

Università degli Studi di Bologna  
DEIS

# Monitoring Business Constraints with the Event Calculus

MARCO MONTALI    FABRIZIO M. MAGGI  
FEDERICO CHESANI    PAOLA MELLO  
WIL M. P. VAN DER AALST

June 4, 2012

# Monitoring Business Constraints with the Event Calculus

MARCO MONTALI<sup>1</sup>      FABRIZIO M. MAGGI<sup>2</sup>  
FEDERICO CHESANI<sup>3</sup>      PAOLA MELLO<sup>3</sup>  
WIL M. P. VAN DER AALST<sup>2</sup>

<sup>1</sup>*Free University of Bozen-Bolzano*  
*Research Centre for Knowledge and Data (KRDB)*  
montali@inf.unibz.it

<sup>2</sup>*Eindhoven University of Technology*  
*Department of Mathematics & Computer Science*  
{f.m.maggi | w.m.p.v.d.aalst}@tue.nl

<sup>3</sup>*University of Bologna*  
*Department of Electronics, Computer Science and Systems (DEIS)*  
{federico.chesani | paola.mello}@unibo.it

June 4, 2012

**Abstract.** 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 behaviours. This in turn trigger the need for runtime verification and monitoring facilities. These are needed to check whether the actual behaviour complies with expected business constraints.

In this work we present MOBUCON, a novel monitoring framework that tracks streams of events and continuously determines the state of business constraints. In MOBUCON, business constraints are defined using *ConDec*, a declarative process modelling language. For the purpose of this work, ConDec has been suitably extended to support quantitative time constraints and non-atomic, durative activities. Then, the logic-based language *Event Calculus* (EC) has been adopted for the formal specification of constraints, and a light-weight, logic programming-based EC axiomatization has been exploited for dynamically reasoning about partial, evolving execution traces. MOBUCON has been integrated within the operational decision support architecture of ProM. To demonstrate the applicability of our proposal, we provide also a concrete case study dealing

with maritime safety and security.

**Keywords:** *Business Constraints, Event Calculus, Runtime Verification, Monitoring, Operational Decision Support*

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>The ConDec Notation</b>	<b>5</b>
2.1	ConDec models . . . . .	6
2.2	Constraints . . . . .	6
2.3	Extending ConDec with Metric Constraints and Non-Atomic Activities	9
<b>3</b>	<b>The Event Calculus</b>	<b>10</b>
3.1	A Brief Introduction to EC . . . . .	10
3.2	The EC Ontology . . . . .	11
3.3	EC Theories . . . . .	12
3.4	Reasoning About EC Theories . . . . .	13
<b>4</b>	<b>Formalizing ConDec in the Event Calculus</b>	<b>14</b>
4.1	Process Execution Traces . . . . .	14
4.2	Formalizing the Activity Lifecycle . . . . .	15
4.3	Business Constraint Instances and Their States . . . . .	18
4.4	Formalization of ConDec Constraints . . . . .	19
4.4.1	Existence Constraint . . . . .	20
4.4.2	Absence Constraint . . . . .	21
4.4.3	Metric Response Constraint . . . . .	22
4.4.4	Metric Chain Response Constraint . . . . .	23
4.5	Implementation . . . . .	23
<b>5</b>	<b>Case Study</b>	<b>25</b>
5.1	Monitoring the behavior of a vessel . . . . .	25
5.1.1	Passenger Ships . . . . .	26
5.1.2	Cargo Ships . . . . .	28
5.2	Benchmarks and Performance Evaluation . . . . .	29
<b>6</b>	<b>Related Work</b>	<b>32</b>
<b>7</b>	<b>Conclusion</b>	<b>34</b>

# 1 Introduction

Many systems operate today in a dynamic, complex and interconnected environment. Larger systems are composed of smaller systems and evolve over time. These characteristics make it difficult to ensure that the composed system behaves 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. Moreover, there is a continuous pressure to meet deadlines and improve response times. We use the term *(business) constraint* [14] to refer to any rule that restricts 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 the constraints can be incorporated in the executable description. First, the integration of diverse and heterogeneous constraints would quickly make models unreadable and tricky. This difficulty would become even more critical when system behavior is modeled using procedural, workflow-like approaches, as business constraints are inherently declarative [18, 15]. Second, 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. Nevertheless, as argued by [22], detailed information about the executed activities of a system is nowadays stored and made available in high-quality event logs. This makes it possible to apply process mining [23] techniques to “evaluate all events in a business process, and do so while it is still running”. The runtime aspect is of particular importance: non-compliant state of affairs could be associated to wrong behaviors, dangerous situations or fraud; they must be therefore promptly detected, possibly generating suitable alerts and triggering recovery or compensation mechanisms. The application of process mining techniques to monitor and guide running cases is relatively new as most authors and systems focused on offline analysis. The application process mining techniques at runtime is referred to as *operational (decision) support* [22]. 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.

In particular, operational support exploits process mining techniques 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 (MONitoring BUbusiness CONstraints), 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. MOBUCON relies on the Event Calculus (EC) [10, 20] as a logic-based, expressive language for the formal specification of constraints, and on a light-weight, logic programming-based EC axiomatization for dynamically reasoning about partial, evolving traces. Unlike

approaches that bind the notion of constraints violation to logical inconsistency (thus halting when the first violation is encountered), MOBUCON provide *continuous support*, without interrupting its functioning even after a violation. This is 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 show how MOBUCON can be equipped with an EC-based formalization of ConDec [18], a graphical language for the declarative and open specification of business constraints. In particular, we focus on an extended version of ConDec which supports quantitative time aspects, such as delays and deadlines [15, 14]. We do not tackle data-related aspects, but they can be incorporated in our approach.

MOBUCON has been fully implemented inside ProM [24], one of the most widely used process mining frameworks. ProM 6 embeds an operational decision support infrastructure. It can be applied to monitor any system whose dynamics is represented by event streams. To demonstrate its applicability, we present a case study in the context of maritime safety and security, where constraints pose requirements on the behavior of vessels that cross a specific area.

The remainder is organized as follows. Sects. 2 and 3 provide some preliminaries, introducing the ConDec language and the EC. Sect. 4 discusses how ConDec constraints and process execution traces can be formalized as an EC theory, and how MOBUCON has been implemented inside the ProM operational support architecture. Sec. 5 then discusses the application of the framework in the context of maritime safety and security. An overview of related work and a conclusion complete the paper.

## 2 The ConDec Notation

We provide a brief introduction to ConDec. For a comprehensive description of the language and its features, including the complete listing of all constraints supported by the language, we refer the interested reader to [18, 16, 14].

ConDec is a graphical, *constraint-based, declarative* language for the 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, ConDec provides a number of control-independent business constraints to interconnect activities, alongside the more traditional ones. 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, Con-

Dec models are *open*: activities can be freely executed, unless they are subject to constraints. On the one hand, openness guarantees flexibility: all the executions that do not violate any constraint are implicitly supported by the business process. On the other hand, to tune the degree of openness supported by the model, suitable abstractions are dedicated to explicitly capture not only what is indispensable, 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 the desired and undesired events, leaving unconstrained all the courses of interaction that are neither mandatory nor forbidden.

Even if concurrency constructs are not explicitly present in the ConDec notation, concurrency is implicitly supported at the best: as long as the stakeholders involved in a process instance behave within the boundaries imposed by the ConDec constraints, they are free to choose the most appropriate way of executing activities. For example, a ConDec constraint stating that “activities  $a$  and  $b$  must coexist in the same process instance” supports executions in which neither  $a$  nor  $b$  are executed, or executions in which  $a$  and  $b$  are both executed in any possible ordering, including the case in which they are executed in parallel.

## 2.1 ConDec models

A *ConDec monitoring model* is composed of a set of business constraints to be monitored, applied to a set of activities. In the basic setting, each activity represents an *atomic* units 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 ConDec constraints applied over atomic activities, and then discuss extensions of such basic setting that are fully covered by our monitoring framework.

A ConDec 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 (*flexibility*). Then, ConDec constraints are added to capture the business constraints of the system, leading to a partially closed model (*compliance*).

## 2.2 Constraints

To support different application domains, ranging from closed, prescriptive settings to more flexible, adaptive ones, ConDec supports a variety 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 ConDec model



contains an *existence 2* and *absence 1* constraints, specifying that activity *get info* must be executed twice and that activity *publish info* should not be executed at all (i.e., it is forbidden to execute it once). 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 inside a set of possible alternatives.

*Relation* constraints are *binary* constraints requiring the execution of some activity when certain circumstances hold. In particular, every time the source activity of the constraint is executed, the relation constraint *expects* the execution of another (target) activity. As shown in Tab. 1, depending on the type of relation constraint, additional requirements may be imposed on the target activity, hardening or softening the constraint.

Furthermore, it is worth noting that relation constraints cover all 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 the Table, but included in Fig. 1) imposes a “before” ordering. In fact, the basic version ConDec only supports a qualitative notion of time: constraints could specify the expected relative positions among two event occurrences, but they cannot express with metric distances between them. This limitation was due a result of the LTL-based semantics of the initial language. Later in this paper, we show that this limitation can be removed by using the Event Calculus as a semantical foundation.

Another important aspect of the language is that relation constraints can be generalized in such a way that multiple source activities and/or multiple 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 that has  $a$  as source and  $b$  and  $c$  as targets is satisfied if, whenever  $a$  is executed, one among  $b$  and  $c$  is executed next. In other words, constraints with a branching target contain an implicit *choice*.

*Negation* constraints are the negative version of relation constraints, since they



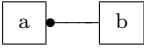
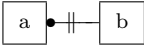
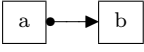
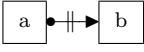
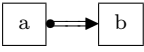
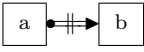
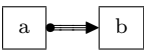
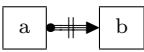
	<b>Responded existence</b> When $a$ is executed, so must $b$ , either before or afterwards		<b>Responded absence</b> If $a$ is executed, $b$ can be never executed in the same case
	<b>Response</b> Every time $a$ is executed, $b$ must be eventually executed as well		<b>Negation response</b> When $a$ is executed, $b$ cannot be executed afterwards
	<b>Alternate response</b> Every time $a$ is executed, $b$ must be consequently executed, before a further occurrence of $a$		<b>Negation alternate response</b> If $a$ is executed twice, $b$ cannot be executed between the two occurrences of $a$
	<b>Chain response</b> Every time $a$ is executed, $b$ must be executed next (i.e., as the first consequent activity)		<b>Negation chain response</b> Every time $a$ is executed, $b$ cannot be executed next (i.e., as the first consequent activity)

Table 1. *Some ConDec relation constraints, together with their corresponding negated version*



Figure 1. *An order management process in ConDec*

*forbid* the execution of some activity when a certain state of affairs is reached. Tab. 1 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 where 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 ConDec. It deals with the flexible management of an order. A customer has the possibility of adding items to an order by means of the *choose good* activity, of submitting the order to the seller, and of 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 good has been chosen (*precedence (1)*), and no further goods can be chosen after having submitted an order (*negation response*). An order can be paid only if 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 ConDec: 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 good*  $\rightarrow$  *choose good*  $\rightarrow$  *submit order* is compliant with the model. Trace *choose good*  $\rightarrow$  *submit order*  $\rightarrow$  *choose good* is instead non-compliant: 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).

### 2.3 Extending ConDec with Metric Constraints and Non-Atomic Activities

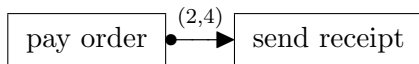
The basic ConDec notation has been extended to deal also with a non-atomic model of activities [16, 14], quantitative time constraints [15], and activity-related data as well as data-aware conditions [14]. In this work, we mainly focus on the first two extensions, showing how they can be both formalized in the EC for monitoring. Nevertheless, the approach used for modeling non-atomic activities can be exploited to incorporate data-aware conditions as well.

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 therefore 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 to properly handle non-atomic activities. In this paper, we adopt the simple activity lifecycle proposed by Pesic for ConDec but the approach is seamlessly able to cover more complex lifecycles as well. In this lifecycle, each activity instance is associated to a *start* event, marking the beginning of the activity instance, and a consequent *completion* or *cancelation* event, respectively marking the proper or premature termination of the activity instance. These three event types are connected by the following lifecycle constraints:

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

Notice that the lifecycle does not only impose a suitable ordering among events, but also requires that there is a one-to-one matching between every start event and a consequent completion or cancelation one.

While non-atomic activities are common in a multitude of different domains, especially those in which complex processes are modeled at multiple levels of abstraction, quantitative time constraints are especially interesting in the context of monitoring. In fact, monitoring is not concerned with the *enforcement* of a desired behavior, but rather in checking whether an uncontrollable process instance is evolving within the expected behavioral boundaries. As illustrated by our case study in Sec. 5, it is common for business constraints to incorporate metric time aspects, such as *delays* and *deadlines*. As proposed in [15], time-ordered ConDec 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 has effect. This time span is interpreted as relative with respect to the time at which the source activity of the constraint is executed. For example, we can easily 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:



### 3 The Event Calculus

We provide an overview of the basic concepts underlying the Event Calculus (EC), which is the formal framework underlying MOBUCON. 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 EC

In 1986, Kowalski and Sergot [10] proposed the EC as a general framework to reason about time, events and change, overcoming the inadequacy of time representation in classical logic. It 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:

- a domain-independent general theory axiomatizing the *meaning* of the predicates supported by the calculus, i.e., the so called *EC ontology*;
- a domain theory which *exploits* the predicates of the EC ontology to formalize the specific system under study in terms of events and their effects, i.e., fluents.

$\text{happens}(\text{Ev}, T)$	Event $Ev$ happens at time $T$
$\text{holds\_at}(F, T)$	Fluent $F$ holds at time $T$
$\text{initially}(F)$	Fluent $F$ holds in the initial state of the system
$\text{initiates}(\text{Ev}, F, T)$	Event $Ev$ initiates fluent $F$ at time $T$
$\text{terminates}(\text{Ev}, F, T)$	Event $Ev$ terminates fluent $F$ at time $T$

Table 2. *The basic Event Calculus ontology*

Our domain theory will focus on the formalization of the ConDec language.

Starting from the seminal work of Kowalski and Sergot, a number of EC dialects have been proposed [19] 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 describe which predicates are 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 [6], we will use Prolog as the specification language.

### 3.2 The EC Ontology

[20] 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 Tab. 2.

“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) impact 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  $t$ . Note that fluents still hold when they are clipped, but they do not hold at the time they are declipped, i.e., the maximal validity interval of fluents are left-open and right-closed.

“What happens when” is the execution trace characterizing a (possibly partial) instance of the system under study. An execution trace is composed of a set of occurred events. As in the case of ConDec, the basic forms of EC assume that events are atomic, i.e., bound to a single time point. In particular, an execution trace is composed of a set of *happens* predicates, listing the occurrences of events and their

corresponding timestamps. Concerning timestamps, the EC adopts a time structure with a minimal element, usually associated to time point 0, that represents the initial state of the system. Since event occurrences are associated to discrete timestamps, we rely on the 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 milliseconds or days). E.g., by choosing ms as the time granularity, each timestamp  $ts$  could be mapped to the number of milliseconds between 1/1/1970 00:00 and  $ts$ .

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  $holds\_at(f, t)$  predicate of the EC ontology is specifically used to test whether  $f$  holds at time  $t$ .

### 3.3 EC Theories

An EC theory exploits the predicates of the EC ontology in order to formalize how domain-specific events affect domain-specific fluents. In our setting, it 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 usual, variables are universally quantified with scope the entire clause. Hence, the fact  $initiates(e, f, T)$ <sup>1</sup> 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 nontrivial example of EC theory is provided in the following example. Sec. 4 will present an EC theory which formalizes ConDec.

**Example 1.** *Let us consider a system characterized by 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 multi-valued fluent  $tot(V)$  to represent that the current total amount corresponds to  $V$ , and use the following EC theory to relate it to the payment event:*

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

*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 ceases to hold. The third clause updates the total amount by initiating a new fluent whose amount corresponds to the current amount plus the paid euros.*

<sup>1</sup>Which corresponds to clause  $initiates(a, f, T) \leftarrow true$ .

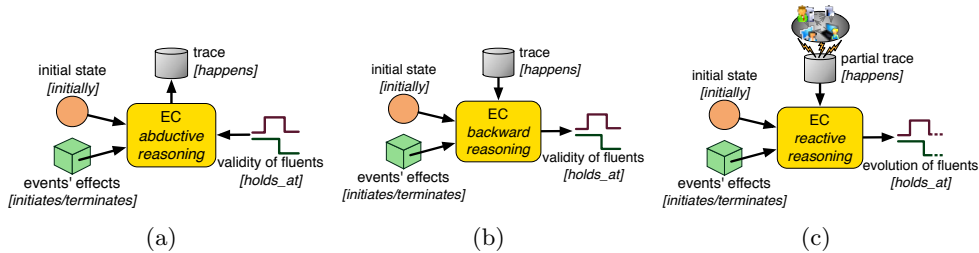


Figure 2. *Abductive, backward and reactive reasoning in the EC setting*

### 3.4 Reasoning About EC Theories

Two main reasoning tasks are usually exploited in the EC setting: *abductive* and *deductive* reasoning [20]. Abductive reasoning (Fig. 2(a)) starts from an EC domain theory and a query representing a desired state of affairs (expressed as a conjunction of  $[\neg]holds\_at$  predicates), and tries to generate a sample trace which respects the domain theory and at the same time achieves that state of affairs. Conversely, deductive reasoning (Fig. 2(b)) takes an external trace and combines it with an EC domain theory, inferring the validity intervals of fluents and answering to given queries (again expressed as conjunctions of  $[\neg]holds\_at$  predicates). Usually, deductive reasoning is carried out in a starting from the query and reasoning backward.

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. Second, 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. Therefore, monitoring calls for *reactive reasoning* (Fig. 2(c)), where fluents' validity intervals are revised/extended 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 *true*. While this approach would in principle work, it is computationally expensive, because it must be restarted from scratch every time the trace is updated, completely forgetting the previously calculated result. This makes backward deductive reasoning practically inapplicable even for small-size problems.

[5] 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. In particular, they developed a Prolog-based Cached EC (CEC), which caches the MVIs of fluents. An MVI is a Maximal Validity Interval inside which a fluent uninterruptedly holds. Upon a new event occurrence,

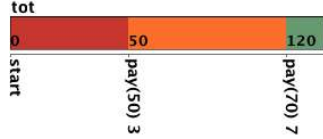


Figure 3. The evolution of a (multi-valued) fluent as graphically depicted by MOBUCON

CEC exploits the cached MVIs to compute the new result. MOBUCON adopts a CEC-inspired implementation for monitoring EC-based specifications (see Sec. 4.5).

**Example 2.** *Let us consider the EC theory described in Ex. 1, and a specific stream of events. 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 ( $\infty$  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. 3 shows how MOBUCON 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 ConDec in the Event Calculus

In this section we show how the EC can be used to formalize ConDec. As mentioned in Sect. 2.3 this formalization overcomes limitations of the classical LTL-based formalization, e.g., the metric time extension allows for the specification of arbitrary intervals, e.g., after each occurrence of activity  $a$ , activity  $b$  should happen within 50 time units. We discuss the representation of process execution traces, and then focus on ConDec 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 much” a running case is complying with the constraints model under study.

### 4.1 Process Execution Traces

A plethora of systems supporting the execution of multi-party interactions and activities provide logging capabilities. For example, most information systems log all

the events tracing the execution of business processes inside the organization.

On the one hand, each organization is characterized by its own specific events, whose semantics and content depends 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, 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 minimal set of information needed to trace the execution of non-atomic activities. In particular, we will take into account, for each event occurrence (or event instance): the *event type*, an event instance *identifier*, the *name* of the involved activity, and the *timestamp* at which the event instance has occurred. We assume that the identifier is used to correlate event instances that belong to the same lifecycle, i.e., to bind each start event instance to a corresponding cancelation/completion ones. The event type corresponds to *e* (“executed”) for atomic activities, whereas it corresponds to one of the characteristic lifecycle event types (*s* for “start”, *x* for “cancelation”, *c* for “completion”) for non-atomic ones. An execution trace, which is simply a set of correlated events (e.g., a business process case), can be then formalized in the EC by means of *happens* predicates. More specifically, an execution trace  $\mathcal{T}$  is a finite set  $\mathcal{T} = \{happens(ev(id_i, type_i, a_i), t_i) \mid id \text{ is an event identifier, } type \in \{e, s, x, c\}, a_i \text{ is an activity name, } t_i \text{ is a timestamp}\}$ .

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 supposed to be implicitly included in each ConDec model, and that its (only) execution marks the termination of the process instance.

**Definition 1.** *An EC trace  $\mathcal{T}$  is well-formed if  $\mathcal{T}$  is partial, i.e.  $happens(ev(-, d, case\_complete), -) \notin \mathcal{T}$ , or  $\mathcal{T}$  is total, i.e., only one *case\\_complete* event belongs to  $\mathcal{T}$  and *case\\_complete* is the last event occurrence in  $\mathcal{T}$ .*

In the following, we will always assume that traces are well-formed.

## 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 Sec. 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 fact, we assume that only “correct” event occurrences have an impact on constraints.



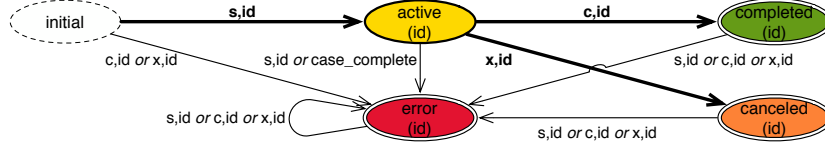


Figure 4. A simple lifecycle for non-atomic activities using a correlation identifier and including an error state

To tackle this issue, we introduce a set of EC rules that formalize the lifecycle shown in Fig. 4, where transitions are mapped onto events, and fluents represent the *active*, *completed* and *canceled* states. We 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 correlating the event identifiers, following the structure described in Sec. 4.1. More specifically, an event occurrence advances some activity instance if it refers to that activity and contains the same identifier. This is needed to properly manipulate activity instances and to support the parallel execution of multiple instances of the same activity. 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, but 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 ConDec, in order to guarantee the aforementioned principle that only correct event occurrences affect the constraints to be monitored. A further set of axioms is used to bind such inferred events to the corresponding state transitions.

**Axiom 1** (Effective start). *An activity instance is effectively started by a start event occurrence, if such occurrence does not lead to an error:*

$$\begin{aligned} happens(start(ID, A), T) \leftarrow & happens(ev(ID, s, A), T) \\ & \wedge \neg initiates(ev(ID, s, A), l\_state(i(ID, A), error), T). \end{aligned}$$

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

$$initiates(start(ID, A), l\_state(i(ID, A), active), T).$$

**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  $T$ .*

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

*The effective completion triggers a transition to the completed state:*

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

**Axiom 3** (Effective cancelation). *The effective cancelation of an activity instance mirrors the axioms used for effective completion.*

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

Beside the three aforementioned states, we also 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. 4. It is worth noting that some errors 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 permanence in the current state (provided that it is not already an error state):*

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

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

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

*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 and is associated to the same identifier:*

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

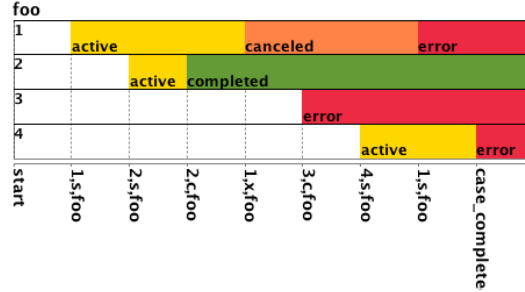


Figure 5. Sample evolutions of four instances of activity “foo”

Finally, 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. 5 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

During the execution of a case, the events composing its partial execution trace impact on the *state* of each modeled business constraint. Furthermore, while some constraints are active from the beginning of the execution and follow a unique evolution as events occur, other constraints are characterized by multiple, parallel and independent evolutions. We present two examples that highlight these aspects.

**Example 3.** 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.** 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 instantiation of the constraint. The specific 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 a single-instance constraint, and response a multiple-instance one. More generally, “cardinality” and choice constraints are single-instance, while all the relation and negation constraints are instantiated multiple times, every time the constraint’s source occurs. 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. In this work, we limit our analysis to activities and temporal aspects, hence the contextual information for event occurrences and constraint instances is composed of the name of the executed activity and its timestamp. More specifically, by associating a unique identifier to each modeled constraint, term  $i(id, a, t)$  denotes the instance of constraint  $id$  that has been created by the execution of  $a$  at time  $t$ . At each time point, every constraint instance can be in one among the following states:

- *pending* is a transient state representing the fact that the constraint is waiting for the occurrence of some event;
- *satisfied* is a transient or permanent state indicating that the execution trace is currently compliant with the constraint;
- *violated* is a permanent state attesting that the instance has been violated by the ongoing case.

In order 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  $I$ , ranging over the three possible values discussed above. For example,  $state(i(id, a, t), pending)$  represents the fact that instance  $i(id, a, t)$  is currently *pending*.

#### 4.4 Formalization of ConDec Constraints

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<sup>2</sup>. The first two rules deal with the creation of an instance, putting it in the specified state, either at the beginning of the execution or when some event

---

<sup>2</sup>Remember that  $state(I, S)$  represents that instance  $I$  is in state  $S$ .

occurs:

$$\begin{aligned} & \textit{initially}(\textit{state}(I, S)) \leftarrow \textit{init\_state}(I, S). \\ & \textit{initiates}(E, \textit{state}(I, S), T) \leftarrow \textit{creation}(E, T, I, S). \end{aligned}$$

The *cur\_state* predicate is instead defined to test the current state of an instance:

$$\textit{cur\_state}(I, S, T) \leftarrow \textit{holds\_at}(\textit{state}(I, S), T).$$

A group of two rules is finally 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 is applied by terminating the fluent associated to  $s_1$  and initiating the one bound to  $s_2$ .

$$\begin{aligned} & \textit{terminates}(E, \textit{state}(I, S_1), T) \leftarrow \textit{trans}(E, T, I, S_1, S_2). \\ & \textit{initiates}(E, \textit{state}(I, S_2), T) \leftarrow \textit{trans}(E, T, I, S_1, S_2) \wedge \textit{holds\_at}(\textit{state}(I, S_1), T). \end{aligned}$$

Let us now describe how the machinery discussed so far can be used to formalize constraints. 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 ConDec 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 event occurrences formalized in Sec. 4.2. The same formalization can be applied to atomic activities as well, by just substituting each mentioned event type associated to the atomic execution.

#### 4.4.1 Existence Constraint

Let us consider an existence constraint of the form  $\boxed{a}^{n..*}$ , identified by *id*. Following Ex. 3, 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*<sup>3</sup>:

$$\textit{init\_state}(i(\textit{id}, \textit{start}, 0), \textit{pend}).$$

**Axiom 6** (existence fulfillment). *A pending existence constraint instance becomes satisfied when the  $n$ -th execution of the involved activity  $a$  is completed:*

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

---

<sup>3</sup>Since no specific event is responsible for the instance creation, we use the keyword “start” as event name.

where  $\text{hap}(Ev, N, T)$  tests whether  $Ev$  occurred  $N$  times before or at  $T$ , and it is obviously called by abstracting away from the specific event identifier<sup>4</sup>:

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

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, 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 execution of some activity, 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:  $\text{trans}(\text{ev}(-, d, \text{case\_complete}), T, i(-, -, -), \text{pend}, \text{viol})$ .*

#### 4.4.2 Absence Constraint

Let us now focus on absence constraints, considering the specific case of  $\boxed{a}^{0..n}$ , with  $id$  as identifier. Like existence constraints, each absence is associated to a single instance. However, the instance follows a complementary evolution with respect to the one of existence. In particular, 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:*

$$\text{init\_state}(i(id, \text{start}, 0), \text{sat}).$$

**Axiom 9** (absence violation). *The active absence constraint instance becomes violated when the  $n$ -th execution of activity  $a$  is completed:*

$$\text{trans}(\text{compl}(-, a), T, i(id, \text{start}, 0), \text{sat}, \text{viol}) \leftarrow \text{hap}(\text{compl}(-, a), N, T).$$

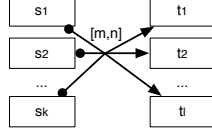
It is worth noting that absence constraint instances are never pending, because they do not place any (positive) expectation about the execution of their associated activity.

---

<sup>4</sup>The 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  $L$ .

### 4.4.3 Metric Response Constraint

Let us consider now the “prototypical” response constraint



It includes branches (i.e., the presence of multiple sources and targets) and quantitative time constraints. We suppose that it is identified by  $id$ . To deal with branching response constraints, we make use of the *member* predicate<sup>5</sup>: 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  $[t_1, \dots, t_l]$ .

**Axiom 10** (response creation). *A new instance of the response constraint identified by  $id$  is created every time an activity  $A$  is completed, and  $A$  is inside the set of 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 then satisfied if a target activity is executed afterwards, within the time boundaries associated to the instance. We use the *current\_state* predicate to extract the reference point from the time contained in the instance context (which corresponds to the time at which the instance has been created). Such reference point is then suitably combined with the constraint’s delay  $m$  and deadline  $n$ , determining the time window inside which one among the target activities is expected to be started.

**Axiom 11** (Response fulfillment). *A pending response instance becomes satisfied when one of its target activities is executed, at a time that falls within the actual time boundaries.*

$$trans(start(-, B), T, i(id, A, T_i), pend, sat) \leftarrow cur\_state(i(id, A, T_i), pend, T) \\ member(B, [t_1, \dots, t_l]) \wedge T \geq T_i + m \wedge T \leq T_i + n.$$

*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  $\infty$ , and we consequently remove constraint  $T \leq T_i + n$  from the axiom.*

<sup>5</sup>*member(El, L)* is true if list  $L$  contains  $El$ .

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 time flow, but “extracts” the current time from the occurring events, a deadline expiration is detected at the *first* following time at which some event occurs. An external component could be employed to constantly send clock events to the EC-based monitor, keeping it updated about the current time and enabling the prompt evaluation of deadlines expiration.

**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.*

$$\mathit{trans}(\_, T, i(id, A, T_i), \mathit{pend}, \mathit{viol}) \leftarrow \mathit{cur\_state}(i(id, A, T_i), \mathit{pend}, T) \wedge T > T_i + n.$$

*This axiom is only present when  $n$  (the deadline) is an actual value.*

#### 4.4.4 Metric Chain Response Constraint

Let us consider the same prototypical constraint used in Sec. 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 one. More specifically, all the axioms discussed in Sec. 4.4.3 are seamlessly maintained, and a new axiom is introduced. Such 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 the chain response identified by  $id$  becomes violated if an event related to a non-target activity occurs.*

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

## 4.5 Implementation

MOBUCON has been fully implemented inside the latest version of the well-known ProM process mining framework [24]. ProM 6 is natively equipped with an Operational Support (OS) service, which accommodates techniques for runtime process



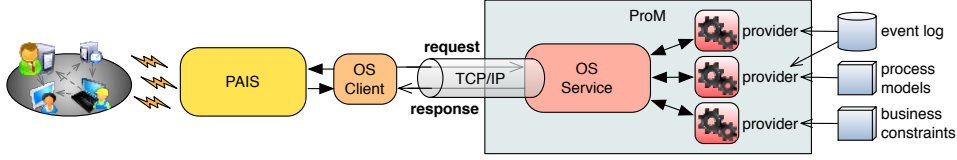


Figure 6. Operational support in ProM 6

monitoring, recommendations and predictions. As sketched in Fig. 6, the OS service takes care of receiving TCP/IP streams of event occurrences from an external system<sup>6</sup>. 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 has been therefore implemented as a monitoring OS provider. Each instance of MOBUCON is associated to a ConDec model, which can be loaded from the XML file produced by the DECLARE [17] editor.

In order to facilitate testing and exploitation of MOBUCON, two client prototypes have been developed, one accepting the realtime acquisition of events, and the other loading partial or complete traces from event logs. Both clients exploit an interface which gives an intuitive, graphical feedback to the user, showing the evolution of each constraint’s instance (see Sect. 5 for examples). 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.

Finally, the top part of the interface is devoted to show the evolution of a metric which measures the *system health*, reporting “how much” the running case is complying with the ConDec model. Different metrics can be easily accommodated into MOBUCON. In the remainder of this paper, we will use a simple but significant metric, which computes the system health by considering the number of violated ( $\#viol$ ) and satisfied ( $\#sat$ ) instances (and ignoring pending states). In particular, at some time  $t$  the system health corresponds to  $1 - \frac{\#viol(t)}{\#viol(t) + \#sat(t)}$  otherwise.<sup>7</sup> Obviously, more sophisticated metrics could be designed, taking, e.g., into account metric temporal aspects or giving different weights to the ConDec constraints.

<sup>6</sup>Typically a workflow system, but any system logging streams of correlated events can seamlessly benefit of the OS support.

<sup>7</sup>If  $\#viol(t) + \#sat(t) = 0$ , then the system health is defined to be 1.

## 5 Case Study

MOBUCON has been applied to a case study in the domain of maritime safety and security. It concerns monitoring a vessel’s behavior in a specific area using a sensor network. The case study has been conducted in the context of Poseidon [12], 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. This collaboration revealed that when monitoring the behavior of vessels, it is crucial to combine information from different data sources. In this sense, MOBUCON demonstrated to be extremely useful in this setting.

We have enriched the ConDec models presented in [13] with quantitative time constraints that have been empirically assessed by analyzing the behavior of the monitored vessels in ideal conditions. In general, our approach can be applied to monitor the behavior of a system (in this case a vessel) evaluating the system health on the basis of the number of anomalies detected.

### 5.1 Monitoring the behavior of a vessel

Every vessel has an on-board AIS (Automatic Identification System [9]) 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 them.

AIS messages contain information such as the *mmsi* number of the reporting vessel, which is a maritime vessel identifier, its navigational status and its ship type. This information can be used to monitor the behavior of the vessel in a specific area. In particular, when monitoring a vessel using the broadcasted stream of AIS messages, we can consider as an event a change in the navigational status of the vessel (e.g. *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 timepoint, and consequently the ConDec model will only contain atomic activities.

Each case corresponds to the sequence of events referring to the same *mmsi* number (i.e., to the same vessel). Vessels are expected to behave differently on the basis of their ship type, i.e. for each ship type only sequences of events with specific characteristics are allowed. Some characteristics 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. The changes in the navigational status of a vessel conform to a less-structured process and their behavior can be effectively represented in ConDec.

Therefore, the first step of our experimentation has been the construction of a ConDec model representing the observed behavior of every possible ship type

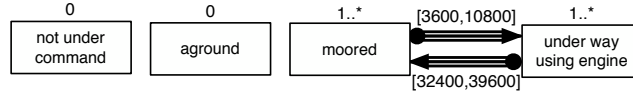


Figure 7. Expected behavior for vessel type passenger ship. Time granularity: seconds.

(e.g. *passenger* or *cargo ship*). These models can be extracted using process discovery techniques as presented in [13]. Each ConDec model has been enriched with metric aspects. The ship type in an AIS message is used to identify the reference ConDec model with respect to which the corresponding vessel must be monitored.

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 while using a time granularity of seconds.

### 5.1.1 Passenger Ships

The observed behavior for *passenger ship* is tackled by the ConDec model in Fig. 7. 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* (represented by the chain response constraints in the model).

The monitored passenger ships are required to go back and forth between two resorts. Considering the distance between the harbors and the speed of this type of ship, we can estimate that each journey must take more than 9 hours (32400 seconds) and less than 11 hours (39600 seconds). Therefore, constraint *chain response*([*under way using engine*], [*moored*]) is associated to this time window. Moreover, for constraint *chain response*([*moored*], [*under way using engine*]) we specify a delay of 1 hour (3600 seconds) and a deadline of 3 hours (10800 seconds) considering the time that each ship must spend in a harbor (for refueling, restocking etc.).

Fig. 8 shows the results for the monitoring of a specific case (i.e. of a specific vessel) with respect to the ConDec model in Fig. 7. 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. 8 also shows the 13 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,

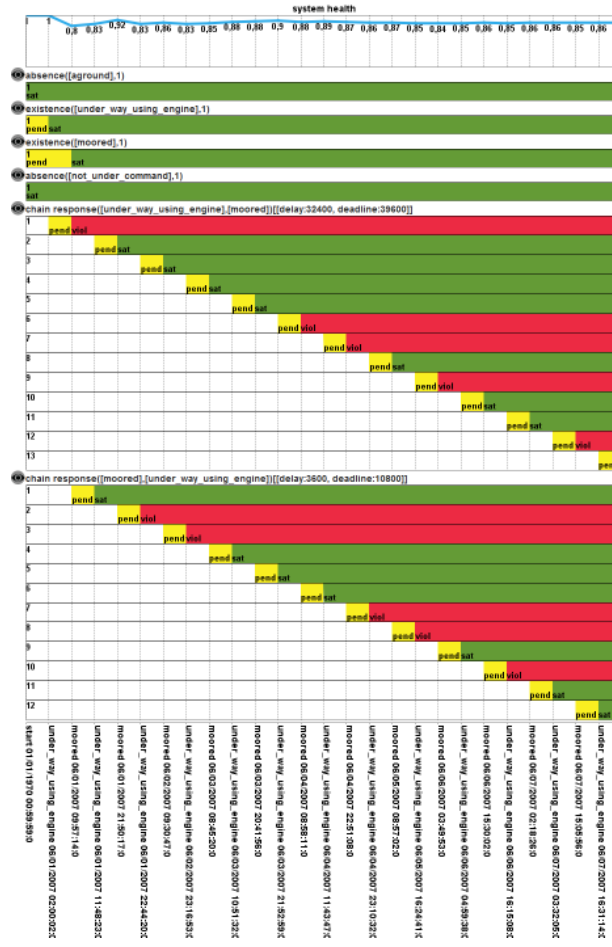


Figure 8. Monitoring of a passenger ship.

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 complete. However, also when *under way using engine* is immediately followed by *moored*, not always the corresponding instances become satisfied. 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 instances 6, 7, 9 and 12 event *moored* occurs after more than 11 hours (after 11 hours and 6

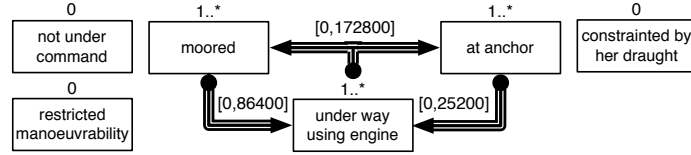


Figure 9. Expected behavior for vessel type cargo ship having a cargo of type “hazard pollutant A”.

minutes, after 11 hours and 8 minutes, after 11 hours and 25 minutes and after 11 hours and 33 minutes respectively).

Constraint  $chain\ response([moored], [under\ way\ using\ engine])$  (delay: 1 hour, deadline: 3 hours) behaves similarly. 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 overall trend of the health of the system is in this case quite homogeneous and varies around an average value of 0.86 (see top of Fig. 8).

### 5.1.2 Cargo Ships

The observed behavior for the ship type *cargo ship* is represented by the ConDec model in Fig. 9. In the remainder, we refer to this as simply a cargo ship (although there are multiple cargo types). Events *constrained by her draught*, *restricted manoeuvrability* 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 seconds for constraint  $chain\ response([at\ anchor], [under\ way\ using\ engine])$ . Event *moored* must be also 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 seconds 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. Considering the distance that must be covered, we can estimate that each journey of this type of ships can take up to 2 days. Therefore, we specify a deadline of 172800 seconds for constraint  $chain\ response([under\ way\ using\ engine], [moored, at\ anchor])$ .

Fig. 10 shows the monitoring of the stream of messages received by two different cargo ships. Let us first focus on Fig. 10(a). Unlike the other absence constraints of the reference model, the instance related to the constraint  $absence([not\ under\ command], 1)$  is initially satisfied but, when at 06-05-2007 18:47:45 *not under command* occurs, it becomes violated. Furthermore, constraint  $existence([at\ anchor], 1)$  is initially pending and it remains pending because *at anchor* does not occur during the monitored period.

There are 6 instances related to the branched constraint *chain response*([*under way using engine*], [*moored, at anchor*]) (deadline: 2 days) corresponding to different journeys of the vessel. In instances 2, 3 and 4 *under way using engine* is followed by *not under command* so that, when this event occurs, they become violated. In instances 1 and 5 *under way using engine* is followed by *moored* within the specified deadline so that they become both satisfied, while instance 6 remains pending. There are 2 instances related to constraint *chain response*([*moored*], [*under way using engine*]) (deadline 24 hours) corresponding to different stops in a harbor. In both of them *moored* is followed by *under way using engine* complying with the specified deadline so that they become both satisfied. There are no instances related to constraint *chain response*([*at anchor*], [*under way using engine*]) (deadline 7 hours).

The health of this vessel is initially 1 (see Fig. 10(a)), then it varies around 0.77.

Let us now briefly comment on the diagram in Fig. 10(b). The constraint *existence*([*at anchor*], 1) is initially pending (because it is waiting for the occurrence of at least one event *at anchor*). However, differently from the previous case, at 06-04-2007 08:11:47 *at anchor* occurs and this instance becomes satisfied. There are 4 instances related to the branched constraint *chain response*([*under way using engine*], [*moored, at anchor*]) (deadline: 2 days) corresponding to different journeys of the vessel. In instances 1, 2 and 3 *under way using engine* is followed by *moored* within the specified deadline so that it becomes satisfied. Instance 4 remains pending. Only 2 instances are related to constraint *chain response*([*at anchor*], [*under way using engine*]) (deadline 7 hours) corresponding to different stops of the considered vessel at anchor. In instance 1 *at anchor* is followed by *under way using engine* complying with the specified deadline so that it becomes satisfied. In instance 2 *under way using engine* occurs after the deadline expiration. There are 3 instances related to constraint *chain response*([*moored*], [*under way using engine*]) (deadline 24 hours) corresponding to different stops in a harbor. In instance 1 *moored* is followed by *at anchor* so that it becomes violated. In instances 2 and 3 *moored* is followed by *under way using engine* complying with the specified deadline so that they become satisfied.

As shown in Fig. 10(b) the health of this vessel is initially 1. Then, it varies from a minimum value of 0.83 and a maximum value of 0.94. Note that the health trend clearly decreases in accord with the violations.

## 5.2 Benchmarks and Performance Evaluation

For the practical application of our approach, the performance of MOBUCON is of the utmost importance. Note that unlike classical process mining approaches, monitoring requires short-term feedback every time the current state of affairs evolves and new event occurrences are acquired.

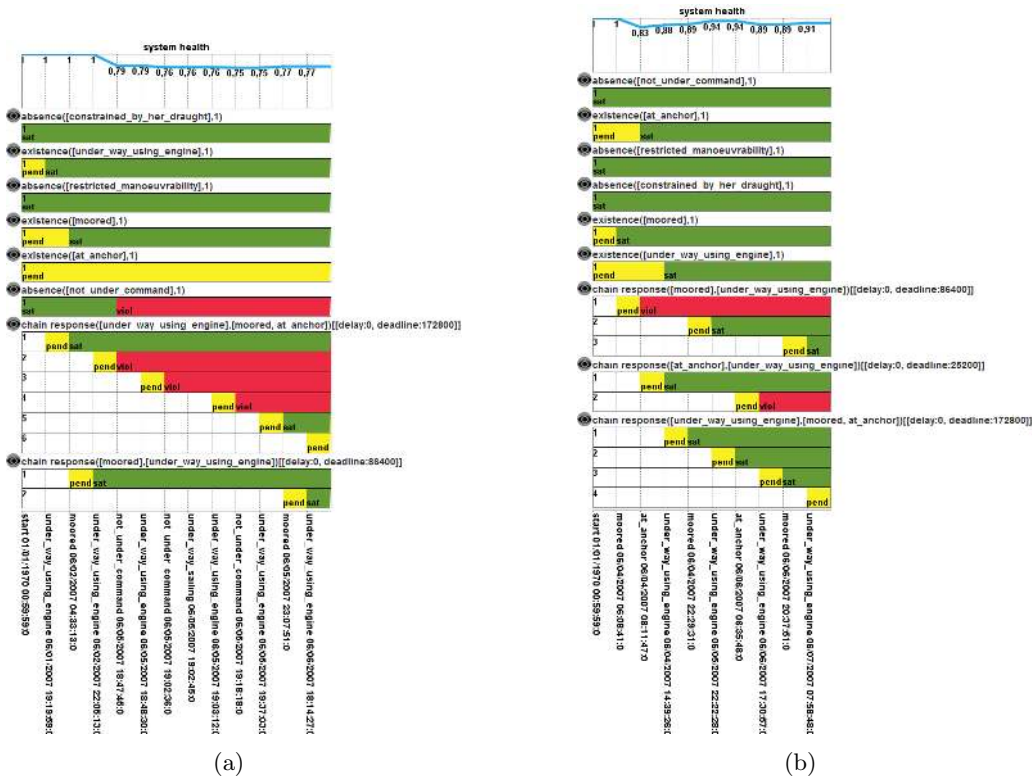


Figure 10. Monitoring of two cargo ships: the health of ship (a) drops to approximately 0.77 while the health of ship (b) varies from 0.83 to 1

The current implementation of MOBUCON is modular, and the reasoning component is loosely connected to the other parts of the system. The formalization of a ConDec model leads to a domain-independent EC specification, which is then combined with a light-weight axiomatization of the CEC (see Sect. 3.4) written in standard Prolog. Hence, virtually any Prolog engine can be seamlessly used as the reasoning component of MOBUCON. We have currently experimented two alternative solutions. The first relies on TuProlog<sup>8</sup>. The main advantage of TuProlog is that, being developed completely in JAVA, it can be seamlessly integrated inside ProM; the main drawback relies however in its performance, which is being currently greatly improved but cannot still compete with the one of mainstream Prolog systems. The other solution relies on YAP<sup>9</sup>, one of the highest-performance state of the art Prolog engines. Unlike TuProlog, YAP cannot be directly invoked from

<sup>8</sup>tuprolog.alice.unibo.it

<sup>9</sup>yap.sourceforge.net

JAVA code, but requires an intermediate bridge (such as InterProlog<sup>10</sup>).

In this work, we are specifically interested in evaluating the performance of our EC-based approach, and therefore 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. Furthermore, we are interested in studying how scalable is MOBUCON when the number of constraints to be monitored increases; therefore, our tests abstract away from the further constraints used to formalize the lifecycle of non-atomic activities, by generating ConDec models that contain atomic activities only.

To empirically assess the performance of MOBUCON, we have set up a generator of ConDec models starting from a set of configuration parameters targeting the “size” and structure of the model. In particular, given:

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

the generator produces a self-contained Prolog test containing a lightweight axiomatization of CEC, a randomly generated trace of  $L$  events occurrences related to the atomic activities in  $\mathcal{A}$ , 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 have used this generator to produce a benchmark containing 1000 tests. All ConDec models contained in such tests are tuned with  $|\mathcal{A}| = 10$ ,  $L = 1000$ ,  $C = 5$ ,  $B = 3$ ,  $d = 0$  and  $D = 50$ . The benchmark contains 10 groups of tests by increasing number of constraints in the ConDec model (with  $N = 10, 20, 30, \dots, 100$ ). 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 (which are ordered according to their timestamps), and measuring the latency time required by the reasoner to produce the updated monitoring result after having acquired a new event. The obtained *average* results are reported in Fig. 11. They attest the scalability of MOBUCON and its feasibility in dealing with models of real-life size. In fact, the 1000 event is processed in less than 100ms for models containing up to 50 constraints, and for large models with 100 constraints, still less than 300ms are required to update the monitoring result.

---

<sup>10</sup>[www.declarativa.com/interprolog/](http://www.declarativa.com/interprolog/)



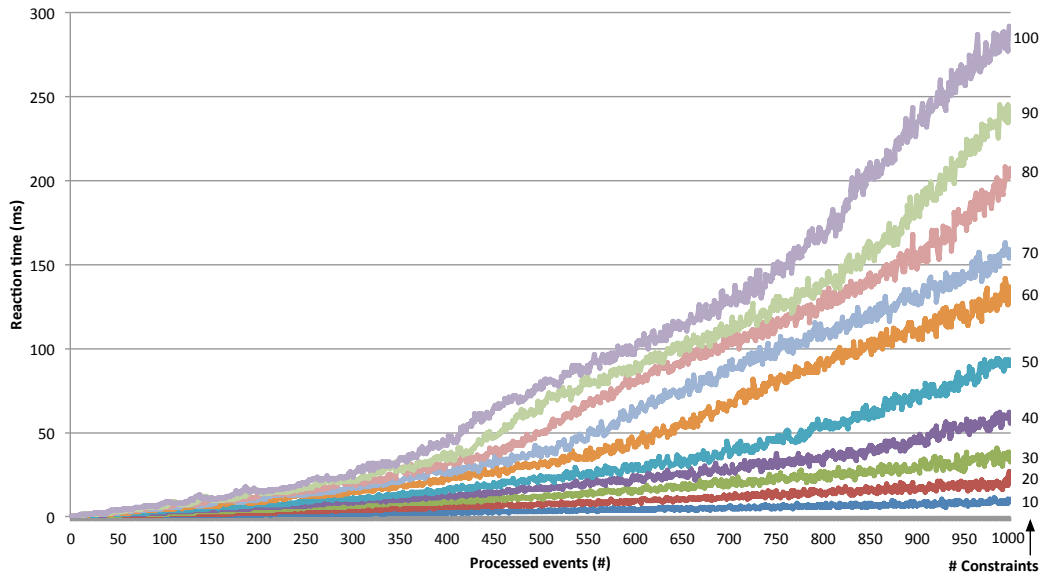


Figure 11. *Monitoring performance of MOBUCON with a set of randomly generated benchmarks; the reaction time denotes the time needed to produce the updated monitoring result after having processed a new event*

## 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.

We will limit our discussion focusing on approaches that adopt logic-based techniques with a well-founded theoretical basis. [21] 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. [7] focus on the application of 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 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.

Although MOBUCON is based on logic programming, it exploits an axiomatization of the CEC by [5], which employs assert and retract predicates to manipulate the MVIs of fluents. Even if there are approaches that aim to define a declarative semantics of such predicates [], in the standard logic programming setting they are considered as extra-logical predicates. An alternative reactive but purely declarative axiomatization of the EC, called REC, has been proposed in [4], exploiting an abductive logic programming-based proof procedure for runtime compliance checking. The application of REC for monitoring service choreographies [3] constitutes a preliminary investigation of the EC-based axiomatization of ConDec 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. While the declarative nature of REC makes it possible to study its formal properties (such as soundness, completeness and termination [4]), the lightweight axiomatization of CEC in Prolog that is exploited by MOBUCON is more efficient.

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 runtime. The construction of monitors requires to modify the LTL semantics 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. [2] 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 to characterize the state of constraint instances. In [1], the authors introduce a four-valued variant of LTL that yields *possibly true* and *possibly false* whenever the system’s behavior is inconclusive in the three-valued sense. [8] 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.

When compared with these approaches, MOBUCON relies on a more expressive logic, supporting quantitative time constraints, non-atomic activities with identifier-based correlation, and data. 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 [11], 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 is the construction of the automaton, which is done before the execution. 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 ConDec, this ability is exploited to identify whether the interplay of two or more constraints will surely lead the execution to incur in a violation. Our EC-based approach is not able to foresee these violations: it treats each constraint separately from the others, combining it only with the trace.

## 7 Conclusion

We have presented MOBUCON, a non-intrusive framework for monitoring systems whose dynamics is characterized by means of event streams. The framework is based on the Event Calculus (EC), which is used to specify the constraints to be monitored. It has been embedded inside the operational decision support infrastructure of ProM.

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 the functionality of MOBUCON, ConDec 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 formalization of ConDec.

An interesting aspect is related to the nature of 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 as new events get to be known. This process could potentially continue forever. From the modeling point of view, this means that the ConDec diagrams could contain cyclic relationships, accounting for expected situations that must be repeatedly achieved. This is the case for the vessel and cargo ships models shown in Figs. 7 and 9. 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 always triggered by the occurrence of new events, taking into account the partial finite trace stored so far.

As ongoing work, we are extending our approach along different dimensions, incorporating data and also recovery and compensation mechanisms to be instantiated in the case of a violation. Thanks to its first-order nature, the EC is able to seamlessly accommodate both aspects. For what concerns data, we will extend the formalization of ConDec including the data-related extensions proposed in [14]. Regarding recovery and compensation, we will rely on the preliminary proposal made in [3], where the violations are reified as special events. Another active research line concerns a comprehensive experimental evaluation of the reasoner, to assess its performance and scalability. Our preliminary evaluation shows that the current implementation can be improved to deal with more challenging application scenarios.

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

## References

- [1] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL Semantics for Runtime Verification. *Logic and Computation*, 2010.
- [2] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, expected 2011.
- [3] F. Chesani, P. Mello, M. Montali, and P. Torroni. Verification of Choreographies During Execution Using the Reactive Event Calculus. In *Proceedings of the 5th International Workshop on Web Service and Formal Methods (WS-FM2008)*, volume 5387 of *LNCS*, pages 55–72. Springer, 2009.
- [4] Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. A Logic-Based, Reactive Calculus of Events. *Fundamenta Informaticae*, 105(1-2):135–161, 2010.
- [5] L. Chittaro and A. Montanari. Efficient Temporal Reasoning in the Cached Event Calculus. *Computational Intelligence*, 12:359–382, 1996.
- [6] Keith L. Clark. Negation as Failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
- [7] Andrew D. H. Farrell, Marek J. Sergot, Mathias Sallé, and Claudio Bartolini. Using the Event Calculus for Tracking the Normative State of Contracts. *International Journal of Cooperative Information Systems*, 14(2-3):99–129, 2005.
- [8] Dimitra Giannakopoulou and Klaus Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 412–416. IEEE Computer Society, 2001.
- [9] International Telecommunications Union. *Technical characteristics for a universal shipborne Automatic Identification System using time division multiple access in the VHF maritime mobile band*, 2001. Recommendation ITU-R M.1371-1.
- [10] Robert A. Kowalski and Marek J. Sergot. A Logic-Based Calculus of Events. *New Generation Computing*, 4(1):67–95, 1986.

- [11] Fabrizio Maggi, Marco Montali, Michael Westergaard, and Wil van der Aalst. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In Stefanie Rinderle-Ma, Farouk Toumani, and Karsten Wolf, editors, *Business Process Management*, volume 6896 of *Lecture Notes in Computer Science*, pages 132–147. Springer Berlin / Heidelberg, 2011.
- [12] Fabrizio M. Maggi, Arjan J. Mooij, and Wil M. P. van der Aalst. *Analyzing Vessel Behavior using Process Mining*, chapter Poseidon book. to appear.
- [13] Fabrizio M. Maggi, Arjan J. Mooij, and Wil M. P. van der Aalst. User-Guided Discovery of Declarative Process Models. In *2011 IEEE Symposium on Computational Intelligence and Data Mining*, 2011.
- [14] Marco Montali. *Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach*, volume 56 of *LNBIP*. Springer, 2010.
- [15] Marco Montali, Maja Pesic, Wil M. P. van der Aalst, Federico Chesani, Paola Mello, and Sergio Storari. Declarative Specification and Verification of Service Choreographies. *ACM Transactions on the Web*, 4(1), 2010.
- [16] Maja Pesic. *Constraint-Based Workflow Management Systems: Shifting Controls to Users*. PhD thesis, Beta Research School for Operations Management and Logistics, Eindhoven, 2008.
- [17] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 287–300. IEEE Computer Society, 2007.
- [18] Maja Pesic and Wil M. P. van der Aalst. A Declarative Approach for Flexible Business Processes Management. In *Proceedings of the BPM 2006 Workshops*, volume 4103 of *LNCS*, pages 169–180. Springer, 2006.
- [19] Fariba Sadri and Robert A. Kowalski. Variants of the Event Calculus. In *Proceedings of the 12th International Conference on Logic Programming (ICLP 1995)*, pages 67–81. MIT Press, 1995.
- [20] M. Shanahan. The Event Calculus Explained. In *Artificial Intelligence Today: Recent Trends and Developments*, volume 1600 of *LNCS*, pages 409–430. Springer, 1999.
- [21] George Spanoudakis and Khaled Mahbub. Non-Intrusive Monitoring of Service-Based Systems. *Cooperative Information Systems*, 15(3):325–358, 2006.

- [22] Wil M. P. van der Aalst, Kees M. van Hee, Jan Martijn E. M. van der Werf, and Marc Verdonk. Auditing 2.0: Using Process Mining to Support Tomorrow's Auditor. *IEEE Computer*, 43(3):90–93, 2010.
- [23] B. Van Dongen, A. K. de Medeiros, and L. Wen. Process Mining: Overview and Outlook of Petri Net Discovery Algorithm. *Transactions on Petri Nets and Other Models of Concurrency*, 2:225–242, 2009.
- [24] Eric Verbeek, Joos Buijs, Boudewijn van Dongen, and Wil van der Aalst. ProM 6: The Process Mining Toolkit. In *Demo at the 8th International Conference on Business Process Management (BPM 2010)*, 2010.