Towards Data-Aware Constraints in Declare

Marco Montali*
Free University of Bozen-Bolzano, Italy
montali@inf.unibz.it

Federico Chesani
Paola Mello
University of Bologna, Italy
name.surname@unibo.it

Fabrizio M. Maggi
Eindhoven University of Technology, the Netherlands
f.m.maggi@tue.nl

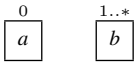
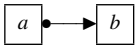
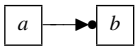
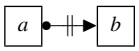
ABSTRACT

In recent years, declarative, constraint-based approaches have been proposed to model loosely-structured business processes, mediating between support and flexibility. A notable example is the Declare framework, equipped with a graphical declarative language whose semantics can be characterized with several logic-based formalisms. Up to now, Declare constraints have been mainly used to tackle control-flow aspects, abstracting away from data. In this work, we extend Declare so as to include task data and data-aware constraints. We show how the Event Calculus (EC) formalization of Declare can be improved to deal with such extensions, and to apply a reactive EC reasoner for monitoring data-aware constraints.
Keywords: process monitoring, data-aware constraints, event calculus.

1. INTRODUCTION

In recent years, declarative languages have been proposed to model business rules and loosely-structured processes, mediating between support and flexibility. A notable example are constraint-based approaches. These approaches enjoy declarativeness by supporting the modeler in the elicitation of the (minimal) set of behavioral *constraints* that must be respected to correctly execute the process, without stating how the involved stakeholders must behave to satisfy them. Acceptable courses of execution are not explicitly enumerated in a pre-defined way; rather, they are implicitly derived as the execution traces that comply with all the modeled constraints, thus enjoying *flexibility by design*. In this work, we focus on the Declare constraint-based framework [13]. Declare provides a graphical, declarative language [14] for the specification of activities and constraints: activities model atomic units of work, while constraints impose expectations about the (non) execution of activities. Constraints range from classical sequence patterns to loose relations, prohibitions and cardinality constraints. They are grouped into four families: *(i) existence* constraints, used to constrain the number of times an activity must/can be executed; *(ii) choice* constraints, requiring the execution of some activities selecting them among a set of available alternatives; *(iii) relation* constraints, expecting the execution of some activity when some other activity has been executed; *(iv) negation* constraints, forbidding the execution of some activity when some other activity has been executed. Tab. 1 lists the Declare constraints that will be used throughout the paper. One of the key advantages of Declare is that its semantics can be characterized in different logic-based approaches, enabling a wide range of reasoning and verification capabilities. At the time being, Declare has been formalized using three logic-based frameworks: Linear Temporal Logic (LTL) over finite traces [14], a rule-based language relying on Abductive Logic Programming [9], and the Event Calculus (EC) [11]. The Event Calculus formalization has proven a successful choice for dealing with runtime verification and monitoring, thanks to its high expressiveness and the existence of reactive, incremental reasoners [3, 2].

Table 1: Some Declare constraints

	**Absence** The target activity *a* cannot be executed **Existence** Activity *b* must be executed at least once
	**Response** Every time the source activity *a* is executed, the target activity *b* must be executed after *a*
	**Precedence** Every time the source activity *b* is executed, *a* must have been executed before
	**Negation response** Every time the source activity *a* is executed, *b* cannot be executed afterwards

In its original shape, Declare is mainly exploited to constrain control-flow aspects, with three main limitations: lack of support for *(i)* non-atomic, durative activities, *(ii)* metric temporal conditions, and *(iii)* data, which are all key requirements when dealing with real-world case studies, such as [16, 8, 5]. In [12], Pesic discussed how to address these issues, but the resulting extensions of Declare were treated only at the tool level, due to the insufficient expressive power of the underlying logic (propositional LTL over finite traces). Consider these illustrative examples:

(C1) In case the respondent bank rating review is completed with outcome "rejected", an account must never be opened [1].
(C2) A loan processing involving more than 20K € can be completed only if the customer has completed her registration.
(C3) Every time a blood glucose test is executed, if the measured level is less than 50 mg/dl, then the patient should start assuming sugar within 3 minutes.

Such rules correspond to specific Declare constraints, in particular a `negation response`, `precedence`, and `response`, but equipped with data. These data limit the range of events that match with the constrained activities. Furthermore, some constraints ex-

*Marco Montali has been partially supported by the EU Project FP7-ICT ACSI (257593).

SAC'13 March 18-22, 2013, Coimbra, Portugal.

plicitly refer to multiple events associated to the involved activities (*start* and *complete* in particular), suggesting that a non-atomic model for activities is needed. Beside custom data, events can be associated, by default, to a set of common information, like the resource who originated the event (*originator*), and the *time* at which the event has been generated. E.g., *registration* in constraint C2 will be executed by some customer, whereas the *patient* is responsible of assuming sugar in C3. In this latter constraint, also timestamps are relevant: the deadline of 3 minutes relates to the actual time at which the glucose test has been completed.

In general, the examples attest that: *(i)* activities could be atomic or non atomic, and involve (input and output) data; *(ii)* data-aware conditions contribute to determine when a business constraint must be applied (e.g., C3 triggers only if the glucose level is lower than 50 mg/dl); *(iii)* data-aware conditions could be either absolute or relative, referring in the latter case to data produced by another activity. Recent works aimed at improving the treatment of these aspects in the Declare framework. In [10], Declare is extended with metric temporal annotations, and the resulting approach is formalized with Abductive Logic Programming rules, to provide reasoning support. A similar version of Declare is used in [21], where however the metric time logic MTL and timed automata are used as the underlying formal framework. In [11], the EC-based formalization of Declare does not only deal with metric time conditions, but also supports non-atomic activities. This is made possible by the first-order nature of EC, which allows to attach data to events, to model parametrized business constraints using variables, and to constrain such data as conditions over the corresponding variables.

In this work, we leverage on this expressiveness making Declare data-aware. We do not make any assumption about the domain of data and resources, which can be arbitrary and even infinite. This makes the standard LTL-based formalization of Declare not applicable: it is intractable with finite domains (it suffers from the state explosion problem, because it requires to translate all data-aware business constraints into corresponding propositional ones), and even undecidable with infinite domains. More specifically, starting from [9], we propose an extension of the Declare notation, where: *(i)* activities are characterized by multiple ports, which denote the different events associated to the activity lifecycle and constitute anchor points for task data; *(ii)* constraints are attached to ports and can employ the connected data to specify data-aware conditions, affecting the constraint behavior. We then improve the EC-based approach of [11] to include data and data-aware conditions. Notice that this approach adopts logic programming (Prolog in particular) to axiomatise the EC and provide the underlying reasoning facilities. In this way, we do not only have the power of EC to formalize business constraints, but we can leverage on the entire Prolog language to specify data-aware conditions. In particular, the choice of EC and logic programming as underlying formal representation allows us to adopt a reactive reasoner [2] to monitor data-aware constraints, providing advanced operational decision support facilities to running processes, in the style of [7].

2. DATA-AWARE Declare

We introduce a data-aware extension of Declare, suitably extending its graphical notation. We tackle task data and embed the extensions proposed in [10] and [11] for metric time conditions and non-atomic activities. For the sake of self-containedness, our examples will employ only the constraints shown in Tab. 1. An in-depth presentation of Declare can be found in [14, 12, 9].

2.1 Atomic and Non-Atomic Activities

As discussed in Sec. 1, often activities are associated to multiple, atomic events such as, for example, *start* and *complete*. This calls for a non-atomic model for activities, where activity instances span over time and are characterized by multiple, correlated event occurrences. Obviously, a non-atomic model requires not only to fix the event types, but also to determine the *activity lifecycle*, i.e., how events affect the state of an instance and which are the allowed orderings. In the literature, there exist many possible activity lifecycle models. They are commonly modeled using a state machine [20]. In [12], Pesic introduced a simplified lifecycle model for Declare, which was then refined in [11] to explicitly account for an *error* state, caused by out-of-order events. In particular, the model in [11] supports the management of multiple lifecycle instances of the same activity, to track its multiple (possibly parallel) executions. To this aim, it assumes that each event comes with an activity name (the activity it refers to), an event type (a specific transition in the activity lifecycle), and an event instance identifier. The supported transitions are t_s, t_c and t_x, respectively modeling the start, completion and cancelation of an activity instance. The lifecycle specifies that each t_s transition causes the creation of a new activity instance, putting it in an *active* state. The t_c and t_x transitions for a certain activity instance can occur only when it is in the active state. If so, they migrate the instance to a *completed*/*canceled* state, and if not, to an *error* state. In order to understand to which activity instance these transitions refer to, the event instance identifier is used. In particular, a start event creates a new activity instance associated to an identifier, and only t_c and t_x transitions referring to the *same* identifier trigger a state transition of the instance. This form of correlation is necessary to manage multiple execution of the same non-atomic activity. This lifecycle has been fully formalized in [11], using the EC. Note that, thanks to the expressiveness of the EC and its ability to capture state machines, the approach could be easily extended to accommodate more complex lifecycles, such as the transactional model of XES[1] (eXtensible Event Stream), the standard defined by the IEEE Task Force on Process Mining for storing, exchanging, and analyzing event logs.

To support such a lifecycle model, we extend Declare with the notion of *port* and use it as the basic building block for atomic and non-atomic activities, and for anchoring constraints. Ports graphically depict the transitions of the lifecycle model, together with the data required or produced by the corresponding activity.

DEFINITION 1 (PORT). *A port $\mathcal{P}$ is a tuple $\mathcal{P} = \langle E, A, I, D, O\rangle$, where E is an event type that belongs to the lifecycle of activity A, while I, D and O are three sets of strings representing the input, internal and output data identifiers of the port.*[2]

DEFINITION 2 (ATOMIC ACTIVITY). *An atomic activity is a pair $\langle N, \mathcal{P}\rangle$, where N is a string representing the activity name and $\mathcal{P} = \langle t_e, N, I, D, O\rangle$ is a port associated to event t_e, representing the atomic execution of the activity.*

From the graphical point of view, a prototypical representation of an atomic activity is provided in Fig. 1 (left). Since the activity is associated to a single port, the port itself is not explicitly shown. Input and output data can be differentiated by looking at the direction of the edges that connect them to the activity. The port therefore corresponds to $\langle t_e, a, \{Id_1, \ldots, Id_n\}, D, \{Od_1, \ldots, Od_m\}\rangle$, where D is a set of *standard attributes* implicitly associated to all activities. Following the approach of the XES standard, we do not fix such attributes once and for all, but we leave instead the choice open to the modeler. A typical situation is the one in which D contains an indication about the *originator* responsible for the execution of an instance of the activity. In the remainder of the paper, we

[1] `http://www.xes-standard.org/`

[2] For simplicity, we do not deal here explicitly with data domains.

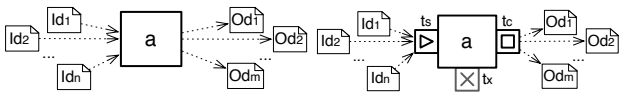


Figure 1: Atomic and non-atomic activities in Declare^{++}

assume that such an originator is indeed always implicitly present for each activity, and we use string "O" to denote it.

DEFINITION 3 (NON-ATOMIC ACTIVITY). *A non-atomic activity is a tuple* $\langle N, \mathcal{P}_s, \mathcal{P}_x, \mathcal{P}_c \rangle$, *where* $\mathcal{P}_s = \langle t_s, N, I, D, \emptyset \rangle$, $\mathcal{P}_x = \langle t_x, N, I, D, \emptyset \rangle$ *and* $\mathcal{P}_c = \langle t_c, N, I, D, O \rangle$, *such that* t_s, t_x *and* t_c *represent the start, cancel and complete events of the activity lifecycle.*

The three ports of a non-atomic activity are part of the graphical notation, as shown in Fig. 1 (right). Input and output data are in this case associated to the start and completion port of the activity, but, according to the definition, input data are also visible to the cancelation and completion ports. Therefore, the three ports for the activity shown in the figure are: $\mathcal{P}_s = \langle t_s, a, \{Id_1, \ldots, Id_n\}, D, \emptyset \rangle$, $\mathcal{P}_x = \langle t_x, a, \{Id_1, \ldots, Id_n\}, D, \emptyset \rangle$ and $\mathcal{P}_c = \langle t_c, a, \{Id_1, \ldots, Id_n\}, D, \{Od_1, \ldots, Od_m\} \rangle$. By definition, we require that ports are associated to the same internal data identifiers, but we also assume that the data corresponding to such identifiers are the same for the start and complete/cancel ports. This reflects the intuition that internal identifiers participate to the definition of the activity instance itself. For example, we have that one single originator must be responsible for the entire execution of a non-atomic activity. A similar situation holds for input data: they are provided when starting an instance of the activity, and they are maintained unaltered also in the completion/cancelation ports, which only access them for "visibility" reasons (see below).

2.2 Data-Aware Constraints

Data-aware constraints extend basic Declare constraints in two directions. First, they are anchored to ports instead of activities, supporting the modeler in the specification of fine-grained constraints regulating start, completion and cancelation of activities. Second, the anchor points may be associated to data conditions, which restrict the range of matching event occurrences. Remember that we accept Prolog predicates as special conditions (such as $member(E, L)$, which checks if element E belongs to list L).

DEFINITION 4 (DATA CONDITION). *A data condition is a Prolog predicate or a formula* $Expr_1\ Op\ Expr_2$ *where* $Op \in \{=, \neq, <, \leq, >, \geq\}$, *and* $Expr_1$, $Expr_2$ *are constituted by data identifiers, (numerical or string) constants or expressions involving data identifiers and constants.*

DEFINITION 5 (ANCHOR). *An anchor is a pair* $\langle \mathcal{P}, Cond \rangle$, *where* $\mathcal{P} = \langle E, N, I, D, O \rangle$ *is a port and* $Cond$ *is a set of data conditions anchored to the port, and defined over the data identifiers in* $I \cup D \cup O$.

As for data-aware constraints, we separate *unary* constraints, extending Declare existence and choice constraints, from *binary* constraints, tackling Declare relation and negation constraints. Some binary constraints can also be associated to metric temporal constraints, following [10]. Examples of unary constraints are the `absence` and `existence` constraints in Tab. 1, whereas examples of binary constraints are all the others reported there.

DEFINITION 6 (UNARY DATA-AWARE CONSTRAINT). *A unary data-aware constraint is a triple* $\langle T, Targets, Card \rangle$, *where* T *is the constraint's type,* $Targets$ *is a set of anchors, and* $Card$ *is the constraint's cardinality. With* $T \in \{$`existence`, `absence`, `exactly`$\}$, *the constraint models a data-aware* existence *constraint,* $Targets$ *contains a single anchor and* $Card \in \mathbb{N}_0$ *represent the minimum, maximum or exact number of times that the anchored port must/can be executed. With* $T \in \{$`choice`, `exclusive choice`$\}$, *the unary constraint models a data-aware* choice *constraint,* $Targets$ *contains at least two anchors and* $Card \in \{1, \ldots, |Targets|\}$ *respectively denotes the minimum or exact number of ports that must be executed among the possible choices.*

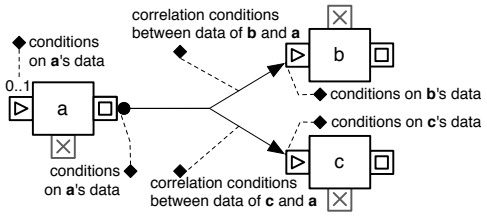


Figure 2: Representation of data-aware Declare constraints

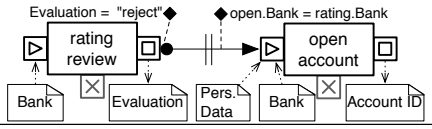
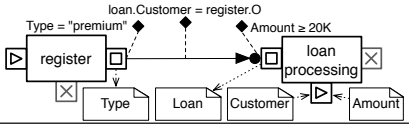
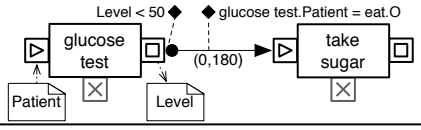
DEFINITION 7 ((TIMED) BINARY DATA-AWARE CONSTRAINT). *A binary data-aware constraint is a triple* $\langle T, Source, CTargets \rangle$, *where* T *is the constraint type (ranging over relation and negation constraints),* $Source$ *is an anchor for the constraint source, and* $CTargets$ *is a set of triples of the form* $\langle Cond, TCond, Target \rangle$. *In each of such triples,* $Target$ *is an anchor,* $Cond$ *is a data-aware condition combining the data identifiers of the ports attached to* $Source$ *and* $Target$, *and* $TCond$ *is an optional metric temporal constraint [10], which can be applied only to time-ordered relation constraints and to* `negation response`.

Intuitively, binary constraints can be augmented with three groups of data conditions, one attached to the source, and two attached to each target. Targets are associated to two groups of conditions to separate those involving only data identifiers of the target port from those that *correlate* the target data with the source ones. We call the latter *correlation conditions*. Note that we support alternative *branches* only on one side of the constraint. In this way, ambiguities and inconsistencies in correlation conditions are avoided: for every branch there is a uniquely determined connection between the two endpoints attached to it. It is worth noting that Declare constraints branching on both sides can be reformulated as a set of constraints branching only on one side. Therefore, our assumption does not lead to a loss of generality, but only affects the compactness of diagrams. Fig. 2 shows how the extended constraints can be depicted in data-aware Declare. Anchor points are denoted by a dashed connection and a diamond, associated to data-aware conditions. The connection points to the associated port or, in case of a correlation condition, to the associated branch. As for correlation conditions, a dot notation $activityName.DataId$ is used to resolve ambiguities. Tab. 2 shows the data conditions present in the four sample constraints of Sec. 1, and the corresponding notation.

EXAMPLE 1 (CONSTRAINT C1). This constraint involves a *rating review* and an *open account* activity. C1 is triggered each time a *rating review* is completed with a rejection of the reviewed bank, and forbids to open accounts to the reviewed bank. This is modeled with a `negation response` that connects the completion of *rating review* to the start of *open account*, and uses a source-anchored data condition to state that C1 is triggered only when $Evaluation =$ "$reject$". We use an (equality) correlation between the reviewed bank and the one of *open account*, so that accounts can still be opened for other banks.

EXAMPLE 2 (CONSTRAINT C2). This constraint uses a numerical condition to determine whether a `precedence` constraint should trigger or not. In particular, its source (the completion of *loan processing*) triggers the constraint only if the amount is greater than 20K €. The target port is the completion of *register*, with the condition that the originator executing the registration is the same as the one who is requesting the loan, and that the chosen registration type is "premium". The first condition is modeled as a correlation condition between the source and the target, while the second one is anchored to the target port. In this way, a 20K+ loan processing can be started only if the loan's customer has completed a premium registration.

Table 2: Data dependencies in the constraints of Sec. 1, and corresponding graphical representation

CONSTRAINT			ACTIVITY	DATA	I/O	CONDITIONS	GRAPHICAL REPRESENTATION
C1	`negation response`	source	*rating review* (complete)	bank outcome	I O	reject	
		target	*open account* (start)	bank accound id	I O	reviewed bank	
C2	`precedence`	source	*loan processing* (complete)	customer amount loan	I I O	more than 20K €	
		target	*registration* (complete)	type	O	premium	
C3	`response`	source	*test glucose* (complete)	patient glucose level	I O	less than 70 mg/dl	
		target	*take sugar* (start)	person	I	patient	

According to Def. 7, metric temporal conditions can be attached to the constraint targets. These are employed to refine the qualitative temporal ordering embedded into a Declare relationship with further metric requirements, in particular with a time interval delimiting the timespan targeted by the constraint. By considering Tab. 1, the (`negation`) `response` constraint can be refined with a time interval (t_1, t_2) stating that if the source port is executed at time t, then the target port is expected (not) to be executed after $t + t_1$ and within the $t + t_2$. Hence, t_1 and t_2 play the role of delay and deadline respectively. Symmetrically, metric `precedence` allows the source port to be executed at time t only if the target one was executed between $t - t_2$ and $t - t_1$.

EXAMPLE 3 (CONSTRAINT C3). This constraint handles a hypoglycemia for a diabetic patient, imposing that the patient must take sugar if her glucose level is below 50 mg/dl. The *glucose test* activity is executed on a patient and returns the glucose level in mg/dl. When a test is completed and the measured level is 50 or less, then the target patient must eventually take sugar. Differently from the other constraints, C3 also imposes a metric temporal requirement on the execution of the target activity: it must be promptly started within 3 minutes after the glucose test. This is modeled by means of a $(0, 180)$ temporal condition, assuming a granularity of seconds.

3. EVENT CALCULUS FORMALIZATION

We now show how the EC can be used to formalize data-aware Declare constraints. The EC is a logic-based formalisms for modeling and reasoning about actions and their effects. Starting from the seminal work in [4], it has been increasingly applied in several domains [17]. Many EC dialects have been proposed, using different underlying logical frameworks. As already mentioned, we adopt one of the most diffused choices, namely logic programming (Prolog in particular). In this setting, an EC specification contains a set of clauses formalizing the calculus predicates (*EC ontology*), and another set of clauses that use this ontology to formalize the domain of interest. The EC ontology is built from the following predicates: *(i)* $happens(Ev, T)$ states that event Ev occurred at time T. A set of ground $happens$ predicates constitutes a (partial) execution trace of the system. *(ii)* $initiates(Ev, F, T)$ and $terminates(Ev, F, T)$ represent the effects of events, i.e., properties that vary over time (*fluents*): $initiates$ and $terminates$ clauses state that event Ev initiates or terminates fluent F at time T. *(iii)* $initially(F)$ models that F holds at the beginning of execution. *(iv)* $holds\_at(F, T)$ checks whether F holds at time T.

Given an EC execution trace and a theory constituted by domain-dependent clauses defining $initially$, $initiates$ and $terminates$ predicates, it is possible to compute the validity intervals of fluents, i.e., to infer the evolution of fluents in response to events. Thanks to the first-order nature of the EC, such clauses can contain universally quantified variables (with scope the entire clause), and match with several events and timestamps. E.g., $initiates(pay(P, A), poor(P), T) \leftarrow A > 1000$ states that each person P paying an amount $A > 1000$ at any time becomes poor.

In our setting, fluents can be used to model and track the state of activity and constraint instances, and to identify noncompliant state of affairs that must be reported to the execution environment. As an activity instance denotes a specific execution of that activity, so a "constraint instance" identifies the instantiation of a constraint on specific events and data, as in [11, 6].

3.1 Formalizing Events and Traces

The execution of activities consists of the execution of their ports, which is represented by an event occurrence $happens(e, t)$, where e is a term that identifies the port and grounds its data, according to Def. 8 below. We assume that data identifiers attached to ports are listed in a pre-defined order (e.g., in alphabetical order), to guarantee an unambiguous matching between data identifiers used when formalising constraints and the real values carried by events.

DEFINITION 8 (EVENT). *The execution of an atomic activity a_A with input data identifiers $I_1, \ldots, I_n$, internal data identifiers $D_1, \ldots, D_r$ and output data identifiers $O_1, \ldots, O_m$ is represented by the event $exec(t_e, a_A, id, [i_1, \ldots, i_n], [d_1, \ldots, d_r], [o_1, \ldots, o_m])$, where id is an instance identifier, $i_1, \ldots, i_n$ are the values of input data identifiers, i.e., $I_j = i_j$ for $j = 1, \ldots, n$ (similarly for internal/output data identifiers). Given a non-atomic activity a_N with input data identifiers $I_1, \ldots, I_n$, internal data identifiers $D_1, \ldots, D_r$ and output data identifiers $O_1, \ldots, O_m$: (i) $exec(t_s, a_N, id, [d_1, \ldots, d_r], [i_1, \ldots, i_n])$ is a (possible) start event for a_N, assigning values to internal and input data identifiers; (ii) $exec(t_x, a_N, id, [d_1, \ldots, d_r], [\,])$ is a (possible) cancelation event for a_N, providing concrete values to internal data identifiers; (iii) $exec(t_c, a_N, id, [d_1, \ldots, d_r], [o_1, \ldots, o_m])$ is a (possible) completion event for a_N, assigning values for internal and output data identifiers.*

DEFINITION 9 (TRACE). *A trace is a set of event occurrences $\{happens(e_1, t_1), \ldots, happens(e_n, t_n)\}$ where $t_1, \ldots, t_n \in \mathbb{N}$ (numerical timestamps) and $e_1, \ldots, e_n$ are events as in Def. 8.*

Starting from these observable, external events, we introduce some EC axioms dedicated to produce suitable corresponding internal events. These are used to mark the proper starting, completion or cancelation of an activity instance, though here we need to account for input and output data. Once these axioms are established, they

can be complemented with the axioms described in [11] for capturing the formalization of the activity lifecycle and tracking the current activity instance states. As for data, Def. 8 states that each complete event only carries output data. However, according to the visibility rules for ports, also input data associated to the previous corresponding start event should be accessible. Hence, events and ports execution do not match completely. The definition of internal events bridges this gap. In the following, $status(i(Id, A), S)$ is a term modeling that instance Id of activity A is in state S.

$happens(start(A, Id, D, I), T) \leftarrow happens(exec(ts, A, Id, D, I), T).$
$happens(compl(A, Id, D, I), T) \leftarrow happens(start(A, Id, D, I), Ts)$
$\wedge\ happens(exec(tc, A, Id, D, Out), T) \wedge T > Ts$
$\wedge\ holds\_at(status(i(Id, A), active), T) \wedge append(I, Out, IOut).$
$happens(cancel(A, Id, D, I), T) \leftarrow happens(exec(ts, A, Id, D, I), Ts)$
$\wedge\ happens(exec(tx, A, Id, D, []), T) \wedge T > Ts$
$\wedge\ holds\_at(status(i(Id, act(A)), active), T).$

3.2 Formalizing Data-Aware Constraints

The EC-based formalization of data-aware constraints is an extension of the one proposed for the Declare language with quantitative time constraints, presented in [11]. To show how this approach can be augmented with data aware aspects, we focus on the `response` constraint, and in particular on constraint C3 of Sec. 1. The idea behind the formalization of `response` constraints is that each time an instance of the source activity is completed, a new, independent instance of the constraint is created. In fact, each instance is associated to a specific "context" (data values, timestamps, actual deadlines and temporal conditions, ...) determined by the involved ports execution. Three states are used for this purpose: *(i) sat* means that the constraint instance is currently satisfied, i.e., that the system trace complies with it; *(ii) viol* states that the constraint instance is violated, i.e., that the trace does not comply with it; *(iii) pend* represents that the constraint instance is currently violated, but some further event can bring it to satisfied. By considering the contribution of data, a `response` constraint states that "every time the source port is executed, and the source data conditions are satisfied, then a future execution of the target port is expected to occur, in such a way that the correlation conditions as well as the target data conditions and the involved metric temporal conditions are met". This behavior can be then formalized as follows: (1) Every time the source port is executed, then a new instance of the constraint is generated and put into the *pend* state, provided that the source data conditions are satisfied; the constraint identifier can be directly inherited from the instance identifier associated to the event occurrence. (2) A transition from the *pend* to the *sat* state is triggered if the target port is executed, so that the correlation and target data conditions, as well as the temporal conditions, are satisfied. (3) A transition from the *pend* to the *viol* state is triggered if the constraint instance is associated to a deadline, and the deadline has expired. The deadline expiration can be recognized by considering the timestamp associated to the currently processed event. (4) A transition from the *pend* to the *viol* state is triggered if the execution trace reaches an end, i.e., a special event is received to attest that no further event will occur. This means that all instances of constraints that are still pending cannot be satisfied anymore. Hence, this transition is applied to all pending constraint instances. All these conditions can be declaratively modeled by means of EC axioms. Tab. 3 reports how they are formally grounded on the hypo constraint, while Fig. 3 shows how they are applied to monitor a sample process instance, using the EC reactive reasoner described in [2] and a JAVA module that graphically depicts the validity intervals of fluents dynamically calculated by the reasoner.[3] The monitored trace in Fig. 3 involves three instances of *glucose_test*. The first does not lead to trigger the corresponding constraint, being the glucose level above 50. The other two trigger the constraints, since they measure a glucose level of 49 (for James) and 42 (for Johanna). Johanna promptly takes sugar, satisfying the pending constraint instance. The completion of this activity instance is the first event that occurs after 180s from the completion of Jame's *glucose_test*: the reasoner detects that the deadline for James' constraint instance (number 2) has expired, and detects a violation of the instance.

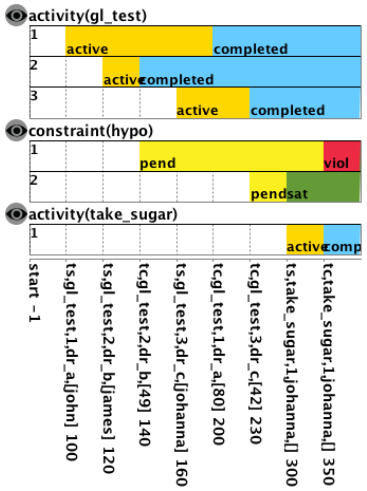


Figure 3: Monitoring the hypoglicemia constraint

4. DISCUSSION AND CONCLUSION

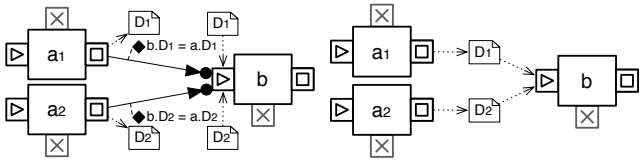
We have proposed a data-aware extension of Declare, showing how its EC-based formalization can be extended to deal with data. The approach is general: all EC reasoners based on logic programming can be used to reason about data-aware constraints. Deductive reasoners can be employed to carry out *ex-post* analysis on event logs, verifying whether they comply with the lifecycle of activities as well as with the modeled constraints. At the same time, thanks to the runtime verification framework presented in [11, 2], this work paves the way for monitoring running processes w.r.t. data-aware Declare constraints (see Fig. 3), leveraging on the promising experimental results presented in [11, 2]. A similar approach to ours is [1], where the BPMN-Q language is extended with a data perspective to model compliance requirements. BPMN-Q rules are used to specify queries to be checked against a BPMN process, evaluating its compliance; a translation to LTL and model checking are exploited for this goal. Declare is instead a framework for modeling loosely-structured processes, and its EC formalization is primarily thought for monitoring and operational support. Our approach currently deals with *task data* [15]. We plan to include case data, which can be seamlessly introduced in EC, by associating each case data identifier to a fluent that models its current value. We also want to integrate constraints with data maintained into an external source, such as a database. This would position our approach in the context of artifact-centric business processes [18], to declaratively capture artifacts' lifecycle dynamic constraints.

An important open issue concerns the usability and effectiveness of the language as the size of constraint models increase. Although

[3] A stand-alone tester, equipped with the full formalization of the example, can be downloaded from `http://www.inf.unibz.it/~montali/stuff/SAC2013-demo.zip`.

Table 3: EC formalization of the hypoglicemia `response` constraint

new	$initiates(complete(glucose\_test, Id, O, [Patient, Level]), status(i(Id, con(hypo)), pend), T) \leftarrow Level < 50.$
pend to sat	$terminates(start(take\_sugar, Id_2, O_2, []), status(i(Id, con(hypo)), pend), T) \leftarrow holds\_at(status(i(Id, con(hypo)), pend), T)$ $\wedge\, happens(complete(glucose\_test, Id, O, [Patient, Level]), T_a) \wedge Patient = O_2 \wedge T > T_a \wedge T \leq T_a + 180.$ $initiates(start(take\_sugar, Id_2, O_2, []), status(i(Id, con(hypo)), sat), T) \leftarrow holds\_at(status(i(Id, con(hypo)), pend), T)$ $\wedge\, happens(complete(glucose\_test, Id, O, [Patient, Level]), T_a) \wedge Patient = O_2 \wedge T > T_a \wedge T \leq T_a + 180.$
pend to viol	$terminates(start(take\_sugar, Id_2, O_2, []), status(i(Id, con(hypo)), pend), T) \leftarrow holds\_at(status(i(Id, con(hypo)), pend), T)$ $\wedge\, happens(complete(glucose\_test, Id, O, [Patient, Level]), T_a) \wedge T > T_a + 180.$ $initiates(start(take\_sugar, Id_2, O_2, []), status(i(Id, con(hypo)), viol), T) \leftarrow holds\_at(status(i(Id, con(hypo)), pend), T)$ $\wedge\, happens(complete(glucose\_test, Id, O, [Patient, Level]), T_a) \wedge T > T_a + 180.$
pend to viol	$terminates(exec(complete), status(i(Id, con(C)), pend), T).$ $initiates(exec(complete), status(i(Id, con(C)), viol), T) \leftarrow holds\_at(status(i(Id, con(C)), pend), T).$



(a) Explicit representation (b) Implicit representation

Figure 4: Data-transfer constraints and a possible compact notation

this topic has not yet been extensively investigated for declarative approaches, some insights have been provided in preliminary studies, such as the cognitive dimensions-based evaluation of ConDec [9] and DecSerFlow [10], and the empirical test of [19]. These studies confirm the benefits of declarative approaches, but point out that their readability tend to decrease for complex models containing a huge number of constraints, unless suitable supporting tools (such as reasoning and planning facilities) are provided. We plan to attack this challenge by studying advanced visual techniques, but also by introducing data-aware patterns that can be used to compactly represent recurrent situations. Fig. 4 shows a specialised representation for a recurrent form of "data-transfer" pattern, used to model that an activity requires data produced by another activity.

5. REFERENCES

[1] A. Awad, M. Weidlich, and M. Weske. Specification, Verification and Explanation of Violation for Data Aware Compliance Rules. In *Proc. of ICSOC*, 2009.

[2] S. Bragaglia, F. Chesani, P. Mello, M. Montali, and P. Torroni. Reactive event calculus for monitoring global computing applications. In *Logic Programs, Norms and Action*. Springer, 2012.

[3] L. Chittaro and A. Montanari. Efficient temporal reasoning in the cached event calculus. *Computational Intelligence*, 1996.

[4] R. A. Kowalski and M. J. Sergot. A Logic-Based Calculus of Events. *New Generation Computing*, 1986.

[5] L. T. Ly, S. Rinderle-Ma, D. Knuplesch, and P. Dadam. Monitoring business process compliance using compliance rule graphs. In *OTM Conferences (1)*, pages 82–99, 2011.

[6] L. T. Ly, S. Rinderle-Ma, D. Knuplesch, and P. Dadam. Monitoring business process compliance using compliance rule graphs. In *On the Move to Meaningful Internet Systems*. Springer, 2011.

[7] F. M. Maggi, M. Montali, and W. M. P. van der Aalst. An operational decision support framework for monitoring business constraints. In *Proc. of FASE*. Springer, 2012.

[8] F. M. Maggi, A. J. Mooij, and W. M. P. van der Aalst. *Analyzing Vessel Behavior using Process Mining*, chapter Poseidon book. 2012.

[9] M. Montali. *Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach*, volume 56 of *LNBIP*. Springer, 2010.

[10] M. Montali, W. M. P. M. Pesic van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative Specification and Verification of Service Choreographies. *ACM Transactions on the Web*, 2010.

[11] M. Montali, F. M. Maggi, F. Chesani, P. Mello, and W. M. P. van der Aalst. Monitoring Business Constraints with the Event Calculus. *ACM Transactions on Intelligent Systems and Technology*, To appear.

[12] M. Pesic. *Constraint-Based Workflow Management Systems: Shifting Controls to Users*. PhD thesis, Beta Research School for Operations Management and Logistics, 2008.

[13] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. Declare: Full support for loosely-structured processes. In *Proc. of EDOC*. IEEE Computer Society, 2007.

[14] M. Pesic and W. M. P. van der Aalst. A Declarative Approach for Flexible Business Processes Management. In *Proc. of BPM Workshops*, LNCS. Springer, 2006.

[15] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In *Proc. of ER*. Springer, 2005.

[16] S. Schulte, D. Schuller, R. Steinmetz, and S. Abels. Plug-and-play virtual factories. *IEEE Internet Computing*, 16(5):78–82, 2012.

[17] M. Shanahan. The Event Calculus Explained. In *Artificial Intelligence Today: Recent Trends and Developments*. 1999.

[18] R. Vaculín, R. Hull, T. Heath, C. Cochran, A. Nigam, and P. Sukaviriya. Declarative business artifact centric modeling of decision and knowledge intensive business processes. In *Proc. of EDOC*, 2011.

[19] B. Weber, H. A. Reijers, S. Zugal, and W. Wild. The declarative approach to business process execution: An empirical test. In *Proc. of CAISE*. Springer, 2009.

[20] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.

[21] M. Westergaard and F. M. Maggi. Looking into the future: Using timed automata to provide a priori advice about timed declarative process models. In *Proc. of COOPIS*. Springer, 2012.