Fundamenta Informaticae 104 (2010) 1–27
DOI 10.3233/FI-2010-330
IOS Press

A Logic-Based, Reactive Calculus of Events

Federico Chesani, Paola Mello, Marco Montali*, Paolo Torroni
DEIS, University of Bologna
V.le Risorgimento, 2
40136 Bologna, Italy
{*federico.chesani* | *paola.mello* | *marco.montali* | *paolo.torroni*}@*unibo.it*

Abstract. Since its introduction, the Event Calculus ($\mathcal{EC}$) has been recognized for being an excellent framework to reason about time and events, and it has been applied to a variety of domains. However, its formalization inside logic-based frameworks has been mainly based on backward, goal-oriented reasoning: given a narrative (also called execution trace) and a goal, logic-based formalizations of $\mathcal{EC}$ focus on proving the goal, i.e., establishing if a property (called *fluent*) holds.

These approaches are therefore unsuitable in dynamic environments, where the narrative typically evolves over time: indeed, each occurrence of a new event requires to restart the reasoning process from scratch. Ad-hoc, procedural methods and implementations have been then proposed to overcome this issue. However, they lack a strong formal basis and cannot guarantee formal properties. As a consequence, the applicability of $\mathcal{EC}$ has been somehow limited in large application domains such as run-time monitoring and event processing, which require at the same time reactivity features as well as formal properties to provide guarantees about the computed response.

We overcome the highlighted issues by proposing a Reactive and logic-based axiomatization of $\mathcal{EC}$, called $\mathcal{REC}$, on top of the SCIFF Abductive Logic Programming framework. Our solution exhibits the features of a reactive verification facility, while maintaining a solid formal background.

Keywords: Event Calculus, Reactive Reasoning, Monitoring, Computational Logic, Abductive Logic Programming.

*Address for correspondence: University of Bologna – V.le Risorgimento, 2 – 40136 Bologna, Italy

1. Introduction

More than 20 years ago, Kowalski and Sergot [21] introduced the Event Calculus ($\mathcal{EC}$) as a general framework to reason about time and events, overcoming limitations of other previous approaches, such as the situation calculus. The event calculus has many interesting features [21, 31]. Among them: an extremely simple and compact representation, symmetry of past and future, generality with respect to time orderings, executability and direct mapping with computational logic frameworks, modeling of concurrent events, immunity from the frame problem, and explicit treatment of time and events. It has therefore been applied to a variety of domains, such as cognitive robotics [30], planning [32], service interaction [23] and composition [28], active databases [16], workflow modeling [11] and legal reasoning [15].

Logic-based implementations of $\mathcal{EC}$ have been based on backward, goal-oriented reasoning, and extensively used in the past to carry out two main reasoning tasks: deductive *narrative verification*, to check whether a certain property (called fluent) holds given a narrative (set of events) [21], and abductive *planning*, to generate a possible narrative which satisfies some requirements [32]. For a survey on the different versions of $\mathcal{EC}$ and the corresponding reasoning tasks, see [31]. These tasks take place after or prior to execution, i.e., in presence of a complete narrative or with no narrative at all.

The existing $\mathcal{EC}$ logic-based approaches are instead unsuitable to perform reasoning during the execution, i.e., to deal with dynamic environments characterized by external narratives which typically evolve over time. However, there are several settings which require run-time reasoning, to the aim of monitoring their dynamics. For example, Business Process Management calls for a constant operational decision support during the execution of a business process, to provide auditing facilities checking whether the workers involved in the process executions comply with regulation and norms [1]. Similarly, in the Service Oriented Computing setting, several services, possibly implemented by different vendors, must properly interact to solve complex tasks; to ensure the correctness of the interactions, the exchanged messages must be dynamically verified against a global contract (called choreography), which specifies the agreed mutual rules of engagement.

In all these situations, the execution traces cannot be controlled nor influenced: they are generated by the autonomous interacting entities. Monitoring is therefore of utmost importance, to provide a constantly updated feedback about the reached state of affairs, and to promptly detect undesired or exceptional situations. Monitoring requires the capability of *reacting* to the occurrence of new events, but while backward, goal-oriented approaches enable a straightforward update of the theory each time an event occurs (it suffices to add the new event occurrence to the knowledge base), they incur a substantial increase of the total time, since reasoning has to be restarted from scratch. Furthermore, monitoring calls for the ability of carrying out "open" reasoning, i.e., of reasoning upon the partial and incomplete information acquired during the execution, extending the inferred results as new events occur.

Reactive $\mathcal{EC}$ implementations based on *ad-hoc*, procedural methods and implementations have been then proposed to overcome this issue. For example, in [23] Mahbub and Spanoudakis present a framework for monitoring the compliance of a service composition w.r.t. behavioral properties. $\mathcal{EC}$ is exploited to monitor the effective behavior of interacting services and report different kinds of violation. Reasoning is implemented by means of an ad-hoc event processing algorithm, which fetches occurred events updating the status of the involved fluents. A JAVA-based $\mathcal{EC}$ engine is instead the reasoning core of the Event Calculus State Tracking Architecture [15], proposed by Farrel et al. to track at run-time the normative state of contracts. The contract's constraints are specified by means of $\mathcal{EC}$ axioms, encoded in a dialect of XML, while a JAVA GUI is exploited to deliver occurring events to the underlying engine and to provide a feedback concerning the computed state of affairs.

Even if all these ad-hoc, procedural approaches exhibit reactivity, they all suffer from the same drawback: they lack a strong formal basis. In particular, it is very difficult to understand and prove their formal properties, and it is therefore impossible to provide guarantees about the computed responses. As a consequence, the applicability of $\mathcal{EC}$ has been somehow limited in large application domains such as run-time monitoring and event processing, which require at the same time reactivity features as well as formal properties to give the aforementioned guarantees.

Following a different line of research, Kowalski and Sadri [20] proposed to use Abductive Logic Programming (ALP) as a way to reconcile backward with forward reasoning inside an intelligent agent architecture. However, besides planning [14], ALP has not been used in combination with the $\mathcal{EC}$, and its forward reactive features have therefore remained unexplored in this context. Nor are we aware of other logical frameworks that implement the $\mathcal{EC}$ in a reactive way: a logic-based implementation of $\mathcal{EC}$ that dynamically reacts to happening events is missing.

Building on Kowalski et al.'s work, in this work we equip the $\mathcal{EC}$ framework with the reactive features of a powerful, general purpose ALP language and proof-procedure named SCIFF. We obtain a version of the calculus, which we call Reactive Event Calculus ($\mathcal{REC}$), which exhibits all the features of a reactive verification facility, while maintaining a solid formal background.

$\mathcal{REC}$ draws also inspiration from the work of Chittaro and Montanari [9], who were the first authors to recognize the importance of proposing forms of $\mathcal{EC}$ which shift the focus from the query perspective to the capability of updating the computed answers when new event occurrences are detected.

In particular, Chittaro and Montanari proposed a mechanism, called *Cached Event Calculus* ($\mathcal{CEC}$), to cache the outcome of the inference process every time the knowledge base is updated by a new event. The difference between $\mathcal{CEC}$ and the classical axiomatization of $\mathcal{EC}$ is twofold:

- While the classical $\mathcal{EC}$ reasons upon the time intervals in which fluents are terminated and initiated, $\mathcal{CEC}$ reasons upon the maximal time intervals in which fluents hold; these intervals are called Maximal Validity Intervals (MVIs).

- The computed MVIs are cached by exploiting assert and retract predicates; this choice has a positive impact on the performance of the calculus, but undermines the declarative semantics of $\mathcal{CEC}$.

$\mathcal{REC}$ relies on the same idea, but its formalization is fully declarative: the caching mechanism is implicitly realized by combining the notion of abduction and the reactivity of the SCIFF proof procedure, which natively supports the dynamic acquisition and treatment of event occurrences. After having described how $\mathcal{REC}$ can be axiomatized on top of SCIFF, we investigate it from a theoretical point of view, discussing its formal properties and the use of negation, as well as from a practical perspective, by means of a representative example dealing with quantitative temporal aspects, violations and compensations.

The paper is organized as follows. After having introduced $\mathcal{EC}$ and the SCIFF framework in Sections 2 and 3, we show how they can be combined to obtain $\mathcal{REC}$ (Section 4). In Section 5 we describe a simple yet challenging case study, where $\mathcal{REC}$ is used to monitor the flow of employees at the entrance of a company. Section 6 is devoted to discuss how standard treatment of negation must be properly adapted in $\mathcal{REC}$, to reflect the intended semantics. Section 7 discusses the formal properties of $\mathcal{REC}$. Discussion and conclusion follow.

2. The Event Calculus

The Event Calculus ($\mathcal{EC}$) was introduced as a logic programming framework for representing and reasoning about events and their effects [21].

Basic concepts are that of *event*, happening at a point in time, and property (called *fluent*), holding during time intervals. Fluents are initiated/terminated by events. Given an event narrative (a set of events), the $\mathcal{EC}$ theory and domain-specific axioms together (called $\mathcal{EC}$ axioms) define which fluents hold at each time. There are many different formulations of these axioms [10]. One possibility is given by the following axioms, where P stands for *Fluent*, E for *Event*, T represents time instants and $\neg$ is interpreted as Negation As Failure [12]:

$$\begin{aligned} \mathit{holds\_at}(P,T) \leftarrow\ & \mathit{initiates}(E,P,T_{Start}) \\ & \wedge T_{Start} < T \wedge \neg \mathit{clipped}(T_{Start},P,T). \end{aligned} \tag{ec_1}$$

$$\begin{aligned} \mathit{clipped}(T_1,P,T_3) \leftarrow\ & \mathit{terminates}(E,P,T_2) \\ & \wedge T_1 < T_2 \wedge T_2 < T_3. \end{aligned} \tag{ec_2}$$

$$\begin{aligned} \mathit{initiates}(E,P,T) \leftarrow\ & \mathit{happens\_at}(E,T) \wedge [\neg]\mathit{holds\_at}(P_1,T) \\ & \wedge \ldots \wedge [\neg]\mathit{holds\_at}(P_M,T). \end{aligned} \tag{ec_3}$$

$$\begin{aligned} \mathit{terminates}(E,P,T) \leftarrow\ & \mathit{happens\_at}(E,T) \wedge [\neg]\mathit{holds\_at}(P_1,T) \\ & \wedge \ldots \wedge [\neg]\mathit{holds\_at}(P_N,T). \end{aligned} \tag{ec_4}$$

Axioms (ec_1) and (ec_2) are domain-independent axioms which formalize the relationship between events and fluents. In particular, Axiom (ec_1) states that a fluent holds at a certain time iff it has been previously initiated by some event, and it has not been clipped in between. Axiom (ec_2) formalizes the meaning of “clipped”: a fluent is clipped inside a given time interval if an event has terminated the fluent in that interval[1].

Axioms (ec_3) and (ec_4) are instead schemas for defining the domain-specific axioms: a certain fluent P is initiated/terminated at a time instant T if an event E happened at the same time, and if some other fluents P_i (do not) hold at that time; testing if these fluents hold or not enables the possibility of defining *contexts*, i.e. of expressing that the relationship between an event and a fluent depends on particular states of affairs. Furthermore, note that $\mathcal{EC}$ adopts a time structure with a minimal element. At the minimal time, the system is in its initial state, which can be characterized by describing the set of fluents holding at the beginning of the execution. This is done by means of $\mathit{initially}$ predicates.

As pointed out in the introduction, the proposed formulation is not suited to deal with dynamic environments, where the narrative of the system evolves in time: every time a new event occurs, reasoning has to be restarted from scratch. To overcome this issue, Chittaro and Montanari proposed a different formulation, which caches the outcome of the inference process every time the knowledge base is updated by a new event. The *Cached Event Calculus* ($\mathcal{CEC}$) [9] computes and stores fluents' *maximum validity intervals* (MVIs), which are the maximum time intervals in which fluents hold, according to the known events. The set of cached validity intervals is then extended/revised as new events occur or get to be

[1]Dual axioms and predicates can be added to define when fluents *do/do not* hold [31]: e.g., an axiom can be added to define *declipped*/3 (an event has made a certain fluent holding).

$happens\_at(Ev, T)$	Event Ev happens at time T
$mvi(F, T_i, T_f)$	Fluent F begins to hold from time T_i and persists to hold until time T_f: $(T_i, T_f]$ is a maximum validity interval for F
$holds\_at(F, T)$	Fluent F holds at time T
$initially(F)$	Fluent F holds from the initial time
$initiates(Ev, F, T)$	Event Ev initiates fluent F at time T
$terminates(Ev, F, T)$	Event Ev terminates fluent F at time T

Table 1. The $\mathcal{EC}$ ontology, extended with the concept of maximum validity interval.

known. Unfortunately, this is done by exploiting assert and retract predicates, which do not have a corresponding declarative semantics.

The complete ontology of $\mathcal{CEC}$ is summarized in Table 1.

3. The SCIFF Framework

The $\mathcal{EC}$ can be elegantly formalized in logic programming, but as we said above, that would be suitable for top-down, "backward" computation. Runtime monitoring, intended as checking the behavior of interacting entities during the execution, requires reactive $\mathcal{EC}$ frameworks, able to deal with a dynamically growing narrative, reacting to the occurrence of new events and inferring new knowledge. For this reason, we resort to a framework which reconciles backward with forward reasoning: the SCIFF language and proof-procedure [2]. In this Section, we describe the main features of the framework, discussing the language as well as the declarative and operational semantics; Section 4 will then show how a reactive form of $\mathcal{EC}$ can be suitably formalized on top of this framework.

3.1. The SCIFF Language

SCIFF is an extension of Fung and Kowalski's IFF proof-procedure for abductive logic programming [17]. It has two primitive notions: events (represented as **H** atoms) and expectations (modeled by **E/EN** atoms). $\mathbf{H}(Ev, T)$ means that an event Ev occurs at time T, and it is a ground atom. $\mathbf{H}(Ev, T)$ exactly corresponds to the $happens\_at$ predicate shown in Table 1. Instead $\mathbf{E}(Ev, T)$ and $\mathbf{EN}(Ev, T)$ can contain variables with domains and CLP constraints [18], and they denote in the first case that an event unifying with Ev is expected to occur at some time in the range of T (T existentially quantified), and in the second case that all events unifying with Ev are expected not to occur, at all times belonging to the domain of T (i.e., T is considered as universally quantified over its domain). SCIFF accommodates existential and universal variable quantification and quantifier restriction, CLP constraints, dynamic update of event narrative and it has a built-in runtime protocol verification procedure.

A SCIFF specification is composed of a knowledge base $\mathcal{KB}$, a set of ICs (integrity constraints) $\mathcal{IC}$, and a goal. $\mathcal{KB}$ consists of backward rules $head \leftarrow body$, whereas the ICs in $\mathcal{IC}$ are forward implications $body \rightarrow head$. As we will see, the reactive axiomatization of $\mathcal{EC}$ will exploit both kinds of knowledge. ICs are interpreted in a reactive manner; the intuition is that when the body of an IC becomes true (i.e., the involved events occur), then the rule fires, and the expectations in the head are generated by abduction. For example, $\mathbf{H}(a, T) \rightarrow \mathbf{EN}(b, T')$ defines a relation between events a and b, saying that if a occurs at time T, b should not occur at any time; $\mathbf{H}(a, T) \rightarrow \mathbf{E}(b, T') \wedge T' \leq T + 300$ says that if a occurs, then an event b should occur no later than 300 time units after a.

3.2. Declarative Semantics

We now briefly introduce the declarative semantics of SCIFF, discussing the two cases in which the execution trace characterizing an instance of the system is complete or partial. For a comprehensive description of SCIFF the reader can refer to [2].

SCIFF's declarative semantics is given by interpreting the SCIFF specifications as Abductive Logic Programs (ALP). In particular, a SCIFF specification is an Abductive Logic Program $\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ where:

- $\mathcal{KB}$ is the knowledge base, composed by a set of clauses.
- $\mathcal{A} \supseteq \{\mathbf{E}/2, \mathbf{EN}/2, \mathbf{H}/2\}$ is the set of abducible predicates, i.e., predicates without definition that are hypothesized/generated during the reasoning process – such a set always contains expectations and happened events, but it could also be extended with other domain-specific abducibles.
- $\mathcal{IC}$ is the set of integrity constraints. Each integrity constraint is used to generate expectations and/or (internal) events when a certain situation making its body true occurs.

The declarative semantics of SCIFF starts from the semantics of ALP introducing the notion of *compliance* of an execution trace with the generated expectations. Therefore, it targets an *instance* of the system, intended as a specific execution trace related to a specification. An execution trace collects the set of events occurred so far (i.e., the partial narrative).

Definition 3.1. (Execution trace)
A SCIFF execution trace $\mathcal{T}$ is a set of (ground) atoms of the form $\mathbf{H}(e, t)$.

Definition 3.2. (Instance)
Given a SCIFF specification $\mathcal{S}$ and an execution trace $\mathcal{T}$, $\mathcal{S}_{\mathcal{T}} = \langle \mathcal{S}, \mathcal{T} \rangle$ is the $\mathcal{T}$-*instance* of $\mathcal{S}$.

Roughly speaking, the declarative semantics states that each positive expectation must be fulfilled by a corresponding event occurrence, while negative expectations must not have any corresponding occurrence in the execution trace. However, two different cases must be separately considered, depending on whether the execution trace characterizing the instance of the system is partial or complete. In the first case, called "open execution trace", further events may still occur to fulfill a positive expectation or violate a negative expectation, and therefore the knowledge about the narrative is incomplete; conversely, in the latter case, called "closed execution trace", no further event will occur to fulfill a positive expectation or violate a negative one. In order to accommodate the open case, the declarative semantics relies on a three-valued logic, i.e., $\models$ will be always interpreted in a three-valued setting, as defined in [22]. In the following, the term "open (closed resp.) instance" will denote an instance whose execution trace is open (closed resp.). Note that, in the general case, open instances can evolve by means of events occurring in the future but also referring to the past. For example, inside an active database a new event occurrence referring to the past could be acquired in the future.

Let us now start with the concept of abductive explanation and goal achievement, used to respectively characterize the admissible abducible sets related to a given instance and to state whether a goal can be achieved or not.

In the open situation, the execution trace is partial and incomplete, and can be possibly extended as new event occurrences get to be known. Hence, abductive explanations are defined without applying the completion to the (partial) execution trace, which reflects the fact that further events may occur to extend it. In this respect, it could be the case that even if a desired goal has not yet been achieved, it will be achieved in the future; therefore, the open situation is associated to the concept of goal "achievability".

Definition 3.3. (open abductive explanation)
Given an open instance $\mathcal{S}_{\mathcal{T}} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle_{\mathcal{T}}$, the abducible set $\Delta^o \subseteq \mathcal{A}$ is an *open abductive explanation* for $\mathcal{S}_{\mathcal{T}}$ iff

$$\mathit{Comp}\,(\mathcal{KB} \cup \Delta^o) \cup \mathcal{T} \cup CET \cup T_{\mathcal{X}} \models \mathcal{IC}$$

where *Comp* is the completion of a theory [22], CET stands for Clark Equational Theory [12] and $T_{\mathcal{X}}$ is the constraint theory [18] (parametrized by the domain $\mathcal{X}$).

The $\mathcal{X}$ parameter allows us to keep the semantics of SCIFF general. In particular, it can be seamlessly grounded to different domains, such as finite domains [13] or reals [19].

Definition 3.4. (goal achievability)
Let $\mathcal{S}_{\mathcal{T}} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle_{\mathcal{T}}$ be an open instance and γ a conjunction of literals representing a goal. γ is *achievable* by $\mathcal{S}_{\mathcal{T}}$ iff there exists an open abductive explanation Δ^o for $\mathcal{S}_{\mathcal{T}}$ s.t.:

$$\mathit{Comp}\,(\mathcal{KB} \cup \Delta^o) \cup \mathcal{T} \cup CET \cup T_{\mathcal{X}} \models \gamma$$

In the closed situation, the execution trace is complete, and therefore it is subject to the three-valued completion as well. Since further events cannot happen to change the state of affairs, there is complete information to evaluate whether a desired goal is entailed or not, and therefore the closed situation is associated to the concept of goal achievement.

Definition 3.5. (closed abductive explanation)
Given a closed instance $\mathcal{S}_{\overline{\mathcal{T}}} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle_{\overline{\mathcal{T}}}$, the abducible set $\Delta^c \subseteq \mathcal{A}$ is a *closed abductive explanation* for $\mathcal{S}_{\overline{\mathcal{T}}}$ iff

$$\mathit{Comp}\left(\mathcal{KB} \cup \Delta^c \cup \overline{\mathcal{T}}\right) \cup CET \cup T_{\mathcal{X}} \models \mathcal{IC}$$

Definition 3.6. (goal achievement)
Given a closed instance $\mathcal{S}_{\overline{\mathcal{T}}} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle_{\overline{\mathcal{T}}}$, a goal γ is *achieved* by $\mathcal{S}_{\overline{\mathcal{T}}}$ iff there exists a closed abductive explanation Δ^c for $\mathcal{S}_{\overline{\mathcal{T}}}$ s.t.:

$$\mathit{Comp}\left(\mathcal{KB} \cup \Delta^o \cup \overline{\mathcal{T}}\right) \cup CET \cup T_{\mathcal{X}} \models \gamma$$

We now focus on the semantics of expectations. First of all, in both the open and the closed situation an abductive explanation must be consistent w.r.t. the contained expectations, i.e., it must not state that an event is expected to happen and not to happen at the same time.

Definition 3.7. (E-consistency)
An abductive explanation Δ is **E**-consistent iff for each ground event e and ground time t:

$$\{\mathbf{E}(e,t), \mathbf{EN}(e,t)\} \not\subseteq \Delta$$

Finally, expectations lead to the central notions of fulfillment and violation, which relate them with the presence/absence of a corresponding occurred event, constituting the basis for defining the notion of *compliance*. As with the other definitions, fulfillment and violation have different meanings depending on whether we are in the open or in the closed situation. During the execution, i.e., in the open phase, the information about the execution trace is incomplete, and therefore it is not always possible to evaluate whether a certain expectation is violated or fulfilled. If this is the case, we say that the expectation is *pending*. It is worth noting that the three possible expectations' "status" exactly correspond to the three possible truth values in three-valued logics.

Definition 3.8. (Fulfilled, violated and pending expectations)
Let us consider a ground (positive) expectation $\mathbf{E}(e_1, t_1)$ and a (ground) negative expectation $\mathbf{EN}(e_2, t_2)$.

Given an open execution trace $\mathcal{T}$:

- $\mathbf{E}(e_1, t_1)$ is $\mathcal{T}$-*fulfilled* iff $\mathbf{H}(e_1, t_1) \in \mathcal{T}$, *pending* otherwise;
- $\mathbf{EN}(e_2, t_2)$ is $\mathcal{T}$-*violated* iff $\mathbf{H}(e_2, t_2) \in \mathcal{T}$, *pending* otherwise.

Given a closed execution trace $\overline{\mathcal{T}}$:

- $\mathbf{E}(e_1, t_1)$ is $\overline{\mathcal{T}}$-*fulfilled* iff $\mathbf{H}(e_1, t_1) \in \overline{\mathcal{T}}$, $\overline{\mathcal{T}}$-*violated* otherwise;
- $\mathbf{EN}(e_2, t_2)$ is $\overline{\mathcal{T}}$-*violated* iff $\mathbf{H}(e_2, t_2) \in \overline{\mathcal{T}}$, $\overline{\mathcal{T}}$-*fulfilled* otherwise.

We have now all the machinery to capture the notion of *compliance*.

Definition 3.9. (compliance)
Let us consider a SCIFF specification $\mathcal{S}$ and a goal γ. An open execution trace $\mathcal{T}$ is *compliant* with $\mathcal{S}$ and γ iff there exists an $\mathbf{E}$-consistent open abductive explanation Δ^o s.t. no expectation in Δ^o is $\mathcal{T}$-violated and γ is achievable. A closed execution trace $\overline{\mathcal{T}}$ is *compliant* with $\mathcal{S}$ and γ iff there exists an $\mathbf{E}$-consistent closed abductive explanation Δ^c s.t. all expectations in Δ^c are $\overline{\mathcal{T}}$-fulfilled and γ is achieved.

3.3. Operational Semantics

The SCIFF proof procedure is an abductive proof procedure able to verify compliance of execution traces with a SCIFF specification. In this respect, it represents the operational counterpart of the declarative semantics described in the previous section, and of the notion of compliance in particular. Starting from an initial (possibly empty) execution trace $\mathcal{T}_i$ and from a SCIFF specification $\mathcal{S}$, the proof procedure is able to dynamically fetch new events and compute abductive explanations for the evolving course of interaction, checking if the expectations contained in such abductive explanations are $\mathbf{E}$-consistent and fulfilled by the occurred events.

Being the language and declarative semantics of the SCIFF framework closely related to the IFF abductive framework [17], the SCIFF proof procedure has taken inspiration from the IFF proof procedure. The IFF proof procedure is one of the most well-known proof procedures that combine reasoning with defined predicates together with reasoning with abducible predicates. While IFF is a general abductive proof procedure, SCIFF is a general abductive proof procedure able to solve the specific problem of compliance verification. In particular, SCIFF is a substantial extension of the IFF and adds, as described in [2], the following features:

- SCIFF supports the dynamic acquisition of events;
- SCIFF supports universally quantified variables in abducibles and quantifier restrictions, in order to properly reason with negative expectations and **E**-consistency;
- SCIFF supports quantifier restrictions [4] and CLP constraints on variables;
- SCIFF supports the concepts of fulfillment and violation, executing a "hypotheses confirmation" step in which the matching between expectations and the execution trace is evaluated.

In particular, as in the case of IFF, SCIFF is based on a rewriting system which transforms one node into a successor node or a set of successor nodes by applying transitions, until quiescence is reached. During the execution, quiescence means that the proof procedure has completed the reasoning phase attesting that the current partial execution trace is compliant with the given specification; the proof then waits for the acquisition of further events. In the closed situation, at quiescence SCIFF provides the definitive evaluation about expectations' status.

Transitions manipulate the integrity constraints of the specification by taking into account how they are affected by the acquired occurred events. The transitions are briefly listed in the following; their complete description is provided in [2].

Unfolding substitutes an atom with its definitions in $\mathcal{KB}$;

Propagation given an implication $(\mathbf{a}(X) \wedge R) \rightarrow Head$ and an abduced literal $\mathbf{a}(Y)$, generates the implication $(X = Y \wedge R) \rightarrow Head$;

Case Analysis Given an implication $(c(X) \wedge R) \rightarrow Head$ in which c is a constraint (possibly the equality constraint '$=$'), generates two children: $c(X) \wedge (R \rightarrow Head)$ and $\neg c(X)$;

Splitting distributes conjunctions and disjunctions;

Logical Equivalences performs usual replacements: $true \rightarrow A$ with A, etc.;

Constraint Solving posts constraints to the constraint solver of choice (currently, CLP($\mathcal{R}$) and CLP(fd) are supported);

Fulfillment declares that an expectation is fulfilled, according to Definition 3.8;

Violation declares a violated expectation: symmetrical to fulfillment (see Definition 3.8);

Consistency checks the **E**-consistency of the current set of expectations;

Happening fetches a new occurred event from an external queue and inserts it into the execution trace;

Closure is triggered when the execution reaches its end, and alerts the proof procedure that the evaluation of fulfillment and violation must "switch" to the closed situation.

In [2], it has been shown that the SCIFF proof procedure is sound and complete with respect to the declarative semantics highlighted in Sect. 3.2. Furthermore, it has been demonstrated that it also terminates when the specification under study obeys to some syntactic restrictions, which is discussed in Sect. 7.1.

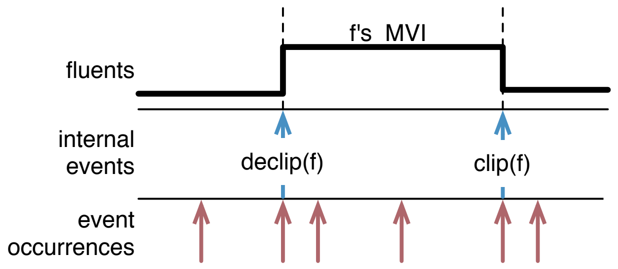

Figure 1. Internal and external events in $\mathcal{REC}$.

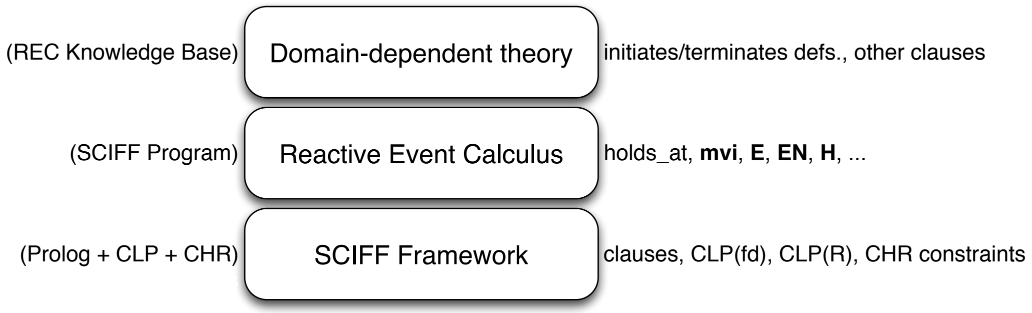
(REC Knowledge Base)
Domain-dependent theory
initiates/terminates defs., other clauses
(SCIFF Program)
Reactive Event Calculus
holds_at, **mvi**, **E**, **EN**, **H**, ...
(Prolog + CLP + CHR)
SCIFF Framework
clauses, CLP(fd), CLP(R), CHR constraints

Figure 2. Overview of $\mathcal{REC}$.

4. The Reactive Event Calculus

The SCIFF axiomatization of $\mathcal{EC}$ that follows draws inspiration from Chittaro and Montanari's $\mathcal{CEC}$ and their idea of MVIs. Events and fluents are terms and times are integer or real (CLP) variables, 0 being the initial time. $\mathcal{REC}$ uses the abduction mechanism to generate MVIs and define their persistence. It has a fully declarative axiomatization (Axioms ax_1 through ax_7): no operational specifications are needed.

A $\mathcal{REC}$ specification is represented as a particular SCIFF specification, constituted by two parts:

- A general part, composed by a mixture of forward and backward rules (i.e., integrity constraints and clauses in the knowledge base), which formalizes the $\mathcal{EC}$ ontology sketched in Table 1, e.g., defining the semantics of MVIs and the relationship between the occurrence of events and MVIs. It uses two special internal events (denoted by the reserved *clip*/*declip* words, differently from generic external events, that are "encapsulated" into the reserved term $event$) to model that a fluent is terminated/initiated, respectively (see Figure 1).

- A domain-dependent part, which binds domain-related events with the initiation and termination of domain-related fluents. This part is composed by clauses that are added to the general knowledge base, completing it with the domain-specific knowledge.

The overall picture is depicted in Figure 2. It shows that a $\mathcal{REC}$ specification is obtained by combining the general axiomatization of the calculus and a specific theory reflecting the domain under study, while SCIFF is used as the underlying reasoning machinery.

4.1. Axiomatization of the Calculus

We now give a more precise definition of a $\mathcal{REC}$ specification, and then focus our attention to the general axioms.

Definition 4.1. ($\mathcal{REC}$ specification)
$\mathcal{REC}$ is a SCIFF specification

$$\langle \mathcal{KB}_{\mathcal{REC}}, \{\mathbf{E}/2, \mathbf{EN}/2, \mathbf{H}/2, \mathbf{mvi}/2\}, \mathcal{IC}_{\mathcal{REC}} \rangle$$

where:

- $\mathcal{KB}_{\mathcal{REC}} = \{ax_1, ax_7\} \cup \mathcal{KB}_{domain}$, being $\mathcal{KB}_{domain}$ the domain-specific set of clauses, following the schemas of Axioms ec_3 and ec_4 in Section 2;
- $\mathcal{IC}_{\mathcal{REC}} = \{ax_2, ax_3, ax_4, ax_5, ax_6\}$.

Axiom ax_1 is a backward rule (clause), as well as Axiom ax_7, whereas Axiom ax_2 through Axiom ax_6 are forward implications (ICs). Such a mixture of backward and forward inference rules is enabled by ALP [20] and it represents the backbone of $\mathcal{REC}$'s reactive behaviour.

Axiom 1. (Holding of fluent)
A fluent F holds at time T if an MVI containing T has been abduced for F. Note that a fluent does not hold at the time it is declipped, but holds at the time it is clipped, i.e., MVIs are left-open and right-closed.

$$holds\_at(F, T) \leftarrow \mathbf{mvi}(F, [T_s, T_e]) \wedge T > T_s \wedge T \leq T_e. \tag{ax_1}$$

Axiom 2. (MVI semantics)
If $(T_s, T_e]$ is an MVI for F, then F must be declipped at time T_s and clipped at time T_e, and no further declipping/clipping must occur in between.

$$\begin{aligned} &\mathbf{mvi}(F, [T_s, T_e]) \\ &\quad \rightarrow \mathbf{E}(declip(F), T_s) \wedge \mathbf{E}(clip(F), T_e) \\ &\qquad \wedge \mathbf{EN}(declip(F), T_d) \wedge T_d > T_s \wedge T_d \leq T_e \\ &\qquad \wedge \mathbf{EN}(clip(F), T_c) \wedge T_c \geq T_s \wedge T_c < T_e. \end{aligned} \tag{ax_2}$$

Axiom 3. (Initial status of fluents)
If a fluent initially holds, a corresponding declipping event is generated at time 0.

$$initially(F) \rightarrow \mathbf{H}(declip(F), 0). \tag{ax_3}$$

Axiom 4. (Fluents initiation)
If an event Ev occurs at time T which initiates fluent F, either F already holds or it is declipped.

$$\begin{aligned} &\mathbf{H}(event(Ev), T) \wedge initiates(Ev, F, T) \\ &\rightarrow \mathbf{H}(declip(F), T) \\ &\quad \vee \mathbf{E}(declip(F), T_d) \wedge T_d < T \\ &\qquad \wedge \mathbf{EN}(clip(F), T_c) \wedge T_c > T_d \wedge T_c < T. \end{aligned} \tag{ax_4}$$

	$\mathcal{CEC}$	$\mathcal{REC}$
New event occurrence	Assertion of the event occurrence, and propagation of its effects by invoking specific clauses	Acquisition of the event occurrence and triggering of a set of transitions, starting from *happening*
Generation of a new MVI	Assertion of the MVI	Abduction of the MVI
Revision of the generated MVIs	Retraction and proper re-assertion of the affected MVIs	Backtracking

Table 2. Comparison between $\mathcal{CEC}$ and $\mathcal{REC}$.

(ax_4) does not use the $holds\_at$ predicate to test if F holds at time T, but employs instead a mixture of positive and negative expectations to test this.

Axiom 5. (Impact of initiation)
The happening of a *declip(F)* event causes the beginning of a new MVI for F, which starts from the time at which the event happens and will terminate sometime in the future.

$$\mathbf{H}(declip(F), T_s) \rightarrow \mathbf{mvi}(F, [T_s, T_e]) \wedge T_e > T_s. \qquad (ax_5)$$

Axiom 6. (Fluents termination)
If an event Ev occurs which terminates a fluent F, F is clipped.

$$\begin{aligned}&\mathbf{H}(event(Ev), T)\\ &\wedge terminates(Ev, F, T) \rightarrow \mathbf{H}(clip(F), T).\end{aligned} \qquad (ax_6)$$

Axiom 7. (Final clipping of fluents)
All fluents are terminated by the special *complete* event.

$$terminates(complete, F, \_). \qquad (ax_7)$$

(ax_7) models the "impact" of the *complete* event (used to indicate that the execution is terminated) on fluents: at the end of the execution, all fluents are terminated, i.e., the time at which the complete event happens fixes the upper limit of the final MVIs. This idea exactly reflects the switch from the open to the closed situation, as described in Section 3: when the special *complete* event occurs, transition *closure* is applied by the proof procedure.

As we have previously sketched, $\mathcal{REC}$ draws inspiration from Chittaro and Montanari's $\mathcal{CEC}$. The main difference is that while reasoning in $\mathcal{CEC}$ is provided by a Prolog engine managing the *assertion* and *retraction* of MVIs, $\mathcal{REC}$ relies on the SCIFF proof procedure, which natively supports reactive reasoning. While $\mathcal{CEC}$ looses its declarative semantics due to the use of *assert* and *retract* predicates, $\mathcal{REC}$ is fully declarative. Table 2 briefly summarizes the comparison between the two approaches[2].

[2]"Revision of the generated" MVIs is needed when event occurrences are not ordered: the acquisition of a new event occurrence referring to the past could lead to modify the inferred MVIs.

4.2. Compliance in $\mathcal{REC}$ and Pure Monitoring

Let us now focus on the SCIFF specification formalizing $\mathcal{REC}$. We suppose that $\mathcal{REC}$ monitors the execution by dynamically acquiring the sequence of occurred events, which are collected in a partial open execution trace. When the execution terminates, the special *complete* event is used to alert $\mathcal{REC}$ that the execution trace must be now considered as closed. Obviously, *complete* is associated to an execution time which is greater than all the times associated to the other event occurrences.

As summarized in Figure 1, in $\mathcal{REC}$ abductive explanations contain MVIs and expectations concerning the internal clip/declip events:

- Positive expectations are employed to delimit MVIs - each MVI starts with a corresponding declip event and terminates with a corresponding clip one.
- Negative expectations are used to state that each MVI is an uninterruptible validity interval for its fluent, i.e., inside an MVI it is expected that no declip nor clip events for the fluent will happen. These expectations are not used to assess compliance, but only to properly delimit MVIs, by imposing the selection of the nearest declipping and clipping event occurrences as matching candidates against the positive expectations.

If we do not consider a goal, i.e., the goal is $true$, $\mathcal{REC}$ is used as a "pure" monitor: it will dynamically collect the occurring events tracking and reporting the fluents' evolution, but it will always evaluate the execution trace as compliant. In fact,

- declipping expectations are automatically fulfilled: fluents are generated only in response of an external event occurrence which leads to the generation of a corresponding (implicit) declipping event occurrence;
- clipping expectations are fulfilled by the special *complete* event at last.

5. $\mathcal{REC}$ illustrated: a personnel monitoring facility

The following real-world case study has been proposed by a local medium-sized enterprise. A company wants to monitor its personnel's time-sheets. Each employee punches the clock when entering or leaving the office. The system recognizes two events:

- $check\_in(E)$: employee E has checked in;
- $check\_out(E)$: employee E has checked out.

The following requirements on employee behavior need a monitoring facility:

(R1) after check in, an employee must check out within 8 hours;

(R2) as soon as a deadline expiration is detected, a dedicated alarm fires at an operator's desk. It reports the employee ID, and an indication of the time interval elapsed between deadline expiration and its detection. The alarm is turned off when the operator decides to handle it.

We assume that the following actions are available to the operator:

- $handle(E)$ states that the operator wants to handle the situation concerning the employee identified by E;
- tic is used to take a snapshot of the current situation of the system, by updating the current time.

We capture requirements (R1) and (R2), using three fluents:

- $in(E)$: E is currently in;
- $should\_leave(E, T_d)$: E is expected to leave her office by T_d;
- $alarm(E, delay(D))$: E has not left the office in time – D represents the difference between the time a deadline expiration is detected and the deadline expiration time itself. In the general case, fluent $alarm(E, A)$ states that an alarm A is currently active for E.

It is possible to model such requirements declaratively using $initiates$ and $terminates$ predicate definitions. We assume hour time granularity.

Let us first focus on the $in(E)$ fluent. E is "in" as of the time she checks in. She ceases to be "in" as of the time she checks out:

$$initiates(check\_in(E), in(E), T). \tag{1}$$

$$terminates(check\_out(E), in(E), T). \tag{2}$$

When E checks in at T_c, a $should\_leave$ fluent is activated, expressing that E is expected to leave the office by $T_c + 8$:

$$initiates(check\_in(E), should\_leave(E, T_d), T_c) \leftarrow T_d \text{ is } T_c + 8. \tag{3}$$

Note that T_c is ground at *body* evaluation time, due to ax_4.

Such a fluent can be terminated in two ways: either E correctly checks out within the 8-hour deadline, or the deadline expires. In the latter case, termination is imposed at the next tic action.

$$\begin{aligned} terminates(check\_out(E), &should\_leave(E, T_d), T_c) \leftarrow \\ &holds\_at(should\_leave(E, T_d), T_c) \wedge T_c \leq T_d. \end{aligned} \tag{4}$$

$$\begin{aligned} terminates(tic, &should\_leave(E, T_d), T) \leftarrow \\ &holds\_at(should\_leave(E, T_d), T) \wedge T > T_d. \end{aligned} \tag{5}$$

The same tic action also causes an alarm to go off:

$$\begin{aligned} initiates(tic, &alarm(E, delay(D), T) \leftarrow \\ &holds\_at(should\_leave(E, T_d), T) \wedge T > T_d \wedge D \text{ is } T - T_d. \end{aligned} \tag{6}$$

Note that in these rules the time of event termination/start (T_c and T) is the same time present in their respective body's $holds\_at$ atoms. This is perfectly normal, but it is not a requirement. In particular, times could be different, as long as the times of $holds\_at$ atoms do not follow event termination/start times. That again would be allowed, but it would amount to define fluents that depend on future events: fluents that are thus not suitable for runtime monitoring. For that reason, we assume that *well-formed*

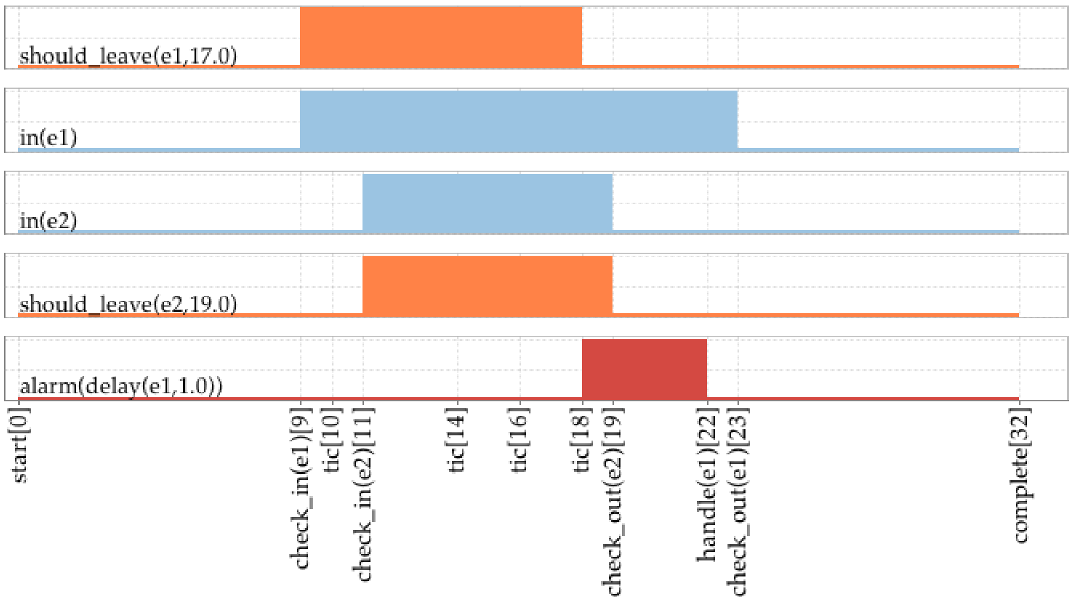

Figure 3. Fluents tracking with $\mathcal{REC}$.

theories do not contain such kind of clauses. More details on this matter will be given below when we discuss the *irrevocability* property in a formal way.

Finally, each alarm related to an employee is turned off when the operator handles that employee:

$$terminates(handle(E), alarm(E, A), T). \tag{7}$$

Based on such a theory, $\mathcal{REC}$ becomes able to dynamically reason from the employees' flow inside the company. In particular, $\mathcal{REC}$ tracks the status of each employee, and generates an alarm as soon as a tic action detects a deadline expiration. As we have seen, alarms are represented by specific fluents, which are initiated by SCIFF when the corresponding exceptional situation is encountered.

Let us consider an event narrative (*execution trace*) involving two employees e_1 and e_2, where e_2 respects the required deadline while e_1 does not:

$$\begin{array}{lll}\mathbf{H}(event(check\_in(e_1)), 9), & \mathbf{H}(event(tic), 10), & \mathbf{H}(event(check\_in(e_2)), 11), \\ \mathbf{H}(event(tic), 14), & \mathbf{H}(event(tic), 16), & \mathbf{H}(event(tic), 18).\end{array}$$

Figure 3 shows the global state of fluents at 18, when $\mathcal{REC}$ generates an alarm because e_1 was expected to leave the office no later than 17, but she has not left yet. The operator can check all pending alarms, and pick an employee to handle in case. Note that, at time 18, there are four pending expectations concerning clipping events, which correspond to four "open" MVIs (i.e., MVIs for which the upper limit is not yet known) related to four different fluents: $alarm(e_1, delay(1))$, $in(e_1)$, $in(e_2)$ and $should\_leave(e_2, 19)$.

The execution now proceeds as follows:

$$\mathbf{H}(event(check\_out(e_2)), 19), \qquad \mathbf{H}(event(handle(e_1)), 22),$$
$$\mathbf{H}(event(check\_out(e_1)), 23), \qquad \mathbf{H}(event(complete), 32).$$

e_2 correctly leaves the office within the deadline, bringing her corresponding in and $should\_leave$ fluents to termination. At 22 the operator handles an alarm involving e_1, who eventually leaves her office at 23.

6. $\mathcal{REC}$ and Negation As Failure

We now discuss the impact of using negated $holds\_at$ predicates inside the definition of the fluents' initiation/termination. The term "negation" will denote, in the following, Negation As Failure. In particular, we show that the standard treatment of negative literals in integrity constraints is inadequate to capture the intended meaning; we then propose a slightly different treatment of negative literals which overcomes these issues while remaining, at the same time, fully declarative. To this purpose, we ground the discussion considering an extended version of the case study described in Section 5.

Let us firstly recall how negation is handled by SCIFF. The SCIFF proof procedure inherits the treatment of negative literals from the IFF proof procedure, through the application of the *logical equivalences* transition. In particular, when a negative literal is encountered by the proof procedure, it is rewritten as follows:

- an integrity constraint $\neg\mathbf{a}(X) \rightarrow \mathbf{b}(Y)$ is rewritten as $true \rightarrow \mathbf{b}(Y) \vee \mathbf{a}(X)$;
- a negated atom in the head $\neg\mathbf{b}(Y)$ is removed and replaced by inserting a new integrity constraint $\mathbf{b}(Y) \rightarrow \bot$.

To show the inadequacy of such a treatment, we extend our case study with a new requirement, used to handle the anomalous case in which an employee forgets to punch in, i.e., she punches out two times in a row without performing a $check\_in$ inbetween. When this situation occurs, a specific alarm is raised, which can be then handled by the operator as in the case of delay.

Using the terminology adopted in Section 5, we can rephrase such a requirement as follows: "A specific alarm is initiated when an employee E checks out, but she does *not* turn out to be in office (i.e., fluent $in(E)$ does not hold)". The corresponding straightforward formalization is:

$$initiates(check\_out(E), alarm(E, forg(check\_in)), T) \leftarrow \neg holds\_at(in(E, T)). \tag{8}$$

Suppose that we have the following (partial) execution trace:

$$\mathbf{H}(event(check\_in(e_1)), 6), \qquad \mathbf{H}(event(check\_out(e_1)), 10).$$

We focus on the last event, occurring at time 10, discussing the transitions of the SCIFF proof procedure. The expected outcome is that fluent $in(e_1)$ maximally holds from time 6 to time 10, and that no alarm is generated.

First of all, a transition *happening* is applied, adding the $\mathbf{H}(check\_out(e_1), 10)$ event occurrence to the current execution trace. Then, a sequence of transitions (starting from *propagation*) is applied in order to match this happened event with some integrity constraints. In particular, the event matches with the body of $\mathcal{REC}$'s Axiom ax_4, eventually leading to:

$$\begin{aligned} initiates(check\_out(e_1), F, 10) \rightarrow & \mathbf{H}(declip(F), 10) \\ & \vee \mathbf{E}(declip(F), T_d) \wedge T_d < 10 \\ & \quad \wedge \mathbf{EN}(clip(F), T_c) \wedge T_c > T_d \wedge T_c < 10. \end{aligned} \tag{9}$$

Transition *unfolding* is then applied to substitute the $initiates$ predicate in the body with its possible definitions (the integrity constraints is replicated for each possible definition of the predicate). Among the unfolded integrity constraints, we find the one obtained by considering clause (8), which leads to rewrite the integrity constraint as:

$$\begin{aligned} \neg holds\_at(in(e_1), 10) \rightarrow & \mathbf{H}(declip(alarm(e_1, forg(check\_in)), 10) \\ & \vee \mathbf{E}(declip(alarm(e_1, forg(check\_in)), T_d) \wedge T_d < 10 \\ & \quad \wedge \mathbf{EN}(clip(alarm(e_1, forg(check\_in)), T_c) \wedge T_c > T_d \wedge T_c < 10. \end{aligned} \tag{10}$$

Let us now consider only the first disjunct in the head. In fact, the other one expects that a previous declip of the alarm occurred, which is not the case; therefore, the latter disjunct will surely lead to a failure. By applying the *logical equivalences* transition on negation, the final result is therefore:

$$\begin{aligned} true \rightarrow & \mathbf{H}(declip(alarm(e_1, forg(check\_in)), 10) \\ & \vee holds\_at(in(e_1), 10). \end{aligned} \tag{11}$$

This rule states that the alarm is raised at time 10 (inclusive) or that fluent $in(e_1)$ must hold at time 10. The problem related to the treatment of negation is now clear: the two disjuncts in the head can be both chosen as suitable candidates for a possible successful derivation; however, while the second disjunct reflects the intended meaning (fluent $in(e_1)$ indeed holds at time 10)[3], the first one leads to generate the alarm, which is a completely "wrong" behavior.

In [29], Sadri and Toni provide different motivating examples to point out such issue, and propose a different treatment of negation to properly deal with them. In particular, they find two separate problems related to the standard treatment of negation when applied to an abducible[4] in the body. Let us consider again the sample rule $\neg\mathbf{a}(X) \rightarrow \mathbf{b}(Y)$, rewritten as $true \rightarrow \mathbf{b}(Y) \vee \mathbf{a}(X)$. The issues are:

- Abducible $\mathbf{a}(X)$ is moved to the head as a new disjunct; therefore, it will be hypothesized by the reasoning machinery, which could be an unexpected behavior or a source of inefficiency.
- The rewritten integrity constraint does not impose mutual exclusion between the head's disjuncts, and therefore it is possible to abduce $\mathbf{b}(Y)$ even if $\mathbf{a}(X)$ can be successfully abduced. This clashes with the intuitive meaning of the integrity constraints, which sounds like "abduce **b**(Y) if **a**(X) is false".

[3]Remember that MVIs are considered as closed on the left and open on the right.

[4]Or a predicate defined in the $\mathcal{KB}$ by means of an abducible, as in the case of $holds\_at$.

To overcome these limits, Sadri and Toni propose to rewrite $\neg\mathbf{a}(X) \rightarrow \mathbf{b}(Y)$ as

$$true \rightarrow \mathbf{b}(Y) \wedge \big(\neg\mathbf{a}(X) \vee provable(\mathbf{a}(X))\big)$$

Negation in the head is then handled in the standard way. Mutual exclusion is imposed by adding $\neg\mathbf{a}(X)$ to the first disjunct, while the second disjunct makes use of the $provable$ meta-predicate, which, roughly speaking, is true when $\mathbf{a}(X)$ is abduced by some other rule.

The main problem of this treatment is that $provable$ does not have a declarative semantics, but only an operational definition. We therefore propose to adopt an intermediate treatment, which allows us to overcome the limits of the standard one but without undermining declarativeness. The idea is to apply mutual exclusion while avoiding the use of $provable$: rule $\neg\mathbf{a}(X) \rightarrow \mathbf{b}(Y)$ is rewritten as

$$true \rightarrow \mathbf{b}(Y) \wedge \big(\neg\mathbf{a}(X) \vee \mathbf{a}(X)\big)$$

We show that this treatment is logically equivalent with the standard one. We discuss the simple case of an integrity constraint containing only a negated abducible in the body and a single abducible in the head; the general case is a straightforward extension.

Theorem 6.1. (Logical equivalence of negation's treatment)
The standard treatment of negative literals

$$\neg\mathbf{a}(X) \rightarrow \mathbf{b}(Y). \quad \Leftrightarrow \quad true \rightarrow \mathbf{b}(Y) \vee \mathbf{a}(X).$$

is equivalent to

$$\neg\mathbf{a}(X) \rightarrow \mathbf{b}(Y). \quad \Leftrightarrow \quad true \rightarrow \big(\neg\mathbf{a}(X) \wedge \mathbf{b}(Y)\big) \vee \mathbf{a}(X).$$

Proof:
Trivially provable, by taking into account the tautology $\neg\mathbf{a}(X) \rightarrow \neg\mathbf{a}(X)$. Indeed, it holds that

$$\neg\mathbf{a}(X) \rightarrow \mathbf{b}(Y). \Longleftrightarrow \left\{ \begin{array}{ll} \neg\mathbf{a}(X) \rightarrow & \mathbf{b}(Y). \\ \neg\mathbf{a}(X) \rightarrow & \neg\mathbf{a}(X). \end{array} \right\} \Longleftrightarrow \neg\mathbf{a}(X) \rightarrow \neg\mathbf{a}(X) \wedge \mathbf{b}(Y).$$

and the standard treatment of the last formulation is exactly $true \rightarrow \big(\neg\mathbf{a}(X) \wedge \mathbf{b}(Y)\big) \vee \mathbf{a}(X)$. □

By adopting this new treatment of negation, rule (11) is rewritten as

$$\begin{aligned} true \rightarrow \quad & \neg holds\_at(in(e_1), 10) \wedge \mathbf{H}(declip(alarm(e_1, forg(check\_in)), 10) \\ & \vee holds\_at(in(e_1), 10). \end{aligned} \tag{12}$$

which captures the intended meaning: the disjunction is interpreted as exclusive, and only the second disjunct can be chosen to have a successful derivation, since $in(e_1)$ holds at time 10.

7. Formal properties of $\mathcal{REC}$

We discuss various formal properties of $\mathcal{REC}$, ranging from general properties (soundness, completeness and termination) to specific properties related to the monitoring setting.

7.1. Soundness, Completeness and Termination

As we have seen in Def. 4.1, a $\mathcal{REC}$ specification is a special kind of SCIFF specification. Reasoning is therefore carried out by using the SCIFF proof procedure without any modification. As a consequence, all the formal properties already proven for SCIFF also hold in the specific case of $\mathcal{REC}$. In particular, $\mathcal{REC}$ inherits both the soundness and completeness properties of the SCIFF's operational semantics with respect to the declarative semantics. Hence, the operational behavior of $\mathcal{REC}$ is faithful to its axiomatization.

Termination of the SCIFF proof procedure has been proven for acyclic and bounded specifications. We briefly recall the concepts of acyclicity and boundedness, and then discuss their impact on $\mathcal{REC}$.

Since SCIFF specifications are abductive logic programs, the acyclicity conditions are extensions of the classical acyclicity conditions defined for logic programs [3]. In particular, the acyclicity conditions defined in [3] apply to the knowledge base of a SCIFF specification, while integrity constraints must obey to the extended conditions defined by Xanthakos in [35][5].

Definition 7.1. (Level mapping)
Let $\mathcal{P}$ be a logic program. A *Level mapping* for $\mathcal{P}$ is a function which maps $\bot$ to 0 and each ground atom in $\mathfrak{B}^{\mathcal{P}}$ to a positive integer:

$$|\cdot| : \mathfrak{B}^{\mathcal{P}} \longrightarrow \mathbb{N} \backslash \{0\}$$

where $\mathfrak{B}^{\mathcal{P}}$ is the Herbrand base of $\mathcal{P}$. Given $A \in \mathfrak{B}^{\mathcal{P}}$, $|A|$ denotes the *level* of A.

Definition 7.2. (Boundedness)
Given a level mapping $|\cdot|$, a literal L is *bounded* with respect to $|\cdot|$ iff $|\cdot|$ is bounded on the set of ground instances of L. In this case, we assume

$$|L| \triangleq \max\{|L_g| \text{ such that } L_g \text{ is a ground instance of } L\}$$

Definition 7.3. (Acyclic logic program)
Given a logic program $\mathcal{P}$, a clause $C \in \mathcal{P}$ is *acyclic* with respect to the level mapping $|\cdot|$ iff, for every ground instance $H \leftarrow B_1 \wedge \ldots \wedge B_n$ of C, it holds that: $\forall\, i \in \{1, \ldots, n\},\ |H| > |B_i|$. The entire logic program $\mathcal{P}$ is acyclic if all its clauses are acyclic with respect to some level mapping.

Definition 7.4. (Acyclic SCIFF specification)
A SCIFF specification $\mathcal{S} = \langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$ is acyclic iff there exists a level mapping $|\cdot|$ such that

1. $\mathcal{KB}$ is acyclic with respect to $|\cdot|$;
2. each $IC \in \mathcal{IC}$ is acyclic with respect to $|\cdot|$: for each ground instance $B_1 \wedge \ldots \wedge B_n \rightarrow H_1 \vee \ldots \vee H_m$ of an integrity constraint in $\mathcal{IC}$, it holds that $\forall\, i \in \{1, \ldots, n\},\ \forall\, j \in \{1, \ldots, m\},\ |B_i| > |H_j|$.

The following theorem states that SCIFF is guaranteed to terminate the computation if the specification under study is acyclic and bounded. This, in turn, implies that SCIFF terminates when reasoning upon a $\mathcal{REC}$ specification, provided that the domain-dependent knowledge base is acyclic and bounded.

[5] In the following, we will consider only the case of positive literals; for negative literals, the interested reader can refer to [35].

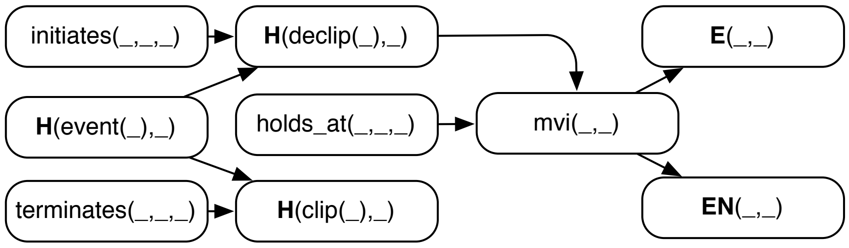

Figure 4. Dependency graph showing the level mapping's constraints among the literals contained in the axiomatization of $\mathcal{REC}$

Theorem 7.1. (Termination of the SCIFF proof procedure [2])
Given a SCIFF instance $\mathcal{S}_\mathcal{T}$, if $\mathcal{S} = \langle \mathcal{KB}, \mathcal{IC} \rangle$ is acyclic and all the literals occurring in $\mathcal{IC}$ are bounded, then every derivation of the SCIFF proof procedure for $\mathcal{S}_\mathcal{T}$ is finite.

Theorem 7.2. ($\mathcal{REC}$ termination)
Let $\mathcal{S}_R$ be a $\mathcal{REC}$ specification having $\mathcal{KB}_{\mathcal{REC}}$ as domain-dependent knowledge base. The SCIFF proof procedure terminates when reasoning upon $\mathcal{S}_R$ if $\mathcal{KB}_{\mathcal{REC}}$ is acyclic and does not contain contradicting predicates $holds\_at(F, T)$ and $\neg\ holds\_at(F, T)$ in the definition of a $initiates/3$ or $terminates/3$ predicate.

Proof:
Let us first consider the general axiomatization of $\mathcal{REC}$ alone. The combination of the general axioms of the calculus, listed in Sect. 4.1, is acyclic and bounded. In particular, even if the ICs of $\mathcal{REC}$ contain happened events both inside the body and inside the head, such happened events are different: happened events in the body refer to $event/1$ terms, while happened events in the head refer to $clip/1$ and $declip/1$ terms. The dependency graph representing all constraints among the level mapping of each literal is shown in Fig. 4. The figure shows that acyclicity is respected and that all the ground versions of each literal can be mapped onto the same level, thus ensuring that boundedness is guaranteed as well.

Let us now focus on the interplay between the general axiomatization of $\mathcal{REC}$ and the domain-dependent knowledge base $\mathcal{KB}_{\mathcal{REC}}$: the only possible source of cyclicity is the adoption of a positive and a negative $holds\_at/3$ predicate in the definition of the same $initiates/3$ or $terminates/3$ predicate. Indeed, due to Axioms (ax_4) and (ax_6) and to the treatment of negation presented in Sec. 6, the positive $holds\_at/3$ belongs to the body of the axiom, while the negative one is moved to the head, thus introducing a cyclic dependency between $holds\_at/3$ and itself. Fortunately, acyclicity and boundedness are still guaranteed provided that the two predicates refer to different fluent or time variables. This is ensured by the noncontradiction hypothesis. □

The next results concern uniqueness and irrevocability, and they are instead relative to the special kind of reasoning needed for the monitoring applications. We thus need to introduce some information about the SCIFF's operational behavior.

7.2. Open, Closed and Semi-Open Reasoning

In general, SCIFF features two main forms of inference, called *open* and *closed derivation*. A derivation starts from a (possibly empty) goal and the integrity constraints, and it generates a list of nodes according to the operational semantics [2], i.e., by applying a SCIFF transition rule at a time. It terminates when no transition is applicable. Open and closed derivations differ by one such transition.

Given a specification $\mathcal{S}$ and two *execution traces* (sets of **H** events) $\mathcal{H}^i$ and $\mathcal{H}^f \supseteq \mathcal{H}^i$, if there exists an *open successful derivation* for a goal $\mathcal{G}$ that leads from $\mathcal{H}^i$ to $\mathcal{H}^f$ we write $\mathcal{S}_{\mathcal{H}^i} \mid\!\sim^{\mathcal{H}^f}_{\Delta} \mathcal{G}$, where Δ is the computed abductive explanation[6]. If $\mathcal{S}$ is a $\mathcal{REC}$ specification, Δ includes the abduced MVIs. When SCIFF executes an open derivation, it assumes that the acquired execution trace is partial. Thus **E** atoms without a matching **H** atom are not considered as violated but only as *pending*: further events may still occur to fulfill them. **EN** atoms can instead be evaluated, because they must never have a matching **H** atom. This approach is used when SCIFF is used for runtime verification, when the narrative is incomplete and events occur dynamically.

SCIFF can also perform *closed derivations*, to reason from narratives known to be complete, or to close the inference process when a dynamic execution comes to an end. In that case, both **E** and **EN** atoms are evaluated: a closed world assumption is made on the collected execution trace, and positive (negative resp.) pending expectations are considered as violated (fulfilled), because no further event will occur to fulfill (violate) them.

SCIFF is sound and complete independently of the order of events. However, there are many important domains in which we can safely assume that events are acquired in increasing order of time. In that case, reasoning is *partially open*: open on the future, when events may still occur, but closed on the past. Expectations on the past can thus be evaluated immediately. To enable this form of *semi-open* reasoning, the SCIFF proof-procedure is equipped with an additional rule, which states that if the execution trace has reached time t, then all pending expectations must be fulfilled at a time $t' \geq t$. We denote such a *semi-open* derivation by $\mid\!\approx$.

7.3. Irrevocability of $\mathcal{REC}$

Monitoring applications enable semi-open derivation. It is required that the generated MVIs are never retracted, but only extended or terminated as new events occur, and that newly generated MVIs do not refer to the past. If that is the case, the reasoning process is called *irrevocable*. Some target applications need irrevocable tracking procedures, which can give a continuously updated view of the status of all fluents. Fluttering behaviours must be by all means avoided. This is true, e.g., when the modeled fluents carry a normative meaning. It would be undesirable, for instance, to attach a certain obligation to an agent at runtime, and see it disappear later only because the calculus revises its computed status.

Two potential sources of irrevocability can be identified:

- the occurring events are not acquired in ascending order, i.e., it could be the case that a happened event referring to the past is analyzed by SCIFF;
- the $\mathcal{REC}$ specification contains nondeterministic choices.

While the first cause is independent from SCIFF, we must guarantee that when the happened events are delivered to SCIFF in ascending order, $\mathcal{REC}$ exhibits an irrevocable behavior. Thus we need to isolate "good" sets of specifications. Once we have them, we must ensure that the reasoning machinery does not make unjustified retractions. In other words, we must guarantee irrevocability as long as the execution traces evolve by ascending times.

[6]Note that Δ only depends on $\mathcal{H}^f$, because $\mathcal{H}^f$ includes $\mathcal{H}^i$. Therefore, in the following we will omit $\mathcal{H}^i$ when possible.

In the reminder of this section, we first define a class of "well-formed" $\mathcal{REC}$ theories, then we show that semi-open reasoning on the resulting $\mathcal{REC}$ specifications is irrevocable. Note that when monitoring the execution, $\mathcal{REC}$ is used with goal $true$.

Definition 7.5. (Well-formed $\mathcal{REC}$ theory)
A knowledge base $\mathcal{KB}$ is well-formed iff

1. $\mathcal{KB}$ is bounded and acyclic;
2. for each clause of the form $initiates(Ev, F, T) \leftarrow body$, fluent F must always be resolved with a ground substitution;
3. by considering all the clauses of the form

   $$initiates(Ev, F, T) \leftarrow body. \qquad\qquad terminates(Ev, F, T) \leftarrow body.$$

   for each $[\neg]holds\_at(F_2, T_2) \in body$, it must hold that $T_2 \leq T$.

Roughly speaking, the first condition ensures that the SCIFF proof procedure will terminate when reasoning upon a $\mathcal{REC}$ specification with a well-formed domain-dependent theory. $\mathcal{REC}$ theories that violate the second or the third condition would instead introduce choice points that hinder irrevocability.

In particular, the second condition prevents non-determinism due to case analysis: when a fluent containing a variable is declipped, it could be the case that its future clipping is non-deterministic. For example, let us consider the following knowledge base:

$$initiates(check\_in(e_1), in(X), T). \tag{13}$$
$$terminates(check\_out(e_1), in(e_1), T). \tag{14}$$

This is an ambiguous specification since it does not clearly state which employee should change status as a consequence of e_1 checking in. When e_1 checks in, the generic fluent $in(X)$ is generated. When e_1 checks out, two possible alternatives are explored by SCIFF: in the first one, it hypothesizes that X/e_1, and thus the fluent is terminated; in the second one, it applies case analysis, supposing that $X \neq e_1$ and maintaining the fluent valid.

Finally, the third condition restricts us to reasoning on stable, past conditions. If the initiation or termination of a fluent depends on future conditions, then SCIFF must speculate on such conditions, exploring the alternative in which they will hold and the possibility in which they will not hold. One of the two choices will eventually lead to a failure. Let us for example consider the following clause:

$$\begin{aligned} initiates(tic, &alarm(delay(E, D), T) \leftarrow \\ &holds\_at(should\_leave(E, T_d), T_1) \wedge D \text{ is } T - T_d \wedge T_1 > T. \end{aligned} \tag{15}$$

The meaning would be that an action is a consequence of an alarm which has not fired yet. Only speculations are possible in that case, and no runtime monitoring algorithm could provide deterministic answers (save freezing until the alarm fires, but in that case the application would no longer be called "runtime").

A well-formed $\mathcal{REC}$ specification $\mathcal{S}$ brings a twofold advantage:

1. given a trace $\mathcal{T}$, there exists exactly one SCIFF semi-open successful derivation for $\mathcal{S}_\mathcal{T}$;

2. if the execution trace is extended with new happened events occurring in ascending order, then the newly computed result satisfies the irrevocability principle.

We discuss both results. The proofs of the presented theorems can be found in [24].

Theorem 7.3. (Uniqueness of derivation [24])
For each well-formed $\mathcal{REC}$ theory $\mathcal{T}$ and for each execution trace $\mathcal{H}$, there exists exactly one successful semi-open derivation computed by SCIFF for the goal $true$, i.e. $\exists 1\Delta$ s.t. $\mathcal{T} \mid\!\approx^{\mathcal{H}}_{\Delta} true$.

Theorem 7.3 ensures that exactly one Δ is produced by a semi-open derivation of SCIFF. This, in turn, means that there exists exactly one "configuration" for the MVIs of each fluent. We give a precise definition of this notion of state, which is the one of interest when evaluating the irrevocability of the reasoning process, and define the notion of progressive extension between states, which gives a formal account to irrevocability.

Definition 7.6. (Current time)
The *current time* of an execution trace $\mathcal{H}$, $ct(\mathcal{H})$, is the latest time of its events:

$$ct(\mathcal{H}) \equiv \max \{t \mid \mathbf{H}(event(e), t) \in \mathcal{H}\}$$

Definition 7.7. (MVI State)
Given a $\mathcal{REC}$ specification $\mathcal{R}$ and an execution trace $\mathcal{H}$ the resulting *MVI state* at time $ct(\mathcal{H})$ is the set of **mvi** abducibles contained in the computed explanation generated by SCIFF with goal $true$:

$$\mathit{MVI}(\mathcal{R}_{\mathcal{H}}) \equiv \{\mathbf{mvi}(F, [T_s, T_e]) \in \Delta\}, \text{ where } \mathcal{R} \mid\!\approx^{\mathcal{H}}_{\Delta} true$$

Definition 7.8. (State sub-sets)
Given a $\mathcal{REC}$ specification $\mathcal{R}$ and a (partial) execution trace $\mathcal{H}$, the current state $\mathit{MVI}(\mathcal{R}_{\mathcal{H}})$ is split into two sub-sets:

- $\mathit{MVI}_{\lrcorner\sqcap\llcorner}(\mathcal{R}_{\mathcal{H}})$, is the set of (closed) MVIs, terminating at a ground time:

$$\mathit{MVI}_{\lrcorner\sqcap\llcorner}(\mathcal{R}_{\mathcal{H}}) = \{\mathbf{mvi}(F, [s, e]) \in \mathit{MVI}(\mathcal{R}_{\mathcal{H}}) \mid s, e \in \mathbb{N}, F\ variable\};$$

- $\mathit{MVI}_{\lrcorner\ulcorner}(\mathcal{R}_{\mathcal{H}})$, is the set of (open) MVIs, terminating at a variable time:

$$\mathit{MVI}_{\lrcorner\ulcorner}(\mathcal{R}_{\mathcal{H}}) = \{\mathbf{mvi}(F, [s, T]) \in \mathit{MVI}(\mathcal{R}_{\mathcal{H}}) \mid s \in \mathbb{N}, F\ and\ T\ variables\}.$$

Definition 7.9. (Trace extension)
Given two execution traces $\mathcal{H}^1$ and $\mathcal{H}^2$, $\mathcal{H}^2$ is an *extension* of $\mathcal{H}^1$, written $\mathcal{H}^1 \prec \mathcal{H}^2$, iff

$$\forall\, \mathbf{H}(e, t) \in \mathcal{H}^2 \backslash \mathcal{H}^1,\ t > ct(\mathcal{H}^1)$$

Definition 7.10. (State progressive extension)
Given a well-formed $\mathcal{REC}$ specification $\mathcal{R}$ and two execution traces $\mathcal{H}^1$ and $\mathcal{H}^2$, the state of $\mathcal{R}_{\mathcal{H}^2}$ is a *progessive extension* of the state of $\mathcal{R}_{\mathcal{H}^1}$, written $\mathit{MVI}(\mathcal{R}_{\mathcal{H}^1}) \trianglelefteq \mathit{MVI}(\mathcal{R}_{\mathcal{H}^2})$, iff

1. the set of closed MVIs is maintained in the new state: $\mathit{MVI}_{\lrcorner\sqcap\llcorner}(\mathcal{R}_{\mathcal{H}^1}) \subseteq \mathit{MVI}_{\lrcorner\sqcap\llcorner}(\mathcal{R}_{\mathcal{H}^2})$

2. if the set of MVIs is extended with new MVIs, these are declipped after the maximum time of $\mathcal{H}^1$: $\forall\, \mathbf{mvi}(f, [s,t]) \in \mathit{MVI}(\mathcal{R}_{\mathcal{H}^2}) \backslash \mathit{MVI}(\mathcal{R}_{\mathcal{H}^1}), s > ct(\mathcal{H}^1)$
3. $\forall\, \mathbf{mvi}(f, [s, T_e]) \in \mathit{MVI}_{\ulcorner}(\mathcal{R}_{\mathcal{H}^1})$, either
   (a) it remains untouched in the new state: $\mathbf{mvi}(f, [s, T_e]) \in \mathit{MVI}_{\ulcorner}(\mathcal{R}_{\mathcal{H}^2})$, or
   (b) it is clipped after the maximum time of $\mathcal{H}^1$: $\mathbf{mvi}(f, [s, e]) \in \mathit{MVI}_{\ulcorner\llcorner}(\mathcal{R}_{\mathcal{H}^2}), e > ct(\mathcal{H}^1)$.

Progressive extensions capture the intuitive notion that a state extends another one if it keeps the already computed closed MVIs and affects the status of fluents only after the time at which the first state was recorded. The extension is determined by adding new MVIs and by clipping fluents which hold at the previous state. We can state the main result leading to irrevocability, namely that extending a trace results in a progressive extension of the MVI state.

Lemma 7.1. (Trace extension leads to a state progressive extension [24])
Given a well-formed $\mathcal{REC}$ specification $\mathcal{R}$ and two execution traces $\mathcal{H}^1$ and $\mathcal{H}^2$,

$$\mathcal{H}^1 \prec \mathcal{H}^2 \Rightarrow \mathit{MVI}(\mathcal{R}_{\mathcal{H}^1}) \trianglelefteq \mathit{MVI}(\mathcal{R}_{\mathcal{H}^2})$$

Theorem 7.4. (Irrevocability of $\mathcal{REC}$ [24])
Given a well-formed $\mathcal{REC}$ specification with goal $true$ and a temporally ordered narrative, each time SCIFF processes a new event, the new MVI state is a progressive extension of the previous one.

8. Conclusion

$\mathcal{EC}$ is a powerful framework for reasoning about time and events. We identified the problem of applying the $\mathcal{EC}$ in order to provide runtime monitoring facilities, to track the dynamics of the system under study and promptly detect exceptional and undesired situations. We observed that there is no satisfactory solution to it in the state of the art. Specifically, related approaches mainly boil down to ad-hoc, tailored procedures that are not easily modifiable and whose formal properties are not easy to determine. We therefore provided the first formal and operational approach to the problem, a $\mathcal{REC}$ implementation in SCIFF. While literature flourishes with procedural implementations of monitoring systems, we have shown how to solve this seemingly procedural problem declaratively, discussing several formal properties of the resulting framework. Particular attention has been given to negative $holds\_at$ predicates, pointing out that the standard treatment of negation by the state of the art abductive proof procedures must be revised to reflect the intended meaning.

$\mathcal{REC}$ is currently subject of different practical research lines and applications. In the Multi-Agent Systems setting, $\mathcal{REC}$ is being investigated to realize a monitoring infrastructure for commitment-based interaction protocols [6]. Social commitments [5, 33] have been already mapped onto $\mathcal{EC}$ [36]; an extended version of this already formalization has been encoded in $\mathcal{REC}$ to provide monitoring support, showing the evolution of commitments in response to the occurring events, and enabling new features such as the possibility of modeling and verifying timed commitments and compensations triggered by the expiration of deadlines [6, 34, 8]. In the Service-Oriented Computing research field, we are applying $\mathcal{REC}$ for monitoring service interaction. In particular, in [25] we have shown that the DecSerFlow language [27, 26], a declarative graphical language for modeling control-flow constraints on the service behavior, can be formalized on top of $\mathcal{REC}$. This makes it possible to dynamically collect the messages exchanged by the interacting services and check how they affect the choreographic DecSerFlow constraints, supporting quantitative time conditions such as delays and deadlines.

Finally, a JAVA-based application is being developed in order to wrap $\mathcal{REC}$ and give a graphical feedback about the outcome produced by the monitoring framework. A preliminary report on this research line can be found in [7, 8].

It is worth noting that $\mathcal{REC}$ adopts the standard $\mathcal{EC}$ ontology. In particular, domain-dependent $\mathcal{REC}$ theories are in fact standard $\mathcal{EC}$ theories, possibly extended with the use of CLP constraints. This feature is of utmost importance, because it guarantees that $\mathcal{REC}$ can be seamlessly applied to monitor arbitrary $\mathcal{EC}$-based specifications. For example, in [11], the authors propose to adopt $\mathcal{EC}$ in order to formalize and execute workflows. To this aim, they integrate the main $\mathcal{EC}$ axioms with a set of activity execution dependency rules and a set of agent assignment rules, showing that the basic control-flow routing mechanisms (sequence, concurrency, alternatives, iteration) can be easily expressed in a declarative way. The obtained formalization could be seamlessly considered as a $\mathcal{REC}$ domain-dependent knowledge base, enabling run-time monitoring facilities. In particular, $\mathcal{REC}$ could be used to infer the current state of affairs, showing which activities have been already completed and which activities are expected to be executed next, according to the control-flow routing elements.

References

[1] van der Aalst, W. M. P., van Hee, K. M., Martijn E. M. van der Werf, J., Verdonk, M.: Auditing 2.0: Using Process Mining to Support Tomorrow's Auditor, *IEEE Computer*, **43**(3), 2010, 90–93.

[2] 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 (TOCL)*, **9**(4), 2008.

[3] Apt, K. R., Bezem, M.: Acyclic Programs, in: *Logic Programming* (D. H. Warren, Ed.), MIT Press, 1990, 617–633.

[4] Bürckert, H.-J.: A Resolution Principle for Constrained Logics, *Artificial Intelligence*, **66**, 1994, 235–271.

[5] Castelfranchi, C.: Commitments: From individual intentions to groups and organizations, *Proceedings of the First International Conference on Multiagent Systems (ICMAS1995)* (V. R. L. L. Gasser, Ed.), The MIT Press, 1995.

[6] Chesani, F., Mello, P., Montali, M., Torroni, P.: Commitment Tracking via the Reactive Event Calculus, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)* (C. Boutilier, Ed.), 2009.

[7] Chesani, F., Mello, P., Montali, M., Torroni, P.: A REC-Based Commitment Tracking Tool, *Demo at the 10th AI*IA/TABOO Italian Joint Workshop "From Objects to Agents" (WOA 2009)*, 2009.

[8] Chesani, F., Montali, M., Mello, P., Torroni, P.: Monitoring Time-Aware Social Commitments with Reactive Event Calculus, *Proceedings of the 7th International Symposium "From Agent Theory to Agent Implementation" (AT2AI-7)*, 2010.

[9] Chittaro, L., Montanari, A.: Efficient Temporal Reasoning in the Cached Event Calculus, *Computational Intelligence*, **12**, 1996, 359–382.

[10] Chittaro, L., Montanari, A.: Temporal Representation and Reasoning in Artificial Intelligence: Issues and Approaches, *Annals of Mathematics and Artificial Intelligence*, **28**(1-4), 2000, 47–106.

[11] Cicekli, N. K., Cicekli, I.: Formalizing the Specification and Execution of Workflows Using the Event Calculus, *Information Sciences*, **176**(15), 2006, 2227–2267.

[12] Clark, K. L.: Negation as Failure, in: *Logic and Data Bases* (H. Gallaire, J. Minker, Eds.), Plenum Press, 1978, 293–322.

[13] Dincbas, M., Hentenryck, P. V., Simonis, H., Aggoun, A., Graf, T., Berthier, F.: The Constraint Logic Programming Language CHIP, *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1988.

[14] Eshghi, K.: Abductive Planning with Event Calculus, *Proceedings of the Fifth International Conference and Symposium on Logic Programming (ILPS 1988)*, MIT Press, 1988, ISBN 0-262-61056-6.

[15] Farrell, A. D. H., Sergot, M. J., Sallé, M., Bartolini, C.: Using the Event Calculus for Tracking the Normative State of Contracts, *International Journal of Cooperative Information Systems*, **14**(2-3), 2005, 99–129.

[16] Fernandes, A. A. A., Williams, M. H., Paton, N. W.: A Logic-Based Integration of Active and Deductive Databases, *New Generation Computing*, **15**(2), 1997, 205–244.

[17] Fung, T. H., Kowalski, R. A.: The Iff Proof Procedure for Abductive Logic Programming, *Logic Programming*, **33**(2), 1997, 151–165.

[18] Jaffar, J., Maher, M. J.: Constraint Logic Programming: a Survey, *Logic Programming*, **19-20**, 1994, 503–582.

[19] Jaffar, J., Michaylov, S., Stuckey, P. J., Yap, R. H. C.: The CLP(R) Language and System, *ACM Transactions on Programming Languages and Systems*, **14**(3), 1992, 339–395.

[20] Kowalski, R. A., Sadri, F.: Towards a Unified Agent Architecture that Combines Rationality with Reactivity, *Proceedings of the International Workshop on Logic in Databases (LID'96)*, 1154, Springer, 1996, ISBN 3-540-61814-7.

[21] Kowalski, R. A., Sergot, M.: A Logic-Based Calculus of Events, *New Generation Computing*, **4**(1), 1986, 67–95.

[22] Kunen, K.: Negation in Logic Programming, *Logic Programming*, 4, 1987.

[23] Mahbub, K., Spanoudakis, G.: Run-Time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience, *Proeedings of the 3rd IEEE International Conference on Web Services (ICWS 2005)*, IEEE Computer Society, 2005.

[24] Montali, M.: *Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach*, Ph.D. Thesis, University of Bologna, 2009.

[25] Montali, M., Chesani, F., Mello, P., Torroni, P.: Verification of Choreographies During Execution Using the Reactive Event Calculus, *Proceedings of the 5th International Workshop on Web Service and Formal Methods (WS-FM2008)* (R. Bruni, K. Wolf, Eds.), 5387, Springer Verlag, 2009.

[26] 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.

[27] Pesic, M., van der Aalst, W. M. P.: A Declarative Approach for Flexible Business Processes Management, *Proceedings of the BPM 2006 Workshops*, 4103, Springer Verlag, 2006.

[28] Rouached, M., Fdhila, W., Godart, C.: A Semantical Framework to Engineering WSBPEL Processes, *Information Systems and E-Business Management*, **7**(2), 2008, 223–250.

[29] Sadri, F., Toni, F.: Abduction with Negation as Failure for Active and Reactive Rules, *AI*IA 99:Advances in Artificial Intelligence, 6th Congress of the Italian Association for Artificial Intelligence, Bologna, Italy, September 14-17, 1999, Proceedings* (E. Lamma, P. Mello, Eds.), 1792, Springer, 1999.

[30] Shanahan, M.: Robotics and the Common Sense Informatic Situation, *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI 1996)* (W. Wahlster, Ed.), John Wiley and Sons, 1996.

[31] Shanahan, M.: The Event Calculus Explained, in: *Artificial Intelligence Today: Recent Trends and Developments* (M. Wooldridge, M. M. Veloso, Eds.), vol. 1600 of *Lecture Notes in Computer Science*, Springer Verlag, 1999, ISBN 3-540-66428-9, 409–430.

[32] Shanahan, M.: An Abductive Event Calculus Planner, *Journal of Logic Programming*, **44**(1-3), 2000, 207–240.

[33] Singh, M. P.: An Ontology for Commitments in Multiagent Systems: Toward a Unification of Normative Concepts, *Artificial Intelligence and Law*, **7**, 1999, 97–113.

[34] Torroni, P., Chesani, F., Mello, P., Montali, M.: Social Commitments in Time: Satisfied or Compensated, *Declarative Agent Languages and Technologies VII (DALT 2009). Revised Selected and Invited Papers* (M. Baldoni, J. Bentahar, M. B. van Riemsdijk, J. Lloyd, Eds.), 5948, Springer, 2009.

[35] Xanthakos, I.: *Semantic Integration of Information by Abduction*, Ph.D. Thesis, Imperial College London, 2003.

[36] Yolum, P., Singh, M. P.: Flexible Protocol Specification and Execution: Applying Event Calculus Planning Using Commitments, *Proceedings of the First International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2002)*, ACM Press, 2002.