Reactive Event Calculus for Monitoring Global Computing Applications

Stefano Bragaglia[1], Federico Chesani[1], Paola Mello[1], Marco Montali[2], and Paolo Torroni[1]

[1] University of Bologna, 40128 Bologna, Italy
{stefano.bragaglia,federico.chesani,paola.mello,paolo.torroni}@unibo.it
http://ai.unibo.it
[2] Free University of Bozen-Bolzano Bolzano, Italy
montali@inf.unibz.it
http://www.inf.unibz.it/∼montali/

Abstract. In 1986 Kowalski and Sergot proposed a logic-based formalism named Event Calculus (EC), for specifying in a declarative manner how the happening of events affects some representation (the *state*) of the world. Since its introduction, EC has been recognized for being an excellent framework to reason about time and events. Recently, with the advent of complex software systems decomposed into sets of autonomous, heterogeneous distributed entities, EC has drawn attention as a viable solution for monitoring them, where monitoring means to represent their state and how events dynamically affect such state.

In this work we present the fundamentals of a reactive and logic-based version of EC, named REC, for monitoring declarative properties, while maintaining a solid formal background. We present some results about its formal as well as practical aspects, and discuss how REC has been applied to a variety of application domains, namely BPM, SOC, CGs and MAS. We also highlight some key issues required by the monitoring task, and finally discuss how REC overcomes such issues.

Keywords: Sergot, Event Calculus, Monitoring, Computational Logic, Business Process Management, Service Oriented Architectures, Multi-Agent Systems, Clinical Guidelines.

1 Introduction

In 1986 Robert A. Kowalski and Marek J. Sergot proposed, in their seminal paper entitled *"A logic-based Calculus of Events"* [15], an approach to representing the effects of action and change in a logic programming framework. The name *Event Calculus* (EC) was coined to draw attention to the contrast with the conception of action and change employed in the situation calculus of McCarthy and Hayes [17]. Instead of thinking primarily in terms of situations and actions as transitions between situations, the authors proposed to think first and foremost about the occurrences of actions–*events*–and the periods of time that they initiate and terminate [25]. The EC is expressed by small set of simple logical axioms, which

A. Artikis et al. (Eds.): Sergot Festschrift, LNAI 7360, pp. 123–146, 2012.

link the happening of events at certain time instants, to relationships and the periods for which they hold. The relationships, to be later known as *fluents*, would be intended as the terms for describing (the state of) a world. The time intervals for which fluents hold, instead, would describe the evolution of the world over time. The axioms were defined in terms of the Horn subset of classical logic, augmented with negation as failure, thus making the formalism executable as a Prolog program.

Since its introduction, the EC has been exploited in a variety of domains, such as cognitive robotics [26], planning [28], service interaction [16] and composition [23], active databases [12], workflow modelling [10] and legal reasoning [11]. Moreover, besides the original implementation presented by Kowalski and Sergot, many other variants and extensions have been proposed, based on logics [24,27,7], as well as on other programming paradigms such as in [11].

Recently, with the advent of distributed software solutions, EC took center stage again. Thanks to the increasing pervasiveness of networks, computers and communication modalities, the complexity of today's systems is often faced by decomposing them into a set of autonomous, possibly distributed entities. Examples of systems embracing this approach are Business Process Management [34], (Computerized) Clinical Guidelines [30], Service-Oriented Computing [13] and Multi-Agent Systems [33]. Such kinds of systems exhibit complex and unpredictable dynamics, emerging from the interaction of multiple, autonomous, and heterogeneous components. As a side effect, assuring the right "behaviour" of such systems is becoming a complex task too. Other problems arise from the limits of classical debugging techniques, which cannot help with dynamic environments where heterogeneous components, often treated as "black boxes", are not internally observable. Monitoring these systems while executing is a practical approach that offer a partial solution to such issue. If assuring the right behaviour *before* the execution can be done only to some extent, monitoring *during* the execution helps to detect wrong happenings/states, and allows to take possible countermeasures.

Monitoring at run-time means to represent the state of such systems, how events affect their state, and to dynamically understand if certain properties are indeed respected. Thus, after 25 years, EC is going to be a suitable candidate for this task.

However, the monitoring task has its own peculiarities. To be effective, the original EC must be adapted to cope with the monitoring's dynamic nature. For example, the classical axiomatization of EC is basically deductive, and is typically used to reason about a fixed history of events. Adding new events require to restart the EC reasoning from scratch. In a monitoring setting, where events continuously happens at a fast pace, such behaviour is not practical.

The aim of this paper is two-fold. On one side, we want to discuss how EC can be extended to cope with the monitoring task, discussing formal aspects, properties as well as performance issues. On the other side, we introduce the reader to four different application domains where EC proved to be a viable and successful solution.

Table 1. The Event Calculus ontology.

$happens\_at(Ev, T)$	Event Ev happens at time T
$holds\_at(F, T)$	Fluent F holds at time T
$mvi(F, [T_1, T_2])$	$(T_1, T_2]$ is a maximal validity interval for F
$initially(F)$	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

To this end, the paper content is organized in two distinguishable parts. In the first part, after a brief recall of the EC original framework (Section 2), we present (Section 3) our extensions to the original event calculus, extensions made to face the peculiarities of the monitoring task. In particular, we present two different implementations of the Reactive Event Calculus, as we named our extensions to the original EC. The first implementation of REC, namely REC_S, is based on a solid, logic background, and it enjoys several formal properties. The second implementation, namely REC_P, is more oriented to performance aspects, a fundamental requirement when dealing with dynamic environments like, e.g., SOA and BPM workflows. A performance comparison between REC_S and REC_P is also presented. While REC_S has been partly published elsewhere, REC_P and the performance comparisons are novel and original contributions.

In the second part of the paper (Section 4) we present our own experiences on applying the REC framework in four different application fields, briefly introducing peculiarities and how REC has been exploited in each context. Our intention is not to provide a survey of all the application fields where EC has been applied. Rather, the aim is to highlight how REC provides a good solution in many fields where monitoring plays a fundamental role.

2 Background on the Event Calculus

The basic concepts of the Event Calculus are those of *event*, happening at a point in time, and of *fluent*, holding during time intervals. Fluents are initiated/terminated by events. Given an event narrative (a set of events), the EC theory and domain-specific axioms together ("EC axioms") define which fluents hold at each time. There are many different formulations of these axioms [9]. A possible formalisation is given by axioms ec_1 and ec_2, where P stands for *Fluent*, E for *Event*, and T represents time instants. Predicates *holds_at* and *clipped* capture, the notions of a fluent that respectively holds at time T, or has been terminated within the time interval $[T_1, T_3]$.

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

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

$$initiates(E,P,T) \leftarrow happens\_at(E,T) \wedge holds\_at(P_1,T) \wedge ... \wedge holds\_at(P_M,T). \quad (ec_3)$$

$$terminates(E,P,T) \leftarrow happens\_at(E,T) \wedge holds\_at(P_1,T) \wedge ... \wedge holds\_at(P_N,T). \quad (ec_4)$$

Roughly speaking, Axiom ec_1 can be intended in the following way: fluent P holds at time T if an event E initiated it at time T_{Start}, and P has not been clipped in the time interval $[T_{Start} \ldots \mathrm{T}]$. Axiom ec_2 then specifies that a fluent has been clipped between time T_1 and T_3 if an event E that terminates P happened within such time interval.

Axioms (ec_3, ec_4) are instead schemas for defining the domain-specific axioms: a certain fluent P is initiated/terminated at time T if an event E happened at the same time, while some other fluents P_i hold. The expression $happens\_at(E,T) \wedge holds\_at(P_1,T) \wedge ... \wedge holds\_at(P_N,T)$ represents the *context* which cause E to initiate P. In general, the context can be any conjunction of literals. To say that a fluent holds at the beginning of time we can use the shorthand $initially(P)$. Dual axioms and predicates can be added to define when fluents *do*/*do not* hold [27]: e.g., an axiom can be added to define *declipped*/3 (an event has made a certain fluent holding). The EC formalization above is called *simple* EC, and uses the Horn fragment of first order logic, augmented with negation as failure.

An EC *theory* is a knowledge base composed by a set of clauses (*initiates*, *terminates*, ...) that relate events to fluents. The set of all EC predicates that will be used throughout the paper is listed in Table 1.

A classic example within the EC research literature is the light switch case. A fluent *light_on* toggles between true and false at every touch of the light switch. To tell whether the light is on or off, we need to know how the light is at the beginning of time, and what is the effect of using the switch. That depends on the current state: if the light is on, then the effect is to switch it off, otherwise the effect is to turn the light on. A sample EC theory for the light switch is given below:

$$initially(light\_on).$$

$$initiates(switch\_pressed, light\_on, T) \leftarrow happens\_at(switch\_pressed, T) \wedge \neg holds\_at(light\_on, T).$$

$$terminates(switch\_pressed, light\_on, T) \leftarrow happens\_at(switch\_pressed, T) \wedge holds\_at(light\_on, T).$$

In Figure 1 it is shown how the fluent *light_on* changes its value when the switch is pressed at time instants 10, 20, 35, and 55, respectively.

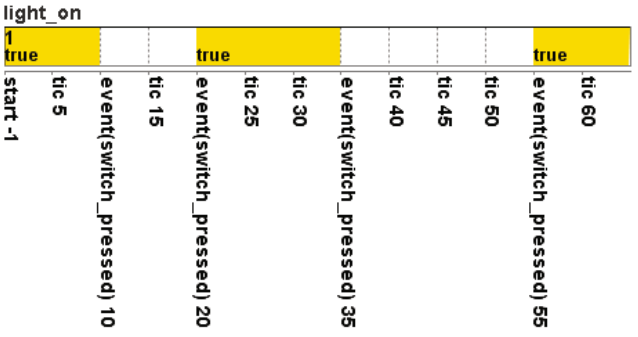


Fig. 1. Fluents for the *light_on* example

3 The Reactive Event Calculus

Since its first axiomatization in 1986, the EC and its many variations (dialects) have been typically subject to two forms of reasoning:

- *deductive reasoning*, to infer the validity interval of fluents starting from an EC specification and a (complete) execution trace.
- *abductive reasoning*, to infer a possible execution trace that can be exhibited by the system according to the corresponding EC specification, starting from a given initial state and leading to a desired goal state.

In the following, we discuss why deductive reasoning must be improved towards the adoption of the EC for runtime monitoring, and show two possible axiomatizations that are suited to this verification task.

3.1 Monitoring, Deductive and Reactive Reasoning

In the monitoring setting, we are interested in reasoning upon a given set of event occurrences produced by the running system, to provide the information about all fluent validity intervals to the stakeholders. This makes deductive reasoning a suitable candidate. However, the main peculiarity of a running system is that it produces a dynamically growing execution trace, which is continuously extended with new event occurrences. Technically:

- the goal is typically fixed, and asks for all fluents validity intervals that hold according to the specification and the (partial) trace accumulated so far;
- the narrative is continuously *updated*, adding new event occurrences to the previous partial trace.

According to the *commonsense law of inertia*, a narrative update typically affects only a small number of fluents, while the majority of them remains unchanged. Therefore, after a new event occurrence is processed, we expect that the corresponding set of inferred validity intervals will be only slightly different from

the former set, computed before the acquisition of such an event. Unfortunately, deductive reasoning does not take advantage from this fact, and recomputes the whole result from scratch every time a narrative update occurs. This source of inefficiency was firstly pointed out by Chittaro and Montanari in [8]. To overcome it, they advocated the need of smarter EC axiomatization able to progress from a result computed previously to a new one by suitably changing it as new events are acknowledged. In particular, they proposed a Prolog-based axiomatization called Cached Event Calculus (CEC), which exploits assert and retract predicates to cache and revise the maximal validity intervals (MVIs) of fluents, where each MVI stores the maximal time intervals inside which a fluent uninterruptedly holds.

3.2 Reactive Event Calculus

The Reactive Event Calculus (REC) stems from the idea of caching the currently computed result for future use. Every time a new event (or set of events) is delivered to the reasoner, it *reacts* by extending the narrative and by consequently extending and revising the previously computed result so as to make it consistent with this new information.

From the technical point of view, we have studied two different axiomatizations of REC. The first axiomatization is based on Abductive Logic Programming (ALP), and exploits a form of "caching by abduction" that is associated to an underlying declarative semantics; the axiomatization is composed of backward (Prolog) clauses and forward implications called *integrity constraints*.

The second axiomatization is Prolog-based. It is a lightweight form of CEC suitable to deal with application domains where events are processed following the order in which they have been generated, or where out-of-order events refer to the recent past (i.e., they are received with a small delay). Typical application domains enjoying this requirement are business processes and (web) services.

Both approaches share the same underlying philosophy of CEC, i.e., caching and manipulation of fluents' MVIs. In particular, they employ the $mvi/2$ predicate listed in Table 1 to reify, generate and cache the maximal validity intervals of fluents: $mvi(f, [t_s, t_e])$ means that f uninterruptedly holds from time t_s (excluded) to time t_e (included). We distinguish between two classes of MVIs: closed MVIs, i.e. MVIs that have a ground value for both its extreme time points, and open MVIs, i.e. MVIs whose termination time is unknown. Furthermore, both axiomatizations formalize the simple EC dialect shown in Table 1, and accept a domain-dependent theory specified in Prolog. Therefore, all specifications written in such an EC dialect can be seamlessly exploited with both approches.

In the following, we review the two axiomatizations and highlight their respective strengths and weaknesses.

3.3 SCIFF-Based REC Axiomatization

SCIFF [1] is an ALP-based framework developed for monitoring the running execution traces of an event-based system, checking whether they comply with a set

of forward "event-condition-expectation" rules. In particular, these rules relate the occurring events and their corresponding conditions to positive and negative expectations. Whenever the current execution trace makes the body of one such rule true, the expectations contained in its head are generated. An execution trace is then flagged as compliant if it eventually satisfies all the expectations, i.e. it contains a matching event occurrence for each positive expectation, and does not contain matching events for the negative ones.

The framework is highly expressive. Events can be arbitrary terms containing variables, and such variables can be subject to Constraint Logic Programming (CLP) constraints [14]. They can also be used inside predicates that are defined in a Prolog knowledge base formalizing the static aspects of the system. Occurred events are represented using the special functor $\mathbf{H}/2$, where $\mathbf{H}(e, t)$ means that event e occurred at timestamp t. Timestamps are explicitly modelled relying on integer or real domain. Roughly speaking, each positive expectation, denoted by $\mathbf{E}/2$, is existentially quantified, reflecting the intuition that one matching event occurrence suffices to fulfil it. Conversely, negative expectations, denoted by $\mathbf{EN}/2$, are universally quantified, because they forbid the presence of all matching event occurrences.

From the technical point of view, a SCIFF specification is represented as an abductive logic program. An abductive logic program is a triple $\langle \mathcal{KB}, \mathcal{A}, \mathcal{IC} \rangle$, where $\mathcal{KB}$ is a partially specified logic program, $\mathcal{A}$ defines the *abducible* predicates, i.e. predicates that are left unspecified in $\mathcal{KB}$, and $\mathcal{IC}$ is a set of *integrity constraints*, used to implicitly isolate the acceptable abductive explanations that can be formulated (hypothesized) to complete the clauses in $\mathcal{KB}$.

In SCIFF, integrity constraints consist "event-condition-expectations" rules, and expectations are abducible predicates. Given an execution trace of the system, modelled as a set of completely ground predicates $\mathbf{H}$, expectations are hypothesized according to such rules. A further hypotheses-confirmation step is then executed in order to check whether they are fulfilled by the event occurrences of the trace or not. Two different semantics for fulfilment are defined, depending on whether the analysed trace is complete or not. In the first case, a "complete knowledge" assumption is made: every positive expectation is declared fulfilled if and only if a corresponding matching event is contained in the trace, and every negative expectation is declared fulfilled if and only if no corresponding matching event exists in the trace. In the second case, further events may still occur to extend the trace, and consequently only the fulfilment of positive expectations and the violation of negative expectations can be checked: a currently violated positive expectation could be still fulfilled if a matching event occurs, and a currently satisfied negative expectation could be still violated if matching event occurs. Further information about the declarative and operational semantics of SCIFF can be found in [1].

In [7], we have shown how REC can be axiomatised on top of SCIFF. We report such an axiomatization here for the sake of completeness. The idea behind the axiomatization is the following:

- **mvi** is modeled as an abducible predicate (to distinguish abducible predicates from the "normal" ones, we use boldface).
- The extreme time points of each **mvi** predicate are bound to two corresponding "internal" events modeling its activation (declipping) and completion (clipping). In order to distinguish the "external", actual event occurrences and such internal ones, we assume without loss of generality that external events are wrapped inside an $ev/1$ term; e.g., $\mathbf{H}(ev(open\_loan(c, id)), 5)$ represents that a loan has been opened at time 5 for customer c, and its identifier is id. Such events correspond to the $happens/2$ predicate in the EC ontology.
- Each of such internal events is generated (i.e., abduced) when an external event occurs and, according to the domain-dependent knowledge base, such an event clips or declips a fluent.

Formally, given an EC domain-dependent theory $\mathcal{KB}_{dom}$, the corresponding SCIFF-based REC specification $\text{REC}_S(\mathcal{KB}_{dom})$ is

$$\text{REC}_S(\mathcal{KB}_{dom}) = \langle \mathcal{KB}_{REC}, \{\mathbf{E}/2, \mathbf{EN}/2, \mathbf{H}/2, \mathbf{mvi}/2\}, \mathcal{IC}_{REC} \rangle$$

where $\mathcal{KB}_{REC} = \{(1),(2)\} \cup \mathcal{KB}_{dom}$, and $\mathcal{IC}_{REC} = \{(3),(4),(5),(6),(7)\}$, and rules (1) through (7) are defined in the remainder of the section.

The first rule of $\mathcal{KB}_{REC}$ is used to check whether a given fluent holds at a certain time. In particular, it states that fluent F holds at time T if a maximal validity interval for F has been generated, and its timespan contains T.

$$holds\_at(F, T) \leftarrow \mathbf{mvi}(F, [T_s, T_e]) \wedge T > T_s \wedge T \leq T_e. \tag{1}$$

Such a rule attests that the reification of maximal validity intervals makes query complexity linear in the number of maximal validity intervals, which is one of the major advantages of CEC (and REC) [8]. The second rule of $\mathcal{KB}_{REC}$ models the effects of a special *complete* event, used to alert SCIFF that the monitored trace has reached its termination. The effect is to terminate every possible fluent, thus clipping all the maximal validity intervals that are still waiting for a terminating event.

$$terminates(complete, F, \_). \tag{2}$$

The first integrity constraint of $\mathcal{IC}_{REC}$ provides an expectation-based semantics to the notion of MVI. In particular, it states that if $(T_s, T_e]$ is an MVI for fluent F, then *(i)* F must have been declipped at time T_s, *(ii)* F will be clipped at time T_e, *(iii)* further declipping/clipping events do not occur in between.

$$\begin{aligned} \mathbf{mvi}(F, [T_s, T_e]) \rightarrow\ & \mathbf{E}(declip(F), T_s) \wedge \mathbf{E}(clip(F), T_e) \\ & \wedge \mathbf{EN}(declip(F), T_d) \wedge T_d > T_s \wedge T_d \leq T_e \\ & \wedge \mathbf{EN}(clip(F), T_c) \wedge T_c \geq T_s \wedge T_c < T_e. \end{aligned} \tag{3}$$

The second rule of $\mathcal{IC}_{REC}$ deals with the *initially* EC predicate. In particular, it states that whenever a fluent initially holds, a corresponding declipping event is generated immediately before the beginning of the execution (e.g., at time -1).

$$initially(F) \rightarrow \mathbf{H}(declip(F), -1). \tag{4}$$

The third and fourth integrity constraints deal with the declipping of a fluent, and the consequent generation of a new MVI. First of all, when *(i)* an event Ev occurs at time T, *(ii)* Ev initiates fluent F at time T, *(iii)* F does not hold at that time, then F becomes declipped, i.e., an internal declipping event is generated.[1]

$$\mathbf{H}(ev(Ev), T) \wedge initiates(Ev, F, T) \wedge \neg holds\_at(F, T) \rightarrow \mathbf{H}(declip(F), T). \quad (5)$$

The occurrence of a *declip(F)* event at time T causes the generation of a new open MVI for F, which starts from$T$ and ends sometime in the future.

$$\mathbf{H}(declip(F), T_s) \rightarrow \ \mathbf{mvi}(F, [T_s, T_e]) \wedge T_e > T_s. \quad (6)$$

The last integrity constraint is symmetrical to (5), and deals with the clipping of a fluent. In particular, when *(i)* an event Ev occurs at time T, *(ii)* Ev terminates fluent F at time T, *(iii)* F holds at that time, then F becomes clipped.

$$\mathbf{H}(ev(Ev), T) \wedge terminates(Ev, F, T) \wedge holds\_at(F, T) \rightarrow \mathbf{H}(clip(F), T). \quad (7)$$

The clipping event has the effect of fulfilling the (pending) expectation regarding the termination time of an MVI for F, thus closing the MVI.

3.4 Prolog-Based REC Axiomatization

The REC axiomatization in Prolog is a light-weight version of CEC. In particular, it provides an efficient treatment of events that are processed in the same order in which they occur. For causal EC specifications, namely specifications in which the initiation and termination of fluents only depend on the present or the past, this assumption implies that the acquisition of a new event can only impact on the current MVIs in two ways: either one or more new MVIs are created, or some currently open MVIs become closed. Obviously, a correct axiomatization must also deal with out-of-order events in a proper way. In CEC, this is achieved by reasoning upon the domain-dependent specification so as to infer the changes that are caused by an event occurrence that is processed out-of-order, suitably revising the affected MVIs and then recursively managing the effect of such modifications. In our axiomatization, instead, we manage an out-of-order event (occurring at time T) in a naive way:

1. a roll-back of the cache is executed, removing all the event occurrences recorded after T as well as all the MVIs that have been started or terminated after T;
2. the out-of-order event is stored and its effect calculated;
3. all the event occurrences removed in the first step are re-played.

Similarly to what we have done for the SCIFF-based axiomatization, given a domain-dependent EC specification $\mathcal{KB}_{dom}$, its Prolog-based REC theory is defined as $\text{REC}_P(\mathcal{KB}_{dom}) = \mathcal{KB}_{dom} \cup \mathcal{KB}_{gen}$, where $\mathcal{KB}_{gen}$ is the general axiomatization of the EC ontology described in the following.

[1] In [7], this Axiom is modeled with a slightly different but logically equivalent rule.

$\mathcal{KB}_{gen}$ exposes a set of predicates that can be invoked by the user or by an external application to interact with the reasoner.

First of all, `status(MVIList)` queries the cache and returns the list of all contained MVIs:

```
status(MVIList):- findall(mvi(F,[T1,T2]),mvi(F,[T1,T2]),MVIList).
```

`reset` is instead used to reset the reasoner by emptying the cache.

```
reset:- retractall(happens(_,_)), retractall(mholds_for(_,_)).
```

The `start` predicate is used to start the reasoner, calculating all the MVIs related to fluents that initially hold. This is done by updating the reasoner with a special fictitious start event occurrence happening just before the beginning of the execution (e.g., at time -1) and by stating that such an event initiates all the fluents that initially hold:

```
start:- update([happens(start,-1)]).
initiates(start,F,-1):- initially(F).
```

The `update(ExtTrace)` predicate is the backbone of the reasoner. It takes a set of event occurrences as input, orchestrating other internal predicates to clean and reorganize such events, to process them and determine their effects, and to eventually update the cache.

```
update(ExtTrace):- remove_duplicates(ExtTrace,NExtTrace),
                   order_events(NExtTrace,OTrace),
                   manage_concurrency(OTrace, Trace),
                   update_events(Trace).
```

The first three predicates are used to pre-process the incoming set of events by removing duplicates, order them by ascending timestamp, and managing concurrency. After that, the trace is now organized as a list of lists, where the each inner list contains event occurrences referring to the same timestamp. The last predicate is instead used to effectively process the (reorganized) trace. The processing phase follows the aforementioned naive approach. We comment the core of the processing phase, which focuses on the acquisition, MVIs calculation and storing from a given set of concurrent events, i.e. events happening at the same time.

```
calc_effects([happens(E,T)|Ev]):- assert_all([happens(E,T)|Ev]),
                                  my_setof(F, clip(F,T),S),
                                  close_mvis(S,T),
                                  my_setof(F, declip(F,T),S2),
                                  open_mvis(S2,T).
```

The first line of the definition inserts all the incoming concurrent events in the cache. The second line individuates all the "negative" effects of such events, i.e., infers all fluents that are clipped at time T, where `clip(F,T)` is defined exactly as in the SCIFF-based axiomatization:

```
clip(F,T):- happens(E,T), terminates(E,F,T), holds_at(F,T).
```

The third line focuses then on extracting all the open MVIs related to such fluents, substituting them with corresponding MVIs that are closed at time T. In particular, for each fluent F, the substitution works as follows:

```
close_mvi(F,T):- holds_from(F,T,T1),
                 retract(mvi(F,[T1,inf])), assert(mvi(F,[T1,T])).
```

where `holds_from` is defined as `holds_at`, but also returns the starting time of the MVI that attests the validity of the fluent at the asked time, and open MVIs are modeled by using the special constant `inf`.

The fourth and fifth line of the `calc_effects` predicate are defined in a specular way. The `declip(F,T)` predicate is defined exactly as in the SCIFF-based axiomatization:

```
declip(F,T):- happens(E,T), initiates(E,F,T), not holds_at(F,T).
```

and the creation of a new MVI for fluent F is modeled as follows.

```
open_mvi(F,T):- assert(mvi(F,[T,inf])).
```

When comparing different timepoints, an extended version of the comparison operators is considered, in order to suitably handle the special "inf" value.

3.5 Discussion

The two proposed axiomatizations provide complementary features. We briefly discuss their main strengths and weaknesses, also taking into account their relation with CEC. The main weakness of REC_P and CEC is that they employ `assert` and `retract` predicates, which are not associated, in Prolog, to an underlying declarative semantics. This makes it difficult to prove formal properties of the calculus. A possible line of research for overcoming this issue is to investigate whether logic programming-based approaches that provide the underlying semantics (such as for example transaction logic [3]) can be adopted for the reactive/cached EC.

Differently from REC_P and CEC, REC_S consists of a purely declarative axiomatization on top of SCIFF, and therefore it directly inherits all the formal properties of SCIFF, such as soundness as well as completeness and termination (under the assumption of acyclicity of the domain-dependent theory [7]). Beside such general properties, we have also formally investigated the class of causal domain-dependent EC specifications, where each fluent initiation/termination only depends on conditions that are tested on current or past timestamps. For these specifications, if events are processed in the same order in which they occur, it is guaranteed that: *(i)* the proof tree generated by the SCIFF proof procedure has only one successful derivation; *(ii)* when new events are processed, MVIs are not revised but only extended (*irrevocability* property [7]).

On the negative side, REC_S does not scale well for large execution traces, mainly because of the complexity of the SCIFF proof procedure. In particular,

two sources of complexity cause a degradation of the reasoner's performance. First of all, REC_S massively relies on the underlying CLP solver, used to manage the consistency of the MVIs' start and termination timestamps. Secondly, negative *holds_at* predicates are rewritten by the proof procedure in such a way that they introduce mutually exclusive decision points in the proof tree [7]. This could require a deep exploration of the search space before finding the right solution.

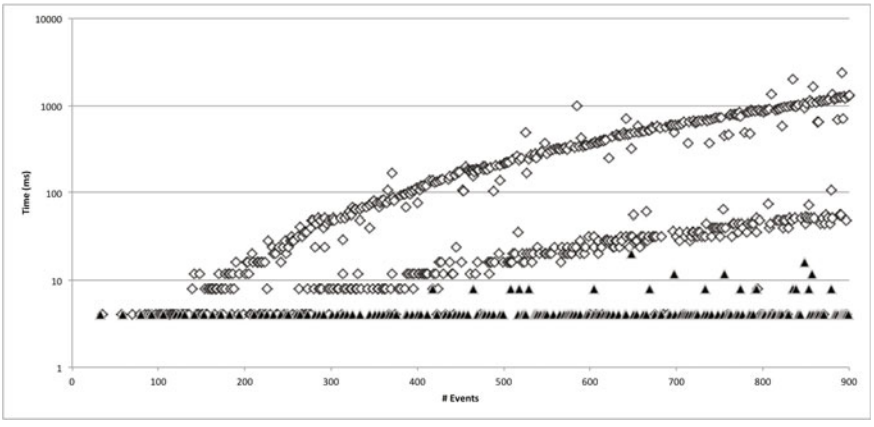
REC_P shows instead good performance, especially for causal specifications and ordered events. Its performance downgrades when an out-of-order event is processed; the degradation is proportional to the number of already processed events that come after such an out-of-order event. The extreme case is the one in which REC_P must process an event whose timestamp comes before the timestamps of all the already processed events: in this case, REC_P recomputes the result from scratch, exactly like a backward, goal-oriented reasoner. However, in the application domains of our interest (see Section 4) either it can be assumed that events are always processed following their execution ordering, or out-of-order events only appear locally, i.e., they were generated recently.

In order to evaluate the performance of REC_P under the assumption of having only "local out-of-order events", and compare its behaviour with the one of CEC, we have set up the following benchmark. We consider a fixed minimal domain-dependent specification composed of 10 rules, and including 4 different event types and 3 different fluents. Each test is generated by constructing a trace starting from two parameters: R, the core trace length, and K, the out-of-order timespan. More specifically, a set of R events is incrementally generated, passing each new event occurrence to the reasoner. Each new event occurrence is generated by randomly choosing an event type among the 4 mentioned in the specification, and by determining its position as follows. Let P denote the last position of the current trace; then the position of the new event is randomly picked between $P - K + 1$ and $P + 1$. This means that if $K = 0$, then events are processed in the order they occur. If $K = 1$, then every new event comes last or second last, and so on. In order to study the trend of the reasoners, this chunk of R events is then repeated at increasing time windows.

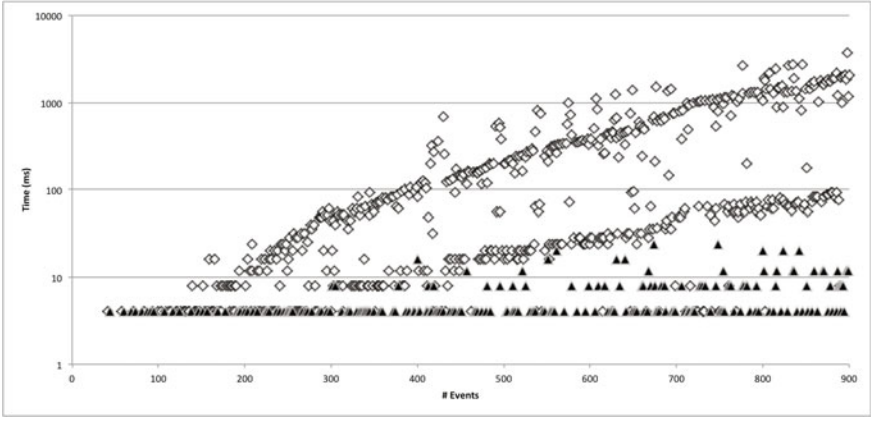
Two tests of this benchmark are shown in Figure 2. On the y-axis the charts present the amount of time needed to process the i-th event. It is easy to see that REC_P outperforms CEC of several orders of magnitude. Furthermore, as expected the sparsity of the reported values increases as K increases.

4 Application Fields

Within the field of distributed systems, new architectural paradigms have recently emerged, in which autonomous, heterogeneous entities collaborate and interact to achieve complex goals. In such paradigms, the focus is shifted from the single participants to their interaction and cooperation. Single participants are viewed as black boxes, thus allowing components' heterogeneity. The interaction instead is typically modelled in terms of (the happening of) *events*, which



(a) R=150, K=30



(b) R=150, K=60

Fig. 2. Comparison benchmarks between CEC (diamond, light) and REC_{P} (triangle, dark). Tests have been carried out with YAP Prolog 5.1.3 on an Intel core® i7 Q 740-1,73GHz architecture, 3GB Ram.

characterize the observable dynamics of a specific systems execution. Notable examples can be observed in many different application domains, such as Business Process Management (BPM), Clinical Guidelines (CG) and Care-flow protocols, Service Oriented Architectures (SOA), and Multi-Agent Systems (MAS).

Within this context, an important line of research is studying new paradigms and modelling languages able to suitably mediate between *compliance support* and *flexibility*. Compliance monitoring of the execution is used to regulate the participants behaviour, so as to be sure the system as whole will exhibit certain properties. Flexibility is intended as the freedom of choice left to the interacting entities, allowing them to take opportunities, to exploit their competences and knowledge, and to cope with highly dynamic environments (thus enhancing overall robustness).

To this end, declarative and open approaches are a suitable perspective [20]. Declarative, to help the modeller in the specification of the behavioural constraints that must be respected, without explicitly enumerating a-priori a predetermined and rigid set of acceptable flows. Open, to leave participants free to execute actions that are not explicitly forbidden, which is a fundamental requirement inside environments where the participants knowledge is incomplete and constantly subject to updates as new events occur. Given these considerations, Event Calculus is a viable answer: being logic-based, it is naturally declarative. Moreover, EC has been introduced exactly for dealing with events, and to reason on how the happenings affect some properties.

In the following Sections, we will briefly recap our previous experiences and results on using EC as a modelling language in different application domains.

4.1 Business Process Management

To achieve their business goals, companies often resort to a variety of means. Among them, there is the careful analysis and organization of the process behind the "production of the goods/services". As defined in [34]:

> "A Business Process consists of a set of activities that are performed in coordination in organizational and technical environments. These activities jointly realize a business goal. Each Business Process is enacted by a single organization, but it may interact with BPs performed by other organizations."

Complex business processes involve also the participation of domain experts, within some technical environment that supports the interaction. Such processes are often characterized by a high degree of unpredictability: e.g., it is impossible to foresee a-priori which behaviour will be exhibited by the interacting entities in a given situation. Hence, automated reasoning techniques must be developed to constantly monitor a running execution of the process and promptly detect deviations between the actual behaviour and the prescriptions of the model.

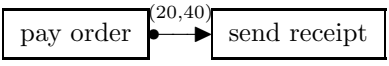
In [20,19] a framework called CLIMB (Computational Logic for the verIfication and Modelling of Business Constraints) is proposed, which combines graphical modelling, logic-based languages and automated reasoning techniques for the rigorous specification of open and declarative interaction models, providing support during their entire life cycle.

CLIMB adopts an extended version of the graphical ConDec [22] notation for the specification of behavioural business constraints. ConDec is a graphical constraint-based language for modelling flexible business processes. In CLIMB, ConDec is extended with quantitative temporal constraints (such as delays, deadlines and latencies), non-atomic activities, data and data-aware conditions. All the constructs of (extended) ConDec are translated into EC theories, making it possible to apply run-time verification and monitoring base on REC. Also, an EC-based operational support provider has been implemented inside ProM 6, the latest version of the widely acknowledged ProM process mining framework[2].

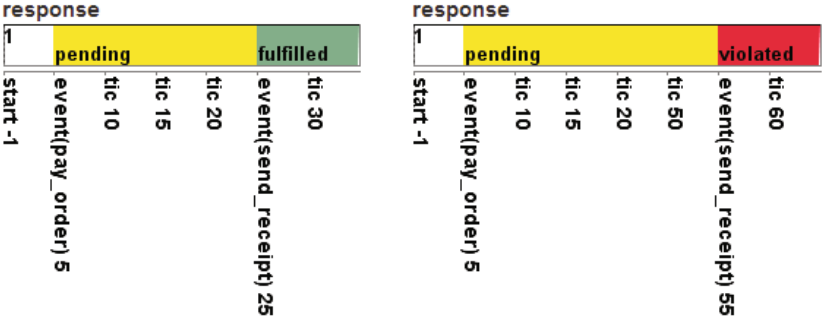
[2] See `www.processmining.org`

The interested reader can refer to [21] for a complete and detailed discussion on how ConDec has been translated in EC, together with its application to the monitoring of some real, complex business cases.

A simple example of a business constraint expressed using the ConDec language is the following:



Given two business activities, named "pay order" and "send receipt" respectively, the business constraint states that upon the execution of the payment, a receipt must be sent between 20 and 40 time units after the payment has been done.



(a) Constraint satisfied (b) Constraint violated

Fig. 3. A simple example of business constraint monitoring with jREC

Such constraint (called *response* in the ConDec terminology) can be easily modelled by means of the EC. In Figure 3a the output of jREC [3] is shown, when the business constraint discussed above is matched against a set of events that respect the constraint. At time instant 5 the "pay order" activity is executed, and then the constraint *response* becomes pending, awaiting for the "send receipt" activity to happen between time instant 25 and 45. At time 25 the receipt is sent, and the constraint becomes *fulfilled*.

In Figure 3b instead an example of the violation of the constraint is presented. Again, at time instant 5 the activity "pay order" is executed, but this time the receipt is sent at time instant 55, well over the established deadline of 45. As a consequence, the *response* constraint is detected as being violated.

4.2 Service Oriented Computing

Run-time verification and monitoring are matter of the greatest importance in the context of service-oriented systems. Indeed, even in the presence of static

[3] jREC is our REC implementation, equipped with a Java-based Graphical User Interface. The GUI can work seamlessly with both REC_P and REC_S. The interested reader can download jREC (running with REC_P) at the url: `http://www.inf.unibz.it/~montali/tools.html`

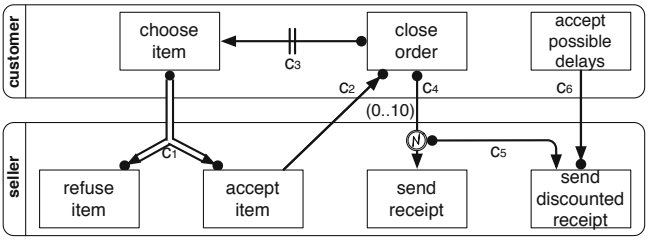
information about the services' behaviour, it is not guaranteed that the behavioural interface exposed by a service effectively corresponds to its internal implementation. This problem is further enhanced by the good level of standardization reached within SOA-oriented systems: while the use of heterogeneous components is definitely easier, inspecting third-party components is not possible any more. Thus, guaranteeing the right behaviour becomes a hard task. A possible way to handle this issue is to continuously check the behaviour of the services during their execution.

In [18], we exploited the REC framework to dynamically check whether the messages exchanged within a service composition comply with a given choreography. Thanks to the REC adoption, the framework is able to constantly provide a feedback that shows how the occurring events impact on the choreography constraints; furthermore, the detected violations can be explicitly represented (i.e., reified) and used to generate alarms or trigger corresponding compensation mechanisms. The starting point is DecSerFlow, a graphical language based on ConDec, but aimed directly at the modelling of services. Again, we provide a semantics of the graphical constraints in terms of EC, and exploit jREC for monitoring.

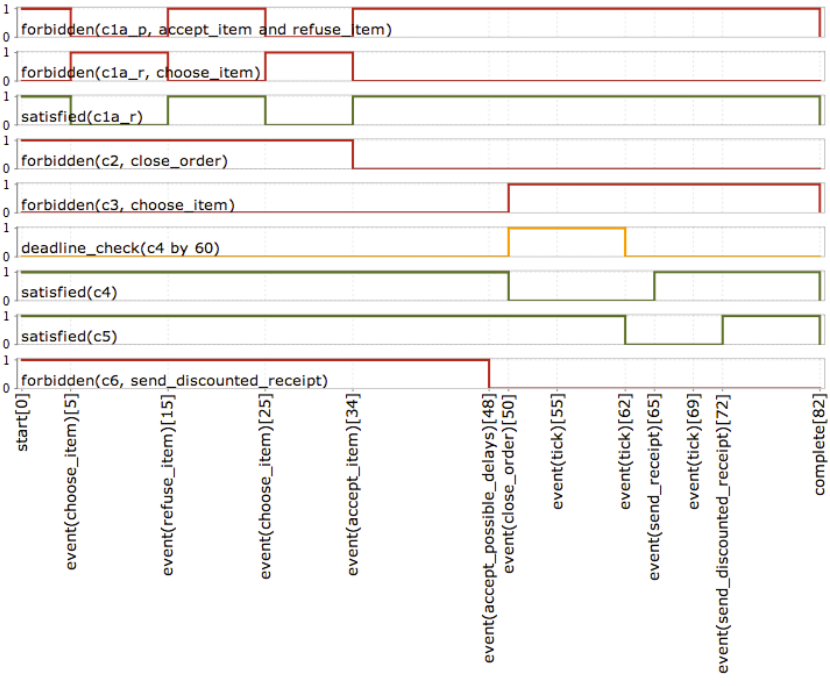
In Figure 4, taken from [18], it is shown a set of constraints over a choreography (Figure 4a). A customer chooses one or more items, and a seller confirm (or refuse) each chosen item. Upon the order finalisation, the seller issues a receipt within a pre-established deadline. A fixed discount is granted if the customer accepts some delays. Anyway, in case of a delay, the seller grants a certain discount. In Figure 4a few constraints are shown. For example, constraint c_1 captures the intuition that every choose item activity must be followed by an answer from the seller, either positive or negative, and that no further item can be chosen until then. When an order has been closed, the seller is expected to send a receipt by at most 10 time units (constraint c_4). Upon the violation of constraint c_4, the seller must deliver a discounted receipt (constraint c_5).

Each DecSerFlow constraint has been modelled by means of one or more particular fluents, depending on the type of constraint. For example, a fluent *satisfied(c1)* is initially true; the intended meaning is that at the beginning of the interaction constraint c_1 is *satisfied*. Upon the event of the customer choosing an item, the fluent is clipped, meaning that c_1 is not satisfied: only when the seller will accept (or refuse) the fluent, it will be declipped, thus returning *satisfied*. In Figure 4b it is possible to observe the fluents and how their states change upon the happening of the events, drawn at the bottom of the Figure. Note that c_1 is also about prohibitions. In particular, it is about the seller accepting and refusing the same item at the same time, and the prohibition for the customer to proceed with another order until the seller has accepted or refused the previous one. For these two aspects, a particular fluent *forbidden(anAction)* shows when a prohibition holds for action *anAction* to be executed.

Summing up, in this domain we have exploited the EC to represent the state of the constraints (by means of fluents), while the activity executions are the events that affect the fluents. Note that the fluents implicitly capture some notion



(a) A DecSerFlow choreography fragment.



(b) Fluents trend generated by jREC, with the time spent for reacting to each happened event.

Fig. 4. A set of constraints about a choreography, together with the generated fluents and the time spent to perform the monitoring. Figures taken from [18].

of satisfaction/violation of the constraints, depending on the observed course of events. Taking into consideration also time-related aspects, it is possible to evaluate if the whole interaction has been compliant w.r.t. the choreography constraints, by simply looking for violated constraints. REC, by showing how fluents changes, provides such information, thus supporting the monitoring of the execution.

4.3 Multi-Agent Systems and Commitments

In the field of open Multi-Agent Systems (MAS), an important issue is related to the specification and verification of interaction protocols, with particular emphasis on social, normative and contractual aspects. A widely adopted approach to the modelling of MAS is the so-called *social approach*, where the allowed interactions and the semantics of the events are given in terms of the consequences that such events have at the social level of the MAS. A notable example of this approach is the one based on *commitments*, firstly introduced by Singh in [29]. Commitments capture the mutual obligations established between the interacting agents during the execution. A commitment states that a debtor agent is bound towards a creditor agent, and must bring about some property to correctly discharge the commitment. A debtor becomes committed towards a creditor as the consequence of its own (communication) acts. The theory of commitments specifies the states through which a commitment goes as certain events (i.e., communication acts) occur.

EC has been exploited for modelling commitments for the first time in [35]. In that work, the authors provide the commitments with a semantics in terms of the EC; at the same time, the authors exploit the EC also for reasoning about the current state of a system, in terms of the commitments that are currently active, satisfied or violated. Taking inspiration from that seminal work, we have further investigated the possibilities offered by the use of the EC for modelling the commitments. A first extension has been presented in [5], where the well-known notion of commitments has been extended with the temporal notion of deadline. This opened up the possibility of modelling commitments in which the debtor agents must bring about a given property within a maximum time instant. Moreover, the formalization presented in [5] supports also MAS in which agents can dynamically take part to or leave the interaction, and in which there is no (a priori) fixed and statically known set of agents.

In [31] the theory of commitments is further extended embracing different possible time-aware commitments. Two classes of time-aware commitments are introduced: *existential* commitments, whose property must be brought about inside a given time interval, and *universal* commitments, whose property must be continuously maintained true during a given time interval. Such powerful extensions have been possible thanks to the use of EC as modelling language.

An important difference between the approach presented in [35], and our works in [5,31], is on how EC has been used to represent commitments. In the former approach fluents directly represent the commitments, hence if a fluent holds then the corresponding commitment is established (*active*). The latter approaches instead exploit fluents to represent the possible commitment states: a fluent is about a commitment being in a certain state. Communication acts among agents are the events that trigger comitment's state transition and, with respect to our EC-based formalisation, the clipping/declipping of the corresponding fluents. As already stated, the latter approach also extends the commitment notion by allowing temporal related properties, with existential/universal quantification over temporal periods.

To illustrate the capabilities of EC used for modelling time-aware commitments, let us present an example taken from [6], where REC has been exploited to monitor commitment-based multi agent systems. In the example, a business contract is considered, about the mutual obligations between a customer and an agency when a car is rented. The following statements are included in the contract:

(S1) the customer is committed of taking the car back to the car rental agency within the agreed number of days;
(S2) the agency, in turn, guarantees that the rented car will not break down for the first three days;
(S3) if the rented car breaks down before the third day has elapsed, the agency promises a "1-day" immediate replacement;
(S4) in case of a car replacement, the customer receives two more rental extra-days for free.

Figure 5 shows the outcome of jREC when used for monitoring an agent interaction that should follow rules S1–S4. In particular, Figure 5 shows both fluents representing the commitment states as well as fluents. For instance, the facts that car `bo123` is available in the agency `ag`, and it is working properly, are represented by the fluents `in_agency(...)` and `great_car(...)`. When the customer `ian` rents the car at time 10, `bo123` is no more in the agency, i.e. the corresponding fluent is clipped. The observed events in the particular interaction are drawn at the bottom of the Figure, together with the time-stamp the events happened.

Different states of the same commitment are represented with different fluents. To easy the comprehension of how commitments change their state, in Figure 5 all the fluents related to the same commitment are drawn on the same line. Hence, upon the event of `rent`, agency `ag` becomes committed towards `ian` to guarantee that car `bo123` will be a `great_car` from time 10 to time 13, persistently along the time period. In Figure 5 such property is wrapped around a `u(...)` term, indicating that the property must hold *universally* along the time period. Looking at the happened events in this particular example, we note that at time 11 car `bo123` breaks down. This leads to the establishment of a new commitment `car_replaced` about the rental agency to substitute the car, while the previous commitment moves from the `active` to the `compensated` state.

Summing up, jREC provides a monitor of how commitments are evolving: such information can be used to detect violations, to determine possible culprits, and to provide an overall judgement about the success or failure of the interactions. Moreover, thanks to its reactive flavour, jREC can be used to inspect, during the interaction, obligations and duties of all the involved parties. It is worthy to note that many other approaches have been investigated for monitoring open societies of agents. In particular, in [2] Sergot and colleagues have applied the EC to the specification of roles (and other deontic notions such as power, permission, obligation, etc.): however, they aim to the simulation rather than run-time monitoring, and their tools are based on the original EC framework. Anyway, the results presented in [2] are complementary to the commitment paradigm,

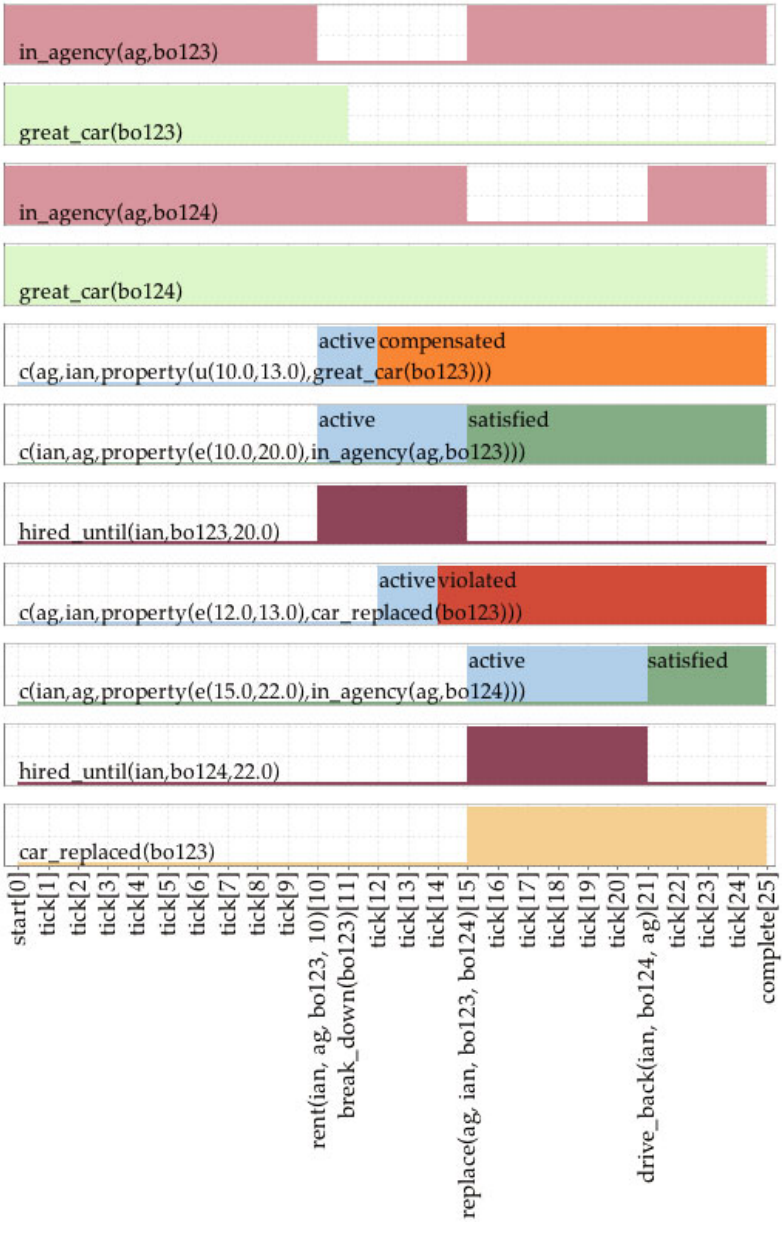


Fig. 5. Sample outcome shown by jREC when monitoring a commitment-based MAS [6]

and a possible future work could be about their integration. Torroni et al. [32] present a retrospective on the REC in the context of Multi-Agent Systems and Commitments, also including a brief survey on recent work by other authors, which is closely related to [31].

4.4 Computerized Clinical Guidelines

Clinical Guidelines (CGs) are, in the definition of the MeSH dictionary, "work consisting of a set of directions or principles to assist the health care practitioners with patient care decisions about appropriate diagnostic, therapeutic, or other clinical procedures for specific clinical circumstances". They are usually defined by means of natural language, and they provide information on how to deal with patients. In particular, CGs quite often provide two types of knowledge: a procedural one, about protocols to be applied to the patient, and often represented in terms of workflows; and a declarative knowledge, in the form of rules, for managing circumstances that would not fit within the procedural part. Moreover, CGs are defined making three important assumptions: (1) the patients are *ideal*: the CG is about an abstract patient suffering exactly the disease the CG is about; (2) physician executing the CG are also *ideal*: they always have all the required competencies; and (3) the context is ideal too: any resource prescribed by the CG is presumed to be readily available.

Given such assumptions, it comes with no surprise that CG executions might differ from the CG model. Indeed, it is the physician that, exploiting her basic medical knowledge, interprets and adapts the CG model to the patient, on a per-patient base. For this reason, monitoring the CG execution w.r.t. the CG model is quite important: the goal is not about evaluating the physician capabilities, but rather on understanding how the CG model fits with the environment, the patients and the physician involved in the CG execution.

Monitoring the execution of a CG requires of course to provide a model of the CG itself. However, CGs comprise procedural knowledge as well as declarative knowledge. In our view, EC has been a very suitable and promising candidate for this task. Indeed, EC implicitly supports the definition of declarative knowledge bases. A previous work [10] proved EC to be powerful enough to represent workflows, and to support reasoning about them. Hence, in [4], we have presented a first prototype based on EC, where, it is possible to encompass in a unified framework all the knowledge provided by a CG. More importantly, we have exploited EC to reason about the CGs models and their execution, captured in the form of a log of happened events: once defined a domain-dependent notion of conformance, we have been able to evaluate different executions and to provide an overall assessment of how the CG model fits the real courses of events.

5 Conclusions

The Event Calculus has been proposed in 1986: since then, it has been widely adopted in a variety of application fields and domains. Despite being 25 years old, it is still subject of research activities. Many features have determined its success, such as its declarative nature, its great simplicity, its overcoming of the frame problem, and the possibility to directly implement it in terms of a Prolog program, just to cite a few of them.

Recently, the advent of ICT solutions heavily based on distributed components has raised the necessity for monitoring applications. Monitoring means to

reason about the state of such systems, and how happening events affect their states. Thus, EC is a strong candidate as a formal framework for supporting the monitoring task. However, some peculiarities of the monitoring setting do not immediately adapt to the usual EC formalization.

In this paper we have provided an overview of our research activity about the use of the EC in the monitoring setting. We have presented a reactive version of EC, named REC, addressing typical characteristics of the monitoring task. In particular, we have discussed two different implementations of REC, each one with its own strengths and weaknesses. Besides developing new implementation of the EC, we have conducted an intense research activity on applications of EC/REC in various domains. We have discussed some of such domains, and briefly shown how EC has been successfully exploited for doing monitoring.

Future research activities will be conducted along two directions. Firstly, we will develop our REC implementation, aiming at better performances, and to assess the formal properties, possibly unifying the two different prototypes. Indeed, applications domain such as SOA are demanding high performances, and monitoring is a task that requires performances and scalability. Also formal properties are a strong requirement, to provide trusted monitoring tools. The second research direction will be about the application domains: in some of them, such as in Clinical Guidelines for example, our work is still preliminary.

Afterword

Some of the authors of this work had the fortune to meet Marek in person. This volume gives us a great opportunity to express our gratitude to him. Marek is not only a great mind, but also a very amusing and entertaining companion. We could enjoy Marek's engaging explanations and witty comments, both in personal communications and from an audience, such as at his opening address of the 2000 Computational Logic conference, or at his tutorial on Norms and Institutions for the 2003 AI*IA conference. Marek is a careful listener and a great communicator. He is extremely sharp and to the point when elaborating on scientific matters, and has a wonderful sense of humour. A conversation with Marek is never boring. This shows a talent, but also his respect for his interlocutor. We are grateful to Marek because he lives by his principles and he inspires us to do the same: to always keep an open mind, to focus on core questions, and to care about the audience. The same attitude Marek shows in personal communications, he also shows in his writings. A most insightful document we came across recently is his introduction to a celebratory volume in honour of Bob Kowalski [25]. In that piece, Marek also talks about the early days of the Event Calculus. That strand of research inspired many others, as it did inspire us, and it still does, after more than a quarter of a century. So thank you, Marek, we hope you enjoy reading this chapter.

Acknowledgements. The authors would like to thank the DEIS DEPICT Project of the University of Bologna that partially sponsored this research. Marco Montali has been partially supported by the EU Project FP7-ICT ACSI (257593). We wish also to thank Fabio Ciotoli for the experimental benchmarks, carried out as part of his MSc project.

References

1. 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), 1–43 (2008)
2. Artikis, A., Pitt, J., Sergot, M.: Animated specifications of computational societies. In: Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent systems (AAMAS), pp. 1053–1061. ACM, New York (2002)
3. Bonner, A.J., Kifer, M.: Results on Reasoning about Updates in Transaction Logic. In: Kifer, M., Voronkov, A., Freitag, B., Decker, H. (eds.) Dagstuhl Seminar 1997, DYNAMICS 1997, and ILPS-WS 1997. LNCS, vol. 1472, pp. 166–196. Springer, Heidelberg (1998)
4. Bottrighi, A., Chesani, F., Mello, P., Montali, M., Montani, S., Terenziani, P.: Conformance Checking of Executed Clinical Guidelines in Presence of Basic Medical Knowledge. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) BPM Workshops 2011, Part II. LNBIP, vol. 100, pp. 200–211. Springer, Heidelberg (2012)
5. Chesani, F., Mello, P., Montali, M., Torroni, P.: Commitment Tracking via the Reactive Event Calculus. In: Boutilier, C. (ed.) Proc. of the 21st Intl. Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 91–96 (2009)
6. Chesani, F., Montali, M., Mello, P., Torroni, P.: Monitoring Time-Aware Social Commitments with Reactive Event Calculus. In: Proceedings of the 7th International Symposium From Agent Theory to Agent Implementation, AT2AI (2010)
7. Chesani, F., Mello, P., Montali, M., Torroni, P.: A logic-based, reactive calculus of events. Fundam. Inform. 105(1-2), 135–161 (2010)
8. Chittaro, L., Montanari, A.: Efficient handling of context-dependency in the cached event calculus. In: Proc. of TIME 1994 - International Workshop on Temporal Representation and Reasoning, pp. 103–112 (1994)
9. Chittaro, L., Montanari, A.: Temporal representation and reasoning in artificial intelligence: Issues and approaches. Annals of Mathematics and Artificial Intelligence 28(1-4), 47–106 (2000)
10. Cicekli, N.K., Cicekli, I.: Formalizing the specification and execution of workflows using the event calculus. Information Sciences 176(15), 2227–2267 (2006)
11. 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), 99–129 (2005)
12. Fernandes, A.A.A., Williams, M.H., Paton, N.W.: A Logic-Based Integration of Active and Deductive Databases. New Generation Computing 15(2), 205–244 (1997)
13. Huhns, M.N., Singh, M.P.: Service-oriented computing: Key concepts and principles. IEEE Internet Computing 9(1), 75–81 (2005)
14. Jaffar, J., Maher, M.: Constraint logic programming: a survey. Journal of Logic Programming 19-20, 503–582 (1994)
15. Kowalski, R.A., Sergot, M.: A logic-based calculus of events. New Generation Computing 4(1), 67–95 (1986)
16. Mahbub, K., Spanoudakis, G.: Run-Time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience. In: Proeedings of the 3rd IEEE International Conference on Web Services (ICWS), pp. 257–265. IEEE Computer Society Press (2005)
17. McCarthy, J., Hayes, P.J.: Some Philosophical Problems From the StandPoint of Artificial Intelligence. Machine Intelligence 4, 463–502 (1969)

18. Montali, M., Chesani, F., Mello, P., Torroni, P.: Verification of Choreographies During Execution Using the Reactive Event Calculus. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 55–72. Springer, Heidelberg (2009)
19. Montali, M.: Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach. Ph.D. thesis, University of Bologna (2009)
20. Montali, M.: Specification and Verification of Declarative Open Interaction Models. LNBIP, vol. 56. Springer, Heidelberg (2010)
21. Montali, M., Maggi, F.M., Chesani, F., Mello, P., van der Aalst, W.M.P.: Monitoring business constraints with the event calculus. Tech. rep., LIA, University of Bologna (2010); currently submitted to ACM TIST
22. Pesic, M., van der Aalst, W.M.P.: A Declarative Approach for Flexible Business Processes Management. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
23. Rouached, M., Fdhila, W., Godart, C.: A Semantical Framework to Engineering WSBPEL Processes. Information Systems and E-Business Management 7(2), 223–250 (2008)
24. Sadri, F., Kowalski, R.A.: Variants of the Event Calculus. In: Sterling, L. (ed.) Proc. of the 12th International Conference on Logic Programming (ICLP 1995), pp. 67–81. MIT Press (1995)
25. Sergot, M.J.: Bob Kowalski: A portrait. In: Kakas, A.C., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond. LNCS (LNAI), vol. 2407, pp. 5–25. Springer, Heidelberg (2002)
26. Shanahan, M.: Robotics and the common sense informatic situation. In: Wahlster, W. (ed.) Proceedings of the 12th European Conference on Artificial Intelligence (ECAI 1996), pp. 684–688. John Wiley and Sons (1996)
27. Shanahan, M.: The Event Calculus Explained. In: Veloso, M.M., Wooldridge, M.J. (eds.) Artificial Intelligence Today. LNCS (LNAI), vol. 1600, pp. 409–430. Springer, Heidelberg (1999)
28. Shanahan, M.: An Abductive Event Calculus Planner. Journal of Logic Programming 44(1-3), 207–240 (2000)
29. Singh, M.P.: Social and psychological commitments in multiagent systems. In: AAAI Fall Symposium on Knowledge and Action at Social and Organizational Levels, pp. 104–106. AAAI Inc. (1991)
30. Ten Teije, A., Miksch, S., Lucas, P. (eds.): Computer-based Medical Guidelines and Protocols: A Primer and Current Trends. Studies in Health Technology and Informatics, vol. 139. IOS Press (2008)
31. Torroni, P., Chesani, F., Mello, P., Montali, M.: Social Commitments in Time: Satisfied or Compensated. In: Baldoni, M., Bentahar, J., van Riemsdijk, M.B., Lloyd, J. (eds.) DALT 2009. LNCS, vol. 5948, pp. 228–243. Springer, Heidelberg (2010)
32. Torroni, P., Chesani, F., Mello, P., Montali, M.: A Retrospective on the Reactive Event Calculus and Commitment Modeling Language. In: Sakama, C., Sardina, S., Vasconcelos, W., Winikoff, M. (eds.) DALT 2011. LNCS, vol. 7169, pp. 120–127. Springer, Heidelberg (2012)
33. Weiss, G. (ed.): Multiagent systems: a modern approach to distributed artificial intelligence. MIT Press, Cambridge (1999)
34. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, Heidelberg (2007)
35. Yolum, P., Singh, M.P.: Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. Annals of Mathematics and Artificial Intelligence 42(1-3), 227–253 (2004)