

Monitoring Business Constraints with the Event Calculus

MARCO MONTALI, Free University of Bozen-Bolzano
 FABRIZIO M. MAGGI, Eindhoven University of Technology
 FEDERICO CHESANI, University of Bologna
 PAOLA MELLO, University of Bologna
 WIL M. P. VAN DER AALST, Eindhoven University of Technology

Today, large business processes are composed of smaller, autonomous, interconnected sub-systems, achieving modularity and robustness. Quite often, these large processes comprise software components as well as human actors, they face highly dynamic environments and their sub-systems are updated and evolve independently of each other. Due to their dynamic nature and complexity, it might be difficult, if not impossible, to ensure at design-time that such systems will always exhibit the desired/expected behaviors. This, in turn, triggers the need for runtime verification and monitoring facilities. These are needed to check whether the actual behavior complies with expected business constraints, internal/external regulations and desired best practices. In this work, we present Mobucon EC, a novel monitoring framework that tracks streams of events and continuously determines the state of business constraints. In Mobucon EC, business constraints are defined using the declarative language Declare. For the purpose of this work, Declare has been suitably extended to support quantitative time constraints and non-atomic, durative activities. The logic-based language *Event Calculus* (EC) has been adopted to provide a formal specification and semantics to Declare constraints, while a light-weight, logic programming-based EC tool supports dynamically reasoning about partial, evolving execution traces. To demonstrate the applicability of our approach, we describe a case study about maritime safety and security and provide a synthetic benchmark to evaluate its scalability.

Categories and Subject Descriptors: D.1.6 [PROGRAMMING TECHNIQUES]: Logic Programming; D.2.5 [SOFTWARE ENGINEERING]: Testing and Debugging - Monitors; F.3.2 [LOGICS AND MEANINGS OF PROGRAMS]: Semantics of Programming Languages - Process Models

General Terms: Languages, Management, Verification

Additional Key Words and Phrases: Business Constraints, Declarative Process Models, Event Calculus, Runtime Verification, Monitoring, Operational Decision Support, Process Mining

1. INTRODUCTION

Today's systems typically operate in dynamic, complex and interconnected environments. Larger systems are composed of smaller systems and evolve over time, becoming different from their initial design: e.g., sub-systems could be updated inde-

This research has been carried out as a part of the Poseidon project at Thales, under the responsibility of the Embedded Systems Institute (ESI) and partially supported by the Dutch Ministry of Economic Affairs - BSIK program. It has been also partially supported by the Netherlands Organization for Scientific Research (NWO) - "Visitors Travel Grant" initiative and by the EU Project FP7-ICT ACSI (257593).

Author's addresses: M. Montali works at the KRDB Research Centre for Knowledge and Data, Free University of Bozen-Bolzano. E-mail: montali@inf.unibz.it. F. M. Maggi is with the Institute of Computer Science at the University of Tartu. E-mail: f.m.maggi@ut.ee. F. Chesani and P. Mello are with the Department of Computer Science and Engineering (DISI), University of Bologna. E-mail: name.surname@unibo.it. W. M. P. Van der Aalst works at the Department of Mathematics & Computer Science, Eindhoven University of Technology. E-mail: w.m.p.v.d.aalst@tue.nl.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2010 ACM 1539-9087/2010/03-ART39 \$15.00

DOI : <http://dx.doi.org/10.1145/0000000.0000000>

pendently of each other or small parts of the system could slightly change interfaces and interaction patterns, etc. Such systems need to deal with dynamic environments, where many unexpected events can happen, thus creating situations that are not possible to foresee at design time. Moreover, in many cases human actors also take part in processes, together with software components. All these characteristics make it difficult to ensure that such large systems behave as expected.

Nevertheless, organizations need to guarantee the correct and safe execution of processes. For example, new legislation is forcing organizations to put more emphasis on compliance to internal/external regulations. Moreover, there is a continuous pressure to meet deadlines and improve response times. Consequently, the number of acceptable systems' behaviors is reduced by such requirements. We use the term (*business constraint*) [Montali 2010] to refer to any rule that specifies or constrains the set of acceptable behaviors. Some constraints can be enforced by an explicit and machine-interpretable model representing the acceptable execution flows for one system in isolation. However, it is unreasonable to think that all constraints can be incorporated in such executable description. First, the integration of diverse and heterogeneous constraints would quickly make models unreadable and tricky. This would become even more critical when the system behavior is modeled using procedural, workflow-like approaches, as business constraints are inherently declarative [Pesic and van der Aalst 2006; Montali et al. 2010]. Secondly, business constraints often target uncontrollable aspects, such as activities carried out by internal entities working in an autonomous way (e.g., people) or by external components, independently from the system itself.

Given that such business constraints cannot be directly incorporated at design time into the system, it would be sufficient that they are satisfied by the system when it is enacted. Obviously, this is possible only if information about the running system is tracked, stored and made available for analysis. Fortunately, as argued by van der Aalst et al. [2010], this is the case for the majority of today's systems: detailed information about the system dynamics and, in particular, the executed activities is typically stored and made available in high-quality event logs. This enables the application of process mining techniques [van der Aalst 2011], to "evaluate all events in a business process and do so while it is still running". The runtime aspect is of particular importance: noncompliant state of affairs could indicate wrong/dangerous situations or fraud; hence, they must be promptly detected, to generate suitable alerts and trigger recovery or compensation mechanisms. The application of process mining techniques to monitor and guide running cases is referred to as *operational (decision) support* [van der Aalst et al. 2010], and it is a relatively new area. Operational support helps business practitioners in the evaluation of all relevant factual data, not only of selected samples, and works in a "push-button" way. It can be used to *check* conformance, *predict* the future and *recommend* what to do next. In the context of this work, we focus on the first task, proposing a novel verification framework, called Mobucon EC (*Monitoring business constraints with Event Calculus*), able to dynamically monitor streams of events characterizing the process executions (i.e., running cases) and check whether the constraints of interest are currently satisfied or not.

The realization of a business constraints monitor poses two key requirements. On the one hand, the monitoring framework must be able to capture the complexity of business constraints, providing at the same time faithful, correct and founded results. To this end, there is a need for expressive languages to suitably define business constraints: such languages must have a formal and precise semantics and should be equipped with sound reasoning algorithms and tools. On the other hand, the monitor must produce such results in a timely fashion, being the analysis carried out on-the-fly, during the system execution. These requirements ask for a suitable trade-off between expressiveness and tractability. In this respect, Mobucon EC relies on *De-*

clare, a declarative language for business constraints [Pesic and van der Aalst 2006]. To meet the expressiveness requirements, we focus on an extended version of Declare constraints, supporting durative actions and quantitative time aspects, such as delays and deadlines [Montali et al. 2010; Montali 2010]. We do not tackle data-related aspects, but they can be seamlessly incorporated in our approach. Here, we provide Declare with a formal semantics based on the Event Calculus (EC) [Kowalski and Sergot 1986; Shanahan 1999]. The EC is a logic-based, expressive framework that allows us to model complex knowledge bases dealing with events and properties (fluents) whose truth value evolves over time. Hence, it is able to declaratively represent and reason about how the events that characterize the execution of a process instance affect the “state” of Declare constraints. Monitoring is then carried out using a light-weight, logic programming-based EC axiomatization for dynamically reasoning about partial and evolving traces [Bragaglia et al. 2012], thus meeting the performance requirement. This is achieved by suitably evolving the seminal idea of Cached Event Calculus (CEC), firstly proposed by Chittaro and Montanari [1996].

Unlike approaches that bind the notion of constraint violation to logical inconsistency (thus halting when the first violation is encountered), through the formalization of “constraint instances”, Mobucon EC provides *continuous support*, without interrupting its functioning even after a violation. This is highly desirable, because the monitored system evolves in an autonomous manner and there is no guarantee that it will halt when a violation is encountered.

To demonstrate the potential of our approach, we discuss two different case studies. The first one focuses on a real process in the context of maritime safety and security, where extended Declare constraints are used to declaratively state the allowed behaviour of vessels that cross a specific area. The second case study focuses on an extensive synthetic benchmark with randomly generated constraint models and traces: the aim is to assess the feasibility, performance and scalability of Mobucon EC. Mobucon EC has been fully implemented inside ProM [Verbeek et al. 2010], the most widely used process mining framework. Version 6.1 of ProM embeds an operational decision support infrastructure. Mobucon EC exploits this infrastructure to monitor any system whose dynamics is represented by event streams.

In summary, in this work we introduce the formalization of time-aware Declare constraints and non-atomic activities using the EC and the application of our reactive reasoner to the setting of process monitoring: this is our scientific contribution. Moreover, this work introduces the Mobucon EC framework and discuss its application to real cases as well as synthetic benchmark with randomly generated constraint models and traces.

The remainder is organized as follows. Sections 2 and 3 provide some preliminaries, introducing the Declare framework and the EC. Section 4 discusses how Declare constraints and process execution traces can be formalized as an EC theory. Section 5 deals with the Mobucon EC implementation and its evaluation. An overview of related work and a conclusion complete the paper.

2. THE DECLARE NOTATION

We provide a brief introduction to Declare. For a comprehensive description of the language, we refer the interested reader to [Pesic and van der Aalst 2006; Pesic et al. 2007; Pesic 2008; Montali 2010]. Declare is a language for the *constraint-based, declarative* specification of processes. Instead of rigidly defining the control flow, it focuses on the (minimal) set of rules that must be satisfied in order to correctly execute the process. It hence accommodates *flexibility by design*, providing a set of modeling abstractions that suitably mediate between control and flexibility. Differently from procedural specifications, where activities can be interconnected by means of sequence patterns, mixed

with constructs that explicitly tackle the splitting and merging of control flows, Declare provides a number of control-independent business constraints to interconnect activities. It is possible to use constraints referring to the future or to the past, as well as constraints that do not impose any ordering among activities. Furthermore, Declare models are *open*: any activity can be freely executed, unless it is explicitly stated otherwise by means of a constraint. On the one hand, openness guarantees flexibility: all the executions that do not violate any constraint are implicitly allowed. On the other hand, to tune the degree of openness supported by the model, suitable abstractions are dedicated to capture not only what is required, but also what is forbidden. In this way, the modeler is not bound to explicitly enumerate the acceptable executions and models remain compact: they specify desired/undesired events, leaving unconstrained all the executions that are neither mandatory nor forbidden.

Even though concurrency constructs are not explicitly present in Declare, concurrency is natively supported: as long as the stakeholders involved in a process instance behave within the boundaries imposed by the business constraints, they are free to choose the most appropriate way of executing activities (including the case in which they are executed in parallel).

2.1. Declare models

A *Declare model* is composed of a set of business constraints, which, in our case, are used to monitor the execution of an external, uncontrollable process. Constraints are attached to one or more activities and are used at execution time to define the acceptable executions of the corresponding activity instances. In the basic setting, each activity represents an *atomic* unit of work and is therefore traced by means of a single event occurring at some time point during the execution of a case. In the following, we first give an overview of Declare constraints applied over atomic activities and then we discuss how non-atomic activities can be seamlessly treated by our approach.

A Declare model is typically designed in two steps. First, the relevant activities of the application domain are elicited and inserted into the model. At this stage, the model is completely unconstrained and the activities can be performed an arbitrary number of times, in whatever order. Then, Declare constraints are added to capture the business constraints of the system, leading to a partially closed model: only those executions that comply with constraints are now accepted.


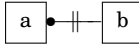

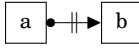




2.2. Constraints

To support several application domains, ranging from closed, prescriptive settings to flexible, adaptive ones, Declare supports a rich (extensible) set of business constraints. They are grouped into four families: *existence*, *choice*, *relation* and *negation* constraints. *Existence* constraints are *unary cardinality* constraints expressing how many times an activity can or must be executed. For example, the Declare model



contains two *existence* constraints: *existence 2*, specifying that activity *get info* must be executed at least twice and *absence*, stating that activity *publish info* cannot be executed. Such constraints are parametric in the actual numbers; in the general case, *existence N* states that the involved activity must be executed at least N times, while *absence N* states that the involved activity can be executed at most $N - 1$ times. *Choice* constraints are an extension of existence constraints that tackles multiple activities at the same time; more specifically, they are *n-ary* constraints expressing that one or more activities must be executed, *choosing* them from a set of possible alternatives.

Table I. Some Declare relation constraints, together with their corresponding negated version

	Responded existence When a is executed, b must be executed either before or afterwards		Responded absence If a is executed, b can never be executed in the same case
	Response Every time a is executed, b must be eventually executed afterwards		Negation response When a is executed, b cannot be executed afterwards
	Alternate response Every time a is executed, b must be consequently executed, before a further occurrence of a		Negation alternate response If a is executed twice, b cannot be executed between the two occurrences of a
	Chain response Every time a is executed, b must be executed next		Negation chain response Every time a is executed, b cannot be executed next

Relation constraints are *binary* constraints requiring the execution of some “target” activity when some other “source” activity is executed. The source activity is graphically identified using a “bullet” notation. Every time the source activity of a relation constraint is executed, the constraint requires the execution of another (target) activity. As shown in Table I, depending on the constraint type, additional requirements may be imposed on the target activity, making the constraint harder or softer.

Relation constraints cover many possible qualitative temporal relationships among two distinct activities: *responded existence* does not impose any ordering, *response* requires an “after” ordering, while *precedence* (not shown in Table I, but included in Fig. 1) imposes a “before” ordering. Note that Declare, in its basic version, only supports a qualitative notion of time: constraints could specify the expected relative positions among two event occurrences, but they cannot express metric distances between them. Later in the paper, we overcome this limitation.

Another important aspect of the language is that relation constraints can be generalized in such a way that multiple source or target activities can be interconnected by a single constraint. In this case, the constraint is called a *branching* constraint. The semantics of branching corresponds to disjunction among event occurrences, which translates in the following behavior: a branching on the source side means that the constraint triggers whenever one of the source activities is executed, whereas a branching on the target side implies that the constraint is satisfied by the (proper) execution of any target activity. For example, a branching chain response with a as source and b and c as targets is satisfied if, whenever a is executed, one among b and c is executed next. This shows that constraints with a branching target involve an implicit *choice*.

Negation constraints are the negative version of relation constraints: they *forbid* the execution of some activity when a certain state of affairs is reached. Table I shows the correspondence between some relation constraints and their negative counterpart; the parallel clearly attests that each negation constraint forbids the presence of an activity, while the same activity is expected by the corresponding positive constraint. Notice also that negation constraints can branch as well, with the same disjunctive semantics adopted for relation constraints.

Fig. 1 shows a sample process modeled in Declare. It deals with the flexible management of an order. A customer has the possibility of adding items, submitting the order to the seller and paying it. The seller, in turn, handles the delivery of orders and of the corresponding payment receipts. The execution of activities is governed by constraints, which implicitly identify the acceptable courses of execution. In particular, an order can be submitted only if at least one item has been chosen (*precedence (1)*) and no further items can be chosen after having submitted an order (*negation response*).



Fig. 1. An order management process in Declare

An order can be paid only if it has been previously submitted (*precedence (2)*) and the payment triggers two expectations: the order must be delivered either before or afterwards (*responded existence*) and a receipt must be consequently sent as well (*response*).

This example shows the flexibility of Declare: the same model accommodates many possible executions. For example, it is acceptable that a case is ended by the customer before submitting an order or after having submitted an order without paying it (but when the order is paid, the seller is expected to deliver the order and the receipt). Hence, trace *choose item* \rightarrow *choose item* \rightarrow *submit order* is compliant with the model. Trace *choose item* \rightarrow *submit order* \rightarrow *choose item* is instead noncompliant: it violates the *negation response* constraint. Thanks to the loose nature of the *responded existence* constraint, the model seamlessly supports the situation in which the seller waits for the payment before delivering the order, but also the case in which an order is delivered before the payment (trusted customer) or even without the payment (free gifts orders). Finally, notice that it is possible to execute *deliver order* and *send receipt* an arbitrary number of times, even without executing other activities. These particular executions are meant to support multiple attempts of such activities, as well as to support cases in which a “gift” order is autonomously delivered by the company or where another receipt is sent again for an order that was previously delivered.

2.3. Extending Declare with Metric Constraints and Non-Atomic Activities

The basic Declare approach has been extended with non-atomic activities, quantitative time constraints, task data and data-aware conditions [Pesic 2008; Montali 2010; Montali et al. 2010; Montali et al. 2013]. In this work, we focus on the first two extensions, showing how they can be formalized in the EC for monitoring purposes. The approach can be easily extended to incorporate data-aware conditions as well (see Section 7).

Non-atomic, durative activities are activities whose execution spans over a time period and is driven by multiple event occurrences. Their incorporation in the language requires three steps: (i) the identification of the atomic events characterizing the execution of an activity; (ii) the definition of the *activity lifecycle*, i.e., a description of the acceptable orderings in which such events may occur; (iii) an extension of the graphical notation so as to properly handle non-atomic activities. In this paper, we adopt the simple activity lifecycle proposed by Pesic [2008] for Declare, but the approach is able to cover more complex lifecycles as well. In the lifecycle considered here, each activity is associated to a *start* event (*s* for short), marking the beginning of the activity, and to a consequent *completion* (*c* for short) or *cancellation* (*x* for short) event, respectively marking the proper or premature termination of the activity. We can graphically extend the notation of Declare activities with “ports” that explicitly account for these three event types, as shown in Fig. 2. In this way, the modeler can attach constraints to any of such ports. This makes it possible to model fine-grained constraints such as “when activity *a* is *completed*, then activity *b* must be eventually *started*” or “activity *c* cannot be *anceled* until activity *a* is *completed*”. These three event types are connected by the following lifecycle constraints:

- *activity termination* - every start event must be eventually followed by a corresponding *single* completion or cancelation event;

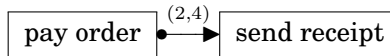


Fig. 2. Representation of atomic and non-atomic activities

- *completion consistency* - every completion event must be preceded by a corresponding *single* start event;
- *cancellation consistency* - every cancellation event must be preceded by a corresponding *single* start event.

The lifecycle does not only impose a suitable ordering among events, but also requires a one-to-one matching between every start event and a consequent completion or cancellation one. In other words, events must be properly *correlated* to each other.

As illustrated by our case study in Section 5, it is common for business constraints to incorporate metric time aspects, such as *delays* and *deadlines*. As proposed in [Montali et al. 2010], time-ordered constraints can be augmented with metric time aspects, by annotating them with two numerical values that delimit the time span inside which the triggered constraints have effect. This time span is interpreted as relative with respect to the time at which the source activity of the constraint is executed. We can, e.g., extend the order management process shown in Fig. 1 to state that when an order is paid, the receipt must be sent between 2 and 4 time units after the payment. The *response* constraint looks, in this case, as follows:



where 2 and 4 represents the delay and deadline associated to the *response* constraint.

3. THE EVENT CALCULUS

We provide an overview of the Event Calculus (EC), which is the formal framework underlying Mobycon EC. In particular, we introduce the calculus, describe its main primitives (called the EC ontology) and then sketch the problem of monitoring EC specifications, relating it to deductive reasoning.

3.1. A Brief Introduction to the EC

In 1986, Kowalski and Sergot [Kowalski and Sergot 1986] proposed the EC as a general framework to reason about time, events and change, overcoming the inadequacy of time representation in classical logic. EC adopts an explicit representation of time, accommodating both qualitative and quantitative time constraints. Furthermore, it is based on (a fragment of) first-order logic, thus providing great expressiveness (such as variables and unification). The three fundamental concepts are that of *event*, happening at a point in *time* and representing the execution of some action and of properties whose validity varies as time flows and events occur; such properties are called *fluents*. An EC specification is constituted by two theories, each containing a set of axioms:

- a general (domain-independent) theory axiomatizing the *meaning* of the predicates supported by the calculus, i.e., the so called the *EC ontology*;
- a domain theory that *exploits* the predicates of the EC ontology to formalize the specific system under study in terms of events and their effects on *fluents*. Our domain theory is focused on the formalization of the Declare language.

Starting from the seminal work of Kowalski and Sergot, a number of EC dialects have been proposed [Sadri and Kowalski 1995] and a plethora of domain-independent theories have been developed to formalize them and provide reasoning capabilities. In this work, we abstract away from the domain-independent theory and limit ourselves to

Table II. The basic Event Calculus ontology

initially(f)	Fluent f holds in the initial state	$\text{happens}(e, t)$	e occurs at time t
$\text{initiates}(e, f, t)$	Event e initiates fluent f at time t	$\text{holds_at}(f, t)$	f holds at time t
$\text{terminates}(e, f, t)$	Event e terminates fluent f at time t		

describe the predicates provided by the EC ontology. Since the majority of EC-based approaches rely on the Horn clause fragment of first-order logic with negation as failure [Clark 1978], we make use of Prolog as the specification language.

3.2. The EC Ontology

Shanahan [1999] intuitively characterizes the EC as “a logical mechanism that infers *what is true when*, given *what happens when* and *what actions do*”. These are the three aspects tackled by the EC ontology, which contains the predicates shown in Table II.

“What actions do” is the domain knowledge about actions and their effects. It is expressed inside the domain theory and captures how the execution of actions (i.e., the occurrence of events) impacts the state of fluents. In the EC terminology, the capability of an event to make a fluent true (false respectively) at some time is formalized by stating that the event *initiates* (*terminates*) the fluent. More specifically, when an event e occurs at time t , so that $\text{initiates}(e, f, t)$ and f does not already hold at time t , then e causes f to hold. In this case, we say that f is *declipped* at time t . There is also the possibility to express that some fluent holds in the initial state, using the *initially* predicate. Conversely, if $\text{terminates}(e, f, t)$ and f holds at time t , then e causes f to not hold anymore, i.e., f is *clipped* at time t . Given two timestamps t_1 and t_2 , we say that $(t_1, t_2]$ is a *maximal validity interval* (MVI) for a given fluent f if $(t_1, t_2]$ is a maximal time window in which f uninterruptedly holds, i.e., f is declipped at time t_1 and then uninterruptedly holds until it is clipped at time t_2 . Note that fluents still hold when they are clipped, but they do not hold at the time they are declipped, i.e., maximal validity intervals are left-open and right-closed.

“What happens when” is the execution trace of a (possibly partial) instance of the system under study, where “partial” means that it can contain a prefix of the full trace. An execution trace is a set of occurred events. The basic forms of the EC assume that events are atomic, i.e., bound to a single time point. In particular, an execution trace is a set of binary, ground *happens* facts, listing the occurrences of events and their corresponding timestamps. As for timestamps, the EC adopts a time structure with a minimal element, usually associated to time point 0, which represents the initial state of the system. Since event occurrences are associated to discrete timestamps, we use natural numbers (\mathbb{N}_0) to represent time values. The mapping of a real timestamp to a corresponding number depends on the chosen time *granularity* (such as msec or day). E.g., by choosing msec as the time granularity, each timestamp t could be mapped to the number of milliseconds between 1/1/1970 00:00 and t .

The combination of the domain knowledge and a concrete execution trace leads to infer “what is true when”, i.e., the intervals during which fluents *hold*. The $\text{holds_at}(f, t)$ predicate of the EC ontology is specifically used to test whether f holds at time t .

3.3. EC Domain Theories

An EC domain theory exploits the predicates of the EC ontology in order to formalize how domain-specific events affect domain-specific fluents. In our setting, such theory is constituted by a logic program whose clauses define the initial state of the system and relate the occurrence of events with the initiation and termination of fluents, possibly providing a set of conditions that should be met to effectively declip or clip them. As

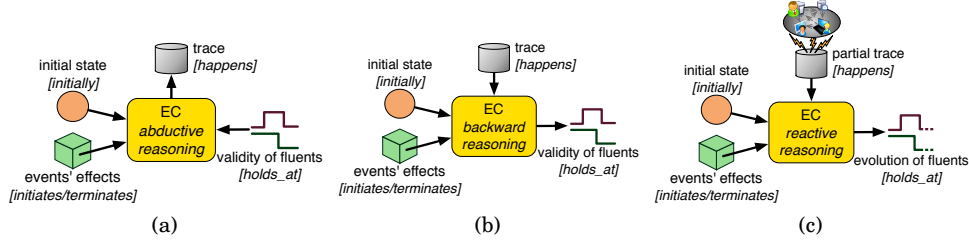


Fig. 3. Abductive, backward and reactive reasoning in the EC setting

usual, variables are universally quantified with scope the entire clause. Hence, the Prolog fact $initiates(e, f, T)$ ¹ states that event e initiates f at every time (with the proviso that f is not already holding; in this case, e has no effect). A simple yet non-trivial example of EC theory is provided in the following example.

Example 3.1. Let us consider a system with a single payment event, used to inform the system that some monetary transaction has occurred: $pay(p)$ represents a transaction of p euros. We would like to infer, timepoint by timepoint, the total amount of exchanged euros. In the EC, we can answer this question by introducing a fluent $tot(v)$ to represent that the current total amount corresponds to v and by exploiting the following EC theory to capture how payment events affect the total amount:

$$\begin{aligned} &initially(tot(0)). \\ &terminates(\text{pay}(P), tot(OV), T). \\ &initiates(\text{pay}(P), tot(NV), T) \leftarrow holds_at(tot(OV), T) \wedge NV = OV + P. \end{aligned}$$

The first clause models that the total amount is initially 0. The second clause states that when a payment event occurs, the currently computed total (old value OV) ceases to hold. The third one updates the total amount by initiating a new fluent whose amount (new value NV) corresponds to the current amount OV plus the paid euros P . These clauses are specified using variables, instantiated when specific payment events occur, with ground values for the payment. Using this modeling pattern, we can represent a sort of “multi-valued” fluents in the EC: the domain theory above represents the correlation between payment events and the current value of the tot fluent.

3.4. Reasoning About EC Theories

Two main reasoning tasks are usually exploited in the EC setting: *abductive* and *deductive* reasoning [Shanahan 1999]. Abductive reasoning (Fig. 3(a)) starts from an EC domain theory plus a query representing a desired state of affairs (expressed as a conjunction of $[\neg]holds_at$ predicates), and seeks a sample trace that respects the domain theory while achieving that state of affairs. Conversely, deductive reasoning takes an external trace and combines it with an EC domain theory, inferring the validity intervals of fluents. More specifically, the main purpose of deductive reasoners is to answer queries that ask about the validity of fluents at some time point(s), expressed using conjunctions of $[\neg]holds_at$ predicates. A special case is the one in which the user is interested in knowing the evolution of *all* fluents up to a certain time point, to reconstruct the overall past and current “global” state of the system. We denote this query as the *global validity* query. Deductive reasoning is typically carried out in a goal-oriented fashion, starting from the query and reasoning *backward* (Fig. 3(b)).

¹Which corresponds to clause $initiates(e, f, T) \leftarrow true$.

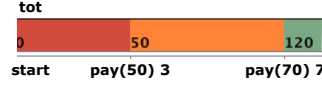


Fig. 4. The evolution of a (multi-valued) fluent, as depicted by Mobucon EC

In our work, we are interested in exploiting the EC as a monitoring framework, which requires a reasoning paradigm able to account for a dynamic, evolving domain. Indeed, monitoring focuses on a running execution, which cannot be described by a single, complete trace, but by a stream of event occurrences. Moreover, there is no explicit query, because the purpose is to track the running execution, inferring how the occurring events impact on the evolution of fluents. Implicitly, monitoring asks for the evolution of all fluents up to the time at which a new event is incorporated and processed by the monitor. Therefore, monitoring calls for *reactive reasoning* (Fig. 3(c)), i.e., for incrementally reasoning on the validity of fluents by revising and extending the produced result as new event occurrences get to be known.

One could think that reactive reasoning can be simply reduced to an iterative application of backward reasoning, where the query is always bound to the global validity query and reasoning is triggered every time a new event is received. While this approach would in principle work, it is computationally intractable: reasoning must be restarted from scratch every time the trace is updated, completely forgetting the previously calculated result. As customary in the reasoning about action literature, every time a new event is processed, typically only a “small” portion of the entire system’s state is affected, while a large extent of such state is maintained unaltered. Hence, the lack of incremental capabilities makes backward deductive reasoning practically inapplicable even for small-size problems.

Chittaro and Montanari [1996] studied this issue in the context of active temporal databases, where the dynamic acquisition of new facts changes the validity of timed data. In particular, they showed the inefficiency of deductive reasoners when dealing with such kinds of update and proposed an alternative reasoning paradigm, which *caches* the computed results for future use. They developed a Prolog-based Cached EC (CEC), an incremental reasoner that caches the MVIs of fluents and revises them every time a new set of facts is added to the database. Mobucon EC adopts the CEC-inspired implementation described in [Bragaglia et al. 2012] for monitoring EC-based specifications (see Section 5.1).

Example 3.2. Let us consider the EC theory described in Ex. 3.1 and a specific stream of events. We use the notation $Res_{\mathcal{T}}$ to represent the result computed for the global validity query for the (possibly partial) trace \mathcal{T} . The result is expressed as the set of all fluent MVIs so far. At the beginning of the execution, CEC infers that a total value of 0 has an MVI spanning from time point 0 to an unknown future time point (∞ if the execution remains quiescent). Using notation $f_{(t_1, t_2]}$ to state that fluent f has an MVI starting from t_1 and ending in t_2 , we thus have: $Res_{\emptyset} = \{tot(0)_{(0, \infty]}\}$. Now suppose that a payment of 50 euros occurs at time 3. According to the EC theory, the current total value of 0 is clipped and a new total amount of 50 is declipped: $Res_{\{happens(pay(50), 3)\}} = \{tot(0)_{(0, 3]}, tot(50)_{(3, \infty]}\}$. Finally, another payment event of 70 euros happens at time 7 and CEC extends the previously computed result as follows: $Res_{\{happens(pay(50), 3), happens(pay(70), 7)\}} = \{tot(0)_{(0, 3]}, tot(50)_{(3, 7]}, tot(120)_{(7, \infty]}\}$. Fig. 4 shows how Mobucon EC visualizes the evolution of the multi-valued fluent. For convenience, all the MVIs related to different total values are grouped together, thus giving an intuitive idea of how values change over time.

4. FORMALIZING DECLARE CONSTRAINTS IN THE EVENT CALCULUS

In this section, we show how the EC can be used to formalize Declare. As mentioned in Section 2.3, this formalization overcomes limitations of the classical LTL-based formalization such as the metric time extension and the formalization of the activity lifecycle. We discuss the representation of process execution traces and then focus on Declare constraints. The core idea of the formalization is to capture the evolution of multiple constraint *instances*, created and manipulated by the occurring events. This enables a fine grained assessment of “how well” a running process instance is complying with the constraint model under study.

4.1. Process Execution Traces

Many systems that support the execution of multi-party interactions and activities provide logging capabilities. This is true, for example, for all the mainstream process-aware information systems [van der Aalst 2011]. On the one hand, each organization is characterized by its own specific events, whose semantics and content depend on the domain and whose format is determined by the underlying information system. On the other hand, all systems share a set of common abstractions, including notions such as the ones of execution trace and event. Throughout this work, we will rely on a small number of such abstractions, which are widespread across different domains. They correspond to the notions in XES², a standard format proposed by the IEEE task force on process mining commonly used for the representation of event logs. In particular, a trace is constituted by *events* that characterize the execution of activities, marking, in particular, the execution steps of activity lifecycle’s instances (*activity instances* for short). Each event contains an *event type*, an *event identifier*, the *name* of the involved activity and the *timestamp* at which the event has occurred. The event type corresponds to *e* (“executed”) for atomic activities. For non-atomic activities, it matches with one of the characteristic lifecycle event types (*s* for “start”, *x* for “cancellation”, *c* for “completion”).

A process execution trace can be then formalized in the EC by means of *happens* predicates. More specifically, a process execution trace \mathcal{T} is a finite set of facts of the form $happens(ev(id, type, a), t)$, where *id* is an event identifier, $type \in \{e, s, x, c\}$ is the event type, *a* is an activity name, and *t* is a timestamp. Note that, by default, traces are interpreted as partial traces, i.e., as traces that could possibly be extended in the future with new event occurrences. However, it could also be the case that a specific execution of the process under study eventually reaches an end, i.e., that the trace becomes a closed, complete trace. For this purpose, a special *case_complete* atomic activity is implicitly included in each Declare model and its (only) execution marks the termination of the process instance. We assume, in the following, that traces are well-formed. An EC trace \mathcal{T} is *well-formed* if \mathcal{T} is *partial* ($happens(ev(-, -, case_complete), -) \notin \mathcal{T}$) or \mathcal{T} is *total*, i.e., one and only one *case_complete* event belongs to \mathcal{T} and *case_complete* is the last event occurrence in \mathcal{T} : all other events have a timestamp less than the one of *case_complete*.

4.2. Formalizing the Activity Lifecycle

The formalization of event occurrence and trace is not sufficient to capture the execution of instances of non-atomic activities. Indeed, when an event occurs, it must obey to the activity lifecycle described in Section 2.3. If this is not the case, a monitoring error must be reported and, at the same time, it must be ensured that such event does not affect the status of any constraint attached to the corresponding activity. In

²<http://www.xes-standard.org/>

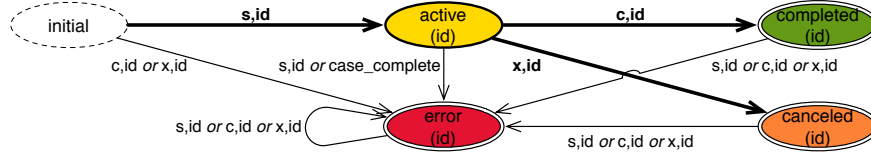


Fig. 5. A simple lifecycle for non-atomic activities using a correlation identifier and including an error state

fact, we assume that only “correct” event occurrences have an impact on constraints. The rationale behind this choice comes from the observation that an error in the activity lifecycle is a critical misbehavior that is caused either by a severe design error or by a problem in the logging infrastructure. In both cases, the logged information is unreliable and it would be unfair to use it for judging compliance.

To tackle this issue, we introduce a set of EC rules that formalize the lifecycle shown in Fig. 5, where transitions are mapped to events and fluents represent the *active*, *completed* and *canceled* states. An important feature of our approach is that the evolution of an activity execution through its lifecycle state is done on a *per-instance* basis. In other words, we foresee the possibility that the same activity is executed multiple times in the same process instance. This requires the ability to track multiple, possibly parallel evolutions of the same activity lifecycle and in particular to distinguish and *correlate* those events that belong to the same lifecycle, by properly binding each cancellation/completion event to the corresponding previous start one. The separation of activity instances and the identification of their lifecycle is important for our purposes because, as we will see in Section 4.3, each of them could impact the prescribed constraints in a different way, once quantitative temporal conditions or other data-aware conditions are taken into account.

To realize such a correlation mechanism, we make use of the event identifier and consider two events to be part of the same activity lifecycle if their activity name and event identifier are equal. We then employ fluent $l_state(i(id, a), s)$ to model that the instance identified by id of activity a is currently in state s . The activity is instantiated every time a start event occurs, using the event identifier attached to the start event as an identifier for the activity instance. The evolution of the instance through the lifecycle is then tracked by following the structure described in Section 4.1. An event occurrence advances some activity instance if it refers to that activity and contains the same identifier. Without such correlation mechanism, when two start events of the same activity are followed by a completion, it would not be possible to decide which of the two active instances has been completed.

Let us first focus on the portion of the lifecycle dealing with the correct transitions, i.e., the ones that do not lead to an error state. To this purpose, we introduce and define three “inferred” event occurrences, which are not explicitly contained in the execution trace. They are used to conceptually identify the situation in which a possible start/completion/cancellation event *correctly* leads to start, complete or cancel an activity instance. Such inferred events will be used in the formalization of Declare, in order to guarantee the aforementioned principle that only event occurrences that respect the activity lifecycle constraints can affect the constraints to be monitored. A further set of axioms is used to bind the inferred events to the corresponding state transitions. This means that, whenever an event is tracked, but it does not lead to generate a corresponding internal inferred event, then the status of all constraints is not affected at all. It is worth noting that the following axioms show the ability of the EC to define

event occurrences in terms of other event occurrences, a la Complex Event Processing (CEP) [Luckham 2001; Artikis and Paliouras 2009].

AXIOM 1 (EFFECTIVE START). *An activity instance is effectively started by a start event occurrence, provided that the same event occurrence did not happen in the past (this case is managed by Axiom 4 below):*

$$\begin{aligned} \text{happens}(\text{start}(ID, A), T) &\leftarrow \text{happens}(\text{ev}(ID, s, A), T) \\ &\wedge \neg \text{happens}(\text{ev}(ID, s, A), T_p) \wedge T_p < T. \end{aligned}$$

The effective start triggers a creation of the corresponding activity instance, transferring the identifier and placing the instance in the active state:

$$\text{initiates}(\text{start}(ID, A), l_state(i(ID, A), \text{active}), T).$$

AXIOM 2 (EFFECTIVE COMPLETION). *An activity instance with name A and identifier ID is effectively completed at time T if a completion event matching with A and ID occurs at some time T, such that the activity instance is active at time T.*

$$\begin{aligned} \text{happens}(\text{compl}(ID, A), T) &\leftarrow \text{happens}(\text{exec}(ID, c, A), T) \\ &\wedge \text{holds_at}(l_state(i(ID, A), \text{active}), T). \end{aligned}$$

The effective completion triggers a transition to the completed state:

$$\begin{aligned} \text{terminates}(\text{compl}(ID, A), l_state(i(ID, A), \text{active}), T). \\ \text{initiates}(\text{compl}(ID, A), l_state(i(ID, A), \text{completed}), T). \end{aligned}$$

AXIOM 3 (EFFECTIVE CANCELATION). *The effective cancelation of an activity instance mirrors the axioms used for effective completion.*

$$\begin{aligned} \text{happens}(\text{cancel}(ID, A), T) &\leftarrow \text{happens}(\text{exec}(ID, x, A), T) \\ &\wedge \text{holds_at}(l_state(i(ID, A), \text{active}), T). \\ \text{terminates}(\text{cancel}(ID, A), l_state(i(ID, A), \text{active}), T). \\ \text{initiates}(\text{cancel}(ID, A), l_state(i(ID, A), \text{canceled}), T). \end{aligned}$$

Beside the three aforementioned states, we introduce a further set of rules used to identify undesired situations, which correspond to a transition to a special *error* state used for monitoring purposes. The transitions to the error state correspond to the ones shown in Fig. 5. Observe that some error configurations capture the *functionality* of instance identifiers with respect to the pair constituted by activity name and event type. This principle guarantees that two active instances of the same activity necessarily have distinct identifiers.

AXIOM 4 (ERROR STATE). *Any event occurrence causing an error has the effect of terminating the current state (provided that it is not already in an error state):*

$$\begin{aligned} \text{terminates}(Ev, l_state(i(ID, A), S), T) &\leftarrow \text{holds_at}(l_state(i(ID, A), S), T) \wedge S \neq \text{error} \\ &\wedge \text{initiates}(Ev, l_state(i(ID, A), \text{error}), T). \end{aligned}$$

Completion and cancelation events cause an error if they occur when the corresponding activity instance is not active:

$$\begin{aligned} \text{initiates}(\text{ev}(ID, c, A), l_state(i(ID, A), \text{error}), T) &\leftarrow \neg \text{holds_at}(l_state(i(ID, A), \text{active}), T). \\ \text{initiates}(\text{ev}(ID, x, A), l_state(i(ID, A), \text{error}), T) &\leftarrow \neg \text{holds_at}(l_state(i(ID, A), \text{active}), T). \end{aligned}$$

The start event causes an error if the corresponding activity instance has been already activated in the past, by means of another start event that refers to the same activity

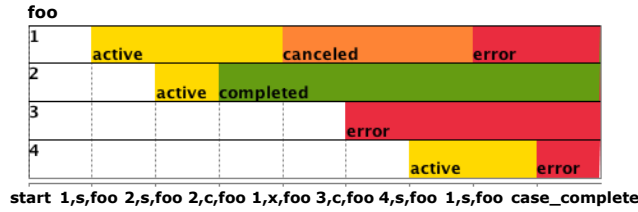


Fig. 6. Sample evolutions of four instances of activity “foo” (event timestamps are not shown)

and is associated to the same identifier:

$$\text{initiates}(\text{ev}(ID, s, A), l_state(i(ID, A), \text{error}), T) \leftarrow \text{happens}(\text{ev}(ID, s, A), T_p) \wedge T_p < T.$$

A last source of error is determined by the execution of a *case_complete* event when the activity instance is still active, which is in fact a temporary state.

$$\text{initiates}(\text{ev}(_, e, \text{case_complete}), l_state(i(ID, A), \text{error}), T) \leftarrow \text{holds_at}(l_state(i(ID, A), \text{active}), T).$$

Fig. 6 shows the outcome produced by applying the EC-based formalization of the activity lifecycle on a simple execution trace. The same activity is executed four times, showing three possible errors as well as a correct execution.

4.3. Business Constraint Instances and Their States

Monitoring business constraints amounts to computing how the event occurrences that characterize the execution of a process instance affect the evolution of constraints *states* over time. Our approach is inspired by existing approaches for runtime verification using temporal logics and enactment of Declare models. However, unlike these approaches we take into account the notion of non-atomic activities and of metric time. On the one hand, Bauer et al. [2007] introduced a framework for the runtime verification of Linear Temporal Logic formulae, where formulae are associated to one among four distinct states: (i) *permanent satisfaction* (the process instance is compliant with the formula); (ii) *temporary satisfaction* (the process instance is currently compliant with the formula, but it is still possible to violate it in the future); (iii) *permanent violation* (the process instance is not - and will not be - compliant with the formula); (iv) *temporary violation* (the process instance is currently violating the formula, but it is still possible to comply with it in the future). On the other hand, Pesic et al. [2007] introduced an enactment environment for Declare, with a similar approach to the one of [Bauer et al. 2007]. In this environment, at each time every constraint is associated to one among three states: two correspond to permanent and temporary violation, while the third (*fulfillment*), is used to represent temporary or permanent satisfaction, attesting that the current trace is compliant with the constraint.

We start from this latter approach, using the term *pending* to denote a constraint that is temporarily violated. The rationale behind this choice is that a temporarily violated constraint expects the execution of a specific activity to become satisfied. However, a major difference between Mobucon EC and the two aforementioned approach relies in the differentiation between constraints and their instances. While in [Bauer et al. 2007; Pesic et al. 2007] the described states are used to track the evolution of constraints over time, in Mobucon EC each constraint can be associated to multiple *instances*, each of which follows an independent evolution. As the following examples highlight, some constraints are active from the beginning of the execution and follow a unique evolution as events occur, while other constraints are characterized by multiple, parallel and independent instances.

Example 4.1. Let us consider a generic existence constraint stating that activity a must be executed at least n times. It is initially in a *pending* state, waiting for n executions of activity a . When the n -th execution of a occurs, it becomes *satisfied*. Conversely, if the case reaches an end and the constraint is still pending, it becomes *violated*.

Example 4.2. Let us consider a metric response constraint, with a and b source and target activity respectively and associated to delay m and deadline n . The constraint is triggered every time activity a is completed, expecting the start of a consequent b occurring inside the desired time interval $[m, n]$. Such a time interval is grounded on the basis of the time at which a occurred. To capture this behavior, every execution of a starts a new, separate constraint instance. Each instance is placed in a pending state, waiting for the occurrence of b inside the time interval obtained by combining the time at which a occurred and the constraint's time window. In particular, an instance created at time t becomes satisfied if b is executed inside $[t + m, t + n]$ or violated if the actual deadline $t + n$ expires and the instance is still pending.

The two examples intuitively introduce the notion of constraint *instance*, with existence being a single-instance constraint and response being a multiple-instance one. In Ex. 4.2, this is due to the presence of metric time constraints, which require to recognize the difference even between multiple occurrences of the same activity, because of the difference between their timestamps. More generally, the notion of constraint instance resembles the one of activity instance: as different executions of the same activity inside a business process determine separate instances of that activity, each one grounded in a specific *context* (data, timestamp, ...), so it can trigger multiple instances of the same constraint.

Summarizing, each constraint instance represents the application of the constraint in a particular context. Since we limit our analysis to activities and temporal aspects, the contextual information for event occurrences and constraint instances is composed of the name of the executed activity and its timestamp. Technically, we associate a global unique identifier to each modeled constraint and use term $i(id, a, t)$ to denote the instance of constraint identified by id that has been created by the execution of activity a at time t . We use the arbitrary constant *start* as a placeholder for the activity identifier for those instances that are active from the beginning of execution, such as the case of Ex. 4.1. The approach can be seamlessly generalized by adding to the context the other relevant pieces of information (resource, data, ...). According to the discussion above, at each time point, every constraint instance can be in one among the following states:

- *pending* (*pend* for short), a transient state representing the fact that the constraint is waiting for the occurrence of one event (possibly, one among many events) and is hence temporarily violated;
- *satisfied* (*sat* for short), a transient or permanent state indicating that the current execution trace is compliant with the constraint;
- *violated* (*viol* for short) is a permanent state attesting that the instance has been violated by the execution trace.

To represent these states, we rely on a (multi-valued) fluent $state(I, S)$, where I is the constraint instance and S the current state of I , with $S \in \{pend, sat, viol\}$. For example, $state(i(id, a, t), pend)$ represents the fact that instance $i(id, a, t)$ is currently *pending*.

4.4. Constraints Formalization

We now describe the axiomatization of constraints semantics as an EC theory. Since the semantics focuses on the creation and state transitions of constraints' instances in response to event occurrences, we first define a set of supporting predicates for state manipulation. The first two rules deal with the creation of an instance, placing it in

the specified state, either at the beginning of the execution or when some event occurs:

$$\begin{aligned} \text{initially}(\text{state}(I, S)) &\leftarrow \text{init_state}(I, S). \\ \text{initiates}(\text{Ev}, \text{state}(I, S), T) &\leftarrow \text{creation}(\text{Ev}, T, I, S). \end{aligned}$$

The *cur_state* predicate is defined to test the current state of an instance:

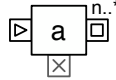
$$\text{cur_state}(I, S, T) \leftarrow \text{holds_at}(\text{state}(I, S), T).$$

A group of two rules is used to capture state transitions. A transition from state S_1 to state S_2 is executable only if the instance is currently in state S_1 , and it is applied by terminating the fluent associated to S_1 and initiating the one bound to S_2 .

$$\begin{aligned} \text{terminates}(\text{Ev}, \text{state}(I, S_1), T) &\leftarrow \text{trans}(\text{Ev}, T, I, S_1, S_2) \wedge \text{cur_state}(I, S_1, T). \\ \text{initiates}(\text{Ev}, \text{state}(I, S_2), T) &\leftarrow \text{trans}(\text{Ev}, T, I, S_1, S_2) \wedge \text{cur_state}(I, S_1, T). \end{aligned}$$

Let us now describe how the machinery discussed so far can be used to formalize some key Declare constraints. The axiomatization of complex Declare models can be found in the benchmark files (see Section 5.3). Each constraint is formalized by means of a set of EC axioms. Two constraints of the same kind share the “form” of such axioms, customized with its specific activities and parameters. The formalization of an entire Declare model consists of the union of axioms used to formalize each one of its constraints. In the following, we assume that the activities involved in the constraints are non-atomic and we make use of the inferred lifecycle events formalized in Section 4.2 to make sure that the state of constraints is affected only by those events that fit into the activity lifecycle constraints. The same formalization can be applied to atomic activities as well, by just considering their single event type.

4.4.1. *Existence*. Let us consider an existence constraint of the form



identified by *id* and stating that activity *a* must be completed at least *n* times. Following Ex. 4.1, it can be formalized by means of three axioms, respectively managing the creation, fulfillment and violation of the unique instance associated to the constraint.

AXIOM 5 (EXISTENCE CREATION). *Each existence constraint is associated to a single instance, created and put in the pending state when the case is started:*

$$\text{init_state}(i(\text{id}, \text{start}, 0), \text{pend}).$$

AXIOM 6 (EXISTENCE FULFILLMENT). *A pending existence constraint instance becomes satisfied when the *n*-th execution of the involved event type (in our case the completion of *a*) occurs:*

$$\text{trans}(\text{compl}(_, a), T, i(\text{id}, \text{start}, 0), \text{pend}, \text{sat}) \leftarrow \text{hap}(\text{compl}(_, a), n, T).$$

where $\text{hap}(\text{Ev}, N, T)$ tests whether *Ev* occurred *N* times before or at *T* and it is obviously called by abstracting away from the specific event identifier³:

$$\text{hap}(\text{Ev}, N, T) \leftarrow \text{findall}(_, (\text{happens}(\text{Ev}, T_p), T_p \leq T), L) \wedge \text{length}(L, N).$$

A violation of the existence constraint is detected when it is pending and a *case_complete* event is received. This attests that the case has reached an end and,

³The Prolog predicate $\text{findall}(\text{Objects}, \text{Goal}, \text{List})$ produces the *List* of all *Objects* that satisfy *Goal*, while $\text{length}(L, N)$ is true if *N* is the length of list *L*.

consequently, that no further event will occur to move the instance from the pending to the satisfied state. This observation does not only hold for the existence constraint, but it captures the inherent semantics of the pending state and is therefore applied to *any* constraint instance in the pending state, independently from the constraint it belongs to. Indeed, a constraint instance is pending if it is waiting for the occurrence of some event and such an expectation cannot be fulfilled anymore if the case is complete.

AXIOM 7 (SEMANTICS OF PENDING). *When the case reaches an end, all pending instances are declared as violated:* $trans(ev(-, e, case_complete), T, i(-, -, -), pend, viol)$.

4.4.2. Absence. Let us now consider an absence constraint, identified by id and modeling that activity a cannot be canceled n times (i.e., it can be canceled at most $n - 1$ times). Like existence constraints, each absence is associated to a single instance. The instance is initially put in the satisfied state and it persists in that state unless the target activity is executed n times, leading to its violation.

AXIOM 8 (ABSENCE CREATION). *Each absence constraint is associated to a unique instance, created and put in the satisfied state when the case is started:*

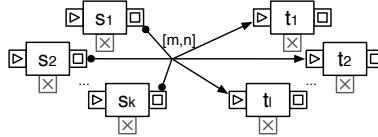
$$init_state(i(id, start, 0), sat).$$

AXIOM 9 (ABSENCE VIOLATION). *The active absence constraint instance becomes violated when the n -th occurrence of the forbidden event (in our case, the cancelation of activity a) is received:*

$$trans(canc(-, a), T, i(id, start, 0), sat, viol) \leftarrow hap(canc(-, a), N, T).$$

It is worth noting that absence constraint instances are never pending. In fact, they do not place any (positive) expectation concerning the occurrence of some event.

4.4.3. Metric Response. We consider the following “prototypical” response constraint



It includes branches (i.e., the presence of multiple sources and targets) and quantitative time constraints. We use again id to identify the constraint. The meaning of the constraint is: whenever one of the source activities is completed at some time t , one of the target activities must be started between m and n time units after t . To deal with branching response constraints, we make use of the *member* predicate⁴: instead of directly checking if an activity is the source or target of a response constraint, we test if it belongs to the set of sources $[s_1, \dots, s_k]$ or targets $[t_1, \dots, t_i]$.

AXIOM 10 (RESPONSE CREATION). *A new instance of the response constraint is created every time some activity A is completed and A is one of the sources. The instance is put in the pending state, waiting for a future suitable execution of a target activity.*

$$creation(compl(-, A), T, i(id, A, T), pend) \leftarrow member(A, [s_1, \dots, s_k]).$$

A pending instance becomes satisfied if a target activity is executed afterwards, within the time boundaries associated to the instance. We use the *cur_state* predicate to obtain the actual timestamp value contained in the instance context (which corresponds to the time at which the instance was created). Such reference point is then suitably

⁴ $member(El, L)$ is true if list L contains El .

combined with the constraint's delay m and deadline n , determining the time window inside which the target event (in our case, the starting event of one of the target activities) is expected to occur.

AXIOM 11 (RESPONSE FULFILLMENT). *A pending response instance becomes satisfied when one of its target activities is started, at a time that falls within the actual, expected time window.*

$$\begin{aligned} \text{trans}(\text{start}(-, B), T, i(\text{id}, A, T_i), \text{pend}, \text{sat}) \leftarrow & \text{cur_state}(i(\text{id}, A, T_i), \text{pend}, T) \\ & \wedge \text{member}(B, [t_1, \dots, t_l]) \wedge \text{inside_future}(T_i, m, n, T). \end{aligned}$$

where $\text{inside_future}(\text{Ref}, V_1, V_2, T)$ is used to test whether T is contained in the time window ranging from $\text{Ref} + V_1$ to $\text{Ref} + V_2$:

$$\text{inside_future}(\text{Ref}, V_1, V_2, T) \leftarrow T_1 \text{ is } \text{Ref} + V_1 \wedge T_2 \text{ is } \text{Ref} + V_2 \wedge T \geq T_1 \wedge T \leq T_2.$$

When m is not specified (no delay), we consider it to be 0, while when n is not specified (no deadline), we consider it to be virtually ∞ . This specific case can be seamlessly treated by interpreting ∞ as a special constant “inf”, providing alternative definitions for the inside_future predicate.

The last two axioms deal with the violation of a *response* instance. Two possible undesired situations may arise: either the instance is pending and the case reaches an end (this behavior is already captured by Axiom 7) or the instance is pending but its actual associated deadline is expired. Both cases identify a state of affairs where no possible future evolution exists, such that the pending instance will be eventually satisfied. The deadline expiration case must be managed only when the constraint is associated to an actual value for the deadline. In particular, this situation is handled with a *best effort* approach: since the EC has no intrinsic notion of the flow of time, but “extracts” the current time from the occurring events, a deadline expiration is detected at the *first* following time at which some event occurs.

AXIOM 12 (RESPONSE VIOLATION DUE TO DEADLINE EXPIRATION). *A response instance becomes violated if it is still pending after the maximum time at which a target activity is expected to be executed.*

$$\text{trans}(-, T, i(\text{id}, A, T_i), \text{pend}, \text{viol}) \leftarrow \text{cur_state}(i(\text{id}, A, T_i), \text{pend}, T) \wedge \text{after_future}(T_i, n, T).$$

where $\text{after_future}(\text{Ref}, V, T)$ checks whether the deadline of V time units after Ref is expired at time T , also dealing with the special case in which V corresponds to ∞ :

$$\text{after_future}(\text{Ref}, V, T) \leftarrow V \neq \text{inf} \wedge T_d \text{ is } \text{Ref} + V \wedge T > T_d.$$

4.4.4. Metric Chain Response. Let us consider the same prototypical constraint used in Section 4.4.3, but now employing a *chain response* instead of a simple response. This constraint can be formalized as an extension of the response constraint. More specifically, all the axioms discussed in Section 4.4.3 are maintained and an additional axiom is introduced. This axiom poses the additional requirement that the target activity must be executed *next* to the source one, i.e., between the execution of the source and the target all other events are forbidden.

AXIOM 13 (CHAIN RESPONSE VIOLATION DUE TO INTERMEDIATE EVENT). *A pending instance of chain response becomes violated if an event related to a non-target activity occurs.*

$$\begin{aligned} \text{trans}(\text{start}(-, X), T, i(\text{id}, A, T_i), \text{pend}, \text{viol}) & \leftarrow \neg \text{member}(X, [t_1, \dots, t_l]). \\ \text{trans}(\text{compl}(-, X), T, i(\text{id}, A, T_i), \text{pend}, \text{viol}) & \leftarrow \neg \text{member}(X, [t_1, \dots, t_l]). \\ \text{trans}(\text{cancel}(-, X), T, i(\text{id}, A, T_i), \text{pend}, \text{viol}) & \leftarrow \neg \text{member}(X, [t_1, \dots, t_l]). \end{aligned}$$

5. IMPLEMENTATION AND EVALUATION

Our monitoring approach has been implemented and evaluated in two ways: (1) using a real-life case study to illustrate its usefulness and (2) a set of experiments that use synthetic data to evaluate its performance. Along the first line, we describe how Mobucon EC has been fruitfully applied in the domain of maritime safety and security. In particular, the case study deals with monitoring a vessel in a specific area using a sensor network, which detects and shares events characterizing a “change of state” in the vessel’s behavior. It shows that Mobucon EC can successfully target a variety of systems beyond classical workflow and process-aware management systems. After showing the usefulness of our approach, we shift attention to various experiments that have been carried out to study the reaction time of the reasoner as the number of events grows and the number of constraints to be monitored increases.

5.1. Implementation

The current implementation of Mobucon EC is composed of four modules:

- (1) *EC module*, the core reasoner of the tool, containing a light-weight axiomatization of CEC (see Section 3.4) written in standard Prolog [Bragaglia et al. 2012].
- (2) *Formalization module*, which translates a Declare model into an EC theory.
- (3) *Information exchange module*, which delivers events to the reasoner and receives the newly computed result.
- (4) *Feedback module*, which provides an intuitive feedback to the user, showing the evolution of each constraint instance.

The reasoning module of Mobucon EC is written in standard Prolog, hence virtually any Prolog engine can be used to perform the computations. We experimented with two alternative solutions. The first relies on TuProlog (tuprolog.alice.unibo.it) and its main advantage is that, being developed completely in JAVA, it can be seamlessly integrated inside JAVA software, such as ProM; the main drawback is that its performance cannot still compete with mainstream Prolog systems. The other solution relies on YAP (yap.sourceforge.net), one of the highest-performance state of the art Prolog engines. Unlike TuProlog, YAP cannot be directly invoked from JAVA code, but requires an intermediate bridge (such as InterProlog⁵).

Hence, Mobucon EC can come in different flavors, from the pure reasoner (just a Prolog theory consisting of CEC, the formalization of constraints and a set of facts describing the execution trace to be analyzed) to a full prototype equipped with a user interface and integrated inside the version 6.1 of the well-known ProM process mining framework [Verbeek et al. 2010].⁶ ProM 6.1 natively provides an Operational Support (OS) service, which accommodates techniques for runtime process monitoring, recommendations and predictions. As sketched in Fig. 7, the OS service takes care of receiving TCP/IP streams of event occurrences from an external system. Each request from the external system must come with a process and a case identifier. In this way, the OS support can handle multiple running cases at the same time, correlating each incoming event to the corresponding stream. Each OS provider encapsulates the business logic of a specific OS functionality, while the OS service mediates the interaction between providers and the external system. Mobucon EC exploits this infrastructure and is implemented as a monitoring OS provider. Each instance of Mobucon EC is associated to a Declare model, which can be loaded from the XML file produced by the Declare editor [Pesic et al. 2007].⁷ In order to facilitate testing and exploitation of Mobucon EC, two

⁵<http://www.declarativa.com/interprolog/>

⁶A stand-alone tester can be found at <http://www.inf.unibz.it/~montali/tools.html>

⁷<http://www.win.tue.nl/declare/>

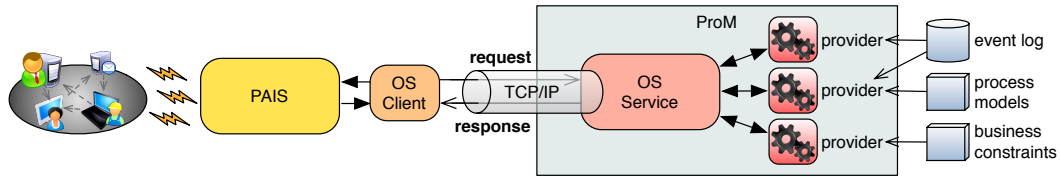


Fig. 7. Operational support in ProM 6

client prototypes have been developed, one accepting the realtime acquisition of events and the other loading traces from event logs.

5.2. Case Study: Monitoring the Behavior of a Vessel

Our case study has been conducted in the context of Poseidon [Maggi et al. 2012], a project partially supported by the Dutch Ministry of Economic Affairs under the BSIK program. In Poseidon, we have closely collaborated with practitioners from the sector of maritime safety and security.

The case study is about monitoring the behavior of vessels as they move inside a certain maritime area. Each vessel has an on-board AIS (Automatic Identification System [International Telecommunications Union 2001]) transponder that uses several message types and reporting frequencies to broadcast information about the vessel. An AIS receiver collects these broadcasted AIS messages and produces a TCP/IP stream of messages. These messages contain information such as the maritime vessel identifier (*mmsi*), the navigational status of the vessel and its ship type. This is the information that can be used to monitor the behavior of the vessel. In particular, when monitoring a vessel using the broadcasted stream of AIS messages, events correspond to changes in the navigational status of the vessel: *moored*, *under way using engine*, *aground*, *at anchor*, *not under command*, *constrained by her draught*, *restricted maneuverability*, etc. Every change in the navigational status is associated to a single timestamp, thus we represent the transition into some navigational status as an atomic activity.

Each vessel is monitored separately from the others, hence, a case is constituted by the sequence of events that refer to the same *mmsi* number. Vessels are expected to behave differently on the basis of their ship type, i.e., for each ship type, only sequences conforming to specific constraints are allowed. Some constraints also involve metric time constraints that the event stream must satisfy, e.g., when a change in the vessel's navigational status is expected to occur within a given interval of time. These constraints are loosely-structured and only limit the behavior of the vessels without explicitly describing how they should operate. Therefore, the compliant behaviors can be successfully represented using Declare. More specifically, the first step of our experimentation has been the construction of a Declare model representing the observed behavior of every possible ship type, such as *passenger* or *cargo ship*. These models can be extracted using process discovery techniques as presented in [Maggi et al. 2012]. In particular, we have enriched the Declare models presented in [Maggi et al. 2012] with quantitative time constraints that have been empirically assessed by analyzing the behavior of the monitored vessels under ideal conditions. In general, our approach can be applied to monitor the behavior of a system (in this case a vessel) evaluating the overall system health on the basis of the number of anomalies detected. For this experimentation, the stream of AIS messages has been recorded in an event log. Experiments have been carried out by loading an excerpt of the log (corresponding to a period of one week) in Mobucon EC, using a time granularity of seconds.

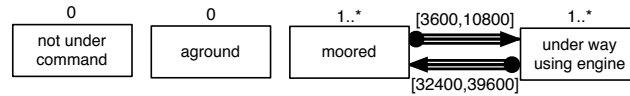


Fig. 8. Expected behavior for vessel type passenger ship. Time granularity: seconds

5.2.1. Passenger Ships. The expected behavior of a *passenger ship* is tackled by the Declare model shown in Fig. 8. Events *aground* and *not under command* must never occur. Moreover, the vessel’s behavior must show a regular alternation of events *moored* and *under way using engine* (modeled with chain response constraints in the model).

The monitored passenger ships are required to go back and forth between two resorts. Together with domain experts, we have estimated that each journey must take more than 9 hours (32400 secs.) and less than 11 hours (39600 secs.). Hence, constraint *chain response([under way using engine], [moored])* is bound to this time window. Moreover, for constraint *chain response([moored], [under way using engine])* we specify a delay of 1 hour (3600 secs.) and a deadline of 3 hours (10800 secs.) considering the time that each ship must spend in a harbor (for refueling, restocking etc.).

Fig. 9 shows the results for the monitoring of a specific case (i.e., of a specific vessel) with respect to the Declare model in Fig. 8. The interface groups together all the instances referring to the same constraint, under the name of the constraint itself. The evolution of each constraint can be visualized in a summarized version, highlighting the number of instances and giving a combined flavor of their contributions or in a detailed way, which separately shows the evolution of each instance. The instance related to constraint *absence([not under command], 1)* and the instance related to constraint *absence([aground], 1)* are always satisfied because in the stream of messages, events *not under command* and *aground* never occur. The instance related to constraint *existence([under way using engine], 1)* is initially pending (because it is waiting for the occurrence of at least one event *under way using engine*). When at 06-01-2007 02:00:02 *under way using engine* occurs, this instance becomes satisfied. Constraint *existence([moored], 1)* has a similar behavior.

Fig. 9 also shows the thirteen instances related to constraint *chain response([under way using engine], [moored])* (delay: 9 hours, deadline: 11 hours). Every instance corresponds to a different occurrence of event *under way using engine*. Note that, in this specific case, each instance of the constraint corresponds to a period of time spent by the considered vessel under way using engine, i.e., for a different journey. The constraint specifies that when *under way using engine* occurs, it must be immediately followed by *moored*. This is always the case except for the last occurrence of *under way using engine* where the instance remains pending (instance 13). This is due to the fact that the considered data, being extracted from a larger event log, is not still complete. However, also when *under way using engine* is directly followed by *moored*, it may be still too early or too late. In fact, only in instances 2, 3, 4, 5, 8, 10 and 11 event *under way using engine* is followed by *moored* complying with the defined temporal specifications. In contrast, in instance 1, event *moored* occurs after less than 9 hours (after 7 hours and 55 minutes). In instances 6, 7, 9 and 12 event *moored* occurs after more than 11 hours (after 11 hours and 6 minutes, after 11 hours and 8 minutes, after 11 hours and 25 minutes and after 11 hours and 33 minutes respectively). Similar patterns can be found for constraint *chain response([moored], [under way using engine])* (delay: 1 hour, deadline: 3 hours). In this case, every instance corresponds to a different occurrence of *moored*, i.e., to a period of time spent by the vessel in a harbor.

The top of Fig. 9 shows the evolution of a metric measuring the overall *system health* over time. This metric provides a global indication about “how much” the running case is complying with the Declare model. Different metrics can be easily plugged into

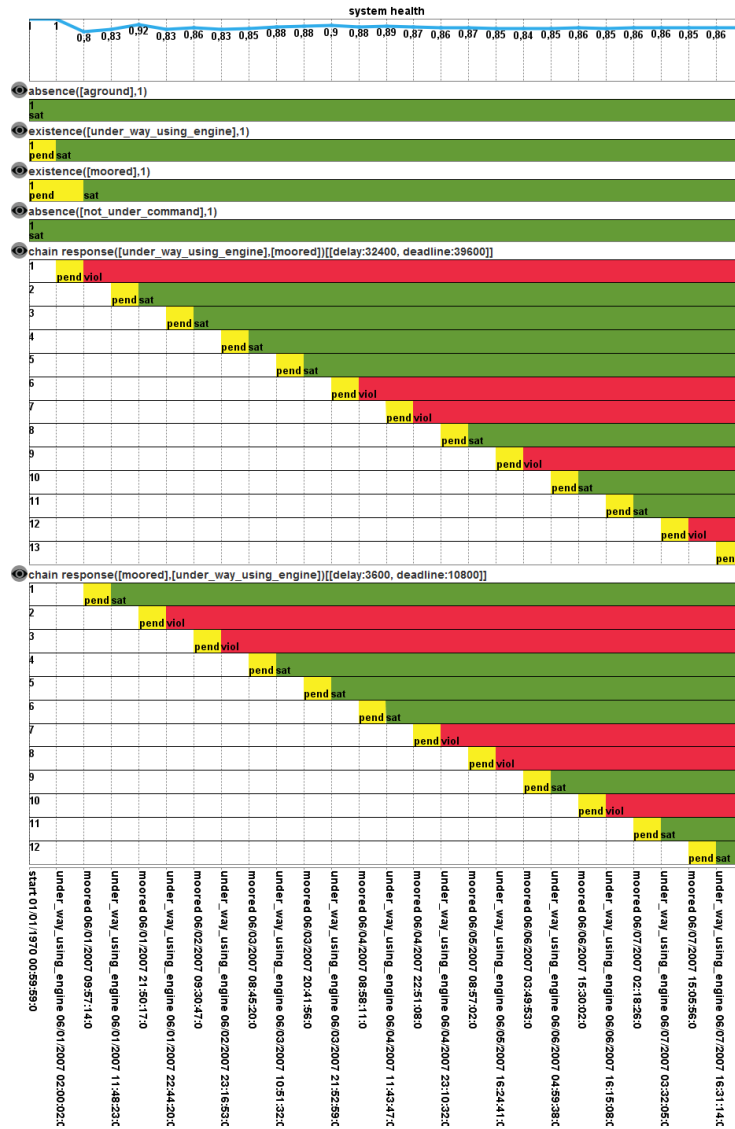


Fig. 9. Monitoring of a passenger ship

Mobucon EC. The basic information needed by the metrics, namely the number of violated, pending and satisfied constraint instances, can be easily obtained from the reasoner. E.g., the number V of violated instances at time T can be obtained by issuing the query $findall(, cur_state(, viol, T), L) \wedge length(L, V)$. By binding T to a specific value t , the query returns the number of instances violated at time t , whereas by leaving T variable, it returns the total number of violated instances so far.

The study of effective compliance metrics is outside from the scope of this paper. Sophisticated metrics can be designed, e.g., exploiting metric times or giving different weights to the Declare constraints. Here, we use a simple but significant metric, which computes the system health by considering the number of violated ($\#viol$) and satisfied ($\#sat$) instances while ignoring the pending ones (being not yet clear whether they will

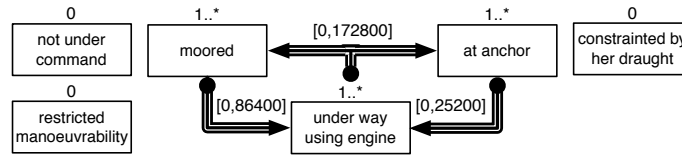


Fig. 10. Expected behavior for vessel type cargo ship having a cargo of type “hazard pollutant A”

be violated or satisfied). In particular, at some time t the system health corresponds to 1 when $\#viol(t) + \#sat(t) = 0$ and to $1 - \frac{\#viol(t)}{\#viol(t) + \#sat(t)}$ otherwise. As can be seen, in this case the system health is quite homogeneous and slightly varies around 0.86.

5.2.2. Cargo Ships. The normative behavior for the ship type *cargo ship* is represented by the Declare model in Fig. 10. In the remainder of this paper we simply refer to this as a cargo ship (although there are multiple cargo types). Events *constrained by her draught*, *restricted maneuverability* and *not under command* must never occur. Moreover, event *at anchor* must be followed by *under way using engine*. A cargo ship can anchor for at most 7 hours, therefore we specify a deadline of 25200 secs. for constraint *chain response([at anchor], [under way using engine])*. Event *moored* must also be followed by *under way using engine* and considering that a cargo ship cannot be moored in a harbor more than 24 hours, we specify a deadline of 86400 secs. for constraint *chain response([moored], [under way using engine])*. Finally, event *under way using engine* can be followed by *at anchor* or *moored*. We express this behavior by using a branched chain response constraint. We have also estimated that each journey of a cargo can take up to 2 days. Therefore, we specify a deadline of 172800 secs. for constraint *chain response([under way using engine], [moored], [at anchor])*.

Fig. 11 shows the monitoring of message streams received from two cargo ships.

5.3. Benchmark and Performance Assessment

For the practical application of our approach, the performance of Mobucon EC is of the utmost importance: unlike classical process mining approaches, monitoring requires short-term feedback every time the current state of affairs evolves and new event occurrences are acquired. Since we are interested in evaluating the performance of our EC-based approach, we focus on the core monitoring task, that is, on the reasoning engine per se, without considering the interaction among components inside ProM nor the time spent for visualization. This is why we chose YAP to carry out our evaluation. For the evaluation, we have set up a generator of Declare models⁸. It requires a set of configuration parameters that target the size and structure of the model: given

- a set \mathcal{A} of activity names,
- a flag selecting atomic vs non-atomic activities,
- the length L of the trace to be monitored,
- the number N of constraints to be included in the model,
- the minimum/maximum cardinality C associated to existence constraints,
- the maximum branching factor B of relation/negation constraints,
- the minimum delay d and maximum deadline D for timed constraints,

the generator produces a self-contained Prolog test theory containing the axiomatization of CEC and of activity lifecycle constraints, a randomly generated trace of L event occurrences of the atomic activities in \mathcal{A} (or their ports, in the non-atomic case) and the EC-based formalization of N randomly generated constraints attached to activities in \mathcal{A} , tuned according to the parameters B , d and D . We used this generator to produce

⁸The generator code and the benchmarks are available at <http://www.inf.unibz.it/~montali/tools.html>

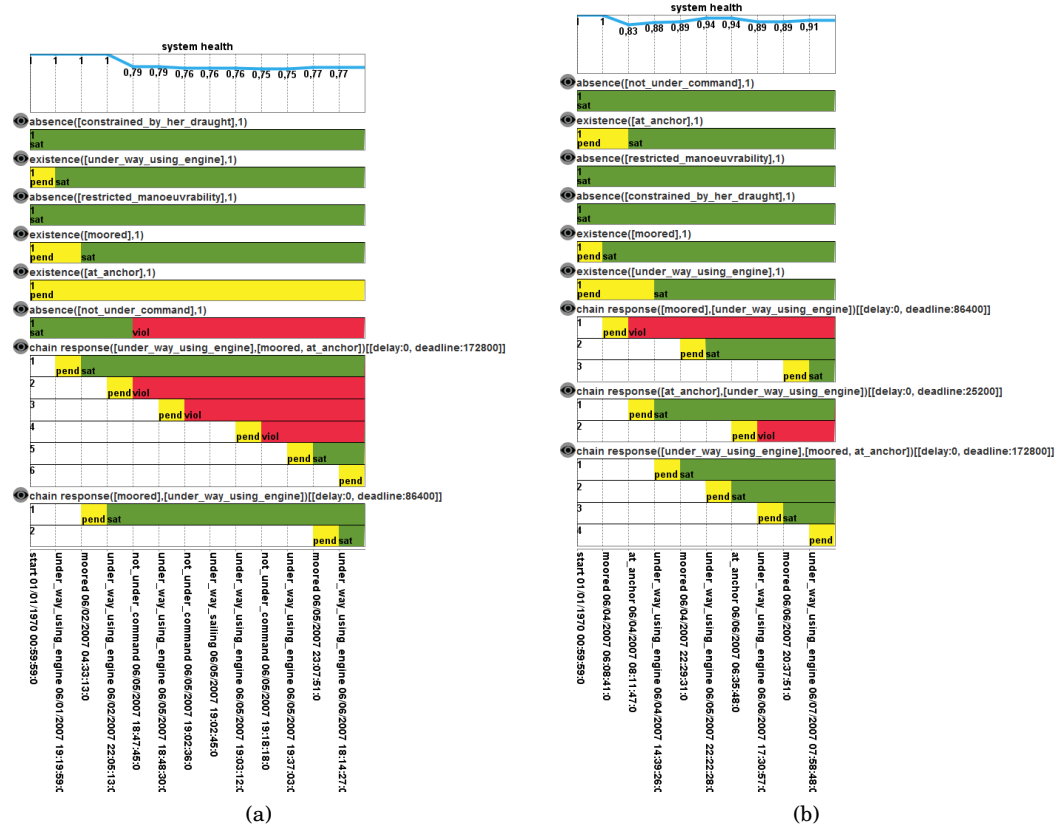


Fig. 11. Monitoring two cargo ships: the health of ship (a) drops to 0.77; the one of (b) varies from 0.83 to 1

two benchmarks, one for atomic and one for non-atomic activities. The first benchmark contains 1000 tests dealing with Declare models with atomic activities, tuned with $|A| = 10$, $L = 1000$, $C = 5$, $B = 3$, $d = 0$ and $D = 50$. These values were artificially tuned to reflect reasonable models and to focus our evaluation on the most critical parameters, namely the number of constraints and the length of the trace. It is worth noting that our approach is insensitive to some parameters. For example, the values for d and D only contribute to determine the average number of violated/satisfied instances of timed relation constraints, but the corresponding checks do not depend on the actual values (since they are tested using Prolog arithmetic built-in functionalities). A similar observation holds for the number of activities as well, since their names are strings tested for unification in the Prolog clauses. The benchmark contains 10 groups of tests by increasing the number of constraints in the model (with $N = 10, 20, \dots, 100$). Each group contains 100 tests obtained by combining 10 randomly generated traces with 10 randomly generated models. Tests are run by simulating the incremental acquisition of the event occurrences contained in the trace (ordered according to their timestamps) and measuring the latency time required by the reasoner to produce the updated monitoring result after each event. The obtained *average* results are reported in Fig. 12. They attest the scalability of Mobucon EC and its feasibility in dealing with models of real-life size. In fact, the 1000-th event is processed in less than 100 msecs for models

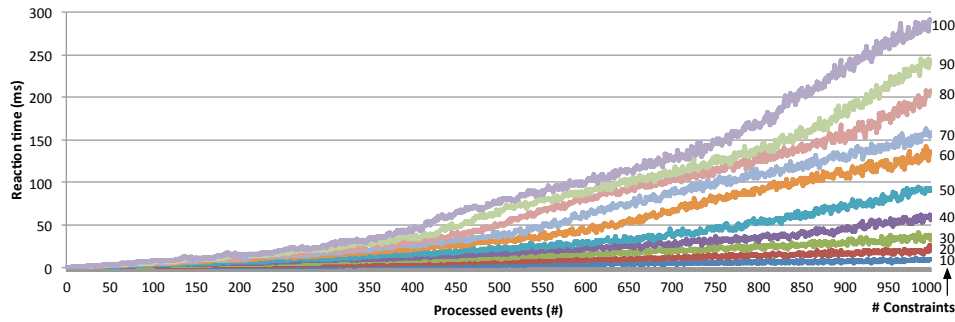


Fig. 12. Performance of Mobucon EC using a set of randomly generated benchmarks with atomic activities

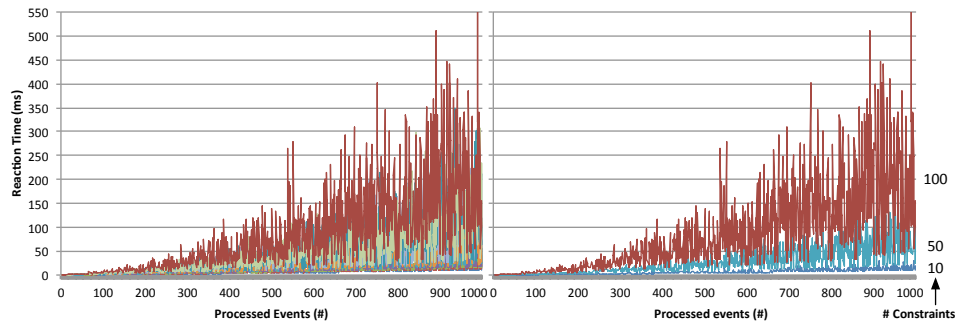


Fig. 13. Performance of Mobucon EC using a set of randomly generated benchmarks with non-atomic activities (for readability, on the right we report only three sets of tests out of the ten shown on the left)

containing up to 50 constraints and for large models with 100 constraints, still less than 300 msecs are required to update the monitoring result.

The second benchmark consists of 250 tests, generated in the same way (and with the same parameters) as for the first benchmark, but with non-atomic activities. These tests were organized in groups of 25, where each group contained from 10 to 100 constraints. Under the non-atomic activities assumption, the generator produces constraints that are randomly attached to one of the three ports associated to the selected activity. An important issue is, in this case, to generate “reasonable” traces, i.e., traces for which events can occur out-of-order (violating the lifecycle constraints), but where the majority of activities are executed properly. We solved this problem by bookkeeping the pair activity name and id attached to each start event, then randomly extracting such pairs so as to generate a consequent cancel/complete event. The average results for each group of tests are reported in Fig. 13. The fluctuation of the plotted values is due to the fact that the processing time is much less for those events that only trigger the evolution of a corresponding activity lifecycle, without affecting the status of constraints. The average processing time for a model containing up to 50 constraints does not exceed 170 msecs for any event in the trace, while, for a large model with 100 constraints, it reaches a maximum peak of ~ 550 msecs.

6. RELATED WORK

The problem of runtime verification and monitoring is omnipresent and has been investigated in different research communities. Similar problems have been studied in a variety of application domains ranging from runtime checking of software execution and hardware systems to self-adaptive and service-oriented applications. Due to this

diversity, comparing the existing approaches is difficult. In fact, each of them is suitable for specific application scenarios. For this reason, we will focus our discussion on the description of the characteristics of the existing techniques and on their conceptual differences with respect to the approach presented in this paper.

Spanoudakis and Mahbub [2006] present a framework for monitoring the compliance of a BPEL-based service compositions with assumptions expressed by the user or with behavioral properties that are automatically extracted from the composition. The EC is exploited to monitor the actual behavior of interacting services and to report different kinds of violations. Farrell et al. [2005] focus on the application of the EC to track the normative state of contracts. They formalize the deontic notions of obligation, power and permission, propose an XML-based dialect of the EC to model contractual statements and dynamically reason upon contract events reporting violations and diagnostics about the reached state of affairs. These proposals exploit ad-hoc event processing algorithms to manipulate events and fluents, written in JAVA. Hence, differently from Mobucon EC they do not have an underlying formal basis and they cannot take advantage of the expressiveness and computational power of logic programming, such as unification and backtracking.

As described in Section 5.1, Mobucon EC is based on logic programming and exploits a lightweight axiomatization of the CEC by Chittaro and Montanari [1996]. An alternative reactive but purely declarative axiomatization of the EC has been proposed in [Chesani et al. 2010], exploiting an abductive logic programming-based proof procedure for runtime compliance checking. A comparison of these two approaches, together with a performance evaluation showing that the axiomatization used by Mobucon EC outperforms the original axiomatization of CEC, is contained in [Bragaglia et al. 2012]. The application of our EC-based approach for monitoring service choreographies [Chesani et al. 2009] constitutes a preliminary investigation of the EC-based axiomatization of Declare proposed in this work. Here, we have extended it with the notion of constraint instances and their states, providing a more systematic support for quantitative time constraints and system health evaluation.

A plethora of authors have investigated the use of temporal logics – Linear Temporal Logic (LTL) in particular – as a declarative language for specifying properties to be verified at runtime⁹. LTL semantics for finite traces needs to be adapted to reflect that reasoning cannot always provide a definitive answer, but must be open to the acquisition of new events and to handle partial, finite traces. Consequently, also the verification techniques must be revised. For example, Giannakopoulou and Havelund [2001] present an approach to LTL runtime checking, where the standard LTL to Büchi automata conversion technique is modified to deal with finite traces. The algorithm produces a finite state automaton that is then used as observer of the system's behavior. Bauer et al. [2011] introduce a three-valued semantics (with truth values *true*, *false* and *inconclusive*) for LTL or timed LTL properties on finite traces. These truth values strictly resemble the *satisfied*, *violated* and *pending* values adopted by Mobucon EC to characterize the state of constraint instances.

When compared with these approaches, Mobucon EC supports quantitative time constraints, non-atomic activities with identifier-based correlation and can be extended with data-aware conditions. Furthermore, techniques based on finite-trace LTL usually associate the notion of violation to the one of logical inconsistency and are therefore not suitable for continuous support. An exception to this is [Maggi et al. 2011], where the automaton constructed for monitoring carries specific information that can be used to realize different recovery strategies for continuous support. An advantage of techniques relying on automata is their efficiency: their most expensive step

⁹See <http://runtime-verification.org/>

is the automaton construction, which is done before the execution and can be greatly optimized [Westergaard 2011]. Furthermore, they are not only able to check whether the current trace is compliant with the given LTL specification, but also to identify whether unavoidable violations will occur in the future. When it comes to Declare, this ability is exploited to identify whether the interplay of two or more constraints will surely lead to a future violation [Maggi et al. 2012]. Our EC-based approach is not able to foresee these violations: it treats each constraint separately from the others.

As described in Section 4.1, we assume that the monitored execution traces contain event identifiers that can be used for correlation, to detect multiple executions of the same activity. As we have seen, this is of utmost importance for the proper assessment of compliance and the consequent computation of overall “system health” metrics. In [Pauw et al. 2005], correlation mechanisms are used to realize an intuitive and comprehensive visualization framework for analyzing web service executions. Such mechanisms are not only used to provide fine-grained information about each web service transaction separately, but also to identify “chatty” communications, where repetitions of the same web service invocations are detected to show potential bottlenecks and suggest potential design improvements. However, as pointed out in [Dustdar and Gombotz 2006], there are cases in which process/activity instances are not directly present in the event logs of the system. For example, correlation identifiers are not present in standard web service logging facilities. How to tune the logging infrastructure and obtain the necessary correlation information is outside from the scope of this work. However, Mobucon EC can benefit of all approaches that aim to achieve correlation in settings where it is not directly provided. In the web service setting, correlation can be obtained by tuning the system so as to account for the needed “process information” [Dustdar and Gombotz 2006], e.g., by means of WS-Addressing or the strategies in [Nezhad et al. 2011]. Although [Nezhad et al. 2011] focuses on process instance-based correlation, i.e., on the isolation of all event occurrences that refer to the same process instance, we believe that a similar approach can be adopted also for activity instances.

7. DISCUSSION AND CONCLUSION

We have presented Mobucon EC, a non-intrusive framework for monitoring systems whose dynamics is characterized by means of event streams. The framework is based on a business constraint language that extends Declare with metric temporal conditions and non-atomic, durative activities. The Event Calculus (EC) is used to formalize the constraints to be monitored. One of the key advantages of logic-based approaches is the separation between declarative knowledge (*what* is the problem about) and control aspects (*how* to solve it). By exploiting Mobucon EC, Declare constraints can be formalized without specifying how to concretely monitor them. Furthermore, as long as the EC ontology does not change, different EC reasoners can be seamlessly applied to deal with alternative forms of reasoning, without affecting the proposed formalization.

It is interesting to note the relation between runtime verification and finiteness of traces. At each time point, a running execution is characterized by a finite trace. However, it is an evolving trace, extended each time another event is observed. This process could potentially continue forever. From the modeling point of view, this means that the Declare diagrams could contain cyclic relationships, accounting for expected situations that must be repeatedly achieved. This is illustrated by the passenger and cargo ship models shown in Figs. 8 and 10. No finite trace will fully comply with them: at least one of their chain response constraints would be pending (if the trace is partial) or violated (if the trace is complete). This has no impact on reasoning, which is triggered by the occurrence of new events, checking the partial finite trace stored so far.

Currently, we are extending our approach in different directions, incorporating data, resources (i.e., activity originators) and also recovery and compensation mechanisms to

be instantiated in the case of a violation. For recovery and compensation, we will rely on the preliminary proposal made in [Chesani et al. 2009], where the violations are reified as special events. Making Declare data-aware is instead concerned with augmenting activities and constraints with additional information and conditions about involved data and resources (beside lifecycle events and timestamps) [Montali 2010]. Technically, the framework here presented provides the basis for this kind of extension: to accommodate data, lifecycle events and the “context” of each constraint instance must be extended with the activity data (such as, e.g., the *price* of an order), while the EC axioms modeling the state transitions of constraints can exploit these additional data to include further conditions (e.g., that the order’s price cannot exceed a given threshold). Consider, e.g., an extended *response* constraint between two activities *a* and *b*, with the additional requirement that the *originator* (i.e., the responsible resource) of *b* must be different than the one of *a* to satisfy the constraint (this is a form of “four-eyes principle”). By introducing the originator as an additional parameter of the lifecycle events, the *response creation* Axiom 10 becomes

$$creation(compl(-, A, O), T, i(id, [A, O], T), pend) \leftarrow member(A, [a]).$$

The presence of *O* in the contextual information of the created instance can then be used in the *response fulfillment* axiom (Axiom 11 in Section 3), to trigger the corresponding state transition only if *b* is executed by an originator O_2 different from *O*:

$$trans(start(-, B, O_2), T, i(id, [A, O], T_i), pend, sat) \leftarrow cur_state(i(id, [A, O], T_i), pend, T) \wedge member(B, [b]) \wedge O_2 \neq O \wedge inside_future(T_i, m, n, T).$$

This form of data-aware extension has been recently pursued in [Montali et al. 2013].

Another active research line concerns a comprehensive comparison of compliance monitoring approaches, to assess their functionalities and performance. Although there is a flourishing literature on monitoring approaches tailored to the measurement of quantitative indicators such as Key Performance Indicators, we are interested in monitoring compliance with complex business constraints. In this respect, we are currently producing a literature review on compliance monitoring approaches, so as to devise a framework for the systematic qualitative comparison of compliance monitoring functionalities. A quantitative evaluation of the state of the art is still difficult because most of the approaches are not supported by publicly available software tools, and due to the lack of shared benchmarks. An interesting matter of investigation will therefore be to check whether the synthetic benchmarks adopted in this work can constitute the basis for this quantitative comparison.

ACKNOWLEDGMENTS

The authors would like to thank Arjan Mooij for his valuable hints and comments on the paper.

REFERENCES

- ARTIKIS, A. AND PALIOURAS, G. 2009. Behaviour recognition using the event calculus. In *Proceedings of AIAI*. Springer.
- BAUER, A., LEUCKER, M., AND SCHALLHART, C. 2007. The good, the bad, and the ugly, but how ugly is ugly? In *Proceedings of RV*. LNCS. Springer.
- BAUER, A., LEUCKER, M., AND SCHALLHART, C. 2011. Runtime verification for LTL and TLTL. *ACM Transaction on Software Engineering Methodologies*.
- BRAGAGLIA, S., CHESANI, F., MELLO, P., MONTALI, M., AND TORRONI, P. 2012. *Logic Programs, Norms and Action*. LNCS. Springer, Chapter Reactive Event Calculus for Monitoring Global Computing Applications.
- CHESANI, F., MELLO, P., MONTALI, M., AND TORRONI, P. 2009. Verification of choreographies during execution using the reactive event calculus. In *Proceedings of WS-FM*. LNCS. Springer, 55–72.

- CHESANI, F., MELLO, P., MONTALI, M., AND TORRONI, P. 2010. A logic-based, reactive calculus of events. *Fundamenta Informaticae* 1-2.
- CHITTARO, L. AND MONTANARI, A. 1996. Efficient temporal reasoning in the cached event calculus. *Computational Intelligence* 12.
- CLARK, K. L. 1978. Negation as failure. In *Logic and Data Bases*. Plenum Press, 293–322.
- DUSTDAR, S. AND GOMBOTZ, R. 2006. Discovering web service workows using web services interaction mining. *International Journal of Business Process Integration and Management*.
- FARRELL, A. D. H., SERGOT, M. J., SALLÉ, M., AND BARTOLINI, C. 2005. Using the event calculus for tracking the normative state of contracts. *Cooperative Information Systems* 2-3.
- GIANNAKOPOULOU, D. AND HAVELUND, K. 2001. Automata-based verification of temporal properties on running programs. In *Proceedings of ASE*. IEEE Computer Society.
- International Telecommunications Union 2001. *Technical characteristics for a universal shipborne Automatic Identification System using time division multiple access in the VHF maritime mobile band*. International Telecommunications Union. Recommendation ITU-R M.1371-1.
- KOWALSKI, R. A. AND SERGOT, M. J. 1986. A logic-based calculus of events. *New Generation Computing*.
- LUCKHAM, D. 2001. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley.
- MAGGI, F. M., BOSE, R. P. J. C., AND VAN DER AALST, W. M. P. 2012. Efficient Discovery of Understandable Declarative Models from Event Logs. In *CAiSE*, J. Ralyté, X. Franch, S. Brinkkemper, and S. Wrycza, Eds. Lecture Notes in Computer Science Series, vol. 7328. Springer-Verlag, Berlin, 270–285.
- MAGGI, F. M., MONTALI, M., WESTERGAARD, M., AND VAN DER AALST, W. M. P. 2011. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *Proceedings of BPM*. Springer.
- MAGGI, F. M., MOOIJ, A. J., AND VAN DER AALST, W. M. P. 2012. *Analyzing Vessel Behavior using Process Mining*. Chapter Poseidon book. to appear.
- MAGGI, F. M., WESTERGAARD, M., MONTALI, M., AND VAN DER AALST, W. M. P. 2012. Runtime verification of ltl-based declarative process models. In *Proceedings of RV*. LNCS. Springer.
- MONTALI, M. 2010. *Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach*. LNBIP Series, vol. 56. Springer.
- MONTALI, M., CHESANI, F., MAGGI, F. M., AND MELLO, P. 2013. Towards data-aware constraints in declare. In *Proceedings of SAC*. To appear.
- MONTALI, M., PESIC, M., VAN DER AALST, W. M. P., CHESANI, F., MELLO, P., AND STORARI, S. 2010. Declarative specification and verification of service choreographies. *ACM Transactions on the Web* 1.
- NEZHAD, H. R. M., SAINT-PAUL, R., CASATI, F., AND BENATALLAH, B. 2011. Event correlation for process discovery from web service interaction logs. *VLDB Journal*.
- PAUW, W., LEI, M., PRING, E., VILLARD, L., ARNOLD, M., AND MORAR, J. 2005. Web services navigator: Visualizing the execution of web services. *IBM Systems Journal*.
- PESIC, M. 2008. Constraint-based workflow management systems: Shifting controls to users. Ph.D. thesis, Beta Research School for Operations Management and Logistics, Eindhoven.
- PESIC, M., SCHONENBERG, H., AND VAN DER AALST, W. M. P. 2007. Declare: Full support for loosely-structured processes. IEEE Computer Society.
- PESIC, M. AND VAN DER AALST, W. M. P. 2006. A declarative approach for flexible business processes management. In *Proceedings of BPM Workshops*. LNCS. Springer.
- SADRI, F. AND KOWALSKI, R. A. 1995. Variants of the event calculus. In *Proceedings of ICLP*. MIT Press.
- SHANAHAN, M. 1999. The event calculus explained. In *Artificial Intelligence Today: Recent Trends and Developments*. LNCS. Springer.
- SPANODAKIS, G. AND MAHBUB, K. 2006. Non-intrusive monitoring of service-based systems. *Cooperative Information Systems* 3, 325–358.
- VAN DER AALST, W. M. P. 2011. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer.
- VAN DER AALST, W. M. P., VAN HEE, K. M., VAN DER WERF, J. M. E. M., AND VERDONK, M. 2010. Auditing 2.0: Using Process Mining to Support Tomorrow’s Auditor. *IEEE Computer* 43, 3.
- VERBEEK, E., BUIJS, J., VAN DONGEN, B., AND VAN DER AALST, W. 2010. ProM 6: The Process Mining Toolkit. In *Proceedings of BPM*.
- WESTERGAARD, M. 2011. Better algorithms for analyzing and enacting declarative workflow languages using ltl. In *Proceedings of BPM*. Springer.