors of business processes and compliance rules. Currently, there is a plethora of related works addressing similar objectives but with different subjects.

7.1 Business processes compliance monitoring

The works in this category share the same objective and subject with this work, but with different compliance languages of various epxressive and monitoring mechanisms.

The work [24] proposed a metamodel REALEM for specifying compliance rules. The metamodel includes three parts: types model, three compliance rule patterns, and meta information. Based on events model and events correlating, the compliance rules are translated into executable rules on IBM infrastructures. As for the three patterns, e.g., $x\_AtLeast(c)\_Before\_y$, they are in fact corresponding to *before* and *after* patterns in ECL. In work [12], the graphic language *Declare* is used as compliance/processes constraints specification language. After translated the *Declare* to LTL, the monitor is created based on colored automata. The weakness of this work is the limitation of *Declare* which only supports control-flow perspective. In work [15], *Declare* is extended to support metric time and life-cycle for activity element and also provided with monitoring semantics based on event calculus formalism. Although the formalism is powerful, the extended *Declare* still supports less for the data perspective. In

[11] Such setting would make the logs compliant with the rule, which enables MonPoly and bpCMon compare the efficiency; Otherwise, if set the *amount* randomly, it would be unreasonable for the comparing, since their different ways to response the violation, i.e., MonPoly prints all the violations and bpCMon stores them in the container.

work [13], it proposed the Linear Dynamics Logic(LDL), which the integration of LTL and regular expression, as well as its monitoring semantics. The LDL enable to specify the compensation for some condition occurring. However, it is still theoretic work and without evaluation performance data. Works of [8][9] proposed compliance graph languages *CRG* and *eCRG*, where *eCRG* extended from *CRG* to support data and resource perspectives. For the monitoring algorithm, both works adopted marking based approach, which is a natural way for the graph language to describe the state of compliance rule. In addition, by graph marking, it is also possible to provide root causes for the violations. However, currently *eCRG* can not support data aggregation and also lack of performance data.

7.2 Business process execution monitoring

Although the works in category share the same subject with our work, they have different objectives as well as the used techniques. Process execution monitoring is close related to business activity monitoring(BAM), it typically includes filtering, correlating, aggregating, and reporting the data related to some PKIs [2] by proper CEP engine [25];

Work [26] proposes an approach which beak the activities into event trails and then translate compliance rules into CEP rules based on these event trails. By using the CEP engine, the compliance can be checked and responded. But it is not clear what kind of compliance rule this method can address. Furthermore, their extending work [7] proposes a model-driven method, which includes an Domain-specific Language based on control-flow occurrence patterns, and also the translations from DSLs to the codes used by specific CEP engine. The drawbacks are the limited expressive of DSLs and the relative limited abilities of CEP to support stateful compliance monitoring.

The work [27] consider the compliance between business process execution with process profile by using CEP. Three basic casual constraints are considered, strict order, exclusive, and interleving, and further translated into CEP queries. By filtering and aggregating events, the violations are reported. Work [28] considers the business process monitoring issue, where the events are related to activity life-cycle and business process is enriched with process event monitoring points. By process models decomposition technique, the process model is transformed into CEP queries. Work [29]propose a framework, named as aPro, which is by evaluating the KPI metrics to understand the business and compliance situations. To evaluate such KPIs, related measure points should be defined first and annotated into the process models. During the processes execution, the related measures data are collected and correlated, and analysis by using of CEP technique. Based on such measures, the regular KPIs and aggregating KPIs are evaluated and then the goals fulfillments are valuated. The key challenges for such methods are how to define relevent KPIs and their related measuring points. Furthermore, it is unclear about how to support temporal compliance rules.

7.3 Services-based system monitoring

The subject of this kind of monitoring is Services-based system and its first class citizen is the messages among services. Comparing to this work, this kind of monitorings have similar objective and also overlapped monitoring mechanisms.

The objective of the monitoring in [30] is to understand the relation between the interactions specified in BPEL and the policies by proposing instance monitor and

class monitor. The policy language is message-based and devised based on PastLTL, and also enables to specify properties of counting based aggregation and staying based time. However, this language still not powerful enough for supporting after-type and metric time policies. Also, the monitor has no root-cause analysis for the violations. Work [31] proposed the method for detecting semantically-upgraded complex events by making use of semantic ontology during the execution of semantic web services composition. However, the objective of this work is different to processes compliance monitoring.

The work [32] proposed a multi-level monitoring framework, named as SERMON, to address the compliance of service agents behaviours with their commitment protocols which are specified with Communicative Act extended Object Constraint Language. The work is the extension of their prior work on requirement monitoring REQMON [33] and the facility used for monitoring is rule based engine Jess.

To enforce the data-aware message contracts for the web services, the work [34] proposed an specification language LTL-FO+, extending from LTL by introducing based on quantifier and XPath an new construct to quantify the data inside the message. The monitoring algorithm for LTL-FO+ formula is designed based on a finite state machine variant, called watcher, which is created on the fly to avoid the huge state space and the state transition is based on formula structure. Regrading to the data-aware contracts, the ECL is also enable to cope with based on event attributes as well as events correlating, furthermore, ECL also support data aggregation which is out of scope of LTL-FO+.

7.4 Runtime verification

The program is the main subject of runtime verification, but it has similar objective and monitoring mechanisms with process compliance monitoring.

The Java-MaC [35] is a prototype implementation of the monitoring and checking architecture for Java programs. From the specification view, it consists of two levels languages, PEDL, for low level event definition and as well as high level events definitions based on low level ones; MEDL is used to describe the property based on high level events. From such two specifications, the filter, event regnizer, and checker can be created automatically. The filter is probed into java codes and send related messages to regnizer to import high level events for checking.

The work [36] compared three typical monitoring approaches regarding to finite state properties, i.e., object-based, state-based, and symbol based, to investigate their relative strengths and weaknesses. By experiments, the object-based has little overhead for multi state changes in property comparing to the other, however, suffers much overhead when the loops within properties. The other two seems exactly converse to the object-based. The work [37] extends the parametric trace slicing method to overcome the weakness of dealing with the operating data and universal of parameters, by proposing the quantified event automata. However, although automata is powerful, lots of works are still needed for specification language and evaluations.

The work [21] implements a rule system based runtime engine, i.e., LOGFIRE, by remifying RETE algorithm. The refimications include introducing the double-indexing among related nodes for speeding up tokens' matching. However, once some fact in the node was updated, each related nodes as well as the indexing mappings would need to be updated synchronizedly. It thereby might be the factor of undermining the efficiency.

In this paper, the tree-like indexing structure is more simple, flexible, and independent. For each fact in value structure, after used, it can be deleted without any influences on others. Besides, from the specification aspect, the supports in [21] for high level, e.g., temporal order, is still limited to two basic patterns.

7.5 Discussions

From the views of the specification language and monitoring mechanism, the relations with closely existed works need to be further clarified to promote the distinct features as well as behind designing principles for the bpCMon.

(1) *the relation between MLTL/LTL and ECL.*

From the expressive aspect, except aggregating part, ECL formula could be described in MLTL/LTL. However, ECL is more abstract than MLTL/LTL, and is specifically designed for compliance monitoring based on the signature including events and event-relation patterns. Within the ECL formula, it includes the monitoring featured elements: *trigger*, *decider*, and *events correlating*, which are essential and also properly organized for creating ERS and efficient monitoring afterwards. These points are not easy to be attained by directly using MLTL/LTL. Besides, MLTL/LTL does not directly support specifying the compliance rule of data aggregation.

(2) *the distinct features of ERS*

ERS is a light-weight rules engine and designed specifically for reactive monitoring. Its distinct features come from two aspects: the rule form and working structure. For the rule form ,as mentioned before, the rules of ERS are of the form, $trigger \rightarrow c\_reaction$. Comparing to classic rule form, e.g., $event\text{-}condition \rightarrow action$ [21], the characters of ERS rule form could speed up finding fireable rules and condition evaluation, and also avoid the issue about fact *staying around* [21] for generic rule engine, e.g., Drools. For the working structure, it is a tree like structure distributed and equipped with *indexing structure* for each independent storing, which is different to the *net* like and *node* sharing structure of RETE-based rules system. Such working memory has its strength for the case of numbers of rules overlapped their left hands. However, for the monitoring, as indicated in the experiment, ERS working structure is more efficient.

(3) *dealing with memory consuming for ERS*

In fact, in bpCMon, time interval plays an important role in reducing memory consuming through the bounded data queue. More specifically, once the instance is out of date comparing the new added one, it would be deleted from the queue. Besides that, the principle, "abandon it after it used", is adopted to apply to each reaction. i.e., for each instances in the afterIIS structures, after it was assessed and used, it would be deleted at once from the queue. In addition, as mentioned before, each storing queues in bpCMon just keep the reference, instead of the clones , of the matched instance object, which can also effectively save the memory consumption. Furthermore, from the programming technique view, garbage collecting and using weak reference object instead of strong reference object, are also some future options for further reducing memory like [37].

8 Conclusion and future works

To target at specifying complex compliance rules as well as their monitoring, this work presents the business processes compliance monitoring framework, bpCMon. It

mainly consists of: an event-pattern based compliance language(ECL) for specifying compliance rules, and events indexed reaction system (ERS) as engine for compliance monitoring. The ECL is devised based on event-pattern and events relation patterns, and also featured with aggregating operators; ERS is a powerful rule based system and designed based on events indexing structure. Experiments on a real life hospital logs over 16 compliance rules indicate the applicability of bpCMon; and the comparisons with two known related works, the MOP and Drools, over the benchmark demonstrate the efficiency of bpCMon; and furthermore the data aggregation functionals of bpCMon is also evaluated by test case.

As for the future works, from the practical view, a friendly interface is needed to support users specifying and managing their compliance rules; from the theoretical view, it is also important to further devise the methods to resolve the rules *conflict* issue and the *intersecting* of working structures as well as their possible relations. In fact, such solutions would be the basis for targeting at the scalable issue of the bpCMon when considering huge number of rules. Finally, further evaluations are also needed for the soundness of the bpCMon.

References

1. Sadiq, S.: A roadmap for research in business process compliance. In: W. Abramowicz, L. Maciaszek, K. Wecel (Eds.) BIS 2011 Workshps, LNBIP 97, pp. 1-4 (2011)
2. Rademakers, T.: Activiti in action. Manning publications(2012)
3. Breaux, T.D., Antn, A. I., and Doyle, J.: Semantic parameterization:a process for modeling domain descriptions. ACM Transations oon Software Enginnering and Methodology, Vol. 18, No.2 (2008)
4. Pesic M. and Van der Aalst W.M.P.: A declarative approach for flexible business processes management. In: J. Eder, S. Dustdar et al.(Eds.): BPM 2006 Workshops, LNCS 4103, pp.169-180 (2006)
5. Awad A., Weidlich M., and Weske M.: Visually specifying compliance rules and explaining their violations for business processes. J. Visual Languages and Computing. 22(2011), pp. 30–55 (2011)
6. Elgammal A., Turetken O., van den Heuvel W.: Formalizing and appling compliance patterns for business process compliance. Software & Systems Modeling, pp 1-28 (2014)
7. Mulo, E., Zdun, U., and Dustdar, S.: Damain-specific language for event-based compliance monitoring in process-driven SOAs. In: SOCA (2013) 7: 59-73 (2013)
8. Ly L.T., Rinderle-Ma S., Knuplesch D., and Dadam P.: Monitoring business process compliance using compliance rule graphs, In: R. Meersman, T.Dillon, and P. Herrero(Eds.): OTM 2011, Part I, LNCS 7044, pp. 82-99 (2011)
9. Knuplesch D., Reichert M., and Kumar A.: Visually monitoring multiple perspectives of business process compliance. In: Hamid, R. M. N., Jan, R., and Matthias, W.(Eds.), BPM 2015. LNCS 9253, pp. 263-279 (2015)
10. Basin D., Klaedtke F., Mller S, etc.: Monitoring of temporal first order properties with aggregations. In: proceeding of RV 2013, LNCS 8174, pp. 40-58 (2013)
11. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns property specifications for finite-state verification. In: ICSE 99, Los Angeles CA, ACM, pp. 411-420 (1999)
12. Maggi F.M., Montali M., Westergaard M., Van der Aalst W.M.P.: Monitoring business constraints with linear temporal logic: An approach based on colored automata. In: S.Rinderle-Ma, F. Toumani, and K. Wolf(Eds.) BPM 2011, LNCS 6896, pp. 132–147 (2011)
13. Giacomo G., Masellis R.D., Grasso M., Maggi M.F., and Montali M.: Monitoring business metaconstraints based on LTL and LDL for finite traces. In: Shazia W. S., Pnina S., Hagen V.(Eds.) BPM 2014. pp. 1–17 (2014)

14. Basin D., Klaedtke F., Mller S., and Zlinescu E.: Monitoring metric first-order temporal properties. In: Journal of ACM, 62(2), pp:1-38 (2015)
15. Montali M., Maggi M.F., Chesani F., Mello P., and Van der Aalst W.M.P.: Monitoring business constraints with the event calculus. ACM Transactions on Embedded Computing Systems, Vol.9, No. 4, 1-29 (2010)
16. Chen F., Jin D., Meredith P.O.N., and Rou G.: Efficient formalism-independent monitoring of parametric properties. In: ASE 2009, IEEE press, pp. 383-394 (2009)
17. Charles L. Forgy.: Rete: a fast algorithm for the many pattern/many object pattern match problem. In: Artificial Intelligence 19(1982), pp. 17-37 (1982)
18. Drools:http://docs.jboss.org/drools/release/6.3.0.Final/drools-docs/html_single/index.html (2015)
19. Friedman-Hill E.: Jess in action: Rule based systems in Java. Manning publications. (2003)
20. Ly L.T., Maggi F.M., Montali M., Rinderle-Ma S., van der Aalst W.M.P.: Compliance monitoring in business processes: functionalities, application, and tool-support. Information Systems. 54, 209–234 (2015)
21. Havelund K.: Rule-based runtime verification revisited. In: Journal of Software Tools Technology Transfer, 17:143-170 (2015)
22. Gnther C.W., and Werbeek E.: XES standard definition. In http://www.xes-standard.org/_media/xes/xesstandarddefinition-2.0.pdf, March 28, version 2.0 (2014)
23. Turetken O., Elgammal, A., and van den Heuvel W.J.: Capturing compliance requirements: a pattern-based approach. IEEE Software, IEEE Computer Society (2012)
24. Giblin, C., Muller, S., Pfitzmann, B.: From regulatory policies to event monitoring rules: towards model-driven compliance automation. Technical report RZ3662, IBM Research. (2006)
25. Esper: http://www.espertech.com/index.php (2015)
26. Mulo, E., Zdun, U., and Dustdar, S.: Monitoring web service event trails for business compliance. In: IEEE International Conference on Service-oriented Computing and Applications(SOCA), IEEE, 2009.
27. Weidlich M., Ziekow H., Mendling J., Günther O., Weske M., and Desai N.: Event-based monitoring of process execution violations. In: S. Rinderle-Ma, F. Toumani, and K. Wolf(Eds.) BPM 2011, LNCS 6896, pp.182-198 (2011)
28. Backmann, M., Baumgrass, A., and Herzberg, N.: Model-driven event query generation for business process monitoring. In: A.R. Lomuscio et al. (Eds.): ICSOC 2013 Workshops, LNCS 8377, pp.406-418 (2014)
29. Koetter, F., and Kochanowski, M.: A model-driven approach for event-based business process monitoring. In: Information Systems and e-Business Managmement, Springer-Verlag, vol 13, issue 1, pp. 5-36 (2015)
30. Barbon, F., Traverso, P., Pistore, M., and Trainotti, M.: Run-time monitoring of instances and classes of web service compositions. In: IEEE Inernational Conference on Web Services(ICWS'06), IEEE computer society. (2006)
31. Vaculin, R., and Sycara, K.: Specifying and monitoring composite events for semantic web services. In: Fifth European Conference on Web Services. (2007)
32. Robinson W.N., and Purao S.: Monitoring service systems from a language-action perpective. In: IEEE transactions on services computing, Vol. 4, No, 1. pp. 17-30 (2011)
33. Robison W.N.: A requirements monitoring framework for enterprise systems.In: Requirements Engineering, 11:17-41 (2006).
34. Hall S., and Villemaire R.: Runtime enforcement of web service message contracts with data. In: IEEE transations on services computing, vol 5, vol 2. pp. 192-206 (2012)
35. Kim M., Kannan S., Lee I., and Sokolsky O.: Java-Mac: a run-time assurance approach for java programs. In: Formal Methods in System Design. 24(2), pp 129-155 (2004)
36. Purandare R., Dwyer M.B., and Elbaum S.: Monitoring finite state properties: algorithmic approaches and their relative strengths. In: S. Khurshid and K. Sen (Eds.): RV 2011, LNCS 7186, pp. 381-395 (2012)

37. Luo Q., Zhang Y., Lee C., Jin D., Rou G., et al.: RV-Monitor: efficient parametric runtime verification with simultaneous properties. In: B. Bonakdarpour and S.A. Smolka(Eds.): RV 2014, LNCS 8734, pp. 285-300 (2014)
38. Havelund K., and Joshi R.: Experience with rule-based anaysis of spacecraft logs. In: C. Artho and P.C.Ölveczky(Eds.): FTSCS 2014, CCIS 476, pp. 1–16 (2015)
39. Barringer H., Falcone Y., Havelund K., etc.: Quantified event automata: towards expressive and efficent runtime monitors. In: D. Giannakopoulou, and D. Méry(Eds.): FM 2012, LNCS 7436, pp. 68-84 (2012)
40. Datar M., Gionis A., Indyk P., and Motwani R.: Maintaining stream statistics over sliding windows. In: SIAM J. COMPUT. 31(6), pp. 1794-1813 (2002)
41. Basin, D., Klaedtke, F. and Zlinescu E.: Greedily computing associative aggregations on sliding windows. In: Information Processing Letters, elsvier, 115(2), pp. 186-192 (2015)

Ulmer Informatik-Berichte
ISSN 0939-5091

Herausgeber:
Universität Ulm
Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
89069 Ulm