Compliance Checking of Execution Traces to Business Rules: an Approach based on Logic Programming *

Federico Chesani[1], Paola Mello[1], Marco Montali[1], Fabrizio Riguzzi[2], Maurizio Sebastianis[3], and Sergio Storari[2]

[1] DEIS - University of Bologna, V.le Risorgimento 2, 40136 Bologna - Italy
{federico.chesani | paola.mello | marco.montali}@unibo.it,
[2] ENDIF - University of Ferrara, Via Saragat 1, 44100 Ferrara - Italy
{fabrizio.riguzzi | sergio.storari}@unife.it
[3] Think3 Inc., Via Ronzani 7/29, 40033 Casalecchio di Reno (BO) - Italy
maurizio.sebastianis@think3.com

Abstract. Complex and flexible business processes are critical not only because they are difficult to handle, but also because they often tend to be less intelligible. Monitoring and verifying complex and flexible processes becomes therefore a fundamental requirement. We propose a framework for performing compliance checking of process execution traces w.r.t. expressive reactive business rules, tailored to the MXML meta-model. Rules are mapped to (extensions of) Logic Programming, to the aim of providing both monitoring and a-posteriori verification capabilities. We show how different rule templates, inspired by the ConDec language, can be easily specified and then customized in the context of a real industrial case study. We finally describe how the proposed language and its underlying a-posteriori reasoning technique have been concretely implemented as a ProM analysis plug-in.

1 Introduction

Recently, Workflow Management Systems (WfMS) have been increasingly applied by companies in order to efficiently implement their Business Processes. A plethora of tools, systems and notations have been proposed to cover all the phases of the Business Process Management life-cycle, from Process Design and Modeling to Execution and Monitoring/Analysis. To deal with needs and requirements of business users, two main dimensions have been recently tackled: *flexibility* and *complextity*. On one side, to be successfully employed WfMS should make a trade-off between controlling the way workers do their business and turning them loose to exploit their expertise during execution [1]; while constraining workers to follow a business process model, flexible WfMS support the

* An short version of this paper, focused on the application, is currently submitted to the 4th Workshop on Business Process Intelligence (BPI2008), held in conjunction with BPM2008.

possibility of deviating from its prescriptions and even changing it at run-time. On the other side, business processes are exploited to model complex problems and domains under different perspectives (e.g. the control flow perspective and the organizational one); to have an idea of such a complexity, just take a look to the Worfklow Patterns [2] initiative[4].

Both dimensions are critical not only because they are difficult to handle, but also because they contribute to make the process less intelligible. Monitoring and verifying complex and flexible processes becomes therefore a fundamental requirement. On the one hand, as claimed in [3] "deviations from the 'normal process' may be desiderable but may also point to inefficiencies or even fraud", and therefore flexibility could lead the organization to miss its strategic goals or even to violate regulations and governance directives. On the other hand, as complexity increases it becomes important to provide support for a business manager in the task of analyzing past/ongoing process executions, in particular to verify whether they meet certain requirements or business rules. This analysis can help the business manager in the process of assessing business trends and consequently making strategic decisions.

In this paper, we focus on this specific task, proposing a framework for performing compliance checking of process execution traces w.r.t. reactive business rules. Such rules are specified by means of a powerful declarative language, inspired by the $\mathcal{S}$CIFF one [4]. $\mathcal{S}$CIFF is a framework based on Abductive Logic Programming, originally devised for modeling interaction within open Multi-Agent Systems and verifying whether interacting agents indeed comply with the prescribed model. Such a compliance verification can be seamlessly exploited at run-time, by dynamically acquiring and reasoning upon occurring events, or a-posteriori, by analyzing log traces of already completed executions.

In the last few years, $\mathcal{S}$CIFF has been applied in the context of Business Process Management and Service Oriented Computing, by investigating its reasoning capabilities to verify a-priori interoperability between a service behavioral interfaces and a choreography [5] and to perform run-time monitoring of exchanged messages checking adherence to choreography rules of engagement [6]. Even more recently, it has been shown that $\mathcal{S}$CIFF is able to formalize all core ConDec [7] constraints, supporting temporal-oriented extensions of such graphical languages and providing different underlying verification capabilities [8]. The work here presented is therefore part of a larger framework, called CLIMB[5], which aims at applying (extensions of) Logic Programming for modeling and verifying business processes. Although CLIMB business rules stem from ConDec constraints, their expressiveness is extended not only as regards temporal aspects but also as regards event data, such as involved originators, event types and activity identifiers.

The need for a flexible and easy-comprehensible language to specify these kind of rules and the importance of a corresponding compliance verification

[4] `http://www.workflowpatterns.com`

[5] The interested reader is referred to `http://www.lia.deis.unibo.it/research/climb`.

framework are motivated by describing its concrete application on a real business process of Think3®[6], a company working in the Computer Aided Design (CAD) and Product Life-cycle Management (PLM) market.

We sketch how the proposed language can be mapped to Logic Programming ($\mathcal{S}$CIFF and Prolog in particular), enabling compliance verification both at runtime and a posteriori. The latter technique has been exploited to implement a ProM [9] plug-in, called $\mathcal{S}$CIFFChecker, that classifies a set of MXML [10] execution traces as compliant/non-compliant w.r.t. a certain business rule, in the style of *LTL Checker* [3].

The paper is organized as follows. Section 2 grounds the compliance checking problem on the Think3 case study. Language and methodology for specifying and applying CLIMB business rules are presented in Section 3. Section 4 briefly sketches how CLIMB rules can be mapped to Logic Programming and illustrates the implementation of a-posteriori compliance checking inside ProM, reporting experiments made on the Think3 case study. Related works and conclusions follow.

2 An industrial case study

An important current challenge in the manufactoring industry is to handle, verify and distribute the technical information produced by the design, development and production processes of the company. The adoption of a system supporting the management of technical data and the coordination of the people involved is of key importance, in order to improve productivity and competitiveness. The main issue is to provide solutions for managing all the technical information and documentation (such as CAD projects, test results, photos, revisions), mainly focusing on the design phase, which produces most such data. Storing and tracking relevant information concerning an item is necessary in this context, because an important part of the design process is spent by testing, modifying and improving previously released versions.

Think3 is one of the leading global players in the field of CAD and PLM solutions: it provides an integrated software which bridges the gap between CAD modeling environments and other tools involved in the process of designing (and then manufacturing) products. All these tools are transparently combined with a non-intrusive information system which handles the underlying product workflow, recording all the relevant information and making it easily accessible to the workers involved, enabling its consultation, use and modification. Such an information system supplies a detailed, shared and constantly updated vision of the life-cycle of each product, as regards documentation of its features as well as traceability of the activities performed on it.

The underlying Think3 workflow centres around the design of a manufacturing product. Different activities can be executed to affect the progress status of an item, involving the modification and even the evolution of multiple co-existing

[6] `http://www.think3.com`

versions of its corresponding project. Such a workflow can be adapted to each single Think3 client company in order to meet different specific requirements.

2.1 Compliance Checking and Decision Making Support: Think3 Requirements

To support a business manager in decision making, and in particular in the tasks of analyzing the life-cycle of different projects and pinpointing problems and bottlenecks, Think3 is investigating the development of a Business Intelligence dashboard. The feasibility of such a dashboard relies on the *traceability* provided by its solution: all the relevant technical and documental information as well as the history of involved executed activities are stored by the information system. Within the TOCAI.IT FIRB Project[7], Think3 and the University of Bologna are collaborating for realizing one of the main dashboard components: a tool supporting compliance verification (both on and off-line) of design processes w.r.t. configurable business rules. This will facilitate the manager in the identification of behavioural trends and non-compliances to regulations or internal policies. In this particular case study, we elicitated the following non-exaustive lists of interesting properties:

- (Br1) Quantifying projects which have been subject to a certain activity within a given time interval (e.g., *How many projects have been modified between 01/2007 and 04/2007?*).
- (Br2) Evaluating the time relationship between the execution of two given activities (e.g. *Was a project committed by 18 days after its creation?*).
- (Br3) Identifying which projects passed too many times through a certain activity (e.g., *Which projects have been modified twice?*).
- (Br4) Analysing activities originators, i.e., workers involved in the process (e.g., *Was a project checked by a person different than the one who published it?*).

Note that such rules can be seamlessly exploited either to analyze process executions or in a prescriptive (deontic) manner; for example, rule (Br2) could be used to obtain an overview about projects throughput as well as to monitor workers in order to detect as soon as possible the violation of a certain deadline (and acting consequently).

3 CLIMB Business Rules

We propose a language, inspired by the $\mathcal{S}$CIFF one [4], for specifying reactive business rules (called CLIMB business rules throughout the paper). Their structure resembles the one of ECA (Event-Condition-Action) rules [11, 12]; the main difference is that, since they are used for checking, they envisage expectations about executions rather than actions to be executed. Expectations represent

[7] `http://www.dis.uniroma1.it/~tocai/`

events that should (not) happen. Therefore, CLIMB rules are used to constrain the process execution when a given situation holds. Both positive and negative constraints can be imposed on the execution, i.e., it is possible to specify what is mandatory as well as forbidden in the process.

Rules follow an `IF` *Body* `having` *BodyConditions* `THEN` *Head* structure, where *Body* is a conjunction of occurred events, with zero or more associated conditions *BodyConditions*, and *Head* is a disjunction of conjunctions of positive and negative expectations (or *false*). Each head element can be subject to conditions as well. An excerpt of the grammar is shown in Figure 1.

$$
\begin{aligned}
\mathit{Rule} &::= [\mathit{IF}\ \mathit{Body}\ \mathit{THEN}]\ \mathit{Head} \\
\mathit{Body} &::= \mathit{Activity\_Exec} \\
&\quad [\mathit{AND}\ \mathit{Activity\_Exec}]^{\star}\ [\mathit{AND}\ \mathit{Constraints}] \\
\mathit{Activity\_Exec} &::= \mathit{Simple\_Activity} \mid \mathit{Repeated\_Activity} \\
\mathit{Simple\_Activity} &::= \mathit{activity}\ \mathit{A\_ID}\ \mathit{is\ performed}\ [\mathit{by}\ \mathit{O\_ID}]\ [\mathit{at\ time}\ \mathit{O\_T}] \\
\mathit{Repeated\_Activity} &::= \mathit{activity}\ \mathit{A\_ID}\ \mathit{is\ performed}\ \mathit{N\ times}\ [\mathit{by}\ \mathit{O\_ID}] \\
&\quad [\mathit{between\ time}\ \mathit{O\_T}\ \mathit{and\ time}\ \mathit{O\_T}] \\
\mathit{Head} &::= \mathit{Head\_Disjunct}\ [\mathit{OR}\ \mathit{Head\_Disjunct}]^{\star} \\
\mathit{Head\_Disjunct} &::= \mathit{Activity\_Exp} \\
&\quad [\mathit{AND}\ \mathit{Activity\_Exp}]^{\star}\ [\mathit{AND}\ \mathit{Constraints}] \\
\mathit{Activity\_Exp} &::= \mathit{Simple\_Activity\_Exp} \mid \mathit{Repeated\_Activity\_Exp} \\
\mathit{Simple\_Activity\_Exp} &::= \mathit{activity}\ \mathit{A\_ID}\ \mathit{should}\ [\mathit{not}]\ \mathit{be\ performed} \\
&\quad [\mathit{by}\ \mathit{O\_ID}]\ [\mathit{at\ time}\ \mathit{O\_T}] \\
\mathit{Repeated\_Activity\_Exp} &::= \mathit{activity}\ \mathit{A\_ID}\ \mathit{should\ be\ performed}\ \mathit{N\ times} \\
&\quad [\mathit{by}\ \mathit{O\_ID}]\ [\mathit{between\ time}\ \mathit{O\_T}\ \mathit{and\ time}\ \mathit{O\_T}]
\end{aligned}
$$

Fig. 1. An excerpt of the CLIMB rules grammar.

The underlying intuitive semantics is that whenever a set of occurred events makes *Body* (and the corresponding conditions *BodyConditions*) true, then also *Head* must eventually be satisfied[8] (see Section 4). Furthermore, it is possible to specify rules without the IF part: such rules are used to impose what the business manager expects (not) to find inside the process instances in any case.

The concept of event is tailored to the one of audit trail entry in the MXML meta-model [10]. An event is atomic and is mainly characterized by:

- the identifier/name of the *activity* it is associated to;
- an *event type*, according to the MXML transactional model [10];
- an *originator*, identifying the worker who generated the event;
- an *execution time*;
- one or more involved *data* items (for simplicity, in the paper we will not take into account this aspect, but it can be seamlessly treated in our framework).

[8] Therefore, rules having *false* in the head are used to express denials.

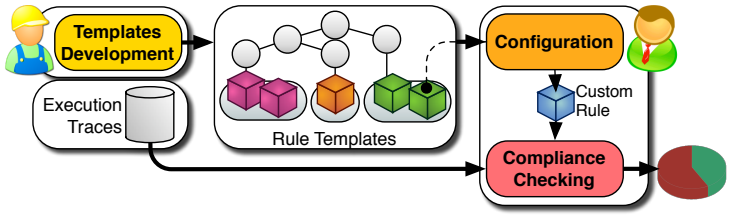

Fig. 2. A methodology for building, configuring and applying business rules.

The main distinctive feature of our rules is that all these parameters are treated, by default, as variables. To specify that a generic activity A has been subject to a whatsoever event, the rule body will simply contain a string like: `activity` A `is` $performed$ `by` O_A `at time` T_A, where A stands for the activity's name, O_A and T_A represent the involved originator and execution time respectively, and $performed$ is a keyword denoting any event type. To facilitate readibility, the part concerning originator and execution time can be omitted if the corresponding variables are not involved in any condition.

Such a generic sentence will match with any kind of event, because all the involved variables (A, O_A and T_A) are completely free, and the event type is not specified. It can then be configured to constrain involved variables or ground them to specific values, and by fixing a specific event type. To deal with the first case, explicit conditions are attached to variables, whereas for the latter case it is enough to substitute the generic $performed$ keyword with the specific one (e.g., *completed* to represent the completion of a certain activity).

Finally, positive and negative expectations are represented similarly to occurred events, by simply changing the `is` part with `should be` or `should not be` respectively.

3.1 A Methodology for Building Rules

To clarify the methodology we propose for developing rules, let us consider a completely configured business rule, namely the specification of the (Br4) rule:

`IF activity` A `is` $performed$ `by` O_A `having` A `equal to` $Check$
`THEN activity` B `should NOT be` $performed$ `by` O_B (Think3-4)
`having` B `equal to` $Publish$ `and` O_B `equal to` O_A

By analyzing this rule, we can easily recognize two different aspects: on one hand, the rule contains generic elements, free variables and constraints, whereas on the other hand it specifically refers to concrete activities. The first aspect captures re-usable patterns: in this case, the fact that the same person cannot perform two different activities A and B, which is known as the *four-eyes principle*. The second aspect instantiates the rules in a specific domain, in this case grounding the four-eyes principle in the context of Think3's workflow.

To reflect such a separation, we foresee a three-step methodology to build, configure and apply business rules (see Figure 2): *(i)* a set of re-usable rules, called *rule templates*, are developed and organized into groups by a technical expert (i.e., someone having a deep knowledge of rules syntax and semantics); *(ii)* rule templates are further configured, constrained and customized by a business manager to deal with her specific requirements and needs; *(iii)* configured rules are exploited to perform compliance checking of company's execution traces.

In the next sections we will introduce the different conditions supported by the language, and we will propose a core set of rule templates, inspired by ConDec constraints, showing how they can be easily customized to deal with the Think3 case study.

3.2 Specification of Conditions

As already pointed out, conditions are exploited to constrain variables associated to event occurences and expectations inside business rules (namely activity names, originators and execution times). Two main families of conditions are currently envisaged: string and time conditions. String conditions are used to constrain an activity/originator by specifying that it is equal to or different than another activity/originator, either variable or constant. An example of a string condition constraining two originator variables is the "O_B `equal to` O_A" part in rule (Think3-4).

Time conditions are used instead to relate execution times, in particular for specifying ordering among events or imposing quantitative constraints, such as deadlines and delays. The semantics of constraints is determined by time operators, which intuitively capture basic time relationships (such as $before$ or at). Absolute time conditions constrain a time variable w.r.t. a certain time/date, whereas relative time conditions define orderings and constraints between two variables. Relative conditions can optionally attach a displacement to the target time variable as well. A displacement is defined by a time duration and by an operator which determines whether the duration will translate the involved time variable forward or backward in time. For example, to specify that the time variable T_B must be within 2 days *after* T_A, we simply write T_B `BEFORE` T_A+2$days$.

3.3 Rule Templates

The first step in our methodology envisages the creation of rule templates, i.e. reusable partially constrained rules. They typically fix the rule structure, e.g. deciding how many events are contained in the body, and use variable string conditions and/or relative time conditions. They do not involve absolute time and constant conditions, which are exploited to ground rules on a specific domain.

We have developed a hierarchy of rules which strictly resembles the one proposed for ConDec. Part of this hierarchy is depicted in Figure 3. Three basic groups, similar to the ConDec ones, are defined: *existence* rules, *IF... THEN* rules and *IF... THEN NOT* rules. Such hierarchy is not fixed: it can be adapted or even replaced by writing other rules and by organizing them differently.

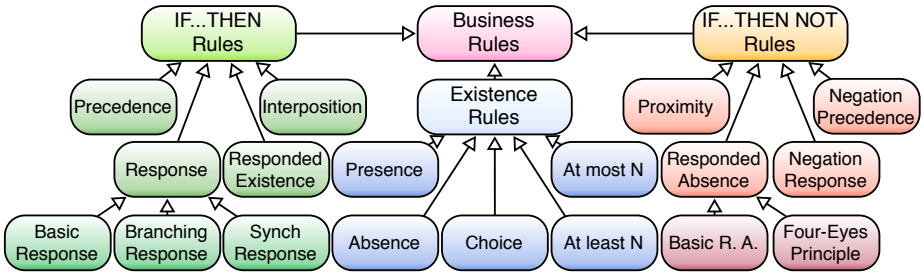

Fig. 3. A templates hierarchy inspired by Condec constraints.

Existence Rules impose the presence/absence of some events in the execution trace, independently from the occurrence of other events (except from the *At Most N* rule), they therefore have a *true* body. The *presence* (*absence*) template simply state that a certain event is expected to occur (not to occur), and is simply formalized as: `activity` A `should (NOT) be` $performed$. *Choice* extends *presence* by introducing disjunction of expectations. The *at least N* (*at most N*) rule extends the presence (absence) one by stating that the specified event should (not) be repeated N times. Such rules are useful for modeling the presence/absence of multiple instances of a certain event in the execution trace, as in the (Br3) Think3 example, but they are rather difficult to be represented, especially when N increases.

For this reason, we have extended the syntax of the language for supporting repetitions as first-class entities. To specify that activity A should be performed at least 3 times, we will then write: `activity` A `should be` $performed$ `3 times between` T_{sA} `and` T_{cA}. The two involved time variables extend the concept of execution time when dealing with multiple events, by identifying the two time points at which the repetition starts and completes[9]. Finally, to express that A must be performed at most 3 times, we can exploit repetitions as follows: `IF activity` A `is` $performed$ `4 times THEN` $false$.

IF... THEN Rules are positive relationships which specify that when certain events happen, then also other events should occur, satisfying the imposed time orderings. The simplest rule belonging to this group is the *responded existence* one, which simply states that when a certain event happens, then another event should happen too, either before or afterward. Starting from this rule, *response* and *precedence* templates extend it by adding respectively an *after* and *before* relative time condition among the involved execution times.

Response and precedence rules can then be specialized to express more complex event patterns, e.g. introducing conjunctions and disjunctions of events. For example, the following template represents a *synchronized response*, i.e. a

[9] This kind of rules obviously involve multiple originators, but for space reasons we do not describe here how they can be constrained.

response triggered by the occurrence of two events:

$$\begin{aligned}
&\texttt{IF activity } A \texttt{ is } performed \texttt{ at time } T_A\\
&\texttt{and activity } B \texttt{ is } performed \texttt{ at time } T_B\\
&\texttt{THEN activity } C \texttt{ should be } performed \texttt{ at time } T_C\\
&\texttt{having } T_C \; after \; T_A \texttt{ and } T_C \; after \; T_B.
\end{aligned}$$

By inverting the last temporal condition (i.e. by imposing $T_C \; before \; T_B$) the user can express an *interposition* template, which states that C should be performed between A and B. Finally, note that more complex ConDec constraints can be specified by combining two different templates: for example, the *alternate precedence* ConDec constraint is specified by means of CLIMB response and interposition.

IF...THEN NOT Rules are the negative version of positive relationships: they express events to be forbidden when other events happen. Roughly speaking, they replace positive expectations with negative ones; e.g., the negation of the interposition template is the *proximity* one, which states that between two given activities A and B another activity C cannot be performed [10].

The *responded absence* pattern is a good example to illustrate templates which involve conditions about originators; in its basic form it states that `IF activity` A `is` $performed$ `THEN activity` B `should not be` $performed$, but by adding an *equal to* constraint between the originators of A and B, it actually models the already cited four-eyes principle.

3.4 From Templates to Customized Business Rules

In a second phase, rule templates are configured by a business manager to deal with her specific requirements. This step exploits constant string conditions, absolute time conditions and relative time conditions with displacements, involving activities, originators and times specifically referring to the company's domain. For example, while Rule (Br3) simply grounds the *at least N* template with the *Modify* activity, rule (Br2) is formalized by adding a $before$ relative time constraint with a displacement of 18 days, thus modeling the presence of a deadline:

$$\begin{aligned}
&\texttt{IF activity } A \texttt{ is } performed \texttt{ at time } T_A\\
&\texttt{having } A \texttt{ equal to } Creation\\
&\texttt{THEN activity } B \texttt{ should be } performed \texttt{ at time } T_B \qquad \text{(Think3-2)}\\
&\texttt{having } B \texttt{ equal to } Commit\\
&\quad \texttt{and } T_B \texttt{ after } T_A \texttt{ and } T_B \texttt{ before } T_A + 18days.
\end{aligned}$$

[10] If activity C is left unspecified also in the customized version, it will match with any performed activity, and therefore the rule will be used for checking that A and B are next to each other.

Rule (Br1) constrains the presence template with absolute time conditions:

$$\begin{aligned} &\texttt{activity } A \texttt{ should be } performed \texttt{ at time } T_A \\ \texttt{having } &A \texttt{ equal to } Modify \texttt{ and} \\ &T_A \texttt{ after } 01/2007 \texttt{ and } T_A \texttt{ before } 04/2007. \end{aligned} \tag{Think3-1}$$

4 Compliance Verification with Logic Programming

Having described syntax and features of our rules language, we now briefly sketch how they can be automatically mapped to two Logic Programming frameworks (namely $\mathcal{S}$CIFF [4] and Prolog), enabling monitoring (run-time compliance verification) and a-posteriori compliance checking of process execution traces w.r.t. CLIMB rules.

Mapping to $\mathcal{S}$CIFF is straightforward because of the deep similarities between the two languages. For space reasons, we report here a mapping example, referring to [8] for more details. Occurred events and positive/negative expectations are directly mapped to $\mathcal{S}$CIFF happened events and expectations, represented respectively by the special predicates $\mathbf{H}(Ev, T)$ and $\mathbf{E}/\mathbf{EN}(Ev, T)$ where Ev is an event with the structure proposed in this paper and T is the corresponding execution time. For example, the customized four-eyes principle shown in rule (Think3-4) is simply specified in $\mathcal{S}$CIFF as follows[11]:

$$\begin{aligned} &\mathbf{H}(event(\_, A, O_A), T_A) \wedge A = \texttt{'Check'} \\ \rightarrow &\mathbf{EN}(event(\_, B, O_B), T_B) \wedge A = \texttt{'Publish'} \wedge O_B \neq O_A. \end{aligned}$$

The constraints among the variables, and in particular those on the activity execution times, are treated in $\mathcal{S}$CIFF by means of Constraint Logic Programming.

The declarative semantics of a $\mathcal{S}$CIFF specification is given in terms of an Abductive Logic Program (ALP). In general, an ALP [13] is a triple $\langle P, A, IC \rangle$, where P is a logic program, A is a set of predicates named *abducibles*, and IC is a set of integrity constraints. Reasoning in abductive logic programming is usually goal-directed and it accounts to finding a set of abduced hypotheses Δ built from predicates in A such that $P \cup \Delta \models G$ and $P \cup \Delta \models IC$ where G is a goal. Abduction is exploited to dynamically *generate* the expectations and to perform the *compliance check*. Expectations are defined as abducibles, and are hypothesised by the abductive proof procedure, i.e. the proof procedure makes hypotheses about the happening of the events. A confirmation step, where these hypotheses must be confirmed by happened events, is then performed: if no set of hypotheses can be fulfilled, a violation is detected.

Given a set **HAP** representing a process execution trace, a knowledge base KB, the set $\mathcal{E}$ of positive and negative expectations, and a set $\mathcal{IC}$ (obtained

[11] The first parameter of event is an anonymous variable because it represents the event type, which is not specified when the keyword *performed* is used.

by translating CLIMB rules), we provide semantics to a $\mathcal{S}$CIFF specification $\mathcal{S} \equiv \langle KB, \mathcal{E}, \mathcal{IC} \rangle$ by defining those sets $\mathbf{EXP} \subseteq \mathcal{E}$ ($\Delta \subseteq A$ in the abductive framework) of expectations which, together with the knowledge base and the happened events **HAP**, imply an instance of the goal (Eq. 1) - if any - and *satisfy* the integrity constraints (Eq. 2).

$$KB \cup \mathbf{HAP} \cup \mathbf{EXP} \models \mathcal{G} \tag{1}$$

$$KB \cup \mathbf{HAP} \cup \mathbf{EXP} \models \mathcal{IC} \tag{2}$$

Moreover, we require the set **EXP** (namely, an *abductive explanation*) to be also **E**-consistent: for any p, **EXP** cannot include $\{\mathbf{E}(p), \mathbf{EN}(p)\}$ (an event cannot be both expected to happen and expected not to happen).

We define the *compliance* of a process execution trace **HAP** with respect to a $\mathcal{S}$CIFF specification (composed of a knowledge base KB and of a set $\mathcal{IC}$ of rules) in terms of the fulfillment of a **EXP** set of expectations: each positive expectation should have a matching happened event, and for each negative expectation there should not be any matching happened event.

Definition 1. (Fulfillment) *Given a process execution trace* **HAP** *and a $\mathcal{S}$CIFF specification, a set of expectations* **EXP** *that is* $\mathbf{E}-$*consistent is* fulfilled *if and only if for all (ground) terms p:*

$$\mathbf{HAP} \cup \mathbf{EXP} \cup \{\mathbf{E}(p) \rightarrow \mathbf{H}(p)\} \cup \{\mathbf{EN}(p) \rightarrow \neg\mathbf{H}(p)\} \not\models \mathit{false} \tag{3}$$

Definition 2. (Compliance) *Given an execution trace* **HAP** *and a $\mathcal{S}$CIFF specification $\mathcal{S}$,* **HAP** *is compliant to $\mathcal{S}$ if and only if there exists an E-consistent set* **EXP** *such that Equations 1 and 2, and Definition 1 hold.*

$\mathcal{S}$CIFF has an operational proof-theoretic counterpart whose main purpose is *(i)* to dynamically acquire occurred events during execution, *(ii)* to generate expectations when the corresponding rule's body is triggered by such happened events, and *(iii)* to finally verify if the execution actually complies with the expectations raised. $\mathcal{S}$CIFF supports the compliance verification task both at run-time and a-posteriori. At run-time, the proof procedure is exploited by the SOCS-SI tool [4] to dynamically monitor and analyze the process behaviour. A posteriori, the same $\mathcal{S}$CIFF proof can be exploited to analyze the execution trace of a process. While in the following we will concentrate on the verification a posteriori, the interested reader can refer to [4] for run-time verification.

4.1 $\mathcal{S}$CIFFChecker: Compliance Checking in ProM

If compliance checking is performed a-posteriori (and therefore reactivity is not required anymore), then also pure Prolog can be exploited to reason upon execution traces, with an appreciable advantage in terms of performances with respect to the $\mathcal{S}$CIFF proof procedure. In this setting, the execution trace under study is treated as a knowledge base storing each audit trail entry as a fact of the type

$$happened(event(EventType, ActivityName, Originator), ExecutionTime).$$

The rule used for checking is instead transformed into a Prolog query by computing the negation of the implication represented by the CLIMB rule. So, if the CLIMB rule is represented by the implication $B \rightarrow H$, then the query would be $B \wedge \neg H$. Such a query tries to find a set of occurred events in the execution trace that satisfy the rule body but violate the rule head. For example, rule (Br4) is translated to the following query:

$$?{-}A = \texttt{'Check'}, happened(event(\_, A, O_A), T_A),$$
$$not(B = \texttt{'Publish'}, O_B \neq O_A, not(happened(event(\_, B, O_B), T_B))).$$

Note that since the analysis is performed a-posteriori, positive expectations are flattened to occurred events, and negative expectations to the absence of events. If the query succeeds, then a counter-example which violates the rule has been found in the execution trace, which is then evaluated as non-compliant. Although Prolog does not provide any explanation about the reasoning outcome (as $\mathcal{S}$CIFF does), it turns out to be very efficient, in particular if a huge number of execution traces have to be checked. However, the Prolog translation can be applied only to a subset of all the possible $\mathcal{S}$CIFF rules: this is not a problem when translating CLIMB rules, since they belong to such a set.

Drawing inspiration from the *LTL Checker* [3], we have therefore developed a ProM [9] analysis plug-in, called $\mathcal{S}$CIFFChecker, for the classification of MXML execution traces w.r.t. CLIMB business rules. In particular, we took inspiration from *LTL Checker* for what regards the functionalities offered, while not relying on temporal logic. $\mathcal{S}$CIFFChecker relies on the three-steps methodology described in Section 3.1, providing a user-friendly graphical interface for the customization of rule templates.

At start-up, rule templates are loaded from a templates file: the available templates follows the classification proposed in Section 3.3. More templates can be added by simply extending the templates file. Once a template has been chosen, it can be easily customized by clicking the "configuration" button (Figure 4): the modifiable elements becomes highlighted, and the user can set specific parameters by clicking on them. To proceed with the compliance checking, the user has to choose also a time *granularity*, which ranges from milliseconds to months and defines the time unit for converting involved time quantities into coherent integer values.

For each execution trace contained in the considered MXML log, three steps are then automatically performed: *(i)* the execution trace is translated into a Prolog knowledge base, converting involved execution times; *(ii)* the CLIMB business rule is mapped to a Prolog query, repetitions are re-written as conjunction of simple sentences, and dates and time displacements are converted; *(iii)*

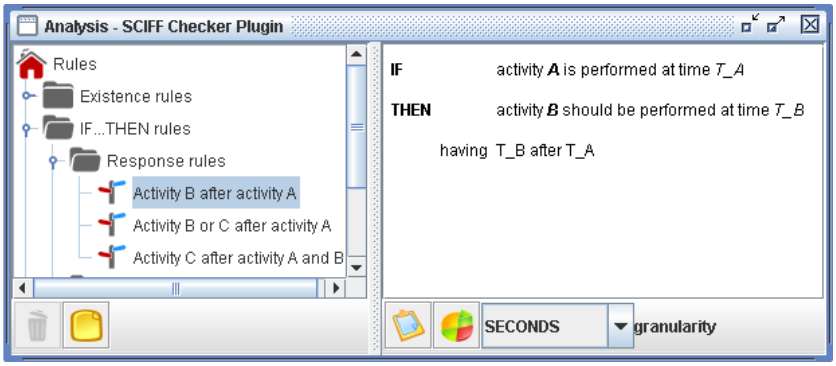

Fig. 4. A screenshot of the main $\mathcal{S}$CIFFChecker window.

finally a Prolog engine based on SWI[12] is exploited to perform the compliance checking.

All verification outcomes are collected and a summarizing pie chart is shown[13], together with the explicit list of compliant/non-compliant traces (Figure 5). At this point, the user can start a new classification by considering either the whole log or only the compliant/non-compliant execution traces. In this way, a conjunction of CLIMB rules can be verified by performing a sequence of tests, each dealing with only one rule, and by then selecting the compliant/non-compliant resulting set.

4.2 Applying $\mathcal{S}$CIFFChecker to the Think3 Case Study

$\mathcal{S}$CIFFChecker has been concretely applied to analyze execution traces of a Think3 client. We have first exploited the ProM Import tool[14] in order to convert the relevant information from the client database into an MXML format, by considering the project name as the case identifier. In particular, we extracted a portion of 9000 execution traces, ranging from 4 to 15 events. We then used, together with a Think3 business manager, the plug-in to express and test the business rules of interest described in Section 3.4. The overall average time for performing compliance checking have been assessed to be around 10-12 seconds. The verification outcomes have been finally analyzed by the business manager, who found them useful in order to obtain a clear overview about the overall process behavior. For example, considering rules (Br3) and (Br4), we discovered that, fortunately, only the 2% of execution traces involved more than two project revisions, and that in only the 3.5% cases the same person was responsible for both publishing and checking the project.

[12] `http://www.swi-prolog.com/`

[13] Thanks to the JFreeChart library, available from `http://www.jfree.org/jfreechart`.

[14] `http://promimport.sourceforge.net`

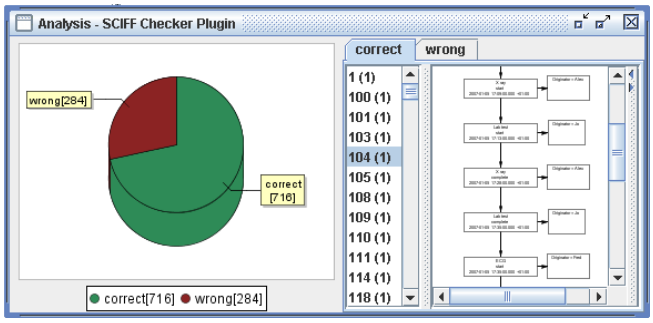

Fig. 5. Compliance chart produced by $\mathcal{S}$CIFFChecker at the end of verification.

The verification of rules like (Br2) was found interesting especially by varying the deadline involved. Indeed, the business manager wanted to detect projects taking too much time as well as projects released too soon, to point out both possible bottlenecks and potential inaccuracies. This kind of rule would be even more useful in a monitoring perspective: it would allow to identify at run-time non-compliant projects and to take as soon as possible specific countermeasures or further analysis.

5 Related Works

The closest work to the one here presented is the ProM *LTL-Checker* [3], that shares with our approach the motivation and purposes. While *LTL-Checker* exploits linear temporal logic for the formalization of properties, our approach belongs to the Logic Programming setting; CLIMB business rules are more expressive for what concerns the possibility of expressing rules triggering when a conjunction of events occur, and regarding the support of relative time constraints. Furthermore, through the mapping to $\mathcal{S}$CIFF, we can seamlessly check compliance at run-time, monitoring running process instances. A posteriori verification of execution traces has been deeply investigated also when the considered model is a procedural specification rather than a set of declarative business rules. For example, in [14] the authors tackle "conformance testing" between execution traces and a Petri Net-based process model, introducing metrics to measure how well a given execution fits into the model.

Our work, as part of the CLIMB framework, belongs to the recently investigated research stream aiming at exploiting declarative models and underlying formal methods for specifying and verifying IT-systems. The application of Logic Programming techniques to formalize and verify loosely-coupled processes and business rules has taken inspiration from ConDec [7].

An interesting research topic concerns the integration of $\mathcal{S}$CIFFChecker with the process mining algorithm described in [15], which follows the opposite direction: it aims at discovering a set of declarative business rules, specified in the $\mathcal{S}$CIFF language, starting from execution traces previously classified as correct

or wrong, s.t. the mined specification evaluates as compliant the correct subset and as non-compliant the wrong one. We could first mine a set of CLIMB business rules, use them to classify new traces, or exploit $\mathcal{S}$CIFFChecker to split a given MXML log into the wrong and correct subsets required as input for the mining algorithm. The latter approach could be exploited to discover a declarative model giving an *explanation* of the $\mathcal{S}$CIFFChecker classification.

6 Conclusions and Future Works

We have described a framework for checking the compliance of process execution traces to declarative reactive business rules, proposing a three-steps methodology for developing and applying rules. The approach has been tested on a real industrial case study, identifying what kind of rules the Think3 company would be able to check and showing how they can be easily expressed by customizing rule templates (re-usable patterns resembling ConDec constraints). In order to effectively use such rules for reasoning, we have sketched how they can be mapped to (extensions of) Logic Programming, namely $\mathcal{S}$CIFF and Prolog. The two mappings allow the monitoring of process executions at run-time or the checking of the compliance of already completed execution traces, helping a business manager in the assessment of business trends and providing decision making support. The latter approach has been implemented as a ProM plug-in that classifies MXML execution traces w.r.t. CLIMB business rules.

Many ongoing and future works concern both the framework itself and its ProM implementation. Regarding the underlying formal framework, we plan to investigate the relationships between Logic Programming and other formal languages, such as temporal logics, and to evaluate to what extent CLIMB rules can be expressed as queries in temporal databases [11]. We are applying our plug-in to different domains, such as a regional screening protocol, and, in cooperation with ENEA, in the context of a chemo-physical process of wastewater treatment plants. The tool will be extended, by introducing the possibility of dealing with case and event data. Finally, we will investigate how $\mathcal{S}$CIFFChecker and the mining algorithm presented in [15] can be jointly exploited to realize a declarative checking-mining cycle, discovering untrivial explanations for a given classification.

Acknowledgments This work has been partially supported by the FIRB project *TOCAI.IT: Tecnologie orientate alla conoscenza per aggregazioni di imprese in internet*.

References

1. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-based workflow models: Change made easy. In Meersman, R., Tari, Z., eds.: Proceedings of the OTM 2007 Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS. Volume 4803 of LNCS., Springer (2007) 77–94

2. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases **14**(1) (2003) 5–51
3. van der Aalst, W., de Beer, H., van Dongen, B.: Process Mining and Verification of Properties: An Approach based on Temporal Logic. In Meersman, R., Tari, Z., eds.: Proceedings of the OTM 2005 Confederated International Conferences CoopIS, DOA, and ODBASE. Volume 3760. (2005) 130–147
4. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. ACM Transactions on Computational Logic **9**(4) (10 2008) To appear.
5. Alberti, M., Gavanelli, M., Lamma, E., Chesani, F., Mello, P., Montali, M.: An abductive framework for a-priori verification of web services. In Bossi, A., Maher, M.J., eds.: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM (2006) 39–50
6. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Storari, S., Torroni, P.: Computational logic for run-time verification of web services choreographies: Exploiting the *socs-si* tool. In Bravetti, M., Núñez, M., Zavattaro, G., eds.: Proceedings of the Third International Workshop on Web Services and Formal Methods (WS-FM 06). Volume 4184 of LNCS., Springer (2006) 58–72
7. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In Eder, J., Dustdar, S., eds.: Proceedings of the BPM 2006 Workshops. Volume 4103 of LNCS., Springer (2006) 169–180
8. Chesani, F., Mello, P., Montali, M., Storari, S.: Towards a decserflow declarative semantics based on computational logic. Technical Report DEIS-LIA-07-002, DEIS, Bologna, Italy (2007)
9. van der Aalst, W.M.P., van Dongen, B.F., Günther, C.W., Mans, R.S., de Medeiros, A.A., Rozinat, A., Rubin, V., Song, M., Verbeek, H.M.W., Weijters, A.J.M.M.: ProM 4.0: Comprehensive Support for Real Process Analysis. In Kleijn, J., Yakovlev, A., eds.: Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007). Volume 4546. (2007) 484–494
10. van Dongen, B.F., van der Aalst, W.M.P.: A Meta Model for Process Mining Data. In Casto, J., Teniente, E., eds.: Proceedings of the CAiSE'05 Workshops (EMOI-INTEROP Workshop). Volume 2., FEUP, Porto, Portugal (2005) 309–320
11. Pissinou, N., Snodgrass, R.T., Elmasri, R., Mumick, I.S., Özsu, T., Pernici, B., Segev, A., Theodoulidis, B., Dayal, U.: Towards an infrastructure for temporal databases: report of an invitational arpa/nsf workshop. SIGMOD Rec. **23**(1) (1994) 35–51
12. Bailey, J., Bry, F., Eckert, M., Patranjan, P.L.: Flavours of xchange, a rule-based reactive language for the (semantic) web. In Adi, A., Stoutenburg, S., Tabet, S., eds.: RuleML. Volume 3791 of LNCS., Springer (2005) 187–192
13. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation **2**(6) (1993) 719–770
14. Rozinat, A., van der Aalst, W.: Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In Bussler et al., C., ed.: BPM 2005 Workshops (Workshop on Business Process Intelligence). Volume 3812. (2006) 163–176
15. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing declarative logic-based models from labeled traces. In Alonso, G., Dadam, P., Rosemann, M., eds.: Proceedings of the 5th International Conference on Business Process Management (BPM 2007). Volume 4714 of LNCS., Springer (2007) 344–359