Rule-based Runtime Monitoring of Instance-Spanning Constraints in Process-Aware Information Systems

Conrad Indiono, Juergen Mangler, Walid Fdhila, Stefanie Rinderle-Ma

University of Vienna, Faculty of Computer Science, Vienna, Austria
{firstname.lastname}@univie.ac.at

Abstract. Instance-spanning constraints (ISC) constitute a crucial instrument to establish coordination between multiple instances in Process-Aware Information Systems. ISC need to be verified and monitored at design as well as runtime. In this work we propose a rule-based approach for runtime monitoring of ISC. We base our work on the well known Rete algorithm and research ways structure the network in such a way that improves matching speed for ISC. We show through a technical evaluation that (1) a rule-based approach is feasible for performing runtime monitoring of ISC and (2) that the heuristics we extract for structuring the Rete network improve the rule matching speed.

Keywords: Business process compliance, Instance-spanning constraints, Runtime Monitoring, Performance Optimization

1 Introduction

Monitoring compliance of processes with rules such as regulatory constraints or security requirements during runtime constitutes a crucial job for many enterprises [8]. Examples comprise the logistics [9], manufacturing [15], and health care [5] domains. The compliance monitoring architecture presented in [8] advocates the collection of events from process execution environments and monitoring them based on compliance constraints by a monitoring engine. Figure 1 substantiates and extends this architecture by proposing the usage of a rule engine as monitoring engine that returns actions to the process execution environments such as process engine and worklists. The actions enable the enforcement of compliance constraints. This is particularly important in the context of instance-spanning constraints (ISC) [3]. ISC span multiple instances of one or several process models and can be utilized to constrain the behavior of these instances and to enforce certain properties. One example is the realization of synchronization between tasks of different process instances [10].

In order to realize a monitoring system as suggested in Fig. 1 it has to be ensured that events and actions are commonly understood between the event sending system and the rule engine. Another important aspect is the language to describe the compliance constraints that is then mapped onto rules. Event Calculus (EC) has been evaluated as suitable for compliance monitoring in [11,8].

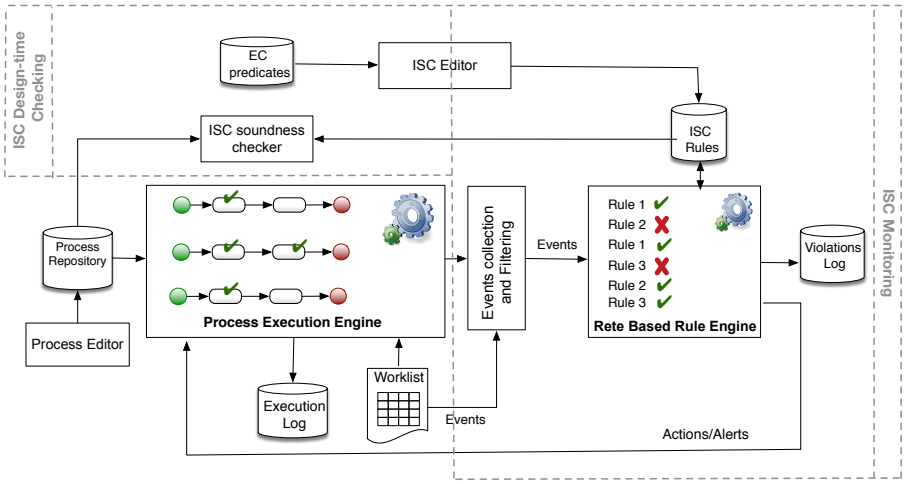


Fig. 1. Event based compliance monitoring in PAIS – overview

Moreover, as shown in [3], Event Calculus (EC) is specifically suited to meet the requirements of compliance monitoring applications and ISC. Hence, in the following, the considerations are based on EC, but the transferability to other formal constraint languages will be discussed. MobuConEC provides a compliance monitoring approach based on EC together with DECLARE (see e.g., [12]) in [11]. As stated in [11], process data values such as resources can be modeled, but the monitoring is not fully supported yet. It remains unclear whether and how the approach can be used for monitoring ISC.

In summary, the following gaps for full support of ISC monitoring in PAIS remain:

1. In the context of interaction between PAIS and rule engines, so far, simple pattern matching techniques have been employed, e.g., by REMAR [10].
2. Existing approaches do only partly focus on runtime optimization. Benchmarking is difficult as most of the tools are not publicly available [11].
3. Existing compliance monitoring approaches do not consider ISC [8].

This leads to the following research questions tackled by the work at hand:

RQ1 Given event data and conditions encoded in Event Calculus, how to execute such rules efficiently?
RQ2 Does condition ordering have an effect on matching performance?
RQ3 Is there an optimal way to structure ISC-specific conditions?

In this paper, a novel approach for monitoring ISC through a rule engine based on the Rete algorithm [4] is presented. This requires matching ISC with Rete. We conduct benchmarks to observe various aspects that affect matching performance and derive heuristics for better structuring the Rete network for

handling ISC. Altogether, the proposed concepts enable a) the monitoring of ISC in PAIS and b) are geared towards increasing performance of the monitoring. The latter is particularly important as possibly a multitude of ISC is to be monitored during process runtime.

The remainder is structured as follows: Section 2 provides a motivating example. In Section 3 it is shown how to match ISC with Rete. Section 4 deals with improving the performance of the matching. Section 5 discusses related work and Sect. 6 concludes the paper.

2 Motivating Example

As an illustrative example, we consider an energy provider that uses an integrated energy management solution to deliver end-to-end advanced metering (cf. [3]). In particular, it provides services for the electricity metering; e.g., meter reading, read-out analysis, and billing. In this example, we abstract the detailed description of the energy provider process, and focus on a subset of tasks (cf. Figure 2). The company provides energy to a set of clients, and for each of these clients regular (daily) read-out measures are carried out. First, information about the client, i.e., the client profile, is checked (e.g., ID, communication protocol, meter type). Then a connection is established and instantaneous values for voltage, power factor and frequency are measured. Additionally, a memory test is performed, and operational irregularities are detected, stored, and transmitted. After the read-out, an analysis of the sampled values is carried out, and the profile analysis is updated and stored. It should be noted that this example requires checks on measurements for the same meter, i.e., instances, in order to detect irregularities in the values. The rule also is an ISC – it involves aggregated events and measurements from several meter read outs.

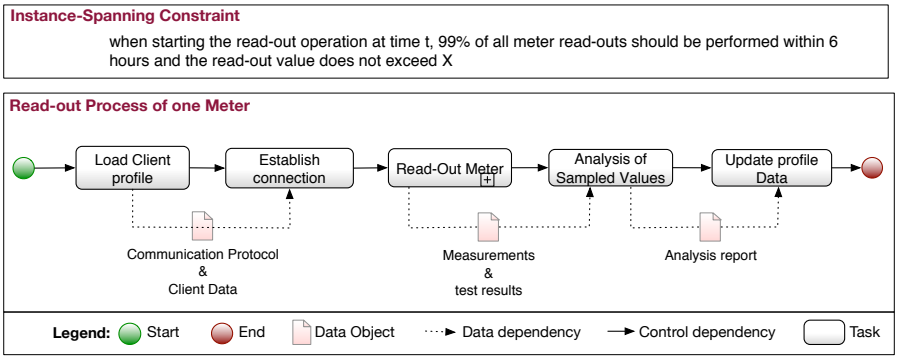


Fig. 2. Process example from the Energy Domain (cf. [3])

Figure 3 describes the execution of the meter read-out process during runtime. In particular, it shows the running instances of N meters' read-outs. De-

pending on the communication protocol for communicating with the corresponding distant meter and the meter type, the time for establishing the connection as well as the time for making the measurements varies. We assume that there is an internal global event $GlobalReadoutStart()$ which launches all read-out instances at the same time t (e.g., daily at 00:00 am). We also assume that for each task of type $read{-}out\ meter$ an event $ReadoutEnd(meter, measurements)$ is emitted, which refers to the completion of that task. All events required for checking the ISC rules are transmitted to the ISC monitor, which checks for any violations and produces alerts if required. In this example, note that the read-out violation for client 3, which happened after 6 hours, can not be directly considered as an ISC violation, unless the aggregated number of read-out violations (of this type) exceeds 1%. However, if the current total value of the finished read-outs exceeds the threshold "x", then a violation alert is emitted immediately (without waiting for the other running read-outs).

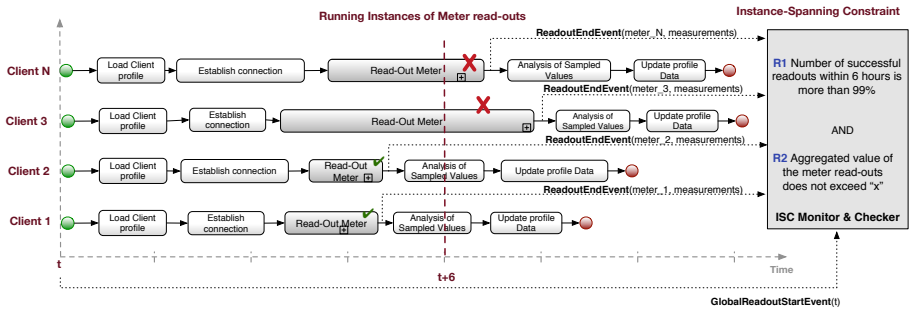


Fig. 3. ISC monitoring example

3 Matching ISC with the Rete Algorithm

3.1 Pattern matching with Rete

A crucial step for evaluating and enforcing constraints during runtime is to match the events emitted by the process execution environments onto the rules and if required to invoke certain actions. In the context of PAIS, mostly simple pattern matching techniques have been employed so far, e.g., [10]. Addressing RQ1, a first step is to analyze whether using more advanced pattern matching techniques is beneficial with respect to performance.

A commonly used pattern matching algorithm in rule engines is the Rete algorithm. It was first proposed in [4]. Figure 4 shows the complete Rete structure for the single rule that adds read-out meter values as they come in from "read-out meter end" events. These are filtered and classified as successful when the timestamp and accumulated_value condition evaluate to true. As the action,

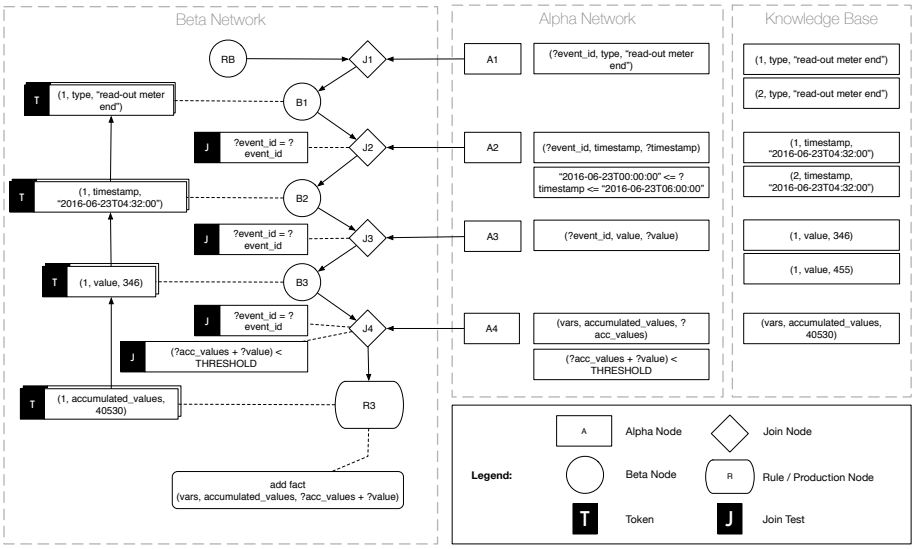


Fig. 4. Basic Rete Network

which occurs on the right-hand side of the rule (RHS) in $R3$, the existing working memory element (WME) representing the total accumulated read-out values ($?acc\_values$) is updated in the knowledge base. Similarly, $R3$ could update another fact increasing the total number of successful read-outs that occured. A Rete network is generally comprised of three parts: the alpha network, the beta network and the knowledge base.

Knowledge Base The knowledge base is the total set of facts that the rule engine is aware of. These facts are also called working memory elements (WMEs). A WME is represented as a triple structure [2] $(id, attr, value)$ on which more complex data structures can be modeled. The first *id* part of the WME is generally a unique id that identifies the fact. The second *attr* part of the triple represents a certain attribute. Having several WMEs with the common *id* one can extract all available attributes and lookup the appropriate values. The final *value* part holds the actual value which the fact represents. In our case we support a *JSON*-like data type excluding the *object* type. Thus we support values of types *number* (*integer* or *float*), *string*, *arrays* and *maps*. The latter two types are recursively defined.

Alpha Network The first contact point to the rule engine for rule activation is the alpha network. As new facts are entered into the knowledge base, the same WME will be sent to the alpha network. We follow [2] and implemented a hash-table variant of the alpha network. Each element within this network is called an alpha node. An alpha node represents a single rule condition and is responsible for processing incoming facts that match the structure of the rule condition. This is also the level where simple constant value checks can be performed to

discriminate relevant WMEs for further checking. As can be seen in $A2$ of Figure 4 the alpha node can directly check each incoming timestamp for matching the condition ("2016-06-23T00:00:00" <=?$timestamp$ <= "2016-06-23T06:00:00").

Rule conditions use the same triple structure applicable for defining WMEs. The difference lies in the ability to set variables to each triple part. These are marked with a preceeding question mark. Thus (?$event_{id}$, $timestamp$, ?$timestamp$) represents the rule condition that matches any kind of $event_{id}$ as long as the *attr* part is equal to constant *timestamp*. All variables are stored and marked for further processing within the beta network. The collection of alpha nodes or rule conditions that lead to rule activation and are thus associated with a rule is called the left hand side (LHS) of the rule.

Beta Network In order to check whether the WME, that has been sent by the alpha node, leads to triggering of rules or not requires a process called activation. A *right-activation* occurs when the join node that is associated by the alpha node is passed a WME. An example would be $A2$ passing the WME $(1, timestamp, "2016\text{-}06\text{-}23T04:32:00")$ to $J2$ in Figure 4. The *right-activation* of $J2$ results into a *join test* (also called *consistency check*) for outstanding variables within the conditions defined in $A2$. In this instance only the ?$event_{id}$ *id* part is checked with tokens from the parent $B1$, since only ?$event_{id}$ occurs in the previous condition. Free-standing variables pass down without contest. Once the *join test* succeeds, the resulting WME is stored as a token inside $B2$. A beta node ($B2$) holds a set of tokens that have passed *join tests*. This essentially stores partial rule condition match results. On storing the token, child join nodes are *left-activated* (e.g. $J3$). This process happens repeatedly with child nodes as long as *join tests* succeed until the point of rule triggering happens where for example $J4$ adds the final token to the production node ($R3$). As can be seen from the Token area of Figure 4, tokens build a chain backwards to map the variables with the actual WMEs that lead to successfull activations. This allows the lookup on the set of WMEs that lead to rule triggering. At rule trigger time the right-hand side (RHS) of the rule is activated and actions are performed. In the case of $R3$ a fact inside the knowledge base is updated representing the total value for read outs.

3.2 Mapping Event Calculus to Rete

[3] has shown that Event Calculus (EC) is a relevant formal language for representing ISC. Specifically, EC [6] is a logic programming approach to model time and change. It uses first order predicate logic (FOL) as the basis and introduces fluents for the ability to model time-varying state. Events are the occurence of actions that might trigger changes to the valuation of fluents. Time is modeled as a one-dimensional horizantal progression and events can occur at any point on this timeline. Fluents change their state within this timeline as well. Both fluents and events can have unlimited parameters giving the ability to represent a multitude of situations to model. EC defines a set of domain independent predicates for dealing with events and fluents: $HoldsAt(f, t)$ asserts that a fluent f evaluates to true at time t. $Happens(e, t)$ asserts that an event e occures at time t. $Initiates(e, f, t)$ first asserts $Happens(e, t)$ then evaluates the fluent f

to true after time t. For setting a fluent f to false when $Happens(e, t)$, then $Terminates(e, f, t)$ can be used.

Figure 5 shows the mapping of concepts available in EC to be executed on top of Rete. In that Figure we distinguish between the LHS and RHS. The LHS are checked for triggering an associated rule, whereas the RHS can be used for either asserting facts and fluents or changing them.

First-Order Logic (FOL) The first four concepts are taken from FOL. The first concept (*conjunction*) is the default way of defining rule conditions in rete. A list of rule conditions are defined using the triple structure and submitted for Rete network construction. A series of alpha node, join node and beta node sequences are newly generated for each condition. If a given condition is equal to another (e.g. shared by another rule) then that one would reuse the existing alpha node (also reusing the subgraph from that point on). *Disjunctions* are handled by splitting the condition list into a number of parts which can be conjunctive. These are then individually created the same way as the *conjunctive* case. The notable point is that the same RHS of the rule is then associated with the final production nodes that are created for each *disjunctive* part. *Quantifiers* can split into three cases (1) *universal quantifiers*, (2) *universal quantifiers* with logical filters, and (3) *existential quantifiers*. Figure 5 maps the latter two cases. The first case is identical to the *conjunctive* case, where each universally quantified condition passes through uncontested to the *beta network*. Case (2) is more interesting: we still need to let the WME matching the condition pass through the *beta network* uncontested, but we need to know whether the WME really passes the filtering logic. In the example, for the universal quantification $\forall x : x > 5$ to be considered true at all times, we need to be notified by the rule engine when it observes the case where this is not true. Thus the RHS action of *assert* can be utilized for this purpose. For case (3) we only need to maintain a WME that indeed the existential quantifier $\exists y$ is true, which is added to the knowledge base when such a WME is observed at least once.

Event Calculus Predicates Fluents are at the heart of EC and supplement facts by adding a time dimension to them. As rete only deals with facts (WME, tokens) in triple structure, we need to either model fluents on top of this triple structure or consider adding native support to fluents inside values. Recall that we support a *JSON*-like data structure for the value part of the triple. Supporting fluents can then mean using the *map* data type to hold *timestamps* as index keys and have as values the *boolean* value representing the fluent's state at that timestamp t. A constant *map* entry for *params* can hold the list of parameters that the fluent is associated with. Thus the triple structure:

$$(?fluent_{id}, \mathrm{f}, ?fluent) \tag{1}$$

carries fluent f where the value part holds the aforementioned fluent instance and represents complete state of f. $HoldsAt(f, t)$ then takes such a condition and performs assertions on the fluent instance f's attributes. $Happens(e, t)$ is mapped for catching the case where a specific event e happens at time t. This could be applied on the LHS for rule activation (as showin Figure 5 or on

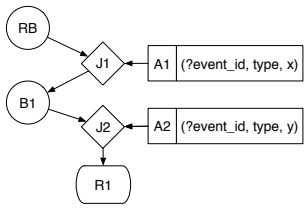
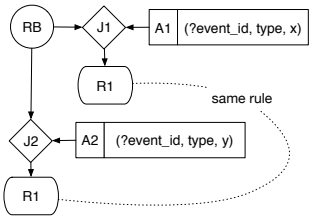
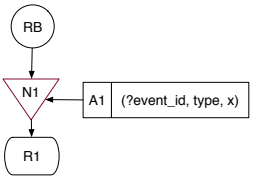
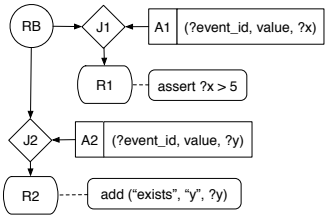
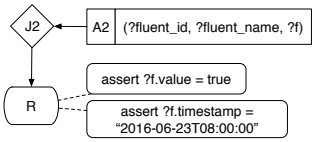
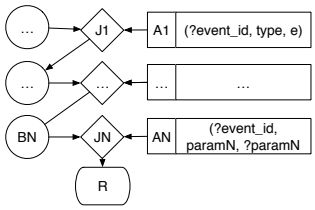
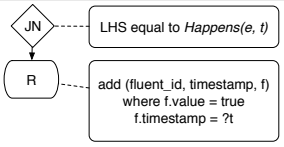
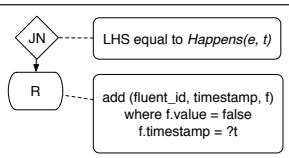
Concept	Triple Form (LHS)	RHS Actions	Network Structure
Conjunction: x ∧ y	Conditions: (?event_id, type, x) (?event_id, type, y)	-	
Disjunction: x v y	Conditions (1): (?event_id, type, x) Conditions (2): (?event_id, type, y)	-	
Negation: ¬x	Conditions: (?event_id, type, x)	-	
Quantifiers: (1) ∀x: x > 5 (2) ∃y	Conditions (1): (?event_id, value, ?x) Conditions (2): (?event_id, value, ?y)	Actions (1): assert ?x > 5 Actions (2): add to shared fact: ?y exists	
HoldsAt(f, t)	Conditions: (?fluent_id, ? fluent_name, ?f)	Actions: assert ?f.value = true assert ?f.timestamp = “2016-06-23T08:00:00”	
Happens(e, t)	Conditions: (?event_id, type, e) (?event_id, timestamp, ?t) (?event_id, param1, ? param1) … (?event_id, paramN, ? paramN)	-	
Initiates(e, f, t)	Conditions: (?event_id, type, e) (?event_id, timestamp, ?t) (?event_id, param1, ? param1) … (?event_id, paramN, ? paramN)	Actions: add (fluent_id, fluent_name, f)	
Terminates(e, f, t)	Conditions: (?event_id, type, e) (?event_id, timestamp, ?t) (?event_id, param1, ? param1) … (?event_id, paramN, ? paramN)	Actions: add (fluent_id, fluent_name, f)	

Fig. 5. Transformation Table of Event Calculus into Rete Networks

the RHS, by adding appropriate assertions on the event instance e's attributes. $Initiates(e, f, t)$ and $Terminates(e, f, t)$ are mapped equally to $Happens(e, t)$ on the LHS for being semantically identical. Only on the RHS does the behaviour differ. There we update the fluent to the appropriate *boolean* value for timestamp t.

4 Improving pattern matching performance

The order of conditions are determined at rule construction time. In this section we show that the order of these conditions (alpha nodes) within the alpha network has an effect on the total rule matching time. In the context of ISC, the number of instances of a process model directly influences how many WMEs are going to be generated for each condition. This effect is especially apparent when there is a big difference between the expected number of occurences for two events which share some conditions from node sharing.

4.1 Effects of the RHS modifying shared condition nodes (IMP1)

First we will explore the question of shared condition node placement. These shared nodes are spread among many rules and may represent a single fact or variable. Shared condition nodes occur frequently such as when using *quantifiers* and the EC predicates $Happens(e, t)$, as well as $Initiates(e, f, t)$ and $Terminates(e, f, t)$ (cf. Figure 5). It is imperative that we understand how to structure these shared condition nodes allowing improved matching time. Specifically in this section we question whether there is a difference in matching time due to the placement of such shared condition nodes. There are two possible placements that we will observe: top or bottom. As shown in Fig. 6(a) a top placement is done by establishing the shared condition before any other conditions of the same rule are defined. Rules defined in this way have the shared condition placed in the beginning of the condition list. This list is taken and during Rete network construction a top level alpha node is generated representing the shared condition. Subsequent conditions that are placed in conjunction below the shared condition are linked through the same beta node. In contrast, Fig.6(b) illustrates the inverse case where the shared condition is defined in the bottom part of the network. In this case, rules place the shared condition in the last part of the condition list, leading to the construction of an alpha node and several join nodes equal to the number of rules that use the shared condition.

The motivating example (cf. Figure 3) includes such a shared condition in the form of the aggregated value of all meter read-outs. This variable can be represented as the triple:

$$(\text{"vars"}, \text{"aggregated\_value"}, ?aggregated\_value) \tag{2}$$

The rule $R2$ of Figure 3 ensures that this aggregated value does not exceed a certain threshold. There is also another variable required for tracking the number of successful read-outs, which can also be implemented as a shared variable and checked by $R1$. For both cases, there are other rules which actually modify the

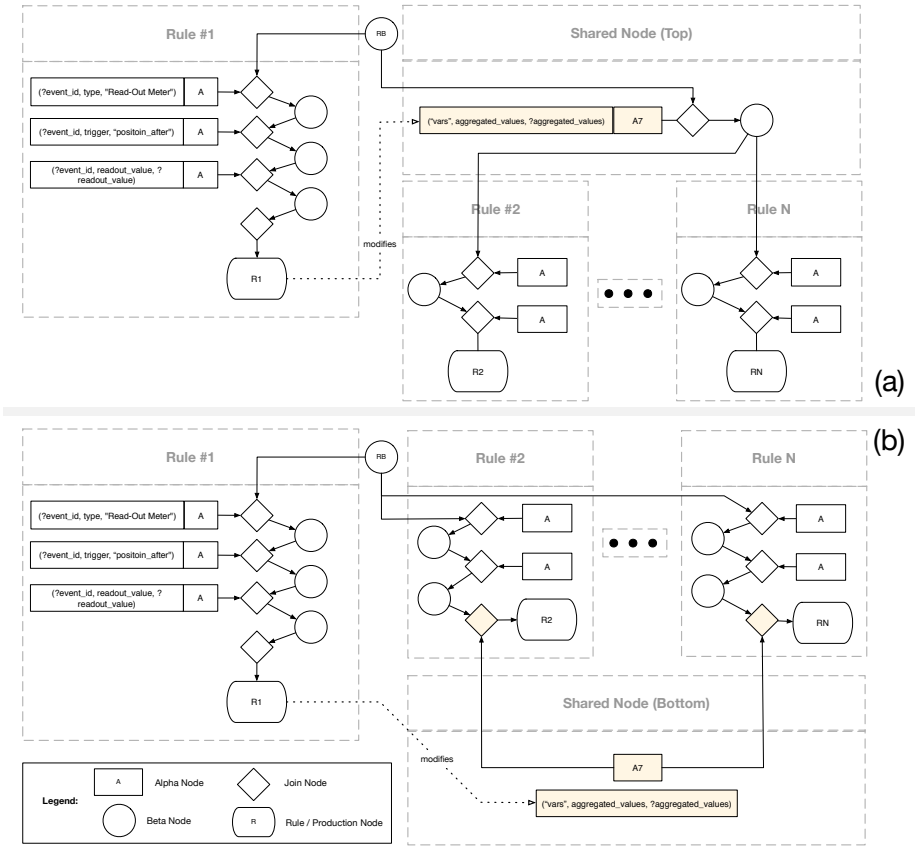


Fig. 6. IMP1 concept. RHS modifies shared variable. Positioning of this shared condition node.

shared variables (see Rule #1 in Figure 6). These rules wait for the correct event type to occur and once activated will execute the RHS of the rule by modifying the appropriate shared conditions. Note that for both variables there is a difference in the expected number of checks required. For the 99% check, we wait for $R1$ to be triggered by listening to the "global readout end" event to occur. The expected number in this example is once per day at 6AM. For the shared condition representing $?aggregated\_value$ we expect $R2$ to be checked every time a "read-out meter end" event occurs, which scales with the number of meters that finish the "Read-Out Meter" activity. Thus we expect the $R2$ rule to be more expensive in term of matching time.

IMP1 Benchmark and Analysis We conduct the following benchmark in order to observe the effects of IMP1. First, our main goal is to identify the effects of shared condition positioning towards matching time. We define two event types: the *main event* is of the type "read-out meter end" which represents the rule

	RHS=Noop	RHS=0%	RHS=100%
Position: Top			
[C_N=1] Pre	373ms	383ms(+2.68%)	**555ms**(+48.79%)
[C_N=1] Post	**367ms**	376ms(+2.45%)	387ms(+5.44%)
Position: Bottom			
[C_N=1] Pre	371ms	376ms(+1.34%)	392ms(+5.66%)
[C_N=1] Post	**367ms**	387ms(+5.44%)	388ms(+5.72%)
Position: Top			
[C_N=100] Pre	**395ms**	407ms(+3.03%)	**19406ms**(+4812.91%)
[C_N=100] Post	398ms	408ms(+2.51%)	459ms(+15.3%)
Position: Bottom			
[C_N=100] Pre	436ms	473ms(+8.48%)	1204ms(+176.14%)
[C_N=100] Post	435ms	443ms(+1.83%)	446ms(+2.52%)
Position: Top			
[C_N=500] Pre	**545ms**	563ms(+3.30%)	**158208ms**(+28928.99%)
[C_N=500] Post	556ms	572ms(+2.87%)	983ms(+76.79%)
Position: Bottom			
[C_N=500] Pre	788ms	962ms(+22.08%)	7545ms(+857.48%)
[C_N=500] Post	789ms	803ms(+1.77%)	827ms(+4.81%)

Table 1. IMP1 Benchmark Results

that modifies the shared condition. The *consequence event* is the event that is triggered as a consequence of the *main event*. In the motivating example (Figure 3) this could be the rule ensuring the aggregated meter read-out value does not exceed a certain threshold ($R2$). Additionally it could be the rule that ensures that 99% of the read-outs succeed ($R1$). The expected number of occurences for the *consequence event* differ: once per day for $R1$ and equal to the number of "read-out meter end" events for $R2$. In order to observe this scaling effect we submit the *consequence event* with differing numbers $C_N \in \{1, 100, 500\}$. Thus $R1$ would be similar to the case $C_N = 1$. The *main event* is submitted with $N = 1000$. Another dimension to observe is the difference in the order of the event streams. In *pre*-order we will submit all events of type *consequence event* first followed by the *main event* instances. The inverse *post*-order would inverse the event submission process: *main event* first, followed by the *consequence event*. Additionally we will observe the effects of triggering the RHS of the *main event* rule which can be the following three cases: $RHS \in \{NOOP, 0\%, 100\%\}$. The $NOOP$ case models the scenario where the RHS does not modify any shared variables. The second $RHS = 0\%$ scenario illustrates the case where the RHS modifies a shared condition, but does not result into further activations due to failure of consistency checks. The last $RHS = 100\%$ would result in full activation up to rule triggering of the *consequence event*. Table 1 shows the benchmark results for IMP1. The values therein are mean values in milliseconds (ms). The bolded and underlined values represent the extremities (lowest and highest) for each cluster. Each cluster of values is grouped by the dimensions: *RHS type*, C_N and *position*. Within each row relative percentage increase from the base ($RHS = Noop$) measurement value is included in parenthesis for better comparison.

The first thing to notice is that for *position* = *top* and $RHS = 100\%$ we have the worst matching time for each cluster. These degraded matching times happen only in *pre*-order and are (1) due to the storing of tokens (partial results) within the beta nodes of *consequence event* rules and (2) due to the number of WMEs stored in the alpha node right below the shared condition node. For each C_N number of events the same amount of WMEs and tokens are stored in the alpha node and beta nodes for the *consequence event* respectively. These are all activated during the submission of the *main event* stream where each of the tokens and facts are iterated for *each main event* instance. The matching complexity for each instance of the *main event* becomes linear to the number of stored WMEs and tokens from the existing *consequence events*, resulting in the degraded matching performance. In the *pre*-order and *position* = *bottom* case the degradation in matching time is softened due to the shorter path to rule activation as the shared node is linked to the last join node. Only the previous token (and not all condition facts) are considered for rule activation. But still this single check is performed N times equal to the *main event* stream. In the *post*-order case, the only WMEs that are activated are those that match the single token stored in the shared variable. Essentially, the N times *main event* activation is reduced as a single token stored in the shared condition's beta nodes. This helps when it is time to match the shared variable with the *consequence event* stream. In real applications events cannot be expected to happen serially blockwise but can come mixed together. In that case the single token stored after RHS modification may not be realistic, but can be simulated by avoiding *consequent event* instances for each N *main event* instances. One way to achieve this is through triggering *consequence events* based on time intervals. Another way to reduce the number of tokens and WMEs is to identify obsolete events that are eligable for pruning.

Concerning the best matching times we can see that for $C_N = 1$ there is not much difference between the two possible positioning options. As C_N scales, the gap between top and bottom for the shared condition node widens. Generally when we know that rules do not affect shared conditions ($RHS = Noop$) as well as do not result into activations due to failure in consistency checks ($RHS = 0\%$) then a top position will fare better regardless of the order of the events being submitted. But once *consequence event* rules are fully activated ($RHS = 100\%$) at each event submission then a bottom position with strict adherence to *post*-order is considerably more stable in terms of matching time. Thus, the probability of a *consequence event* rule being triggered has a considerable affect in the positioning of the shared condition node. Of course, when the *consequence event* stream is relatively small (e.g. $C_N = 1$) then the position of the shared condition node does not affect the matching time as long as *post*-order is maintained.

4.2 Effects of sharing event instance data nodes (IMP2)

When applying the triple structure for representing conditions as well as facts we stumble on a drawback that can be illustrated in Fig. 7. One disadvantage is the need to use several triples for representing a single event instance. These event instances consist of the event type, timestamp for the point in time of occurence,

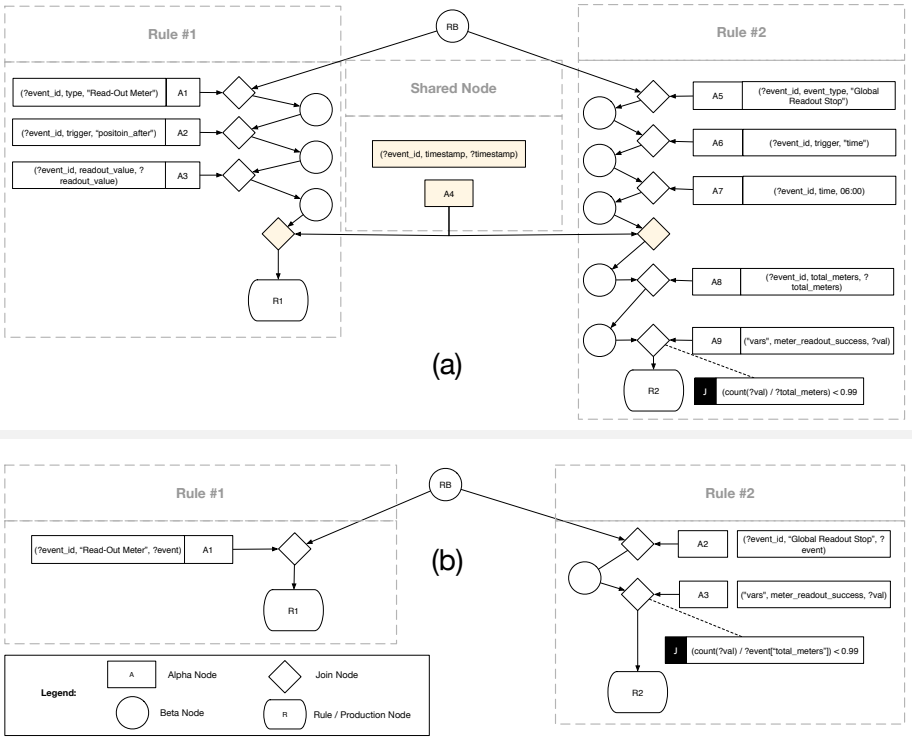


Fig. 7. False sharing of Event Instance Data. (a) uses the triple condition representation which leads to false sharing for the timestamp condition. (b) uses the native event representation, leading to less triples and no event instance data false sharing.

as well as a variable number of parameters that are tied to the event. Each of these elements have to be represented as triples, tied together by a common $?event_{id}$:

$$\begin{aligned}(?event_{id}, \text{"type"}, \text{"readout meter"}) \\ (?event_{id}, \text{"timestamp"}, ?timestamp) \\ (?event_{id}, \text{"value1-N"}, ?value1-N)\end{aligned} \tag{3}$$

The drawback materializes in terms of prolonged matching time, as each node activation has a non constant cost associated with it, amplified through sharing of conditions. This cost is relative to the number of child nodes and the therein stored number of tokens as well as WMEs. Unnecessary matching time can be avoided by having the Rete structure identify opportunities to break off activation for avoiding subgraph traversal. But this is not always possible. The second disadvantage is related to the first: the usage of triples, specifically for representing timestamps. Each timestamp consists of this triple:

$$(?event_{id}, \text{"timestamp"}, ?timestamp) \tag{4}$$

The requirements for representing a timestamp are the following: the value part of the triple is a variable as each incoming fact representing a timestamp is not constant. Each timestamp is associated with a $?event_{id}$ to map it to the correct event instance. We need to include *timestamp* as attribute to properly find the correct timestamp attribute for the event instance. But observe how this fact representation results into false sharing in Fig 7. Due to the hash table optimization for the alpha network [2], triples of this form are shared among all other conditions. Thus for a set of event ids: $\{event_1, event_2, event_3\}$ only a single alpha node is generated to hold all event instance timestamps. This false sharing results in a number of activations and consistency checks among all event instances: as a new fact $(event_4, \text{"timestamp"}, \text{"2016-05-05T10:39:00"})$ enters the rule engine, it will check consistency for $event_{id}$ with all existing tokens $\{event_1, event_2, event_3\}$. Thus event instance additions become linear in time relative to the number of existing event instances. These consistency checks are unnecessary as we know all instance data are relevant solely to the current event instance.

One way to avoid this false sharing can be achieved by defining the timestamp condition to be the following triple:

$$(?event_{id}, \text{"timestamp\_"} + ?event_{id}, ?timestamp) \tag{5}$$

This solution appends the event id to the attribute, which would create a seperate alpha node for each event instance. Unfortunately this solution won't work, as the Rete structure is compiled in advance before submitting an event stream. Most applications of the rule engine won't know in advance which event ids will be submitted to the engine, unless a statically set total number of events is defined which may limit the rule engine's applicability for different domains. Thus another solution would natively implement event objects inside the value part of the triple. In section 3 we define the value part of the triple to be a JSON compatible data structure. Due to false sharing of event instances, we add support for event objects as well. As such, a condition holding such an event instance for the event types *readout meter* is mapped with the following triple:

$$(?event_{id}, \text{"readout meter"}, ?event) \tag{6}$$

Each condition listening to event conditions is represented as a single triple with the $?event_{id}$ representing the event instance's unique id, the second parameter representing the event's type and the value part a variable holding the actual event instance. An example event object can be seen in Listing 1.1.

Listing 1.1. Event Object Instance

```
event_t event = event_t();
event.type = "readout meter";
event.trigger = POSITION_AFTER;
event.timestamp = "2016-05-05T10:39:00";
event.data["value1"] = 147;
```

In this way, unique alpha nodes are created for each event type and each ?*event* variable inside the value part of the triple holds the actual event instance. Constant as well as consistency checks are performed among the attributes of these event instances.

IMP2 Benchmark and Analysis To observe the effects of IMP2 we have setup the following benchmark, for which the results can be seen in Fig. 8. We implement both versions of representing events. First the triple-based method is used where each event's attribute is realized as a triple for both defining conditions as well as submitting facts as WMEs. Secondly, the native method is used where we add support for event objects as possible values inside triples. Therefore, event attributes are aggregated inside this object. We setup an event stream (size $N = 1000$) appropriate for both methods and submit them to the rule engine incrementally. Additionally, we scale the number of existing events $\{0, 10, 100, 200, 250, 500, 750, 1000\}$ allowing us to observe the change in matching time as tokens are added to the Rete structure before the event stream is actually submitted.

We define two rules which follow Equation 3 with conditions for event *type*, *timestamp* and an arbitrary value *value1*. The first rule listens for the type "readout meter" and the second rule is triggered for the type "global readout end". As can be seen in Fig. 8(a) the linear matching time for the triple-based method is indeed a problem. This is especially apparent once more tokens come into play, each representing an attribute for an event. The linear matching time occurs due to the join node's requirement to check for consistency among the tokens (e.g. that the $event_{id}$ is equal between tokens). The effect can be seen in the amount of activations being performed as well, where the join node is leading in number (cf. Fig. 8(c)). At one extreme where $N = 1000$ and existing events $= 0$ we can observe 7000 join node activations. For each incoming event there are 7 join node activations for the following reason: 1 *right-activation* for each incoming event *type* fact and 2 *right-activations* for each incoming *timestamp* as well as 2 *right-activations* for each incoming *value1* facts. The latter two attributes are shared conditions among the rules and are thus submitted once for each rule. The remaining two *left-activations* occur when the $event_{id}$ consistency check passes between the *type* and *timestamp* tokens as well as between the *timestamp* and *value1* tokens. When a secondary event of type "global readout end" is submitted, the equal amount of 7 join node activations are triggered. This explains the 14000 join node activations for the case $N = 1000$ and existing events $= 1000$. In contrast to the triple-based method, the native method stays nearly constant in matching time as no consistency checks are required among the event attributes. With the native method the number of join node activations are equal to the total number of events being submitted.

4.3 Heuristics for performant ISC event matching

In this section we summarize the analysis for the benchmarks performed for IMP1 and IMP2 (cf. Figure 9). We categorize these heuristics by their origin (IMP1 or IMP2) as well as the type of the heuristic. *Static* heuristics are those

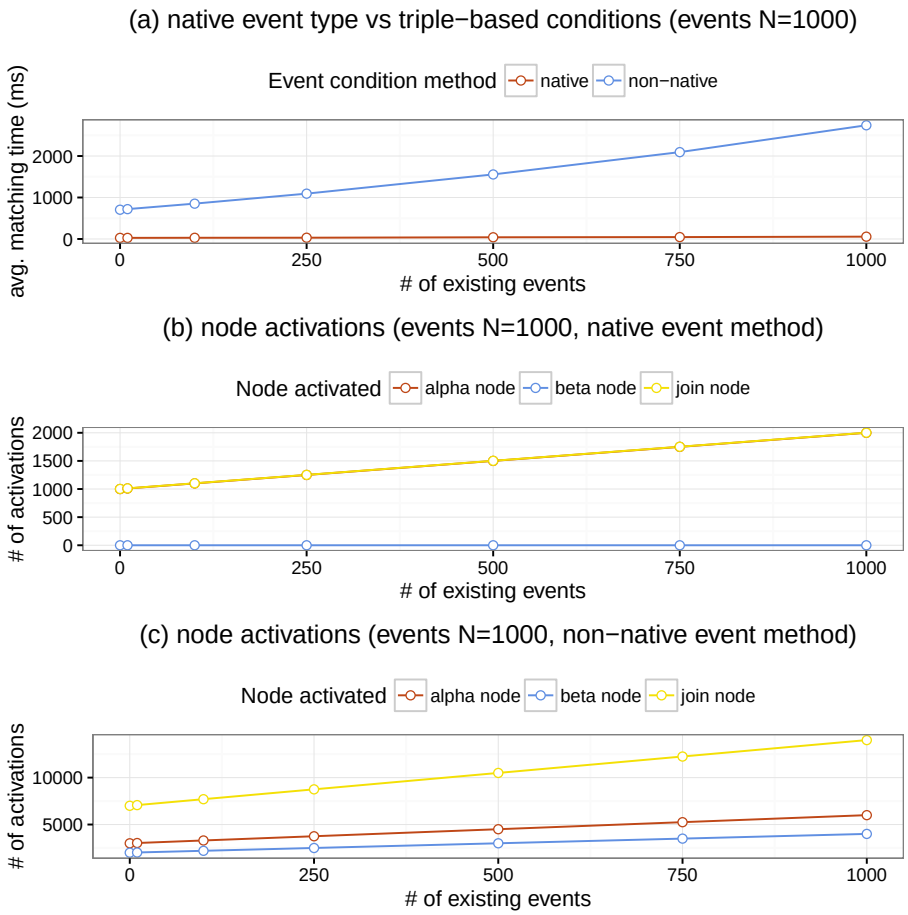


Fig. 8. IMP2 Benchmark Result

that are applicable at Rete network construction time. *Dynamic* heuristics are those that are only applicable during runtime of the rule engine and may require information that are only available then. Some heuristics can be applied in both states and are marked *static/dynamic*. The *condition* column states the precondition that need to be met before applying the heuristic. IMP1 deals with the positioning of shared condition nodes, the timing for emitting *consequence event* instances, as well as the purging of obsolete events which store partial results in the form of tokens as well as WMEs in the shared condition node.

5 Related Work

Checking the compliance of business process with constraints has been researched well. Monitoring constraints over process executions constitutes one task next

Heuristic	Type	Condition	Description
IMP1-1	Static / Dynamic	IF post-order of consequence event is ensured	Place shared condition nodes to position = top
IMP1-2	Dynamic	IF order of main and consequence events are mixed	Emit consequence events at specific time intervals to avoid frequent matching
IMP1-3	Static / Dynamic	IF probability of consequence event rule being triggered is low (RHS near 0%)	Place shared condition nodes to position = top
IMP1-4	Static / Dynamic	IF probability of consequence event rule being triggered is high (RHS near 100%)	Place shared condition nodes to position = bottom
IMP1-5	Static / Dynamic	IF $C_N > 1$ and scales further	Prefer position = top
IMP1-6	Dynamic	IF # of tokens and WMEs in shared condition nodes increases beyond threshold	Identify and purge obsolete events to reduce number of partial results (tokens inside beta nodes)
IMP1-7	Dynamic	IF # of tokens and WMEs in shared condition nodes increases beyond threshold	Restructure to position = bottom
IMP2-1	Static	IF need to represent concepts that do not require matching between their attributes.	Aggregate these concepts as single value instances (e.g. Events and Fluents)

Fig. 9. Summary of heuristics extracted from IMP1 and IMP2

to design time or post mortem compliance checking [14,1] and guaranteeing compliance-by-design [7]. For compliance monitoring the survey [8] provides an overview on existing approaches and compares them based on Compliance Monitoring Functionalities (CMFs). When it comes to compliance monitoring mostly ISC have been neglected yet [8]. Hence, the approach at hand tackles an open gap. It is related to MobuConEC [11] as both approaches use EC. However, MobuConEC is stated of not being fully data-aware and operable yet [11]. Moreover, it is not evident whether and how MobuConEC supports ISC.

This work employs and extends the Rete algorithm proposed in [4]. Several extensions for Rete have been suggested. [2] introduces Rete/UL which extends Rete with an unlinking strategy for avoiding *null activations*. These are activations where it is known that either the beta node does not hold tokens or the alpha node contains no WMEs. In such cases, it is possible to *unlink* the connection between the alpha network and the beta network to avoid unnecessary activations. The benchmarks we have shown through IMP1 handles the cases where too many tokens and WMEs cause degradation in matching time and show heuristics to avoid the worst cases. As such, we can still augment the unlinking strategy to further improve the case where $RHS = 0\%$ (where the activation of a shared condition node does not lead to actviations).

6 Conclusion and Outlook

In this work we have tackled the problem of realizing a monitoring engine specifically for matching Instance-spanning constraints (ISCs). We base our work on Event Calculus (EC) and show the feasability of mapping EC to Rete Networks (RQ1) (cf. Section 3.2). Further we conducted experiments in regards to the Rete Structure to identify aspects to improve matching performance and extracted these insights as heuristics (RQ2, RQ3) (cf. Section 4). As future work we would like to implement further ISC examples collected in [13] with the hopes of extracting more heuristics. Based on the heuristics IMP1-2, IMP1-6, and IMP1-7 (cf. Figure 9) we can tackle more dynamic runtime data aspects to

identify more ways for improving pattern matching performance. Furthermore, we will show the applicability of the heuristics on complex rule definitions.

References

1. Van der Aalst, W., Adriansyah, A., van Dongen, B.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 2(2), 182–192 (2012)
2. Doorenbos, R.B.: Production Matching for Large Learning Systems. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1995), uMI Order No. GAX95-22942
3. Fdhila, W., Gall, M., Rinderle-Ma, S., Mangler, J., Indiono, C.: Classification and formalization of instance-spanning constraints in process-driven applications. In: International Conference on Business Process Management 2016 (2016)
4. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. 19(1), 17–37 (1982), http://dx.doi.org/10.1016/0004-3702(82)90020-0
5. Garbe, C., Peris, K., Hauschild, A., Saiag, P., Middleton, M., Spatz, A., Grob, J., Malvehy, J., Newton-Bishop, J., Stratigos, A., et al.: Diagnosis and treatment of melanoma: European consensus-based interdisciplinary guideline. European Journal of Cancer 46(2), 270–283 (2010)
6. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Generation Computing 4(1), 67–95 (1986)
7. Lohmann, N.: Compliance by design for artifact-centric business processes. Inf. Syst. 38(4), 606–618 (2013)
8. 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. Inf. Syst. 54, 209–234 (2015), http://dx.doi.org/10.1016/j.is.2015.02.007
9. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: Analyzing vessel behavior using process mining. In: Situation Awareness with Systems of Systems, pp. 133–148. Springer (2013)
10. Mangler, J., Rinderle-Ma, S.: Rule-based synchronization of process activities. In: 13th IEEE Conference on Commerce and Enterprise Computing, CEC 2011, Luxembourg-Kirchberg, Luxembourg, September 5-7, 2011. pp. 121–128 (2011)
11. Montali, M., Maggi, F.M., Chesani, F., Mello, P., van der Aalst, W.M.P.: Monitoring business constraints with the event calculus. ACM TIST 5(1), 17 (2013), http://doi.acm.org/10.1145/2542182.2542199
12. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA. pp. 287–300 (2007)
13. Rinderle-Ma, S., Gall, M., Fdhila, W., Mangler, J., Indiono, C.: Collecting examples for instance-spanning constraints. Technical report, arXiv.org (2016), http://eprints.cs.univie.ac.at/4634/
14. Sadiq, S., Governatori, G., Namiri, K.: Modeling control objectives for business process compliance. In: International conference on business process management. pp. 149–164 (2007)
15. Schulte, S., Schuller, D., Steinmetz, R., Abels, S.: Plug-and-play virtual factories. IEEE Internet Computing 16(5), 78–82 (2012)