Classification and Formalization of Instance-Spanning Constraints in Process-Driven Applications

Walid Fdhila(✉), Manuel Gall, Stefanie Rinderle-Ma, Juergen Mangler, and Conrad Indiono

Faculty of Computer Science, University of Vienna, Vienna, Austria
{walid.fdhila,manuel.gall,stefanie.rinderle-Ma,
juergen.mangler,conrad.indiono}@univie.ac.at

Abstract. In process-driven applications, typically, instances share human, computer, and physical resources and hence cannot be executed independently of each other. This necessitates the definition, verification, and enforcement of restrictions and conditions across multiple instances by so called instance-spanning constraints (ISC). ISC might refer to instances of one or several process types or variants. While real-world applications from, e.g., the logistics, manufacturing, and energy domain crave for the support of ISC, only partial solutions can be found. This work provides a systematic ISC classification and formalization that enables the verification of ISC during design and runtime. Based on a collection of 114 ISC from different domains and sources the relevance and feasibility of the presented concepts is shown.

Keywords: Instance-spanning constraints · Compliance · Process-Aware Information Systems

1 Introduction

Checking and enforcing constraints such as regulations or security policies is the key concern of business process compliance [29]. Enterprises have to invest significantly into compliance projects, e.g., for large companies $4.6 million only for the management of internal controls [31]. BPM research has provided several solutions for compliance at design time, e.g., [6] and runtime (cf. survey in [15]). Despite these large efforts, an important type of constraints has not been paid sufficient attention to, i.e., *Instance-Spanning Constraints (ISC)*. ISC are constraints that refer to more than one instance of one or several process types. Logistics is a domain where ISC play a crucial role for the bundling or rebundling of cargo over several transport processes [4]. Other domains craving for ISC support are health care [7] and security [33]. Specifically, in highly adaptive process-driven applications where processes dynamically evolve during runtime [10] ISC provide the means for ensuring a certain level of control.

© Springer International Publishing Switzerland 2016
M. La Rosa et al. (Eds.): BPM 2016, LNCS 9850, pp. 348–364, 2016.
DOI: 10.1007/978-3-319-45348-4_20

ISC support is scattered over a few approaches [7,13,17,18,27,33], but a *comprehensive* support for ISC formalization, verification, and enforcement is missing. Here, the property *comprehensive* refers to the context of ISC such as multiple instances or processes, the expressiveness, e.g., ISC referring to data or time, and the process life cycle phase the ISC is referring to. For a sufficient understanding of these requirements, a systematic classification of ISC is needed. An ISC formalization can then be chosen based on the ISC classification and additional requirements such as complexity of the verification. The following research questions address these needs:

1. *How to systematically classify ISC?*
2. *How to formalize ISC based on ISC classification?*
3. *Do ISC classification and formalization meet real-world ISC requirements?*

Questions 1–3 will be tackled following the milestones set out in Fig. 1. At first, objectives are harvested from literature that must be met by an ISC classification (*Question 1*) and formalization (*Question 2*). The ISC classification will be created as new artifact. The ISC formalization choice (*Question 2*) is based on an analysis of existing languages. Based on an ISC collection of 114 examples from practice, literature, and experience, relevance and feasibility of the ISC classification are evaluated (*Question 3*). Moreover, the ISC formalization will be validated by formalizing and implementing representatives along the provided ISC classification (*Question 3*). In summary, this work provides an ISC classification and formalization as well as an evaluation based on an extensive meta study on ISC examples (cf. [26] for a complete description and all 114 ISC examples).

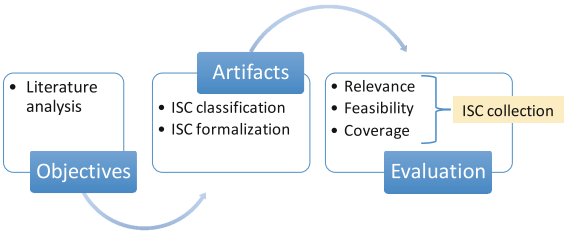


Fig. 1. Milestones following the research methodology in [22]

Section 2 provides ISC objectives and the ISC classification. Section 3 discusses alternatives for formalization languages. In Sect. 4, relevance and feasibility of the ISC classification is evaluated. ISC representatives are formalized and implemented in Sect. 5. Section 6 discusses related approaches and Sect. 7 closes with a summary.

2 ISC Classification

Following the milestones set out in Fig. 1, a collection of objectives on the ISC classification and formalization is harvested from literature. ISC have a strong runtime focus [33] and can thus be estimated as related to compliance monitoring in business processes. In [15], objectives on compliance monitoring have been selected and evaluated as Compliance Monitoring Functionalities (CMF). The CMFs are grouped along *modeling*, *execution*, and *user* requirements. For the ISC classification the focus is at the moment on modeling and execution requirements. User requirements will play an important role later on when investigating feedback options and handling of ISC violations and conflicts. According to [15], modeling and execution requirements are *CMF 1: Constraints referring to time*, *CMF 2: Constraints referring to data*, *CMF 3: Constraints referring to resources*, *CMF 4: Supporting non-atomic activities*, *CMF 5: Supporting activity life cycles*, *CMF 6: Supporting multiple instances constraints*.

Although *CMF 6* suggests the use of CMFs for ISC, the CMF framework does not deal with ISC, but rather with multiple activity instantiations. Hence, we complement the elicitation of objectives by including requirements stated in literature on ISC, i.e., [7,13,17,18,27,33]. These works partly confirm *CMF 1–CMF 6* and extend it by the *context* of a constraint [13,17,18], i.e., whether it refers to a single/multiple processes and/or single/multiple instances. An example for an ISC spanning multiple instances of a single process is a security constraint restricting the loan sum granted by one employee over all her customers [33]. An example for an ISC spanning single instances of multiple processes is imposing an order between two activities of different treatment processes [7].

Concluding, we state as objectives for ISC classification and formalization:

Objective 1: coverage and support of *CMF 1–CMF 3* (*modeling*)
Objective 2: coverage and support of *CMF 4–CMF 6* (*execution*)
Objective 3: coverage and support of *context* single/multiple instances for single/multiple processes
Objective 4: support during design/runtime

Regarding **Objective 4**: ISC might not only become effective during runtime, but also during design time, e.g., imposing restrictions on different process variants and their instances that can be checked during design time, such as static information about roles in a process spanning separation of duty scenario. Thus, support of ISC during design time is added to the objectives.

Figure 2 depicts the proposed ISC classification designed along **Objective 1–4** . **Objective 1** suggests a classification along the modeling requirements time, data and resource. Here, the classification of an ISC into several requirements is conceivable. ISC *A user is not allowed to do t2 if the total loan amount per day exceeds $1M* [33], for example, can be classified as time and data. For a selective classification, ISC should not fit into multiple categories, but be assigned to exactly one category. For this reason, the modeling requirements are grouped into *single* and *multiple* requirements. Multiple modeling requirements describe ISC for which more than one modeling requirement is existing such as in the

example above. An ISC is classified as single modeling requirement if none or one modeling requirement is present. **Objective 2** is not considered for the ISC classification. In turn, the underlying CMFs are relevant for the formalization and for the interplay with a process execution engine which manages task states and multiple instances of a task.

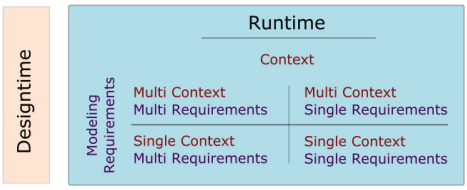


Fig. 2. ISC classification according to objectives.

Objective 3 requires to extend the classification by the spanning property of constraints, e.g., imposing a restriction that must hold across several process instances. In the iUPC logical description [13,17,18,27], for example, the spanning part is described as *context*. ISC can span over processes and/or instances. An ISC is considered *single* spanning if the constraint spans over processes **or** instances and *multi* spanning when the constraint spans across both.

ISC can be enforced during design and run time (**Objective 4**). The proposed ISC classification considers both, but due to the strong runtime focus of ISC design-time will be a single group and run-time is divided into the four classifications provided by modeling requirements and context. A more extensive discussion on design and runtime support of ISC is provided in Sect. 3.1.

3 Analysis of Existing Formalisms for ISC Support

In Sect. 2, we have identified 4 objectives primordial for the classification and formalization of ISC. In the following, we use these 4 objectives to evaluate a list of existing formalisms and compare them to ISC requirements.

3.1 ISC Support During Design and Runtime

We start with a discussion of ISC requirements on verification at design time and runtime (cf. **Objective 4**).

Design time checking aims at verifying the process model compliability with respect to the defined ISC, detecting and resolving conflicts between multiple ISC, and checking the reachable states of the instances with respect to the defined ISC. This might imply generating and combining possible traces to be checked against the ISC. One of the techniques used at design time is model checking. This technique suffers from well known problem of state explosion and is not well suited for checking constraints that refer to runtime data.

Runtime checking becomes necessary as soon as ISC refer to execution data, time, or resources. Moreover, at runtime it is possible to deviate from the original process model, and therefore a monitoring approach to check possible violations becomes primordial. In contrast to design time checking, the process models are not used in the monitoring of constraints (unless for conformance checking), but the runtime events instead. At runtime, we differentiate between two checking possibilities: (i) using partial traces, where events are analyzed against the constraints when they arrive, and (ii) post checking, i.e., using complete traces, which assume that the analyzed instances have completed. ISC span multiple instances. Hence, the fact that an instance or a set of instances satisfy an ISC at the time of their completion does not necessarily ensure that this ISC will not be violated by the executing of future instances, i.e., combined with the completed ones. Consequently, it becomes crucial for ISC monitoring to define correctly the window for analyzing the instances against the constraints.

3.2 Analysis of Formal Languages

In this section, we have analyzed the commonly used formalisms in the areas of business process compliance and concurrent systems as follows.

Event-B is a specification language that describes how the system is allowed to evolve. In particular, it specifies the properties that the system must fulfill [1]. Event-B is mainly used for distributed systems, using artifacts; i.e. blueprints, to reason about the behavior and the constraints of the future system. The main advantage of Event-B is that it allows different level of abstractions through stepwise refinement. Event-B is based on events, expresses the constraints between them, and supports modality; i.e. time operators (CMF 1). In the context of business processes, Event-B has been used for verifying cloud resource allocation and consumption [3] (CMF 2–3).

TLA+ is a syntactic extension of TLA (Temporal logic of Action), a specification language for describing and reasoning about asynchronous, nondeterministic concurrent systems [9]. TLA+ combines temporal logic with logic of action, is suited for reasoning about protocols, and can be used to specify safety and liveness properties. Similarly to Event-B, TLA+ allows different levels of abstraction through refinement.

Both TLA+ and Event-B can be appropriate for specifying and checking ISC at design time. In particular, structural parts of ISC might checked before runtime to detect inconsistencies or incorrect specifications. Both formalisms are very expressive, support time, data and resources (**Objective 1**), and can ensure properties such as liveness, fairness or safety at design time. However, this does not prevent deviations from the specified model at run time. To our knowledge, TLA+ and Event-B are meant to be used for specifying correct and compliant models, but not for monitoring the system properties at run-time; i.e. they does not satisfy **Objective 4**. Both languages are used for distributed and concurrent systems and can support **Objective 3.**

LTL (Linear Temporal Logic) is a formal language, introduced by Pnueli [24], referring to the temporal modality (CMF 1), and used for reactive and

concurrent systems. LTL is an extension of propositional logic, and expresses properties of computation traces; i.e., is interpreted over execution traces. Recently, LTL has been used for modeling and checking compliance constraints of business processes at both design and run-time [2,16] (**Objective 4**). While most of the approaches for design time verification would use a Kripke Structure for model checking LTL properties, some monitoring approaches rely on a transformation of the constraints to a monitor (automata) that evaluates the runtime events. Several extensions of LTL have been proposed to cover other aspects not originally considered. For example, DLTL (Dynamic Linear Temporal Logic) strengthen the UNTIL modality with regular expression of the propositional dynamic logic. Similarly, RTL (Regular Temporal Logic) extends LTL with semi-extended regular expressions, and MLTL extends it with metrics.

CTL (Computation Tree Logic), is a branching time logic that, in contrast to LTL, expresses constraints on dynamic evolution of states rather than traces. Unlike LTL, in CTL the evolution of time is nondeterministic, and every instant of time has several successors, rather than, exactly one [32]. While LTL reasons about events along a single computation path, CTL quantifies over paths that are possible from a given state, through a computation tree. LTL and CTL are not really comparable and have different expressive powers; i.e., there are formula that can be expressed in CTL but not in LTL, and inversely. The strong fairness property, which guarantee a fair behavior between concurrent instances cannot be expressed in CTL. While LTL is better in expressiveness, the problem of model-checking CTL formulae of a Kripke structure is of polynomial complexity [32]. Several extensions of CTL has been proposed; e.g. CTRL extends it with regular expressions [19].

CTL* can express all formulae of both LTL and CTL [32]. However, the problem of model checking becomes P-space complete. While LTL can be used for monitoring, CTL and CTL* are mostly used for model checking at design time (**Objective 4**).

PDL is a dynamic logic with several modalities that extends modal logic by associating action to the operators; i.e.; multimodal logic [5]. It particularly expresses formulae of the form: *after executing an action, it is necessary or possible that the proposition holds*. PDL can also express nondeterministic behavior through regular expressions and compound actions. The complexity of PDL decidability is proved to be in deterministic exponential time which makes it not appropriate for monitoring (**Objective 4**).

μ-Calculus is an extension of modal logic with two operators μ and v corresponding to the least and greatest fixpoints operators [14]. μ-Calculus is a superset of CTL* and PDL, and is also used for the formal verification of concurrent systems. Despite its expressive power, the complexity of model checking systems specified with μ-Calculus is considerably high.

Although CTL* and μ-Calculus are powerful branching-time logics, both of which subsume CTL and LTL (μ-Calculus subsumes PDL as well), they are complex to understand and to use by non-experts [19]. ISC can be conveniently and concisely formulated in terms of regular expressions that are not provided

by standard temporal logics such as CTL and LTL [13,18]. Besides, LTL, CTL* and μ-calculus adopt an inherent qualitative notion of time but when it comes to quantitative time or metrics they become insufficient (CMF 1) [15]. LTL is also not suitable for constraints that deal with data and resources (CMF 2–3), or mult-instances (CMF 6), which are aligned with **Objectives 1–2** of ISC.

EC (Event Calculus) is a general logic programming treatment of time and change [12]. Event calculus is based on first order predicate logic FOL and expresses properties in terms of Fluents. A Fluent is a time-varying property whose valuation is changing according to effect axioms defined in the theory of the problem domain. The time in EC is linear rather than the branching time used in other logics, where time is a tree. Accordingly, Fluent valuation is relative to time points instead of successive situations. EC provides an inherent support for concurrent events [12], where events occurring in overlapping time intervals, from different sources can be deduced (**Objective 3**). EC has benefited enormously from several extensions; e.g. for expressing different properties such as non deterministic actions, gradual changes, compound events, indirect effects, actions with duration or actions with delayed effects [21]. There exist a multitude of reasoners or solvers for EC; e.g. Discrete Event Calculus reasoner, F2LP [21]. EC supports abductive reasoning to generate hypothetical events. In other words, it permits constructing a rule based on the observed events. In the context of business processes, EC has been widely used for either formalizing process models, process choreographies (process interactions) [28], or obligations and compliance rules [20]. As already mentioned in [15,20], EC adopts an explicit representation of qualitative and quantitative time (CMF 1), and supports the CMF 2–6 that we pointed as relevant for ISC checking. Moreover EC supports checking at both design and runtime (**Objective 4**).

Other Languages: In particular, **SQL-like languages** such as PQL or APQL [8] as declarative languages based upon temporal logic seem to be good candidates for expressing complex constraints and querying instance events at runtime. In contrast to the logic based reasoning, they are data-centric and can deal with the CMFs that we have defined. Currently, PQL is used for querying process model instances. Also **eCRG** (extended Compliance Rule Graph) is a visual monitoring language for business process compliance which supports control and data flow including time and resource perspectives [11] (CMF 2–3). eCRG is based on FOL and can be used at both, design and runtime (**Objective 4**).

ISC checking at design-time is not always decidable due to loops or quantification over infinite sets (e.g., time, integer, arbitrary data objects). While the assumption of finite sets is made implicit for LTL and CTL, and therefore they are considered as decidable at design time, it does not hold for more expressive language such as EC. The expressive power of EC precludes its decidability at design time, but meanwhile can cope with most ISC specifications. Since temporal logic properties are decidable over finite-state models, adopting this assumption makes EC also decidable at design time. LTL, CTL, PDL, EC, eCRG and SQL-like languages are all decidable at runtime (monitoring) since they check over traces. However, they have different complexity.

Table 1 elaborates on the above discussion and classifies the studied languages with respect to **Objectives 1–4**. eCRG, SQL-like languages, and EC seem to be good candidates for ISC formalization. SQL-like languages are more data-centric, but remain as a good alternative to support ISC. Overall, EC is adopted for ISC formalization and used as basis for design time checking and runtime monitoring.

Table 1. Evaluation of formalisms with respect to Objectives 1–4

		TLA+	Event B	LTL	CTL	PDL	μ-Calculus	eCRG	SQL-Like	EC
Objective 1		+	+	+/−	+/−	+/−	+/−	+	+	+
Objective 2		+	+	+/−	+/−	+/−	+	+	+	+
Objective 3		+	+	+	+	+	+	+	+	+
Objecjtive 4	Design	+	+	+	+	+/−	+/−	+/−	+/−	+/−
	Runtime	−	−	+	+/−	+/−	+/−	+	+	+

Caption: (Full support (+), Not Supported (−), Partly supported (+/−))

4 Relevance and Feasibility of ISC Classification

114 ISC examples were collected during a meta study described in [26]. Manufacturing, logistics/transport, health care, security and energy/smart grid were identified as relevant application domains which were complemented by other domains such as teaching and insurance during the study. Altogether, 42 % of the ISC examples stem from the energy domain, 16 % from automotive and manufacturing, 10 % from security, 9 % from logistics and transport, 7 % from health care, and 16 % from other domains. Among the analyzed sources were EU and WWTF projects (16 %), regulatory documents (42 %), industry papers (15 %), literature (9 %), as well as ISC examples from experiences; i.e., own working projects (18 %). The complete collection of ISC examples is provided in [26].

In order to show the relevance and feasibility of the ISC classification (cf. Fig. 2), the ISC were manually categorized with respect to the following aspects[1].

- Application: design/runtime
- Context: single/multiple processes/instances
- Modeling requirements: structure, data, time, resource, execution data

Regarding *application*, it can be observed that all examples refer to runtime (except those in category undef). Hence, the classification into *design time* and *runtime* is not reflected by the examples. Nonetheless, ISC examples for design time can be envisaged (e.g., static role assignment), however, the emphasis seems to be ISC support during runtime. *Execution data* [18] can be observed as additional modeling requirement when compared to the CMFs in [15]. *Structure* is present in every ISC (as implicitly also the case for the CMF framework [15]).

[1] Note that ISC for which no categorization was possible without further information were categorized as *undef*. The reason behind is that the ISC in many cases did not have a specified connected process model.

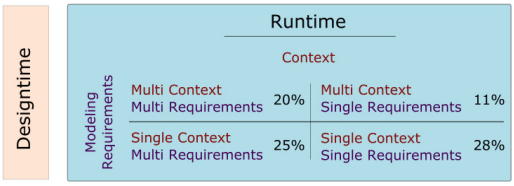
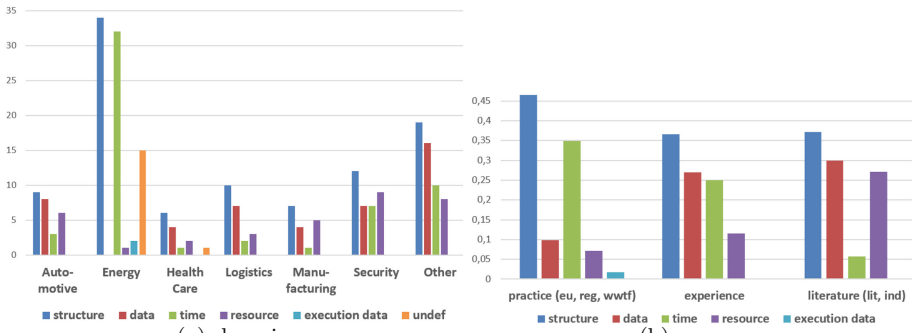


Fig. 3. Distribution of classification

The distribution of the examples with respect to *context and modeling requirements* is depicted in Fig. 3. About 20 % of the examples can be classified as spanning multiple processes, instances, and modeling requirements. 11 % span multiple context and are categorized to fit a single modeling requirement. In total, 53 % of the ISC are classified as single context spanning either processes or instances. 25 % of the ISC in category single context are further categorized as referring to multiple modeling requirements and 28 % as single modeling requirement. 16 % of the examples are not considered due to unclear context (12 %) or missing modeling requirements (4 %). For this data set, each ISC fits exactly one of the classification categories.

To learn more about the modeling requirements, they were plotted against the domain and the source (cf. Fig. 4). *Structure* is a modeling requirement present in every domain (cf. Fig. 4(a)) ranging from about 35 % to 45 %. There are differences for modeling requirements *data*. Specifically, *data* is not present at all for domain energy whereas for the other domains the amount of ISC referring to *data* ranges from about 20 % to 32 %. *Time* plays some role for all domains, but seems to be especially represented for the energy domain (about 38 %) compared to a range from about 6 % to 20 % for all other domains. Looking into the energy examples, many ISC refer to a certain time frame (Service Level Agreements (SLA)). *Resource* is present throughout all domains, again the energy domain shows less ISC referring to *resources* (about 1 %) than the other domains (about 14 % to 29 %). All ISC referring to *execution data* fall into domain energy. *Resources* seem to play a particularly important role in manufacturing and automotive as well as in security. The latter is not very surprising as the assignment of resources is an essential security measure.

For analyzing modeling requirements along source (cf. Fig. 4(b)), it was decided to aggregate sources into categories *practice* (covering projects and regulatory documents), *experience*, and *literature* (covering literature and industry papers) in order to compare practice and research. Industry papers could have also been categorized under practice as these paper mostly describe real-world use cases. Figure 4(b) shows that practice has more emphasis on *time* as literature, whereas literature emphasizes on *resources*. Literature also contains more examples with modeling requirement *data* then the practical examples. Experience seems to balance out modeling requirements from practice, e.g., *time*, and literature, e.g., *data*. Only practice refers to example with *execution data*. One can interpret this as follows: category *practice* is dominated by the energy



(a) domain (b) source

Fig. 4. Modeling requirements for domain and source (normalized, grouped barcharts)

domain where *time* plays an important role. Nonetheless, the rather marginal coverage of *time* by literature in contrast to practice is interesting to look into. Also the practice category introduces *execution data* which has not been considered by literature at all. The experience examples intentionally try to resemble a balanced coverage of all modeling requirements.

To round off the explorative analysis of the ISC collection, the *usage* of the ISC examples was analyzed. [18,27] distinguish categories *compliance, attribution, behavior*, and *meta* where compliance refers to checking certain properties, attribution to, for example, runtime assignments, behavior to enforcement of certain actions during runtime such as synchronization, and meta to constraints defined on other constraints. Figure 5 shows the distribution of ISC example usage for the different domains. For automotive and manufacturing, compliance and behavior are present with an emphasis on compliance. Energy refers to compliance, but no other category (except undef). For health care and security

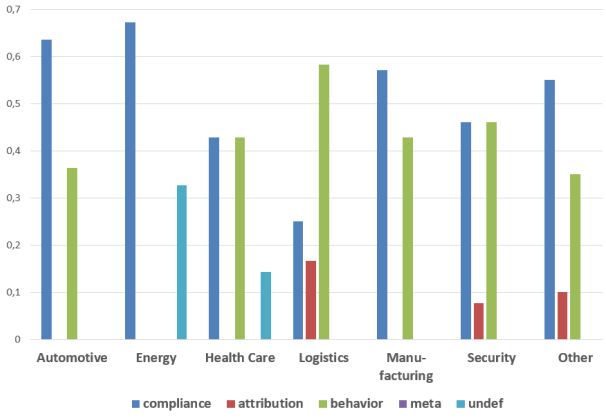


Fig. 5. Usage of ISC examples along domain (normalized, grouped barchart)

compliance and behavior are equally presented, where for health care also some undef cases are present. Logistics, security, and others exhibit also examples for attribution. For logistics the category with highest presence is behavior. Trying an interpretation, automotive and manufacturing show a similar distribution of usage, i.e., compliance and behavior with an emphasis on compliance. For the energy domain, only compliance is present. This can be explained by the sole existence of SLAs in the respective regulatory document which are to be checked rather than to be enforced. For health care and security behavior seems to play an equally important role as compliance because certain regulations are to be enforced or synchronization plays an important role. Security also demands for attribution, e.g., for assignment of roles. Logistics has more demand for behavior (e.g., synchronizing deliveries) and a relatively high demand for attribution.

5 Formalization of ISC Representatives

Preliminaries: As aforementioned, EC is a temporal formalism that can specify properties of dynamic systems in terms of events and the effects of their occurrence on predefined fluents (properties). While fluents are conditions regarding the state of a system, events are occurrence of actions that might change the state of the system and consequently the valuation of the fluents. A typical fluent would indicate that a process variable holds a specific value at a given time. EC mainly defines a set of domain independent predicates, which can be augmented by a domain related predicate. Figure 6 describes a subset of the basic predefined predicates of EC [21]. Specifically, the occurrence of an event e at a time t is represented by the predicate $Happens(e, t)$. This can influence a fluent f by terminating its old valuation that holds until point in time t, and initiating it with a new valuation that holds after t (through the predicates $Terminates$ and $Initiates$ respectively). The reader can refer to [21] for the complete set of domain independent fluents.

As evaluation of the applicability of EC in the context of ISC, we have formalized 4 representative scenarios derived from the ISC classification and implemented them with a reasoner. The scenarios are described in Fig. 7 and refer to the following categories of the ISC classification (cf. Fig. 2): **Scenario 1:** single context/multi modeling; **Scenario 2:** single context/single modeling; **Scenario 3:** multi context/single modeling; **Scenario 4:** multi context/multi modeling.

PREDICATE	MEANING
InitiallyN(f)	f is false at timepoint 0
InitiallyP(f)	f is true at timepoint 0
HoldsAt(f,t)	f is true at time t
Happens(e, t)	e occurs at time t
Initiates(e, f, t)	if e occurs at time t, then f is true and not released from the commonsense law of inertia after t
Terminates(e, f, t)	if e occurs at time t, then f is false and not released from the commonsense law of inertia after t
Release(e, f, t)	if e occurs at time t, then f is released from the commonsense law of inertia after t

Fig. 6. A subset of EC predicates (cf. [21])

SCENARIOS	EVENT CALCULUS	FORMULA
Type: [Single / Multi] "When starting the read-out operation at time t, 99% of all meter readouts should be read out within 6 hours and the read out value does not exceed X."	**EVENTS** **FLUENTS** GlobalReadoutStart() ReadoutFinished(meter) ReadoutEnd(meter, data) Value(counter, value) **STATEMENTS** ∀meter, time, counter, value **Happens**(GlobalReadoutStart(), time) => **Terminates**(GlobalReadoutStart(), ReadoutFinished(meter), time)∧ **Initiates**(GlobalReadoutStart(), Value(counter, 0), time) ∧ **Terminates**(GlobalReadoutStart(), Value(counter, value), time) ∧ ¬(value = 0);	∀time1, time2, meter, data, counter,value **Happens**(GlobalReadoutStart(), time1) ∧ **Happens**(ReadoutEnd(meter, data), time2) ∧ (time2 > time1) ∧ (time2 < time1 + 6) => (data <= X) ∧ **Terminates**(ReadoutEnd(meter, data), Value(counter, value), time2) ∧ **Initiates**(ReadoutEnd(meter, data), Value(counter,value+1), time2) ∧ **Initiates**(ReadoutEnd(meter, data), ReadoutFinished(meter), time2); ∀time, counter, value **Happens**(GlobalReadoutStart(), time) => **HoldsAt**(Value(counter, value), time+6) ∧ (value = n)
Type: [Single / Single] "For 100 (simultaneous) ad hoc readouts of end devices/"activate/deactivate customer interface" readouts/ meter checks, 99 % <= 5 min is required."	**EVENTS** **FLUENTS** GlobalReadoutStart() ReadoutFinished(meter) ReadoutEnd(meter) Value(counter, value) **STATEMENTS** **InitiallyP**(Value(counter,0)) **InitiallyP**(Value(violations,0)) ∀meter, time **Happens**(ReadoutStart(meter), time) => **Terminates**(ReadoutStart(meter), ReadoutFinished(meter), time); ∀meter, time2 **Happens**(ReadoutEnd(meter), time2) => ∃time1 **Happens**(ReadoutStart(meter), time1) ∧ (time2 > time1); ∀meter, time1, time2, counter, violations, value1, value2 **Happens**(ReadoutEnd(meter), time2) ∧ **Happens**(ReadoutStart(meter), time1) ∧ (time2-time1 > 5) ∧ **HoldsAt**(Value(counter, value1), time) ∧ (value1 modulo N > 0) => **Initiates**(ReadoutEnd(meter), ReadoutViolation(meter), time2) ∧ **Terminates**(ReadoutEnd(meter),Value(violations, value2),time2)∧ **Initiates**(ReadoutEnd(meter), Value(violations,value2+1), time2);	∀time, meter, counter,value1 **Happens**(ReadoutEnd(meter), time) ∧ **HoldsAt**(Value(counter, value1), time) ∧ (value1 modulo N > 0) => **Terminates**(ReadoutEnd(meter), Value(counter, value1), time2) ∧ **Initiates**(ReadoutEnd(meter), Value(counter,value1+1), time2); ∀time1,time2, meter, counter , violations ,value1, value2 **Happens**(ReadoutEnd(meter), time2) ∧ **Happens**(ReadoutStart(meter), time1) ∧ (time2-time1 > 5) ∧ **HoldsAt**(Value(counter, value1), time) ∧ (value modulo N = 0) => HoldsAt(Value(violations,value2), time2)) ∧ (value 2 +1 < (99*N/100)) ∧ **Terminates**(ReadoutEnd(meter), Value(violations, value2), time2) ∧ **Initiates**(ReadoutEnd(meter), Value(violations,value2+1), time2); ∀time1,time2, meter, counter , violations ,value1, value2 **Happens**(ReadoutEnd(meter), time2) ∧ **Happens**(ReadoutStart(meter), time1) ∧ (time2-time1 <= 5) ∧ **HoldsAt**(Value(counter, value1), time) ∧ (value modulo N = 0) => HoldsAt(Value(violations,value2), time2)) ∧ (value 2 < (99*N/100)) ∧ **Terminates**(ReadoutEnd(meter), Value(violations, value2), time2) ∧ **Initiates**(ReadoutEnd(meter), Value(violations,value2+1), time2);
Type: [Multi / Single] "A user is not allowed to execute more than 100 tasks (of any workflow) in a day."	**EVENTS** **FLUENTS** TaskStart(user, task) TaskCount(user, value) TaskEnd(user, task) LastTaskDay(user, day) **FUNCTIONS** getday(time) : Day **STATEMENTS** ∀user **InitiallyP**(TaskCount(user,0)); ∀user, task, value, time **Happens**(TaskStart(user, task), time) => **HoldsAt**(TaskCount(user, value), time) ∧ (value < n); ∀user, task, value, day, time **Happens**(TaskStart(user, task), time) ∧ **HoldsAt**(LastTaskDay(user, day), time) ∧ (day = getday(time)) =>	**Terminates**(TaskStart(user, task), TaskCount(user, value), time) ∧ **Initiates**(TaskStart(user, task), TaskCount(user, value + 1), time); ∀user, task, value, day, time **Happens**(TaskStart(user, task), time) ∧ ¬**HoldsAt**(LastTaskDay(user, day), time) => **Initiates**(TaskStart(user, task), LastTaskDay(user, getday(time), time) ∧ **Terminates**(TaskStart(user, task), TaskCount(user, value), time) ∧ **Initiates**(TaskStart(user, task), TaskCount(user, value + 1), time); ∀user, task, value, day, time **Happens**(TaskStart(user, task), time) ∧ **HoldsAt**(LastTaskDay(user, day), time) ∧ (day < getday(time)) => **Terminates**(TaskStart(user, task), LastTaskDay(user, day), time) ∧ **Initiates**(TaskStart(user, task), LastTaskDay(user, getday(time), time) ∧ **Terminates**(TaskStart(user, task), TaskCount(user, value), time) ∧ **Initiates**(TaskStart(user, task), TaskCount(user, 0), time);
Type: [Multi / Multi] "Print similar jobs together."	**EVENTS** **FLUENTS** PrintStart(printer, queuetype) Printing(printer, queuetype) PrintEnd(printer, queuetype) PrintQueue(printer, queuetype, integer) **STATEMENTS** ∀printer, queuetype **InitiallyP**(PrintQueue(printer, queuetype, 0)) ∧ **InitiallyN**(Printing(printer, queuetype)); ∀printer, queuetype1, queuetype2, integer, time **Happens**(PrintStart(printer, queuetype1), time) ∧ ¬**HoldsAt**(Printing(printer, queuetype1), time) ∧ ¬**HoldsAt**(Printing(printer, queuetype2), time) ∧ (queuetype1 != queuetype2) => **HoldsAt**(PrintQueue(printer, queuetype1, integer), time) => **Initiates**(PrintStart(printer, queuetype1), Printing(printer, queuetype1), time) ∧ **Initiates**(PrintStart(printer, queuetype1), PrintQueue(printer, queuetype1, integer + 1), time); ∀printer, queuetype1, queuetype2, integer, time **Happens**(PrintStart(printer, queuetype1), time) ∧ ¬**HoldsAt**(Printing(printer, queuetype1), time) ∧ **HoldsAt**(Printing(printer, queuetype2), time) ∧ (queuetype1 != queuetype2) => **HoldsAt**(PrintQueue(printer, queuetype1, integer), time) => **Initiates**(PrintStart(printer, queuetype1), PrintQueue(printer, queuetype1, integer + 1), time);	∀printer, queuetype, integer, time **Happens**(PrintStart(printer, queuetype), time) ∧ **HoldsAt**(Printing(printer, queuetype), time) ∧ **HoldsAt**(PrintQueue(printer, queuetype, integer), time) => **Terminates**(PrintStart(printer, queuetype), PrintQueue(printer, queuetype, integer), time) ∧ **Initiates**(PrintStart(printer, queuetype), PrintQueue(printer, queuetype, integer + 1), time); ∀printer, queuetype, integer, time **Happens**(PrintEnd(printer, queuetype), time) ∧ **HoldsAt**(Printing(printer, queuetype), time) ∧ **HoldsAt**(PrintQueue(printer, queuetype, integer), time) => **Initiates**(PrintEnd(printer, queuetype), PrintQueue(printer, queuetype, integer - 1), time); ∀printer, queuetype1, queuetype2, integer, time **Happens**(PrintEnd(printer, queuetype1), time) ∧ **HoldsAt**(Printing(printer, queuetype1), time) ∧ ¬ **HoldsAt**(Printing(printer, queuetype2), time) ∧ (queuetype1 != queuetype2) ∧ **HoldsAt**(PrintQueue(printer, queuetype1, 0), time) ∧ **HoldsAt**(PrintQueue(printer, queuetype2, integer), time) ∧ (integer > 0) => **Terminates**(PrintEnd(printer, queuetype1), Printing(printer, queuetype1), time) ∧ **Initiates**(PrintEnd(printer, queuetype1), Printing(printer, queuetype2), time);

Fig. 7. ISC scenarios based on [26] and formalized using EC

Scenario 1: The scenario is taken from the energy domain and adapted from the engergy domain, and states that when starting the readout operation at time t, 99 % of all meter readouts should be read within 6 h and the readout values not exceeding X. The ISC includes time as well as data and concerns all instances of the same meter readout process (Single/Multi). First, we define the events that have to be caught by the ISC checker, which are the starting action for launching meter readouts and an event related to each meter readout that finished. Note that the readouts of the different meters are simultaneous. If we assume n as the number of all meters, then the checker needs to wait for all instances to complete until 6 h from the start time, in order to check whether the condition of 99 % is met. The status of each meter is represented by the fluent $ReadoutFinish(meter)$, whose value is set to true if the readout is finished and false otherwise. The fluent $Value(counter, value)$ is used to check the value of the counter; i.e., number of finished readouts, after 6 h. Each event of type $ReadoutFinish(meter)$; i.e. Happens(ReadoutEnd(meter, data), time2), increments the value of the counter by terminating the old valuation of the fluent $Value(counter, oldvalue)$ to false; i.e., $Terminates(ReadoutEnd(meter, data), Value(counter, value), time2)$, and initiating the fluent $Value(counter, oldvalue + 1)$ to true:

$$Initiates(ReadoutEnd(meter, data), Value(counter, value + 1), time2).$$

Scenario 2: The second scenario (Single/Single) removes the data constraint from the first one but extends it by limiting the constraint to each 100 finished instances, which requires to reinitialize the counter after each 100 readouts. For each group of 100 finished readouts, 99 % of the instances should have finished within 5 min. This makes the constraint selective, since it selects the first 100 completed readouts first, than applies the deadline constraint. To this endeavor, we have added a violation counter that increments each time a readout takes more than 5 min to finish. We use the modulo function to reinitialize the number of violations after 100 readouts. If the number of violations exceeds 99 %, the last statement will evaluate to false. It is possible to consider another fluent for each meter to express if its readout exceeded 5 min; e.g., $Readoutviolation(meter)$.

Scenario 3 is of type Multi/Single and states that a user is not allowed to execute more than 100 tasks of the same or different workflows in the same day. The ISC clearly spans multiple processes, but here we assume that a user can instantiate each process only once. For the formalization (cf. Fig. 7), we use a predefined function $getday(time)$ that extracts the day as an integer value from the given discrete time. At each new day, the counter is reset allowing the user to execute more tasks for the day. A simple counter is incremented on the execution of a task.

Scenario 4: is of type Multi/Multi and states that similar jobs of different processes are printed together (cf. Fig. 7). The modeling requirements are resource for the printers as well as data for the print job type. Scenario 4 can be interpreted in various ways. For this simple implementation, we have opted to

represent a queuing system, incremented as new print jobs of the same type are added. Each job type is added to an associated queue. Only the currently active job type represented in the $Printing(printer, queuetype)$ fluent are worked on by the limiting resource. Jobs are finished in batches and printing jobs are switched as the queue empties at a $PrintEnd(printer, queuetype)$ event. To improve the queuing system, an additional time-based counter could be added.

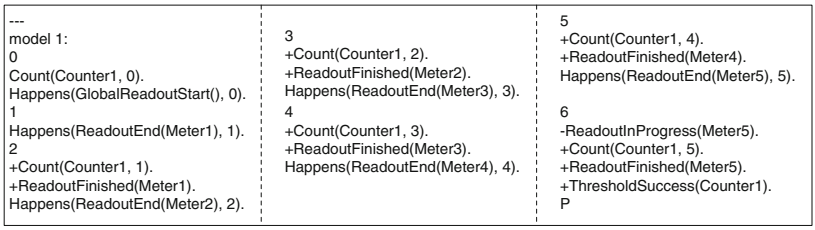


Fig. 8. ISC scenarios checking results with Decreasoner

Implementation. Each of the representative scenarios has been formalized with EC, and implemented and simulated with Decreasoner (Discrete Event Calculus Reasoner)[2]. Decreasoner uses discrete time representation, and transforms the problem into a satisfiability problem (SAT). Since the examples have been taken from the aforementioned domains; e.g., energy or healthcare, where no processes were provided, we have simulated the generation of the events in a separate module. These events are represented as $Happens(event(..), time)$. statements, applicable for each scenario. We specified event occurrences at different times and with different data. This replaces the simulation using a replay of the process models or logs. Checking results of the first scenario are depicted in Fig. 8. In particular, it shows the trace for one model, where it shows the valuations of the fluents as well as events occurrence at different time points. A fluent preceded by a "+" means that the fluent is evaluated to true, while a fluent preceded by "−" means that it is evaluated to false.

6 Related Work

A multitude of approaches for business process compliance exist that can be mainly categorized into design time, e.g., [6,29] and runtime approaches (see, for example, the survey on compliance monitoring approaches in [15]). However, there are only a few approaches that directly deal with ISC. Heinlein [7] addresses ISC at structural level only, i.e., offering means to define constraints on process activities between different instances. Other approaches focus on certain usage scenarios for ISC in Process-Aware Information Systems (PAIS) such as access control [33], batching [25], and queuing [23,30]. These usage scenarios provide

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

valuable input for the objectives and evaluation of a comprehensive approach for ISC support in PAIS.

The iUPC approaches [13,17,18,27] provide a comprehensive logical description for constraints in general, i.e., the iUPC framework. Moreover, the design and enactment of ISC in PAIS are preliminarily addressed in [13]. A special kind of ISC usage, i.e., for synchronization is formalized and implemented in [17]. However, a systematic and integrated approach for formalizing, verifying, and implementing ISC in PAIS fulfilling the ISC objectives is missing.

7 Conclusion and Outlook

ISC are the means to define restrictions and behavior across multiple instances of the same or different process types. This enables a required level of control, even for ultra-dynamic process-driven applications for which each instance evolves in a different way. This work provides the fundament for comprehensive ISC support in process-driven applications by an ISC classification and a corresponding ISC formalization based on Event Calculus. The feasibility is evaluated based on a collection of 114 ISC examples from different domains and resources. It could be observed that ISC requirements exist for many domains from manufacturing to health care and can be harvested from different sources such as regulatory documents or project deliverables. Future work will include user requirements in ISC support as well as an integration with existing process engines.

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

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, New York (2010)
2. Awad, A., Weidlich, M., Weske, M.: Consistency checking of compliance rules. In: Abramowicz, W., Tolksdorf, R. (eds.) BIS 2010. LNBIP, vol. 47, pp. 106–118. Springer, Heidelberg (2010)
3. Boubaker, S., Gaaloul, W., Graiet, M., Hadj-Alouane, N.B.: Event-b based approach for verifying cloud resource allocation in business process. In: International Conference on Services Computing, pp. 538–545 (2015)
4. Cabanillas, C., Baumgrass, A., Mendling, J., Rogetzer, P., Bellovoda, B.: Towards the enhancement of business process monitoring for complex logistics chains. In: Lohmann, N., Song, M., Wohed, P. (eds.) BPM 2013 Workshops. LNBIP, vol. 171, pp. 305–317. Springer, Heidelberg (2014)
5. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. J. Comput. Syst. Sci. **18**(2), 194–211 (1979)
6. Ghose, A., Koliadis, G.: Auditing business process compliance. In: International Conference on Service-Oriented Computing, pp. 169–180 (2007)
7. Heinlein, C.: Workflow and process synchronization with interaction expressions and graphs. In: International Conference on Data Engineering, pp. 243–252 (2001)

8. ter Hofstede, A.H.M., Ouyang, C., La Rosa, M., Song, L., Wang, J., Polyvyanyy, A.: APQL: a process-model query language. In: Song, M., Wynn, M.T., Liu, J. (eds.) AP-BPM 2013. LNBIP, vol. 159, pp. 23–38. Springer, Heidelberg (2013)
9. Joshi, R., Lamport, L., Matthews, J., Tasiran, S., Tuttle, M., Yu, Y.: Checking cache-coherence protocols with TLA+. Form. Methods Syst. Des. **22**(2), 125–131 (2003)
10. Kaes, G., RinderleMa, S., Vigne, R., Mangler, J.: Flexibility requirements in real-world process scenarios and prototypical realization in the care domain. In: OTM Workshops, pp. 55–64 (2014)
11. Knuplesch, D., Reichert, M., Kumar, A.: Visually monitoring multiple perspectives of business process compliance. In: International Conference on Business Process Management, pp. 263–279 (2015)
12. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Gener. Comput. **4**(1), 67–95
13. Leitner, M., Mangler, J., Rinderle-Ma, S.: Definition and enactment of instance-spanning process constraints. In: International Conference on Web Information Systems Engineering, pp. 652–658 (2012)
14. Lenzi, G.: The modal μ-calculus: a survey. Task Q. **9**(3), 293–316 (2005)
15. 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)
16. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.: Monitoring business constraints with linear temporal logic: an approach based on colored automata. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 132–147. Springer, Heidelberg (2011)
17. Mangler, J., Rinderle-Ma, S.: Rule-based synchronization of process activities. In: Commerce and Enterprise Computing, pp. 121–128 (2011)
18. Mangler, J., Rinderle-Ma, S.: IUPC: identification and unification of process constraints. CoRR abs/1104.3609 (2011). http://arxiv.org/abs/1104.3609
19. Mateescu, R., Monteiro, P.T., Dumas, E., de Jong, H.: Ctrl: extension of CTL with regular expressions and fairness operators to verify genetic regulatory networks. Theoret. Comput. Sci. **412**(26), 2854–2883 (2011)
20. Montali, M., Maggi, F.M., Chesani, F., Mello, P., van der Aalst, W.M.P.: Monitoring business constraints with the event calculus. ACM Trans. Intell. Syst. Technol. **5**(1), 1–30 (2014)
21. Mueller, E.T.: Commonsense Reasoning: An Event Calculus Based Approach. Morgan Kaufmann, Burlington (2006)
22. Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S.: A design science research methodology for information systems research. J. Manag. Inf. Syst. **24**(3), 45–77 (2007)
23. Pflug, J., Rinderle-Ma, S.: Dynamic instance queuing in process-aware information systems. In: Symposium on Applied Computing, pp. 1426–1433 (2013)
24. Pnueli, A.: The temporal logic of programs. In: Annual Symposium on Foundations of Computer Science, pp. 46–57 (1977)
25. Pufahl, L., Herzberg, N., Meyer, A., Weske, M.: Flexible batch configuration in business processes based on events. In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (eds.) ICSOC 2014. LNCS, vol. 8831, pp. 63–78. Springer, Heidelberg (2014)
26. Rinderle-Ma, S., Gall, M., Fdhila, W., Mangler, J., Indiono, C.: Collecting examples for instance-spanning constraints. Technical report, arXiv:1603.01523 (2016)

27. Rinderle-Ma, S., Mangler, J.: Integration of process constraints from heterogeneous sources in process-aware information systems. In: International Workshop on Enterprise Modelling and Information Systems Architectures, pp. 51–64 (2011)
28. Rouached, M., Fdhila, W., Godart, C.: A semantical framework to engineering WSBPEL processes. Inf. Syst. e-Bus. Manag. **7**(2), 223–250 (2008)
29. Sadiq, W., Governatori, G., Namiri, K.: Modeling control objectives for business process compliance. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 149–164. Springer, Heidelberg (2007)
30. Senderovich, A., Weidlich, M., Gal, A., Mandelbaum, A.: Queue mining – predicting delays in service processes. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) CAiSE 2014. LNCS, vol. 8484, pp. 42–57. Springer, Heidelberg (2014)
31. Ulfelder, S.: Building a compliance framework. Compt. World **38**(27), 34–35 (2014)
32. Vardi, M.Y.: Branching vs. linear time: final showdown. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, p. 1. Springer, Heidelberg (2001)
33. Warner, J., Atluri, V.: Inter-instance authorization constraints for secure workflow management. In: Symposium on Access Control Models and Technologies, pp. 190–199 (2006)