Evolution of Instance-Spanning Constraints in Process Aware Information Systems

Conrad Indiono*, Walid Fdhila**, and Stefanie Rinderle-Ma*

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

Abstract. Business process compliance has been widely addressed resulting in works ranging from proposing compliance patterns to checking and monitoring techniques. However, little attention has been paid to a specific type of constraints known as instance spanning constraints (ISC). Whereas traditional compliance rules define constraints on process models, which are checked separately for each instance, ISC impose constraints that span multiple instances. This paper focuses on ISC evolution and its impact on process compliance. In particular, ISC change operations, as well as change strategies are defined, and the impact on both the ISC monitoring engine and the process instances during run time are analyzed. The concepts are prototypically implemented.

1 Introduction

Constraints imposed on business processes evolve constantly, for example, when new constraints are added or existing constraints are updated. Whereas business process changes have been investigated in detail (e.g., [18,19]), constraint changes – also in interplay with process changes – have lacked attention so far. Some approaches address the impact of business process change on constraint checking [12], but concepts for the evolution of the constraints and the impacts on other constraints and business processes are missing. This holds particularly true for so called instance-spanning constraints (ISC), i.e., constraints that span multiple instances of one or several business processes [4]. In this work we revisit change strategies as formulated for business process evolution, i.e., versioning, migration, and clean state [1] for ISC changes and investigate the related change impacts along the following research questions:

RQ1 How to define atomic ISC change operations?
RQ2 How can ISC changes be handled ($\mapsto$ change strategies)?
RQ3 How to visualize and measure the impact of ISC changes?
RQ4 How to realize versioning in the context of ISC change?

Answering the questions is challenging due to the complexity of the ISC: ISC consist of structural patterns referring to the underlying processes and conditions concerning data, time, and resource aspects [14]. Whereas these parts can be also found for intra-instance constraints, i.e., independent constraints that can

be verified for each process instance in a separated way, ISC can also contain trigger and action parts. These parts define the "active" components of an ISC such as putting the instance execution to a suspend state for synchronization [13,4]. Moreover, ISC might "share" data and resources. Hence, changing one ISC might affect other ISC even over different versions. Finally, ISC change and impact have to be considered during both, design and runtime.

This work tackles RQ1 – RQ4 as follows: At first, ISC change operations are defined ($\mapsto$ RQ1) in accordance with business process change patterns [17] and constraint changes proposed in literature [10], i.e., ISC change operations for adding, deleting, and updating ISC are proposed. The complexity of ISC adds several elements to defining change operations as each of the parts concerning structure, data, time, resources, and trigger/actions might be adapted. Established process change strategies such as versioninig, migration, and clean state are transferred to ISC change ($\mapsto$ RQ2). This is also connected with a motivation on which ISC formalism and inference techniques can be selected, in this work Event Calculus and RETE. The impact of ISC changes on the ISC themselves as well as on associated process instances is systematically studied at an abstract level and implemented using the RETE matching algorithm ($\mapsto$ RQ3). As the field of ISC evolution is entirely new, a first ISC versioning algorithm is proposed and a prototype proof of concept is presented ($\mapsto$ RQ4).

2 Preliminaries

While process instances reflect the actual execution of a business process, ISC are means to define and enforce restrictions over multiple process instances. As an illustrative example, we use the process scenario of Fig. 1 [8]. It depicts an integrated energy management solution to deliver end-to-end advanced metering. Assume that the provider aims at ensuring that 99% of all readouts (of different instances) are performed within 6 hours and the aggregate read out value does not exceed x. This constraint is considered as an ISC since it imposes restrictions over multiple instances. The ISC meta model follows the IUPC structure [14].

Definition 1. ***[ISC]*** *An ISC is defined as a tuple ISC(context$^+$, connection$^+$, condition*, behavior$^+$) where:*

- *a context refers to the process/process instances subjected by an ISC.*
- *a connection defines the events required to check an ISC (e.g., activity started or time trigger event).*
- *a condition is a constraint on either resources, time or process/rule data.*
- *a behavior is an action triggered when all conditions are met (e.g., suspend a task, resume a task).*
- $^+$ *denotes an at-least-one, while* * *denotes an optional quantifier*

Definition 2. ***[Event]*** *An event represents the occurrence of an action (e.g., execution of a process task) at a given time and is defined as a tuple* (*event_id*, *type*, *timestamp*, *payload**) *where type is the event name, timestamp is the time of its occurrence and payload holds the event related data.*

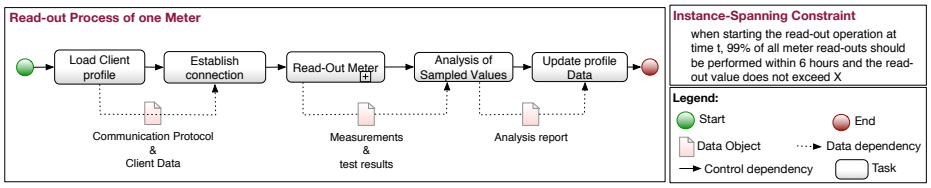

Fig. 1. Process Example from the Energy Domain, taken from [8]

Example 1 ***[Event]*** $(id, readout\_meter, time, meter, readout\_value)$ *is an event of type* $readout\_meter$ *that provides the value of the readout of a given meter.*

Example 2 ***[ISC]*** *The ISC described in Fig. 1 can be defined as follows:* **ISC(Context(**$readout\_process$**), Connection(**$readout\_meter$, $global\_readout\_start$**), Condition(**$total\_readout > x$**), Condition(**$at\ global\_readout\_start.t + 6$ *less than* 99% *of total meters are finished* **), Behavior (***send alert***)).** *The event* $(id, global\_readout\_start, t)$ *is a time trigger to launch all read-out instances at the same time* t*.*

The ISC representation employed in this paper is simplified and needs to be converted into a rule engine language in order to be executed and monitored [8] by an ISC monitor. ISC formalization using, e.g., Event Calculus (EC) is proposed in [4,8]), which we implement on top of a Rete rule engine acting as the ISC monitor. EC is a logic programming approach to model time and change. It uses first order predicate logic (FOL) as the basis and introduces fluents and domain-independent predicates to assert fluents for the ability to model time-varying state. Sect. 4.2 gives an overview of ISC implementation based on EC.

3 Atomic ISC Change Operations

ISC changes can range from deleting an ISC attribute (e.g., condition or connection) to adding or updating new or existing attributes respectively. Similar to process change operations [17], three main change operation groups can be identified, i.e., delete, add, and update. Each of these groups include various change operations with different impacts on both the process instances and the ISC monitor, cf. Def. 3.

Definition 3 (Change Operations).

$$
\begin{aligned}
Change\_operation ::= {} & Operation\_type(ISC, Component) | Delete(ISC) \\
& | Add(ISC, Context^{+}, Connection^{+}, Condition^{*}, Behavior^{+}) \\
Operation\_type ::= {} & Delete | Add | Update \\
Component ::= {} & Context | Connection | Condition | Behavior
\end{aligned}
$$

While the first change operation acts on a specific attribute of an existing ISC using one of the three change types (i.e., Delete, Add and Update), the second change operation deletes an entire ISC and, the third adds a new ISC. Based on Fig. 2 we selectively illustrate some *delete* change operations, while

DELETE	DELETE - BEFORE	DELETE - AFTER
CONTEXT	For both **gas** and **electricity,** 99% of all meters should be readout within 6 hours.	For **electricity,** 99% of all meters should be readout within 6 hours.
CONNECTION	At **12:00** and **14:00**, the average readout of all meters should have a value less than X	At **12:00**, the average readout of all meters should have a value less than X
CONDITION	When starting the read-out at 00:00, 99% of all meters should be read out within 6 hours and **readout value should not exceed X**.	When starting the read-out of 00:00 values, 99% of all meters should be read out within 6 hours.
BEHAVIOR	For 100 (simultaneous) ad hoc readouts , if 10 meter checks exceed 6 hours then send an alert and **stop the readouts.**	For 100 (simultaneous) ad hoc readouts , if 10 meter checks exceed 6 hours then send an alert.
ISC	When starting the read-out of 00:00 values 99% of all meters should be read out within 6 hours.	-
ADD	**ADD - AFTER**	**ADD - BEFORE**

Events: ***e1*** (id, Readout_meter ,time, meter, value) - ***e2*** (id, GlobalReadoutStart ,time)

Fig. 2. Change Operations: Delete and Add Examples

omitting *add* operations due to space constraints. Note that the action part is considered as "send alert" by default.

Delete Context A context represents the process model to which an ISC refers. Monitoring an ISC requires execution events of the corresponding process. Multiple contexts might be defined within the same ISC. Deleting a context implies that all related events are no longer required for its monitoring, and automatically removes all corresponding linkages, i.e., (context, connection). In Fig. 2 (CONTEXT), the ISC has changed from considering both processes of electricity and gas meter readouts respectively, to only electricity.

Delete Connection A connection refers to the events necessary for checking an ISC. Within a single ISC, multiple connections can be specified, which might refer to different contexts, i.e., events from different process executions. Therefore, deleting a connection reduces the number of event types to be checked by an ISC. Note that deleting all connections of a same context implies the deletion of the latter. In Fig. 2 (CONNECTION), it is checked at both times 12:00 and 14:00 whether the aggregate readout value of all meters is less than the threshold x. Deleting trigger time 12:00 means that the monitor will still receive the readout events, but will check the threshold condition at 14:00 only.

Delete Condition An upcoming event or set of events, in order to be considered for an ISC, needs to match its conditions. Conditions are defined as constraints on the data associated with the events; e.g., process data, resources, time. Multiple conditions might be combined for the same ISC, and therefore, deleting a condition releases a restriction on the events used for the ISC. In Fig. 2 (CONDITION), the condition on the threshold value was removed, which means that the latter will not be checked when the change becomes effective.

Delete Behavior As aforementioned, a behavior is an action that is executed if an ISC has fired; e.g., a stop or wait task. Deleting a behavior reduces the number of actions to be enacted when all conditions evaluate to true. In Fig. 2 (BEHAVIOR), the action stop is removed. Consequently, only the action send alert is executed when the ISC fires. This might have impacts on the process instances that are stopped before the change. Compensation actions might be required in order to continue those instances.

4 ISC Evolution Approach

This section introduces the general approach for performing ISC Evolution, starting with an overview of the concepts applied.

4.1 General ISC Evolution Strategies

Inspired by the strategies for process evolution in [1], we introduce three general change strategy approaches for managing ISC evolution. These are versioning, migration, and clean state. Fig. 3 illustrates the differences between these strategies by comparing how the same change sequence is handled in each case. The illustrative case shows how an initial set of ISC, i.e., $\{ISC1_0.ISC2_0\}$ are changed over time, abstracting from the concrete atomic change operation applied. At time t $ISC1_0$ is changed into $ISC1_1$. At time $t+1$ there are three changes: (1) the previously changed $ISC1_1$ transformed to $ISC1_2$, (2) the change of $ISC2_0$ to $ISC2_1$, and (3: only in the versioning case) the original $ISC1_0$ change to $ISC1_3$. Critical to the evolution of ISC is state management, where state refers to any part of an *ISC(context, connection, condition, behaviour)* being shared among different ISC. For example, state $S1$ is the state shared between two ISC: $ISC1_0$ and $ISC2_0$, which in the running example (Fig. 1) might represent the aggregate read-out value from all smart meters, e.g., parts of the condition attribute of both ISC. State management is performed differently among the three strategies.

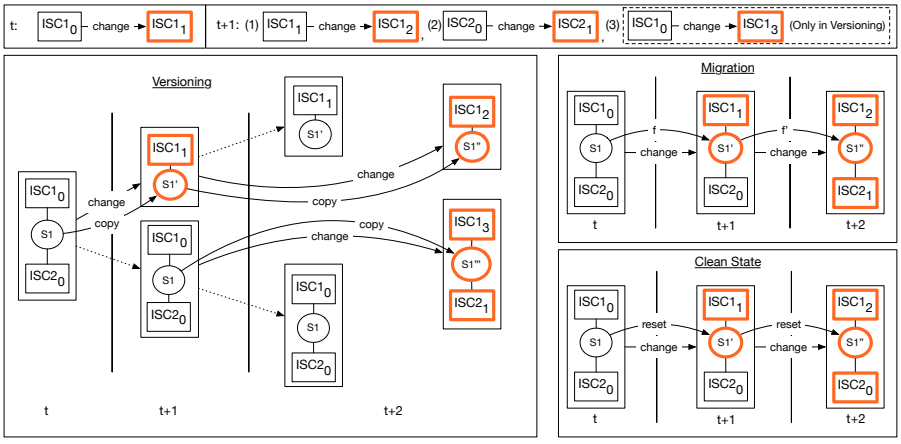

Fig. 3. General ISC Evolution Strategies

Migration In the migration case, a domain-specific abstract transformation function f is applied to $S1$ at time t. A concrete function f needs to be determined on a case by case basis for each change scenario. Furthermore, in this case f is bound by the change of $ISC1_0$ to ISC_1, as well as by the actions that $ISC1_0$ has already performed. In the latter case, compensation actions inverse to the original ones may need to be executed. An example could be in a medical

scenario where 10 patients are pre-approved for a novel operation, where new test results suggest the requirement to reevaluate these patients. The change from $ISC1_0$ to $ISC1_1$ represents the change in verification of the approval process. The concrete function f in this case reduces the pre-approved 10 patients using the now correct approval logic, e.g., 6 pre-approved patients. Similarly, at time $t+1$ f' is applied for migrating state $S1'$ to $S1''$, which is bound by both changes $ISC1_1$ to $ISC1_2$ and $ISC2_0$ to $ISC2_1$. In that way, the migration transformation function f and f' are domain and case dependent.

Clean State The clean state strategy can be seen as an instance of the migration strategy where the function f is fixed as the function *reset*. This state management function, as the name implies, resets all the information within a state to their default values, which depend on their data types. In the running example (Fig. 1), this could be the resetting of the aggregate read-out values to the default value: 0. This *reset* function does not depend on the changes being performed, but only on the data types being modified within the states. This makes the *reset* function domain and case independent.

Versioning The versioning strategy also has a domain and case independent state transformation function: *copy*. Also unique to the versioning strategy is the concept of namespacing ISC and associated states, to allow the differing ISC versions to coexist. At time t, the versioning strategy creates a new namespace for each changed ISC to occupy, together with any associated state. For example, at time t, the change of $ISC1_0$ to $ISC1_1$ leads to the creation of a new namespace for $ISC1_1$ and its associated copied state $S1'$. Notice that the original namespace spanning $ISC1_0$, $ISC2_0$, and $S1$ are still maintained, representing the previous version of $ISC1$. At t+1: while case (1) leads to the usual namespacing for the new $ISC1_2$, cases (2) and (3) lead to the creation of a common namespace due to the common state $S1'''$. Namespacing is directly tied to the state being copied, establishing a separate context for the newly changed ISC, allowing them to be processed independently from coexisting ISC versions. It can be imagined how the versioning approach can be merged with the other two approaches. For example, after establishing the new namespace for $S1'$ at time t, a domain and case dependent transformation function f could be applied to the *copied* state $S1'$ to further customize the state.

4.2 ISC Implementation Overview

We now focus the discussion on a concrete ISC evolution implementation. As depicted in Fig. 4, there are two views on the implementation of ISC: (1) the formalism view and (2) the inference technique view. In [4] we have conducted an extensive analysis of various formalisms in regards to applicability of expressing ISC. From that analysis we have chosen Event Calculus due to its ability to reason on events as well as facts over time, i.e., fluents, and its expressitivity by extending domain-specific functions to cover all components of an ISC (cf. Def. 1). For the purpose of runtime checking using EC, we have adapated the RETE algorithm [8] as the inference technique, specifically due to its forward chaining, reactive nature of dealing with events. Additionally, the visualization of the RETE network helps with understanding the impact of the atomic ISC change

operations. Alternatives to RETE are available, e.g., in the form of Mobucon EC [15], which uses an embedded prolog inference technique combined with cached fluents for improved performance. In this case, EC is compiled to embedded prolog as the target language. Alternatively, CEP implemented via Drools could be employed. While rule management and offline versioning is supported through a plugin (Guvnor), runtime versioning is not available due to lack of state management. In all cases, the general algorithms proposed in this paper need to be specialized for the chosen inference technique to support runtime ISC evolution.

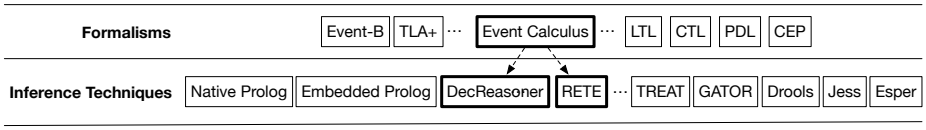

Fig. 4. Formalisms and Inference Techniques

Internally, a RETE rule engine represents an ISC in a specific structure (e.g., Rete graph), which reflects all its attributes (e.g., connections, conditions, behavior). While a process instance represents the actual execution of a process model (within the process engine), an ISC instance represents the actual execution of an ISC structure (within the rule engine). This implies that the conditions and fact evaluations (e.g., aggregate readout value) of an ISC define its state at run-time, i.e., ISC instance. Each event occurrence might change the state of an ISC instance and consequently the evaluation of a condition or a variable over time. An ISC instance terminates when all its actions are enacted. Changing an ISC not only impacts its structure (i.e., static impacts), but also the related process instances as well as the ISC instance itself (i.e., dynamic impacts).

Additionally, ISC can share state (i.e., conditions, connections, or contexts). This means that deleting a connection or condition from an ISC does not necessarily result in its removal from the ISC monitor as it can still be used by another ISC. Similarly, adding a new condition to an ISC does not necessarily result in its implementation in Rete as it might have been already implemented by another ISC that shares the same condition. Deploying a new ISC does not always result in a completely new structure (e.g., Rete graph) inside the monitor. The monitor will only add those parts of the structure that did not exist before.

Another challenging problem is that multiple ISC are running simultaneously, and consequently changing an ISC might cause conflicts with other ones. Therefore, it is necessary to ensure that no conflicts among ISC are generated as a result of a change. Furthermore, ISC generally refer to events related to task executions or process context data at run-time. As such, it is primordial to check whether a change affects the ISC compliability (cf. [3]) with the corresponding process models (e.g., referring to events that are not produced by the process execution). As discussed in a previous work [4], ISC can be checked at both design and runtime. While the former focuses on verifying ISC compliability with process models as well as detecting conflicts between multiple ISC, runtime checking aims at identifying ISC violations by monitoring execution events. A priori checking of ISC using Event Calculus (EC) requires both process models

and ISC to be transformed into EC and then fed into an EC solver (e.g., DecReasoner[1]) to detect either conflicting constraints or incorrect specification. As mentioned in [4], design time checking is not always decidable due to loops or quantification over infinite sets (e.g., arbitrary data objects). In this work, we assume that all ISC as well as change operations are specified correctly, which means that the application of a change operation does not result in inconsistencies within the ISC structure, e.g., adding a condition that refers to connections not specified in the changed ISC. Analyzing the correctness of ISC is not the focus of this paper. For more details on how ISC are modeled and checked with EC, the reader may refer to [4].

4.3 Rete-based ISC Monitoring

Having established EC as the chosen formalism and RETE as the inference technique for executing EC, we now introduce the fundamentals of ISC monitoring based on the RETE algorithm [8] on which we specialize and implement the ISC evolution strategies introduced in Sect. 4.1. In this context, the paper shows only the fundamentals of the RETE algorithm for discussing ISC evolution. To see how EC can be mapped to RETE for complete runtime execution, please refer to [8]. The ISC Monitor listens to a stream of events sent by the process engine and performs actions defined by the ISC, e.g., suspend/continue activities. This process follows the knowledge-based world view, where each event structure is deconstructed as individual facts and fed to a knowledge base. From there, the inference engine applies the Rete pattern matching algorithm [6] to derive the updated states based on the newly submitted facts. We use a variation of the Rete algorithm as outlined in [8] to improve the matching performance for ISC. An ISC is *fired* when all of its conditions match some subset of facts from the knowledge base and the associated behaviour is to be executed. This behaviour can consist of one or many actions and may affect the process engine (e.g., when suspending a running process instance). Figure 5 shows a sample Rete network representing parts of the ISC (cf. Fig. 1) for verifying that (1) all meters are read out within six hours from the read out event starting at 00:00 and (2) the aggregated value of these read out values does not exceed a certain value. A Rete network consists of three main components: knowledge base, alpha network and beta network.

Knowledge Base The knowledge base collects the set of facts previously submitted in the form of events sent by a governing process engine. These facts, also called *working memory elements* (WME) are defined as follows.

Definition 4 (Fact). *A fact is an element that is decomposed from an event structure (cf. Def. 1):* (*id*, *attribute*, *value*)*, where value represents the value of attribute and* (*id*, *attribute*) *is a composite key.*

An event occurrence might change the valuation of one or multiple facts. For example, an incoming event holding a single *readout_value* as the payload of a single meter, might affect the fact representing the aggregated value: *accumulated_values*. We define the *id* part to be a tuple of (event_{id}, instance_{id}).

[1] http://decreasoner.sourceforge.net

Whereas the event_{id} is a globally unique identifier for the specified event (e.g., completion of activity "Readout Value"), the instance_{id} is a globally unique identifier referring to a process instance.

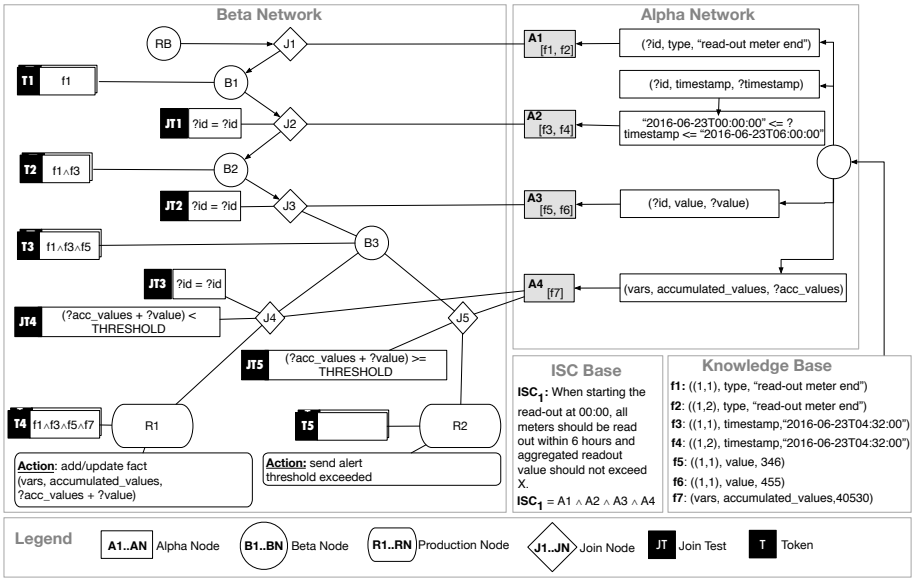

Fig. 5. ISC Monitor: Inside view of the Rete Network (adapted from [8])

Alpha Network The alpha network is a projection network to match facts from the knowledge base and store them inside *alpha nodes*. A pattern is associated to each alpha node where the triple structure for facts is reused for matching: (*id const* | *?id*, *attribute const* | *?attribute*, *value const* | *?value*). Each part of the triple structure can be either a constant value or a variable. The latter is marked with a prepended *?* to its name. For example, the triple (*?id*, type, "read-out meter end") is a pattern that matches all facts having *any* value inside the *id* part, the exact value *type* in the *attribute* part and the exact value "read-out meter end" inside the *value* part. Thus a fact ((event_1, instance_1), type, "read-out meter end") represents the event which has been emitted by the process instance id = 1 and event id = 1 where the activity "read-out meter" has completed. In Fig. 5 this fact would be stored as *f1* in the alpha node *A1*.

Beta Network The beta network is responsible for joining together facts that match certain conditions. This is accomplished by *join nodes*, each one connecting a single alpha node with its parent beta node. Attached to join nodes can be an arbitrary number of *join tests*, matching facts stored in alpha nodes with tokens stored in beta nodes. The default join test behaviour is to check whether the *id* part of the facts contained in the beta and alpha nodes are equal, i.e., $(\text{event}_{id}^{alpha}, \text{instance}_{id}^{alpha}) = (\text{event}_{id}^{beta}, \text{instance}_{id}^{beta})$. This enables

joining together the string of facts originating from the same instance as well as event, which is simplified as $?id = ?id$ in Fig. 5. Arbitrary complex join tests at this level can be employed to relate two facts in various ways. For example, time-based comparison operators would be employed here. In the case of a successful join test, tokens referencing the matching WMEs are stored to all children beta nodes, allowing subsequent join tests to be performed. Following this series of successful join tests until the end will lead to the firing of *production nodes* which contain the sequence of actions that need to be executed.

4.4 Change Strategy: Versioning

We now discuss the specialization of the ISC evolution strategies using the RETE-based ISC monitor. In Sect. 4.1 we have discussed the differences between the three ISC evolution strategies, as well as the necessary operations required to conduct state management. In this section we will specialize the versioning change strategy and propose a concrete versioning algorithm, as well as define proper state management for RETE-based ISC. First we discuss the role of the time of change t_c. Even after employing the necessary versioning techniques (namespacing and state copy), events need to be relayed to the correct version of the ISC at runtime. For this purpose, the time of change t_c becomes critical and necessitates the concept of a **router**, that routes events to the correct ISC instance. Given two ISC: ISC_old and ISC_new, versioning aims at keeping both in place at the same time. Notably, process instances started before t_c are monitored through ISC_old, while the instances started after t_c are checked against ISC_new. Thus, it is important for the ISC monitor to have the information about each instance start time. The latter helps correlating future instance events with the appropriate version. Indeed, each event includes information to which process instance it belongs, i.e., instance identifier. Using the latter in combination with the event related to instance start time, it becomes possible to find out whether the event belongs to an old or new instance.

Definition 5 (Shared Variable and Shared ISC). *A variable is an alpha node that follows the fact structure* (*id*, *attribute*, *value*) *(cf. Def. 4). A* shared variable *is a variable that satisfies one of the following conditions:*

Condition 1: Given two ISC Δ and Δ' with variables $v \in V$ and $v' \in V'$, v is a shared variable *iff $v.id = v'.id \wedge v.attribute = v'.attribute \wedge v.value = v'.value$. In this case Δ and Δ' are also* shared ISC.

Condition 2: Given a function $\pi : (var, action) \mapsto Bool$, which returns true when a variable var is modified by action, an ISC Δ with variables $v \in V$ and actions $\alpha \in \mathcal{A}$, then any variable where $\pi(v, \alpha) = true$ holds is also a shared variable *due to it being modified in the behaviour part of an ISC.*

Isolation of versioned ISC *Shared variables* become a source of complexity when versioning is considered. Imagine an ISC (ISC_old) from the energy domain (cf. Fig. 1), implementing the readout example, which alerts whenever a certain threshold is exceeded. Furthermore, consider a new version of the ISC (ISC_new), which changes the threshold value to be higher before triggering an alert. The usage of the same *shared variable* causes both versions to be evaluated, and in consequence, both ISC_old and ISC_new could be triggered, which is not the

intended behaviour. Since we want all process instances that have started after the time of change ($> t_c$) to fall under ISC_new, we can introduce isolation of *shared variables* to achieve the intended behaviour. This isolation can be conducted by creating a copy of the original *shared variable* and storing it under a unique *namespace* intended for ISC_new. *Namespacing* can be realized in Rete by defining the *shared variables* to have a unique prefix in the id part of the fact triple (cf. Def. 4). Furthermore, the production node needs to know which namespace of the *shared variable* it is supposed to access (i.e. when updating the aggregated read out value after each successful readout event). Namespacing thus affects both the alpha nodes (representing the *shared variables*) as well as the production nodes.

Definition 6 (Namespacing of Shared Variables). *Given a* shared variable v, $\eta(v)$ *returns a new variable* v', *such that* $\neg\exists v'' \in V : v''.id = v'.id$ *and thus does not satisfy Condition 1 of Def. 5. Furthermore, any action that modifies a shared variable,* $\alpha \in \mathcal{A} : \pi(v, \alpha) = true$, *needs to be namespaced as well* $\eta(\alpha, v') \mapsto \alpha'$, *such that* $\alpha.id \neq \alpha'.id \wedge v'.id = \alpha'.id$.

Now that all *shared variables* are namespaced uniquely for ISC_new, fact evaluation and ISC triggering happen in isolation from ISC_old. What happens if there is a *shared ISC* (cf. Def. 5), ISC_shared, connected to the same *shared variable*, having unrelated behaviour to both ISC_old and ISC_new? At the current state, due to ISC_new being isolated using its unique *namespace*, anytime ISC_old evaluates the fact related to the *shared variable*, the *shared ISC* will also be affected. ISC_shared needs to independently evaluate the *shared variable* from both ISC_old and ISC_new. Therefore, both ISC_old and ISC_new need to be *namespaced*, leading to three distinct namespaces for the same initial *shared variable*. Creating isolated namespaces for ISC_old and ISC_new raises the question of how the *shared variables* should be initialized. Two options are available: 1) copying the original value of the *shared variable* (default) or 2) applying a custom function f to re-initialize the *shared variable*, which could be just resetting it to the default value, depending on its data type. For example, in the case where the *shared variable* is an integer variable, re-initializing could be set to the default value 0. It is part of the versioning specification to deal with the *initialization* of namespaced *shared variables*.

Definition 7 (Initializing Shared Variable). *Given the set of options* $ops = \{copy, reinit\}$, *a* shared variable s, *a namespaced* shared variable s' *(where* $\eta(s) = s'$*), and a custom transformation function that performs a domain-specific action to re-initialize* s: f, *the initialization of the* shared variable s' *is defined as*

$$init(s, s', op, f) = \begin{cases} s.value = s'.value, & \text{if } op = copy \\ s.value = f(s.value), & \text{otherwise} \end{cases}$$

Preserving previously evaluated facts Namespacing an ISC Δ transforms it into a new ISC Δ' to be added to the Rete graph. Since only one of those ISC two can exist at any one time, the original ISC Δ needs to be removed, while at the same time trying to avoid loss of previously evaluated facts

Algorithm 1: Unbounded Versioning Algorithm for Rete-based ISCs

Input: $\{\Delta_1, ..., \Delta_n\}$, Δ', t_c, op_{old}, op_{new}, f_{old}, f_{new}

Begin
// (1) adjust the last ISC for proper routing based on t_c, if necessary
if $length(\{\Delta_1, ..., \Delta_n\}) = 0$ **then**
  // no previous ISC to adapt, Δ' is submitted as is
  $rete\_add(\Delta')$; **return**
else if $length(\{\Delta_1, ..., \Delta_n\}) = 1$ **then**
  // This is the first time the ISC is versioned, add router condition
  $\Delta_{last} = last(\{\Delta_1, ..., \Delta_n\})$
  $\Delta'_{last} = \Delta_{last}.alpha\_nodes \cup \{?id, instance\_start\_time, ?ist, \{?ist < t_c\}\}$
else if $length(\{\Delta_1, ..., \Delta_n\}) > 1$ **then**
  // Router condition exists, add proper join test to this condition
  $\Delta_{last} = last(\{\Delta_1, ..., \Delta_n\})$
  $\Delta'_{last} = \Delta_{last}.router.join\_tests \cup \{?ist < t_c\}$
// (2) adapt the new ISC for proper routing
$\Delta' = \Delta'.alpha\_nodes \cup \{?id, instance\_start\_time, ?ist, \{?ist >= t_c\}\}$
// (3) Namespacing and Initialization of shared variables (cf. Def. 6 and Def. 7)
$\Delta'_{last}.S = \{init(s, \eta(s), op_{old}, f_{old}) \mid \forall s \in \Delta_{last}.S, s' \in \Delta'.S : s' = s\}$
$\Delta'_{last}.A = \{\eta(\alpha, s) \mid \forall \alpha \in \Delta'_{last}.A, s \in \Delta'_{last}.S : \pi(s, \alpha) = true\}$
$\Delta'.S = \{init(s, \eta(s), op_{new}, f_{new}) \mid \forall s \in \Delta_{last}.S, s' \in \Delta'.S : s' = s\}$
$\Delta'.A = \{\eta(\alpha, s) \mid \forall \alpha \in \Delta'.A, s \in \Delta'.S : \pi(s, \alpha) = true\}$
// (4) ISC Fact evaluation preserving change from Δ_{last} to Δ'_{last}(cf.Def.8)
$rete\_add(\Delta'_{last})$; $safe\_delete(\Delta_{last}, \Delta'_{last})$; $rete\_add(\Delta')$

(i.e. tokens inside the beta nodes). Simply removing the original ISC Δ causes all related nodes within the beta network to be removed, including the tokens which represent the previously evaluated facts. A safe delete of the original ISC Δ needs to be performed which ensures that such tokens shared by both Δ and Δ' are not removed when deleting Δ.

Definition 8 (Safe Deletion of an ISC). *Given an ISC Δ and a shared ISC Δ^s, a $safe\_delete(\Delta, \Delta^s)$ operation ensures that the structurally shared beta and join nodes, which hold the previously evaluated facts (i.e. tokens) remain intact when deleting Δ. A safe delete can be realized by recursively deleting from the bottom production node (representing the ISC Δ), up to the root beta node, while only deleting those nodes which do not reference Δ^s.*

ISC Versioning Algorithm We are now able to define an ISC Versioning Algorithm (cf. Algorithm 1) that utilizes the Rete graph structure for unbounded versioning of an ISC. The algorithm deals with the three previously discussed aspects: (1) utilizing the *router* concept to detect which version of the ISC should handle the incoming event, (2) *Namespacing* (cf. Def. 6) and the subsequent *Initialization* (cf. Def. 7) of *shared variables* to isolate the different versions of an ISC, and (3) the preservation of previously evaluated facts (i.e. tokens) using *safe delete* (cf. Def. 8). As input, the algorithm requires the list of previous versions of the ISC $\{\Delta_1, ..., \Delta_n\}$, the modified ISC itself Δ', where $\gamma(\Delta_n) \mapsto \Delta'$ is assumed, the time of change t_c and the type of operation for initializing the shared variables op_{old} and op_{new}. Whereas op_{old} is the operation to be performed

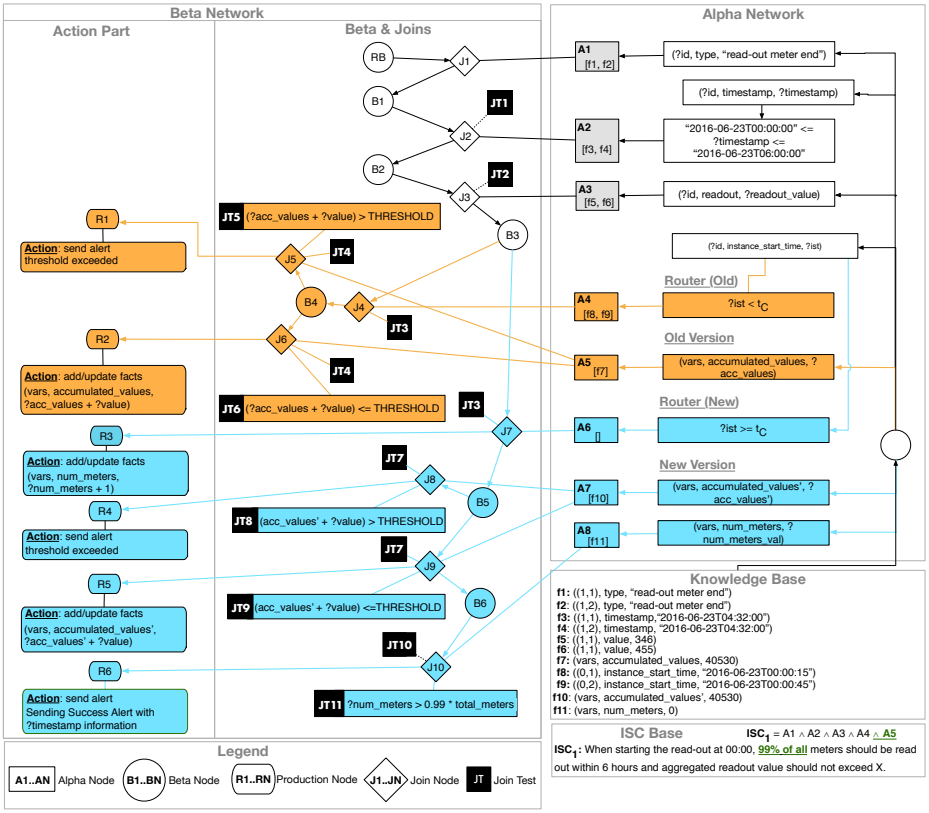

Fig. 6. Change Impact Evaluation: Adding Condition + Versioning

when initializing the *shared variables* in Δ_n, op_{new} is responsible for initializing Δ'. Both are assumed to be one of $\{copy, reinit\}$ (cf. Def. 7).

For illustrating the algorithm, we take the original ISC defined in Fig. 5, which does not include the 99% of all smart meters condition, and apply it using the *add condition* operation combined with the *versioning* change strategy. This change is visualized in Fig. 6. The color of the nodes identifies the associated ISC: white being the common path for shared attributes (i.e, same context, connection and condition). Orange nodes and edges represent the old ISC, whereas the blue elements representing the new ISC.

Lines 1 - 16 of Algorithm 1 deal with the *router* aspect (1). Here three subcases are handled. The simplest case is where Δ' has never been versioned before, signaled by an empty $\{\Delta_1, ..., \Delta_n\}$. In this case neither routing, nor isolation of shared variables, nor the preservation of previous facts are necessary. In the case where Δ' represents the first versioning event of Δ_n, the router needs to be setup for the first time. This is accomplished by adding the pattern *(?id, instance_start_time, ?ist)* to the alpha nodes and linking it to both Δ_n and Δ'. As previously mentioned, the process instance's *starting time* is a unique discriminator to identify which ISC version is responsible for handling an event.

Here Δ_n is picked for dealing with cases where the process instance's *starting time* is smaller than t_c, otherwise Δ' is responsible for handling the event. The appropriate *join tests* are added to the routing pattern (e.g. $\{?ist < t_c\}$ for Δ_n, or $\{?ist >= t_c\}$ for Δ'). In the case where the routing pattern already exists, which happens when the versioning algorithm has been applied once, then the same routing pattern can be reused in the alpha nodes. Only the new *join tests* are added in that case. Applying the algorithm recursively in this fashion allows unbounded versioning of the same ISC. In the illustration, the routing behaviour is reflected in the creation of the alpha nodes A4, representing old process instances whose instance start time is $< t_c$, and correspondingly A6, representing new process instances started after t_c. On the dynamic impact level, we trigger a reindexing process where facts are matched for the newly created alpha nodes A4 and A6. We assume that there is an event with $\text{event}_{id} = 0$ that registers the process instance's start time, which in Fig. 6 are transformed to facts $f8$ and $f9$ representing the two process instances goverened under ISC_old. There are no facts yet matching A6 for ISC_new.

The second aspect is concerned with the isolation of *shared variables* (lines 17-22) through *namespacing* and *initialization*. *Namespacing* both the *shared variable* for Δ', as well as for Δ_n, ensures that all *shared ISC* are independent of each other allowing new process instances after t_c to trigger independently from those before t_c. Noticeable here is that A5 is reused from the old ISC, responsible for maintaining the *shared variable* of accumulated readout values for ISC_old. For maintaining the state for ISC_new, A7 is created as the result of *namespacing* the shared variable *accumulated_values* by *copying* its value to a new shared variable *accumulated_values'*. Additionally for ISC_new, a new shared variable is introduced (A8) to maintain the actual number of meters being read out. On the dynamic impact for this part of the alpha network, the facts $f10$ and $f11$ are initialized and matched to A7 and A8 respectively. Whereas $f10$ is a simple copy of an existing shared variable (*accumulated_values*), $f11$ is initialized as the counter 0. From this point on the two *shared variables accumulated_values* and *accumulated_values'* diverge in processing of subsequent facts and represent the ISC instances of ISC_old and ISC_new respectively, effectively isolating the two different ISC versions.

5 Technical Evaluation

We implemented the concepts introduced in this paper as a prototypical proof-of-concept, extending the ISC Monitor [8][2]. For conducting this technical evaluation we followed the methodology outlined in Figure 7 in order to tackle research questions RQ2 to RQ4. Concretely, we (1) analyse the correctness of the ISC versioning algorithm, (2) observe the effectiveness of $safe\_delete$ for avoiding costly reindexing of facts during ISC versioning and (3) highlight the change impact on the Rete graph aggregated by change operation type.

Collected Dataset In [4] and [20] we collected ISC examples from five different domains (i.e., health care, security, transport / logistics, manufacturing

[2] The full source code, as well as supplementary material can be found under http://gruppe.wst.univie.ac.at/projects/crisp/index.php?t=iscevolution

and energy). Some parts of these example scenarios have been simulated on the Cloud Process Execution Engine (CPEE) [21] and subsequently logged as event logs in the Extensible Event Stream (XES) format. Through this collection phase we can already map the common ISC types related to the domain. Consequently, the most likely change operation for the ISC operating under certain domains can be classified. For example, 53% of the collected ISC examples are classified as single-context, meaning that for those cases an *add context* operation is appropriate.

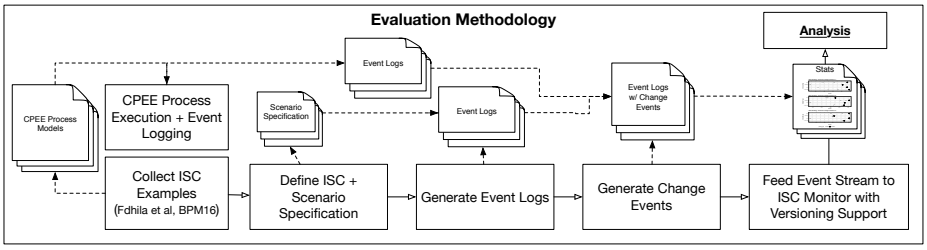

Fig. 7. Evaluation Methodology

Scenario Specification In addition to the modeled examples, we aim to extend the number of event logs to test multitude of realistic scenarios in various domains. For this purpose, we have implemented a *scenario generator* that takes as input a *scenario specification*. The latter is a formal specification of the scenario, which is used to generate event logs. The specification allows the realistic definition of structural control flow as well as data elements. In addition, care has been taken to generate realistic timestamps which follow the known probabilistic distribution from collected scenarios. In this fashion we have specified in total 10 scenarios, each varying in the number of process instances being generated [10, 100, 1000, 10K]. Along with the *scenario specification* we associate the corresponding ISC that governs the monitoring.

Generate Change Events In order to study the impacts of ISC changes, as well as verify the change process, we embed change events into the event logs. We perform this embedding step randomly over the event logs, following the probability distribution of change operations that could occur depending on the scenario domain (e.g. a change on condition referring to *resource* occurs only very rarely in scenarios in the energy domain. Most ISC in that domain refer to *execution data*). The change events are not completely random, as to avoid invalid change operations. We thus specify the valid base of change operations per scenario, and limit the randomization based on these operations. To ensure an equal distribution of ISC_old and ISC_new events, we automatically pick t_c for each scenario based on the generated event stream, such that the difference in number of events between the ISC versions remains within a max. deviation of 5%.

	ADD Cond.	ADD Conn.	ADD Context	Avg.
N=**1000**				
(**M2**)[avg. evaluation time $< t_c$ in ms]	0	0	0	0
(**M2**)[avg. evaluation time $>= t_c$ in ms]	2	14	17	11
(**M3**)[avg. evolution time in ms ($\triangle$/†)]	$39^{\triangle}/71^{\dagger}$	$43^{\triangle}/64^{\dagger}$	$48^{\triangle}/73^{\dagger}$	$43^{\triangle}/69^{\dagger}$
(**M4**)[change impact on alpha/beta/join]	2/1/2	5/5/6	6/6/7	4.33/4/5
(**M4**)[change impact on tokens ($\triangle$/†)]	$83^{\triangle}/332^{\dagger}$	$83^{\triangle}/332^{\dagger}$	$83^{\triangle}/332^{\dagger}$	$83^{\triangle}/332^{\dagger}$
	DEL Cond.	DEL Conn.	DEL Context	Avg.
N=**1000**				
(**M2**)[avg. evaluation time $< t_c$ in ms]	1	1	1	1
(**M2**)[avg. evaluation time $>= t_c$ in ms]	27	10	9	15.33
(**M3**)[avg. evolution time in ms ($\triangle$/†)]	$48^{\triangle}/71^{\dagger}$	$41^{\triangle}/64^{\dagger}$	$42^{\triangle}/65^{\dagger}$	$43^{\triangle}/66^{\dagger}$
(**M4**)[change impact on alpha/beta/join]	6/5/6	4/1/2	5/2/3	5/2.66/3.66
(**M4**)[change impact on tokens ($\triangle$/†)]	$83^{\triangle}/175^{\dagger}$	$83^{\triangle}/332^{\dagger}$	$83^{\triangle}/332^{\dagger}$	$83^{\triangle}/279.66^{\dagger}$

$\triangle$=with $safe\_delete$, †=without $safe\_delete$

Table 1. Representative Metrics for ISC Versioning from the energy domain

Feed event streams to the ISC Monitoring Engine Finally, the generated event streams for each domain are fed into the Rete-based ISC Monitoring Engine, which implements the proposed ISC versioning algorithm. Whenever a change event is met, the versioning algorithm is applied. Any other event is fed normally for processing. Key metrics (cf. Table 1) are determined per domain allowing us to analyse various aspects of the algorithm. Additionally, we also track (**M1**) the actual number of ISC activations for each version to ensure correctness of the ISC versioning algorithm.

Analysis: ISC Versioning Algorithm Correctness. The first question to tackle is the correctness of the ISC versioning algorithm. Towards that goal we split up three different ISC sets to the same event stream for a given t_c. The first set (**S1**) consists solely of the original ISC (ISC_old), with the additional pattern that it only handles process instances with *starting time* $< t_c$. Similarly, the second set (**S2**) consists of ISC_new handling instances with *starting time* $>= t_c$. The third set (**S3**) utilizes the ISC versioning algorithm to maintain both ISC_old and ISC_new. **M1** confirms that the number of ISC activations in **S1** equals the ISC_old activations of **S3**, and the same for **S2** for ISC_new in **S3**. This confirms that the *router* logic, *namespacing* and *initialization* logic of *shared variables* work correctly as intendend.

Analysis: Change Impact by change operation type. Table 1 shows the collected metrics **M2**-**M4** for scenarios within the energy domain. Generally we can observe that the lowest cost (in terms of evaluation time, evolution time and number of nodes affected) is by performing an *ADD Condition* operation, which only affects a single alpha node (twice due to *namespacing*). The next level in complexity is the operation *ADD Connection*, which can be explained due to the necessity of adding facts related to another activity in the same event stream. Finally, *ADD Context* can be interpreted as an additional alpha node, representing the new process model, being added on top of a *ADD Connection*. The effectiveness of utilizing $safe\_delete$ in terms of evolution time and reduced fact reindexing can be seen through metrics **M3**+**M4**. **M3** shows that the aver-

age evolution time *with safe_delete* for both *add* and *delete* is nearly 37% faster compared to the variant *without safe_delete*, i.e., 43ms vs 69ms. Similary, only 25% of the tokens need to be processed when using *safe_delete*, compared to without, i.e., 83 vs 332 (**M4**).

6 Related Work

Change and evolution in PAIS have been research topics for many years, focusing on the definition, soundness, and realization of process schema and instance changes [17]. The impact of process schema and instance changes on compliance constraints has been addressed in [12] by reducing the effort of compliance verification to those compliance constraints that are affected by the changes. Different change scenarios have been considered ranging from ad hoc instance changes to process evolution with concurrent instance changes. Compliance of constraints after ad hoc instance changes has been also subject to the work presented in [11]. Here three states can be distinguished after an instance change, i.e., valid, partially valid, and invalid. Changes of compliance constraints for central processes have been addressed in [10]. The work describes a unified compliance management framework that also considers changes of constraints (adding, changing, and deleting). Some approaches address change and compliance in distributed processes. In [9], algorithms are proposed in order to detect the effects of changing private processes on global compliance rules. [9,5] follows up by stating challenges for the evolution of collaborative processes and their compliance rules. In a similar direction, [2] aims at finding alignments between the monitoring of business networks and compliance at the presence of change at both levels. If a change occurs, a set of actions is determined in order to maintain the ability to monitor the business network. Despite the interest in studying the impacts of process and compliance change, evolving ISC and their impacts on processes and vice versa have not been considered yet. In general, supporting ISC in PAIS is a emerging topic where some approaches have dealt with ISC selectively in the area of, for example, batching [16] or security [22]. In [4], the relevance and modeling of ISC has been addressed in a general way. [8] has proposed techniques for checking and monitoring ISC. A visual notation for ISC has been introduced in [7]. However, none of these approaches has dealt with evolving ISC and processes.

7 Conclusion

This paper addressed ISC evolution in process aware information systems. In particular, it identifies atomic ISC change operations (RQ1) and revisits general change strategies for ISC evolution (RQ2). On the basis of EC as the ISC formalism and Rete as the implementation technique, ISC versioning has been concretely discussed in detail, implemented and evaluated (RQ3+RQ4).

Acknowledgment This work has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT15-072.

References

1. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data Knowl. Eng. 24(3), 211–238 (1998)

2. Comuzzi, M.: Aligning monitoring and compliance requirements in evolving business networks. In: On the Move to Meaningful Internet Systems. pp. 166–183 (2014)
3. Fdhila, W., Knuplesch, D., Rinderle-Ma, S., Reichert, M.: Change and compliance in collaborative processes. In: Services Computing (2015)
4. Fdhila, W., Gall, M., Rinderle-Ma, S., Mangler, J., Indiono, C.: Classification and formalization of instance-spanning constraints in process-driven applications. In: Business Process Management. pp. 348–364 (2016)
5. Fdhila, W., Rinderle-Ma, S., Knuplesch, D., Reichert, M.: Change and compliance in collaborative processes. In: Services Computing. pp. 162–169 (2015)
6. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. 19(1), 17–37 (1982)
7. Gall, M., Rinderle-Ma, S.: Visual modeling of instance-spanning constraints in process-aware information systems. In: Adv. Information Systems Eng. (2017)
8. Indiono, C., Mangler, J., Fdhila, W., Rinderle-Ma, S.: Rule-based runtime monitoring of instance-spanning constraints in process-aware information systems. In: On the Move to Meaningful Internet Systems. pp. 381–399 (2016)
9. Knuplesch, D., Fdhila, W., Reichert, M., Rinderle-Ma, S.: Detecting the effects of changes on the compliance of cross-organizational business processes. In: Conceptual Modeling. pp. 94–107 (2015)
10. Koetter, F., Kochanowski, M., Renner, T., Fehling, C., Leymann, F.: Unifying compliance management in adaptive environments through variability descriptors. In: Service-Oriented Computing and Applications. pp. 214–219 (2013)
11. Kumar, A., Yao, W., Chu, C.: Flexible process compliance with semantic constraints using mixed-integer programming. INFORMS Journal on Computing 25(3), 543–559 (2013)
12. Ly, L.T., Rinderle, S., Dadam, P.: Integration and verification of semantic constraints in adaptive process management systems. Data Knowl. Eng. 64(1), 3–23 (2008)
13. Mangler, J., Rinderle-Ma, S.: Rule-based synchronization of process activities. In: Commerce and Enterprise Computing. pp. 121–128 (2011)
14. Mangler, J., Rinderle-Ma, S.: IUPC: identification and unification of process constraints. CoRR abs/1104.3609 (2011), `http://arxiv.org/abs/1104.3609`
15. Montali, M., Maggi, F., Chesani, F., Mello, P., van der Aalst, W.: Monitoring business constraints with the event calculus. ACM Transactions on Intelligent Systems and Technology 5(1), 17 (2013)
16. Pufahl, L., Herzberg, N., Meyer, A., Weske, M.: Flexible batch configuration in business processes based on events. In: Service-Oriented Comp. pp. 63–78 (2014)
17. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer (2012)
18. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer (2012)
19. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems - a survey. Data Knowl. Eng. 50(1), 9–34 (2004)
20. Rinderle-Ma, S., Gall, M., Fdhila, W., Mangler, J., Indiono, C.: Collecting examples for instance-spanning constraints. Tech. Rep. arXiv:1603.01523, arXiv (2016)
21. Stuermer, G., Mangler, J., Schikuta, E.: Building a modular service oriented workflow engine. In: 2009 IEEE International Conference on Service-Oriented Computing and Applications (SOCA). pp. 1–4 (Jan 2009)
22. Warner, J., Atluri, V.: Inter-instance authorization constraints for secure workflow management. In: Symposium on Access Control Models and Technologies. pp. 190–199 (2006)