An Operational Decision Support Framework for Monitoring Business Constraints

Fabrizio Maria Maggi[1,⋆], Marco Montali[2,⋆⋆], and Wil M.P. van der Aalst[1]

[1] Eindhoven University of Technology, The Netherlands
{f.m.maggi,w.m.p.v.d.aalst}@tue.nl
[2] KRDB Research Centre, Free University of Bozen-Bolzano, Italy
montali@inf.unibz.it

Abstract. Only recently, *process mining* techniques emerged that can be used for *Operational decision Support* (OS), i.e., knowledge extracted from event logs is used to handle running process instances better. In the process mining tool ProM, a generic OS service has been developed that allows ProM to dynamically interact with an external information system, receiving streams of events and returning meaningful insights on the running process instances. In this paper, we present the implementation of a novel business constraints monitoring framework on top of the ProM OS service. We discuss the foundations of the monitoring framework considering two logic-based approaches, tailored to Linear Temporal Logic on finite traces and the Event Calculus.

Keywords: Declare, process mining, monitoring, operational decision support.

1 Introduction

Process mining has been traditionally applied on historical data that refers to past, complete process instances. Recently, the exploitation of process mining techniques has been extended to deal also with running process instances which have not yet been completed. In this setting, process mining provides *Operational decision Support* (OS), giving meaningful insights that do not only refer to the past, but also to the present and the future [1]. In particular, OS techniques can be used to: *check* the current state of affairs detecting deviations between the actual and the expected behavior; *recommend* what to do next; *predict* what will happen in the future evolution of the instance.

In order to enable the effective development of OS facilities, the widely known process mining framework ProM 6 [2] incorporates a backbone for OS [3]. Here,

⋆ This research has been carried out as a part of the Poseidon project at Thales under the responsibilities of the Embedded Systems Institute (ESI). The project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

⋆⋆ This research has been partially supported by the NWO "Visitor Travel Grant" initiative, and by the EU Project FP7-ICT ACSI (257593).

© Springer-Verlag Berlin Heidelberg 2012

all the common functionalities needed for OS are implemented, such as management of requests coming from external information systems, dynamic acquisition and correlation of incoming partial execution traces (representing the evolution of process instances), and interaction with different process instances at the same time. The OS backbone relies on a client-server architecture. The client is exploited by an external stakeholder to send a partial trace to ProM and ask queries related to OS. On the server side, an OS service (running inside ProM) takes care of coordinating the available OS functionalities in order to answer such queries. Multiple *OS providers* that encapsulate specific OS functionalities can be developed and dynamically registered to the OS service.

In this work, we present the implementation of a novel runtime compliance verification framework on top of the ProM OS. The framework is called *Mobucon* (*Mo*nitoring *bu*siness *con*straints) and its focus is to dynamically *check* the compliance of running process instances with business constraints, detecting deviations and measuring the *degree of adherence* between the actual and the expected behavior.

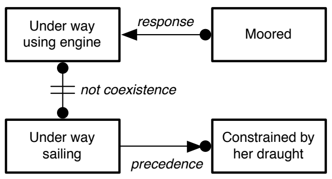
Given a business constraints reference model and a partial trace characterizing the running execution of a process instance, Mobucon infers the status of each business constraint. In particular, it produces a constantly updated snapshot about the state of each business constraint, reporting whether it is currently violated. Consequently, it determines whether the process instance is currently complying with the reference model or not. Beside this, other meaningful insights can be provided to end users, such as, for example, indicators and metrics related to the "degree of compliance", e.g., relating the number of violated constraints with their total number.

The paper is organized as follows. Section 2 presents the *Declare* language [4] and its extension to include metric temporal constraints and constraints on event-related data. The language is declarative and graphical. Moreover, *Declare* has been formalized using a variety of logic-based frameworks, such as Linear Temporal Logic (LTL) with a finite-trace semantics[1] [5,6] and the Event Calculus (EC) [7,8]. Section 3 describes the architecture of our proposed framework. In Sect. 4 and 5, we describe the implementation of two different reasoning engines as OS providers based on LTL and on the EC respectively. We are currently applying our framework to various real-world case studies; in Sect. 6, we report on the monitoring of *Declare* constraints in the context of maritime safety and security. Finally, Sect. 7 includes a comparison of the two approaches and discusses related work and conclusion.

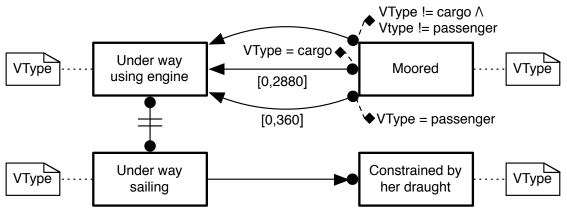
2 Declare

Declare is a declarative, constraint-based process modeling language first proposed in [5,4]. In a constraint-based approach, instead of explicitly specifying all the acceptable sequences of activities in a process, the allowed behavior of

[1] For compactness, in the following we will use the LTL acronym to denote LTL on finite traces.



(a) Basic *Declare* model



(b) *Declare* model augmented with data, data-aware conditions and metric temporal constraints

Fig. 1. Two *Declare* models in the context of maritime safety and security

the process is implicitly specified by means of declarative constraints, i.e., rules that must be respected during the execution. In comparison with procedural approaches, that produce "closed" models, i.e., models where what is not explicitly supported is forbidden, declarative languages are "open" and tend to offer more possibilities for execution. In particular, the modeler is not bound anymore to explicitly enumerate the acceptable executions and models remain compact: they specify the mandatory and undesired behaviors, leaving unconstrained all the courses of interaction that are neither mandatory nor forbidden.

Declare is characterized by a user-friendly graphical front-end and is based on a formal back-end. More specifically, the formal semantics of *Declare* can be specified by using LTL [5,6], abductive logic programming with expectations [6], or the EC [7,8]. These characteristics are crucial for two reasons. First, *Declare* can be used in real scenarios being understandable for end-users and usable by stakeholders with different backgrounds. Second, *Declare*'s formal semantics enable verification and automated reasoning. This is a key aspect in the implementation of monitoring tools for *Declare* models.

Figure 1a shows a simple *Declare* model elicited in the context of a real case study related to the monitoring of vessels behavior in the context of maritime safety and security. We use this example to explain the main concepts. It involves four *events* (depicted as rectangles, e.g., *Under way using engine*) and three *constraints* (shown as arcs between the events, e.g., *not coexistence*). Events characterize changes in the navigational status of each monitored vessel. Constraints highlight mandatory and forbidden behaviors, implicitly identifying the acceptable execution traces that comply with (all of) them. In our case study, a vessel can be either *Under way using engine* or *Under way sailing* but not both, as indicated by the *not coexistence* between such two events. A vessel can be *Constrained by her draught*, but only after being *Under way sailing* (a vessel equipped with an engine cannot be constrained by draught and a sailing vessel cannot be constrained before it is under way). This is indicated by the *precedence* constraint. Finally, after being *Moored* each vessel must eventually be *Under way using engine*, as specified by the *response* constraint.

In [7], an extension of this constraint-based language has been proposed; this extension incorporates also non-atomic activities (i.e., activities whose execution

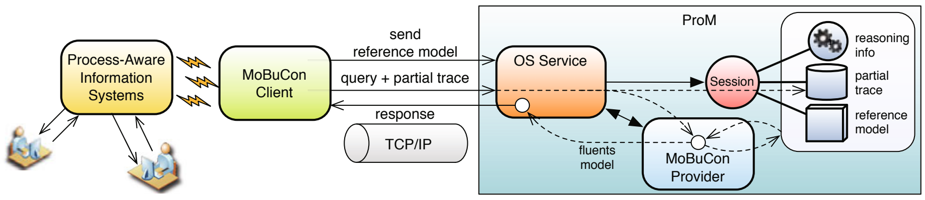


Fig. 2. Mobucon Architecture

is characterized by a life cycle that includes multiple events), event-related data and data-aware conditions and metric temporal constraints (for specifying delays, deadlines and latencies). This extended language is exploited in Fig. 1b to augment the aforementioned constraints with conditions on time and data. More specifically, we assume that each event is equipped with two data: the identifier of the vessel and its type. In particular, the *response* constraint is now differentiated on the basis of the vessel type, introducing different timing requirements (which are specified with the granularity of *minute*). The first *response* constraint indicates that if the type of the vessel is *Passenger ship* and event *Moored* occurs, then *Under way using engine* must eventually occur within 6 hours at most. The second one indicates that if the type of the vessel is *Cargo ship* and *Moored* occurs, then *Under way using engine* must eventually occur within 48 hours. A last standard *response* constraint is employed to capture the behavior of all other vessels, without imposing any deadline. Finally, although not explicitly shown in the diagram, each constraint is applied to events that are associated to the same vessel identifier. This correlation mechanism makes it possible to properly monitor also a unique event streams collecting the evolving behaviors of multiple vessels at the same time.

3 Mobucon Architecture

Figure 2 shows the overall architecture of Mobucon. Mobucon relies on the general architecture of the OS backbone implemented inside ProM 6. Such backbone has been introduced and formalized using colored Petri nets in [3]; in Sect. 3.1, we will therefore sketch some relevant aspects of the general architecture. In Sect. 3.2, we ground the discussion to the specific case of Mobucon, discussing the skeleton of our compliance verification OS provider. The data exchanged between the Mobucon client and provider is illustrated in Sect. 3.3. Finally, in Sect. 3.4, we describe the implemented Mobucon clients. The two concrete instantiations of the Mobucon skeleton in the LTL and EC settings are discussed in Sect. 4 and 5.

3.1 General Architecture

The ProM OS architecture relies on the well-known client-server paradigm. More specifically, the ProM OS service manages the interaction with running process instances and acts as a mediator between them and the registered specific OS providers.

Sessions are created and handled by the OS Service to maintain the state of the interaction with each running client. To establish a stateful connection with the OS Service, the client creates a session handle for each managed running process instance, by providing host and port of the OS Service. When the client sends a first query related to one of such running instances to the OS service, it specifies information related to the initialization of the connection (such as reference models, configuration parameters, etc.) and to the type of the queries that will be asked during the execution. This latter information will be used by the OS Service to select, among the registered active providers, the ones that can answer the received query. The session handle takes care of the interaction with the service from the client point of view, hiding the connection details and managing the information passing in a lazy way. The interaction between the handle and the service takes place over a TCP/IP connection.

3.2 Mobucon Skeleton

In Mobucon, the interaction between a client and the OS service mainly consists of two aspects. First of all, before starting the runtime compliance verification task, the client sends to the OS service the *Declare* reference model to be used. This model is then placed inside the session by the OS service. The reference model is an XML file that contains the information about events and constraints mentioned in the model. This format is generated by the *Declare* editor (`www.win.tue.nl/declare/`). The client can also set further information and properties. For example, each constraint in the *Declare* reference model can be associated to a specific weight, that can be then exploited to compute metrics and indicators that measure the degree of adherence of the running instance to the reference model.

Secondly, during the execution, the client sends queries about the current monitoring status for one of the managed process instances. The session handle augments these queries with the partial execution trace containing the evolution that has taken place for the process instance after the last request. The OS Service handles a query by first storing the events received from the client, and then invoking the Mobucon provider.

The Mobucon provider recognizes whether it is being invoked for the first time w.r.t. that process instance. If this is the case, it takes care of translating the reference model onto the underlying formal representation. The provider then returns a fresh result to the client, exploiting a reasoning component for the actual result's computation. The reasoning component, as well as the translation algorithm, are dependent on the chosen logical framework (LTL or EC), while the structure of the skeleton is the same for the two approaches. After each query, the generated result is sent back to the OS service, which possibly combines it with the results produced by other relevant providers, finally sending the global response back to the client.

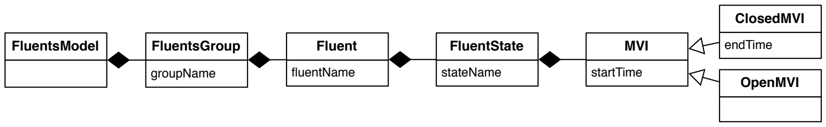


Fig. 3. Fluent model used to store the evolution of constraints

3.3 Exchanged Data and Business Constraints States

We now discuss the data exchanged by the Mobucon client and provider. Note that these data are common to both instantiations of the provider (Mobucon LTL and Mobucon EC). The partial execution traces sent by the client to the OS use the XES format (`www.xes-standard.org/`) for event data. XES is an extensible XML-based standard recently adopted by the IEEE task force on process mining.

The response produced by the Mobucon provider is composed of two parts. The first part contains the temporal information related to the evolution of each monitored business constraint from the beginning of the trace up to now. At each time point, a constraint can be in one state, which models whether it is currently: *satisfied*, i.e., the current execution trace complies with the constraint[2]; *(permanently) violated*, i.e., the process instance is not compliant with the constraint; *pending* (or *possibly violated*), i.e., the current execution trace is not compliant with the constraint, but it is possible to satisfy it by generating some sequence of events. This state-based evolution is encapsulated in a *fluent model* which obeys to the schema sketched in Fig. 3. A fluent model aggregates fluents groups, containing sets of correlated fluents. Each fluent models a multi-state property that changes over time. In our setting, fluent names refer to the constraints of the reference model. The fact that the constraint was in a certain state along a (maximal) time interval is modeled by associating a closed MVI (Maximal Validity Interval) to that state. MVIs are characterized by their starting and ending timestamps. Current states are associate to open MVIs, which have an initial fixed timestamp but an end that will be bounded to a currently unknown future value.

The Mobucon provider also computes the current value of a *compliance indicator* of the running monitored instance. This number gives an immediate feeling about the "degree of adherence" between the instance and the reference model. A low degree of adherence can be interpreted differently depending on the application domain. In general, it is used to classify a process instance as "unhealthy". However, it can also be used to show that a reference model is obsolete and it must be improved to better reflect the reality. The compliance indicator can be computed using different metrics, that can consider the current state of constraints, as well as other information such as the weight of each individual constraint. For example, the compliance indicator shown in Fig. 5a, implemented in Mobucon LTL, is evaluated, at some time t, through the formula $1 - \frac{\sum_i weight_i \#viol_i(t)}{\#events(t) \sum_i weight_i}$, and takes into account the number of violations of each

[2] Mobucon LTL also differentiates between possibly and permanently satisfied.

individual constraint of the reference model ($\#viol_i$) and its weight ($weight_i$). On the other hand, the compliance indicator shown in Fig. 5b, implemented in Mobucon EC, considers the number of violated ($\#viol$) and satisfied ($\#sat$) instances. In particular, at some time t the compliance indicator corresponds to $1 - \frac{\#viol(t)}{\#viol(t)+\#sat(t)}$[3].

3.4 Mobucon Clients

We have developed three Mobucon clients, in order to deal with different settings: (a) manual insertion of the events, (b) replay of a process instance starting from a complete event log, and (c) acquisition of events from an information system. The first two clients are mainly used for testing and experimentation. The last client requires a connection to some information system, e.g., a workflow management system. The three clients differ on how the user is going to provide the stream of events, but all of them include an interface with a graphical representation of the obtained fluent model, showing the evolution of constraints and also reporting the trend of the compliance indicator (Fig. 4).

4 Mobucon LTL

As discussed earlier, there are two Mobucon providers for monitoring business constraints: one based on LTL and one based on the EC. We now describe the LTL-based provider [9]. The basic idea is that a stream of events is monitored w.r.t. a given *Declare* reference model. Each LTL constraint implied by the *Declare* model is translated to a finite state automaton. Moreover, the conjunction of all LTL constraints is also translated to a finite state automaton. The generated automata are used to monitor the behavior. Using the terminology introduced in [9], we call the automaton corresponding to a single *Declare* constraint *local automaton* and the automaton corresponding to their conjunction *global automaton*. Local automata are used to monitor each single constraint in isolation, whereas the global automaton is used to monitor the entire system and detect non-local violations originated by the interplay of multiple constraints.

4.1 Modeling and Implementation

When Mobucon LTL receives a request from a new process instance, it first initializes the session for that instance. In particular, each single constraint of the *Declare* model associated to the session by the client and their conjunction are translated into finite state automata. For the translation, we use the algorithm introduced for the first time in [10] and optimized in [11]. Local and global automata are stored in the session. After that, the provider processes the event (or a collection of events) received with the first request from the client. The following requests will provide again single events or collections of events. The

[3] If $\#viol(t) + \#sat(t) = 0$, then the compliance indicator is defined to be 1.

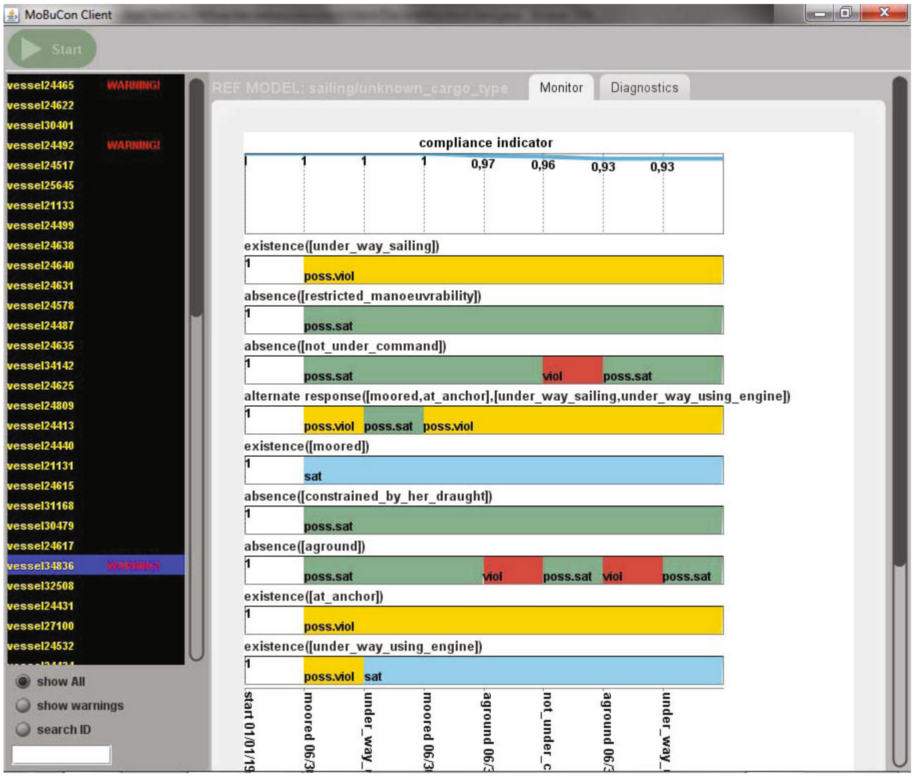


Fig. 4. Screenshot of one of the Mobucon clients

events are processed one by one by using the automata every time retrieved from the session. In this way, the state of each automaton is always preserved from the last request. The set of fluents' MVIs associated to each constraint is recomputed accordingly and returned by the reasoner.

4.2 Approach

Both global and local automata are reduced so that, if a transition violates the automaton from a certain state, this transition does not appear in the list of the outgoing transitions from that state. More specifically, a transition can be positive if it is associated to a single positive label (representing an event, e.g., *moored*), or negative if it is associated to negative labels (e.g., $\neg aground$). Positive labels indicate that we follow the transition when exactly the event corresponding to that label occurs, whereas negative labels indicate that we can follow the transition for any event not mentioned. Hence, acceptable events correspond to the label associated to a positive outgoing transition from the current state or to no one of the labels associated to a negative outgoing transition.

The Mobucon LTL provider checks first whether the processed event is acceptable by the global automaton. If the event is allowed, the provider fires the corresponding transition on the global automaton. In this case, to compute the state of every single constraint in isolation as well, it also fires the transitions corresponding to the processed event on the local automata (note that, if the event is acceptable by the global automaton, it is also acceptable by all local automata). If, after having fired the transition, a local automaton is in an non-accepting state, the corresponding constraint is possibly violated. If a local automaton is in an accepting state, the corresponding constraint is (possibly or permanently) satisfied. To distinguish between possibly and permanently satisfied constraints, the provider checks whether all possible events correspond to a self loop on the current state. If this is the case, the constraint is permanently satisfied, otherwise it is possibly satisfied. If the processed event violates the global automaton, from the point of view of the automata, the violating event is ignored. However, the provider still informs the client that the event caused a violation w.r.t. the reference model. Moreover, it also gives intuitive diagnostics about the violation. Indeed, the global automaton allows the provider to precisely identify which events were permitted instead of the one that caused the violation. This information is derived from the labels of the outgoing negative and positive transitions from the current state in the global automaton.

In some cases, a violation in the global automaton can be directly reduced to a violation of a local automaton. However, in other cases none of the individual local automata is violated as the problem stems from the interplay of multiple constraints [9]. In the latter case, the Mobucon LTL provider is able to identify the conflicting sets of constraints, i.e., the minimal sets of constraints that cause the violation.

5 Mobucon EC

Mobucon EC exploits a reactive EC-based reasoner to provide monitoring facilities. When a first query is received for some process instance, the provider applies a translation algorithm which analyzes the reference model stored in the corresponding session, producing a set of corresponding EC axioms. It then creates a new instance of the reasoner, initializing it with the EC theory obtained from the translation procedure. The reasoner instance is then stored into the session. Every time a new partial trace must be checked, the reasoner is extracted from the session and updated with the new events. This triggers a new reasoning phase in which the previously stored fluents' MVIs are revised and extended. The set of all MVIs is then returned by the reasoner.

In the following, we first sketch how *Declare* constraints, possibly augmented with data and metric temporal aspects, can be tackled by means of EC axioms. We then discuss the implementation of the reasoner.

5.1 Modeling

A comprehensive description of how the EC can be used in the *Declare* setting can be found in [8]. Here, we consider one of the constraints mentioned in Fig. 1b, namely the *response* constraint over a *Cargo ship*, to give an intuition about such a translation, considering data and metric temporal constraints as well.

Broadly speaking, an EC theory is a logic program which employs special predicates for modeling how fluents change over time, in response of the execution of certain events. For example, $initiates(e, f, t)$ ($terminates(e, f, t)$) is used to say that event e initiates (terminates) f, i.e., makes f true (false), at time t; $holds\_at(f, t)$ is used to run queries over the validity of fluents, in this case verifying whether f is true at time t. For a comprehensive description of the EC, we refer the reader to [12].

In the context of *Declare*, and differently from the LTL-based approach, the runtime characterization of business constraints is not given over the constraints themselves, but is tailored to constraints' instances. A constraint instance represents a specific "grounding" of the constraint inside a specific context, i.e., with specific data, specific instantiation time, and so on. According to this observation, in the EC-based formalization of *Declare* fluents have the form $state(i(ID, Params), State)$, where ID is the identifier of the constraint, $Params$ is a list of parameters characterizing a specific instance of the constraints, and $State$ is the current state of the instance, i.e., one among *sat*, *viol* and *pend* (to respectively model that the constraint instance is satisfied, violated or pending). In our example, the *response* constraint over a *Cargo ship* will be identified by *cr*, and the params characterizing each instance will be the identifier of the vessel (needed to properly correlate events) and the creation time (needed to properly check the metric temporal constraints).

EC axioms are given over event types, which are then subject, during the execution, to unification with each occurring concrete event. Event types have the form $exec(Name, Who, Data)$, where $Name$ is the name of the event, Who identifies the entity that originated the event, and $Data$ is a list of further data. The *response* over a *Cargo ship* is associated to the *moored* and (*Under way using*) *engine* events, which can be represented by the two event types $exec(moored, V_{id}, [V_{type}])$ and $exec(engine, V_{id}, [V_{type}])$. It is instantiated every time a *moored* event happens for a cargo vessel; the instance is put in a pending state, waiting for the occurrence of a corresponding *engine* event:

$$initiates(exec(moored, V_{id}, [cargo]), status(i(cr, [V_{id}, T]), pend), T)$$

A state transition from the pending to the satisfied state happens for an instance, if the following conditions hold: (1) the instance is currently pending; (2) an *engine* event occurs for a *Cargo ship*; (3) the event has the same vessel identifier of the instance; (4) the timestamp of the event is after the creation time of the instance, but before the actual deadline (which corresponds to the creation time plus 2880 minutes). Such state transition is modeled by terminating the previous state and initiating the new one, if all conditions are satisfied:

$$\begin{aligned}
&terminates(exec(engine, V_{id}, [cargo]), status(i(cr, [V_{id}, T_c]), pend), T) :- \\
&\qquad holds\_at(status(i(cr, [V_{id}, T_c]), pend), T), T > T_c, T \leq T_c + 2880. \\
&initiates(exec(engine, V_{id}, [cargo]), status(i(cr, [V_{id}, T_c]), sat), T) :- \\
&\qquad holds\_at(status(i(cr, [V_{id}, T_c]), pend), T), T > T_c, T \leq T_c + 2880.
\end{aligned}$$

Contrariwise, if a (generic) event happens at a time which is greater than the creation time of the instance plus 2880, and the constraint instance is still pending, this attests that the deadline has expired, and that a transition from the pending to the violated state must be triggered:

$$\begin{aligned}
&terminates(\_, status(i(cr, [V_{id}, T_c]), pend), T) :- \\
&\quad holds\_at(status(i(cr, [V_{id}, T_c]), pend), T), T > T_c + 2880. \\
&initiates(\_, status(i(cr, [V_{id}, T_c]), viol), T) :- \\
&\quad holds\_at(status(i(cr, [V_{id}, T_c]), pend), T), T > T_c + 2880.
\end{aligned}$$

Finally, a further general rule is added to state that each pending instance becomes violated when the process instance is completed.

The visualization depicted in Fig. 5b shows the status of the various constraints for a running trace and is based on the above axioms (together with the ones modeling the other constraints in Fig. 1b).

5.2 Reasoner Implementation

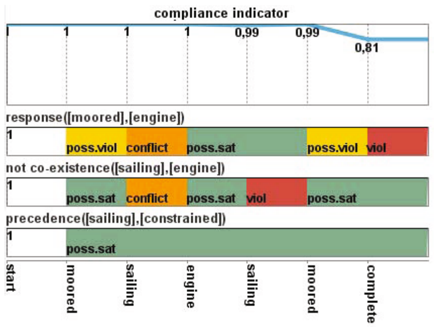
To effectively compute the MVIs characterizing the evolution of each constraint instance, Mobucon EC relies on a reactive EC reasoner and three translation components. A first translator converts the XML representation of a *Declare* reference model to a corresponding set of EC axioms. A second one converts a XES (partial) trace to a set of logic programming facts, also applying a translation of timestamps using the chosen granularity; such facts are then matched against the EC axioms that formalize the reference model. A last translator is used to convert the outcome produced by the reasoner (a set of strings) to a fluent model according to the schema of Fig. 3.

The reactive reasoner is inspired by the Cached EC (CEC) developed by Chittaro and Montanari [13]. It uses a Prolog-based axiomatization of the EC predicates following the CEC philosophy, i.e., already computed MVIs of fluents are cashed and subsequently revised and extended as new events are received.

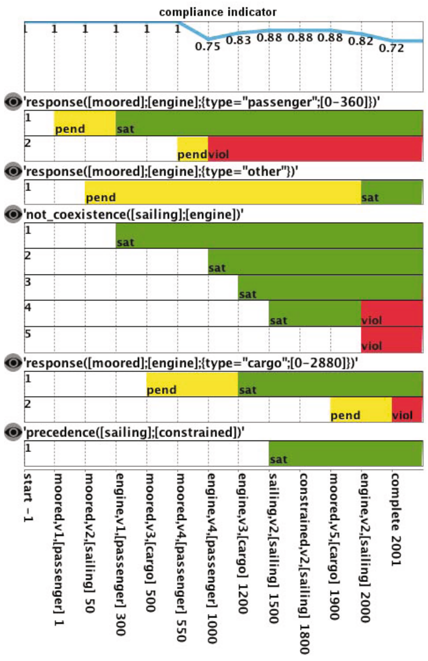
Different underlying Prolog engines can be plugged into the tool. In particular, we experimented TuProlog (`tuprolog.alice.unibo.it/`) which is completely implemented in JAVA and thus guarantees a seamless integration inside ProM, and YAP (`yap.sourceforge.net/`), which is one of the highest-performance Prolog engine available today.

6 Case Study

In this section, we present the application of the two Mobucon providers (LTL and EC) as part of a case study conducted within the research project Poseidon (`www.esi.nl/poseidon/`) and focused on the analysis of vessel behavior in the domain of maritime safety and security. The case study has been provided by Thales, a global electronics company delivering mission-critical information systems and services for the Aerospace, Defense, and Security markets. In our experiments, we use logs collected by an on-board maritime Automatic Identification System



(a) Mobucon LTL.



(b) Mobucon EC.

Fig. 5. Examples of monitoring results in our case study

(AIS) [14], which acts as a transponder that logs and sends events to an AIS receiver. An event represents a change in the navigational status of a vessel (e.g., *moored* or *Under way using engine*). Each event has an associated vessel ID and vessel type (e.g., *Passenger ship* or *Cargo ship*). The logs are excerpts of larger logs and correspond to a period of one week. The standard behavior of the vessels is described by domain expert using *Declare*, where constraints are used to check the compliance of the behavior of vessels as recorded in the logs.

Let us first focus on the Mobucon LTL provider. Figure 1a shows the reference model used to monitor vessels behavior. Each vessel corresponds to a process instance in the log. Figure 5a shows a graphical representation of the constraints' evolution for a specific instance. Events are displayed on the horizontal axis (for the sake of readability, a more compact notation is used). The vertical axis shows the constraints, reporting their evolution as events occur.

When event *moored* is executed the *response* constraint becomes possibly violated. Indeed, the constraint is waiting for the occurrence of another event (execution of (*Under way using*) *engine*) to become satisfied. After *moored*, (*Under way*) *sailing* is executed, leading to a conflict caused by the interplay of the *not coexistence* and the *response* constraints. The conflict is due to the fact that the first constraint forbids whereas the other constraint requires the presence of event *engine*. Note that, after a conflict or a (local) violation, constraints can become non-violated. In fact, Mobucon LTL implements a recovery strategy where the violating events are ignored (after having been reported). In this case, for instance, when *sailing* occurs, the conflict is raised but the event is, in fact, ignored. The next event is *engine* and *response* (that was possibly violated before the conflict) becomes possibly satisfied. After that, when event *sailing* occurs, *not coexistence* becomes permanently violated because *engine* and *sailing* cannot coexist in the same trace (note that also in this case the violating event is ignored after that the violation has been reported). The next event is *moored* and *response* becomes possibly violated. When the case completes, the *response* constraint becomes violated because it is not possible to satisfy it anymore.

Finally, note the trend of the compliance indicator in Fig. 5a. The indicator decreases in correspondence of each (local) violation. This example also shows clearly that a violation of the *response* constraint influences the indicator more than a violation of the *not coexistence* constraint.

Let us now consider the Mobucon EC provider, which employs the reference model shown in Fig. 1b. In order to show the potentiality of the approach, we consider in this case the unique events stream generated by the AIS receiver; correlation between events referring the same vessel is under the responsibility of the framework itself, using the formalization discussed in Sec. 5. Figure 5b shows a graphical representation of the constraints' evolution. Events (with attached data and timestamps) are displayed on the horizontal axis. The vertical axis shows the constraints and their instances, reporting their evolution as time flows.

Every time event *moored* occurs, a new instance of the *response* constraint (for the specific vessel type) is created. At first, the state of the instance is pending because it is waiting for the occurrence of an (*Under way using*) *engine* event referring to the same vessel ID, and within the deadline specific for the corresponding vessel type. Event *engine* occurs for *Passenger ship v1* less than 6 hours after *moored*. For *v4* this takes more than 6 hours, thus resulting in a violation. Similar to the example used for the Mobucon LTL provider, also in this case, the occurrence of *sailing* for *Sailing boat v2* generates a conflict between the instance of the *response* constraint and the instance of the *not coexistence* constraint corresponding to this vessel. They can never become both satisfied, the first requiring and the other forbidding the presence of event *engine* for this vessel. However, unlike the LTL-based provider, the Mobucon EC provider does not point out any problem when the conflict arises. Only when, as the last event of the trace, *engine* occurs for *v2*, the instance of the *not coexistence* constraint for vessel *v2* becomes violated. This example shows that, on the one hand, the Mobucon EC provider is able to monitor constraints augmented with data conditions and metric temporal constraints. On the other hand, the Mobucon LTL provider supports the early detection of violations originating from a conflict among two or more constraints.

As explained in Sect. 3.3, the compliance indicator is computed differently in both providers. For both providers the indicator decreases after each violation. However, in EC-based provider, the compliance indicator increases when new satisfied instances are created.

Table 1. Comparison between the Mobucon LTL and EC providers (I = implemented, I* = partially implemented, + = supported by the formal framework, – = not supported by the formal framework)

	LTL	EC		LTL	EC
1. single constraints monitoring	I	I	5. recovery and compensation	+	+
2. non-local violations	I*	–	6. metric temporal aspects	–	I
3. continuous support	I	I	7. data and data-aware conditions	–	I*
4. diagnostics	I	–	8. non-atomic activities	–	+

7 Discussion and Conclusion

This paper presents a new Operational decision Support (OS) framework for monitoring business constraints. The framework implementation exploits the functionalities provided by the OS service in ProM. Mobucon comes with a general flexible architecture able to accommodate multiple reasoning engines. In this paper, we demonstrate two such engines, one based on (finite-trace) Linear Temporal Logic (LTL) and automata, and the other on the Event Calculus (EC) and a Prolog-based reactive reasoner.

In the literature, most of the proposed approaches for compliance verification either work on static models at design time [15,16] or on off-line a-posteriori conformance checking [17] using only historical data. The majority of approaches for *online* business process monitoring focus on measuring numerical attributes, such as Key Performance Indicators (KPIs). For example, in [18], a framework is introduced for modeling and monitoring of KPIs in Semantic Business Process Management. In particular, the authors integrate the KPI management into a semantic business process lifecycle, creating an ontology that is used by business analysts to define KPIs based on ontology concepts. In [19], an integrated framework is presented for run-time monitoring and analysis of the performance of WS-BPEL processes. In particular, this framework allows for dependency analysis and machine learning with the ultimate goal of discovering the main factors influencing process performance (KPI adherence).

An exception to this trend is the work by Ly et al. on semantic constraints in business processes [20]. This work is more related the one presented here. Both approaches recognize the importance of runtime compliance verification of processes with rules and constraints. However, while Ly et al. aims to describe a comprehensive framework for compliance of semantic constraints over the whole process lifecycle, here we have proposed concrete ways for attacking this problem during the execution of processes.

Table 1 provides a comparison of our two OS providers for monitoring business constraints (LTL-based and EC based). Analysis of this table provides some interesting insights. First of all, both approaches are able to manage the monitoring of individual business constraints. Non-local violations refer to the situation in which no single constraint is currently violated, but there is a conflicting set of constraints. Whereas the LTL-based approach can discover non-local violations thanks to the construction of the global automaton, the EC-based approach does not support this. Note that the detection of non-local violations is currently only partially supported by the Mobucon LTL provider: the non-local violations is detected, but the minimal conflicting set is not yet computed efficiently. We are currently working on extending the colored-automata based approach to more efficiently identify minimal sets of conflicting constraints [21]. Both approaches support continuous support, i.e., the monitoring framework is able to provide support even after a violation takes place. While the Mobucon EC provider is only able to detect that a violation has taken place, Mobucon LTL also provides diagnostics about which events were expected (not) to occur. Although recovery and compensation mechanisms have not yet been included in our implementation, both approaches can support them [9,22].

The last three rows in Tab. 1 refer to the extension of the *Declare* language. Metric temporal aspects have been already incorporated into the Mobucon EC provider [8]. Metric temporal logics and timed-automata will be investigated to improve the LTL-based approach in this direction. Data and data-aware conditions are not-expressible in LTL, while the EC-based tool is being extended to accommodate them. Its ability to support data is attested by the formalization example shown in Sec. 5 and Fig. 5b. Similarly, EC is also able to support non-atomic activities.

Finally, let us briefly comment on the performance of the two approaches. For the Mobucon LTL provider, a recent investigation has revealed that very efficient algorithms can be devised for building local and global automata [11]. Once the automata are constructed, runtime monitoring can be supported in an efficient manner. The state of an instance can be monitored in constant time, independent of the number of constraints and their complexity. According to [11], the time to construct an automaton is 5-10 seconds for random models with 30-50 constraints. For models larger than this, automata can no longer routinely be constructed due to lack of memory, even on machines with 4-8 GiB RAM. For the Mobucon EC provider, some complexity results are inherited from the seminal investigation by Chittaro and Montanari [13]. An initial investigation of the performance of this approach (with YAP Prolog as underlying reasoner) can be found in [8]. Differently from the LTL-based approach, whose most resource-consuming task is the generation of the automaton, which is done before the execution, the EC-based approach triggers a reasoning phase every time a new event is acquired. Despite this, our investigation shows that, for randomly generated models and traces, the reasoner takes an average time of 300ms to process the 1000th acquired event with a model containing 100 constraints.

References

1. van der Aalst, W.M.P., Pesic, M., Song, M.: Beyond Process Mining: From the Past to Present and Future. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 38–52. Springer, Heidelberg (2010)
2. Verbeek, E., Buijs, J., van Dongen, B., van der Aalst, W.: Prom 6: The process mining toolkit. In: Demo at BPM 2010 (2010)

3. Westergaard, M., Maggi, F.M.: Modeling and Verification of a Protocol for Operational Support Using Coloured Petri Nets. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 169–188. Springer, Heidelberg (2011)
4. Pesic, M., Schonenberg, H., van der Aalst, W.: DECLARE: Full Support for Loosely-Structured Processes. In: EDOC 2007, pp. 287–300 (2007)
5. Pesic, M., van der Aalst, W.M.P.: A Declarative Approach for Flexible Business Processes Management. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
6. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative Specification and Verification of Service Choreographies. ACM Transactions on the Web 4(1) (2010)
7. Montali, M.: Specification and Verification of Declarative Open Interaction Models. LNBIP, vol. 56. Springer, Heidelberg (2010)
8. Montali, M., Maggi, F., Chesani, F., Mello, P., van der Aalst, W.: Monitoring Business Constraints with the Event Calculus. Technical Report DEIS-LIA-002-11, University of Bologna (Italy) (2011), LIA Series no. 97, `http://www.lia.deis.unibo.it/Research/TechReport/LIA-002-11.pdf`
9. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 132–147. Springer, Heidelberg (2011)
10. Giannakopoulou, D., Havelund, K.: Automata-Based Verification of Temporal Properties on Running Programs. In: ASE 2001, pp. 412–416 (2001)
11. Westergaard, M.: Better Algorithms for Analyzing and Enacting Declarative Workflow Languages Using LTL. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 83–98. Springer, Heidelberg (2011)
12. Shanahan, M.: The Event Calculus Explained. In: Artificial Intelligence Today: Recent Trends and Developments, pp. 409–430 (1999)
13. Chittaro, L., Montanari, A.: Efficient Temporal Reasoning in the Cached Event Calculus. Computational Intelligence 12, 359–382 (1996)
14. International Telecommunications Union: Technical characteristics for a universal shipborne Automatic Identification System using time division multiple access in the VHF maritime mobile band. Recommendation ITU-R M.1371-1 (2001)
15. Governatori, G., Milosevic, Z., Sadiq, S.W.: Compliance Checking Between Business Processes and Business Contracts. In: EDOC 2006, pp. 221–232 (2006)
16. Awad, A., Decker, G., Weske, M.: Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 326–341. Springer, Heidelberg (2008)
17. van der Aalst, W.M.P., de Beer, H.T., van Dongen, B.F.: Process Mining and Verification of Properties: An Approach Based on Temporal Logic. In: Meersman, R., Tari, Z. (eds.) CoopIS/DOA/ODBASE 2005. LNCS, vol. 3760, pp. 130–147. Springer, Heidelberg (2005)
18. Wetzstein, B., Ma, Z., Leymann, F.: Towards measuring key performance indicators of semantic business processes. In: BIS 2008, pp. 227–238 (2008)
19. Wetzstein, B., Leitner, P., Rosenberg, F., Brandic, I., Dustdar, S., Leymann, F.: Monitoring and analyzing influential factors of business process performance. In: EDOC 2009, pp. 141–150 (2009)

20. Ly, L.T., Göser, K., Rinderle-Ma, S., Dadam, P.: Compliance of Semantic Constraints - A Requirements Analysis for Process Management Systems. In: GRCIS 2008 (2008)
21. Maggi, F.M., Westergaard, M., Montali, M., van der Aalst, W.M.P.: Runtime Verification of LTL-Based Declarative Process Models. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 131–146. Springer, Heidelberg (2012)
22. Chesani, F., Mello, P., Montali, M., Torroni, P.: Verification of Choreographies During Execution Using the Reactive Event Calculus. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 55–72. Springer, Heidelberg (2009)