An Event Calculus Formalization of Timed Automata

Nicola Falcionelli, Paolo Sernani,
Dagmawi Neway Mekuria, Aldo Franco Dragoni

Università Politecnica delle Marche, Ancona, Italy
{n.falcionelli, d.n.mekuria}@pm.univpm.it,
{p.sernani, a.f.dragoni}@univpm.it

Abstract. Both the Event Calculus and Timed Automata are two very well known Knowledge Representation and Model Checking techniques. Specifically, the Event Calculus is a logic framework that allows the concept of time to explicitly appear inside formulas, while at the same time avoiding to step outside First Order Logic. It provides the machinery to perform what is known as Commonsense Reasoning, and an ontology to express Domain Knowledge with three main elements: fluents (time-dependent properties), events and timepoints. This ontology is particularly suited to intuitively model intelligent agents' knowledge, thanks also to the powerful reasoning capabilities of the formalism. On the other hand, Timed Automata are a special kind of Finite State Machines, in which state transitions can be also governed by time contraints. One of their main applications is to formally verify and validate Real-Time systems, expressed as Timed Automata networks. This kind of validation has been already successfully used to demonstrate the soundness of several algorithms, via the help of Model Checkers such as UPPAAL. This work proposes a common point among these techniques, or more precisely, a methodology to translate the graph representations of (Discrete) Timed Automata into Event Calculus logic predicates. Additionally, a performance evaluation of a Prolog implementation of this technique is shown, looking for possible relationships between Model Checking and the various Event Calculus reasoning types.

1 Introduction

Logic programming and declarative languages have always been particularly suited to represent graph, list and tree data structures. Algorithms such as traversing, searching, insertion and deletions on these data structures rely on recursion, a typical feature at the core of declarative paradigms. On a parallel track from the operations themselves, representing knowledge in terms of predicates and facts allows to easily and intuitively model even such complex data structures. For example, an instance of an oriented graph can be easily represented by a list of 3-tuples in the form of $< EdgeLabel, NodeOut, NodeIn >$.

Graphs are also the fundamental structures of many mathematical modeling techniques, such as Finite State Machines, Timed Automata, and Petri Networks.

Such techniques are commonly used to model complex dynamical systems, with the goal of providing mathematical abstractions, simulating their execution, or formally proving certain properties. To do such, the common practice is to involve ad-hoc model checkers such as UPPAAL [4]: they usually have their built-in languages, that allow to (graphically) represent, simulate and check the desired model. More specifically, Timed Automata are a more powerful version of Finite State Machines, in which regular state transitions are extended with timing constraints (guards), in order to be able to explicitly describe time-dependent behaviour.

With the goal of establishing a common underlying logic language as a support for those model checking techniques, this work focuses on explaining how (Discrete) Timed Automata can be implemented and simulated within the Event Calculus, a First Order Logic formalism designed to represent how actions affect different properties of a chosen scenario [17]. Being EC one of the few that combine a linear temporal structure (Modal Logic and Situation Calculus have branching time), it has been chosen as the reference language, thanks to its feasible reasoning time/complexity (with the help of caching strategies [13, 6]), good expressiveness/readibility and quite broad community coverage.

The work is organized as follows: sections 2 and 3 provide background for this research, section 4 focuses on explaining the Event Calculus-Timed Automata modeling technique by means of a Prolog implementation, section 5 puts such implementation to the test to analyze how the reasoning performance change with respect to two parameters, and lastly, section 6 draws the conclusions of the paper and outlines the future work.

2 Related Work

The Event Calculus (EC) has been particularly useful as a knowledge representation and reasoning tool in several domains. [8] suggests a way to standardize the definition of (assistive) monitoring rules in EC, providing a graphical representation and showing how reasoning can be performed on them. Such monitoring rules can be potentially modeled as Timed Automata, and thus translated into EC with the technique presented in this work. [9] shows how the EC can be exploited to build intelligent agents, within an underlying Mobile Multi-Agent platform such as MAGPIE. In [14] a way to model workflows and business processes in EC is proposed. Concepts such as AND/OR splits/joins and reasoning strategies are thoroughly explained, and some potential applications are discussed as well. Notably, the use of EC as a programming model for AI in games is discussed in [20], opening for possible connections with Finite State Machines. The presented work also stresses EC's capabilities, widening the spectrum of formal models that can be represented by means of the EC.

Execution time and complexity have always been a major constraint in logic programming and in EC specifically. Even though the Cached-Event Calculus and the Reactive-Event Calculus already improve the performance of plain EC a lot, this is still not enough for most practical purposes. In this direction, [16, 7] propose several EC-machinery integrated indexing strategies that provide a more efficient Knowledge-Base management and query execution. A different approach is the one took in [5], in which, thanks to logic-theory-compiling, part of the EC flexibility is traded off for performance. In this work, the same indexing of [16] is applied to the proposed EC implementation of TA, in order to investigate how it impacts reasoning performance.s

In the field of model checking, EC have been extended to include Modal Logic operators, as shown in [12]. [19] instead presents four several EC reasoning strategies, which can be potentially exploited for Model Checking purposes also with the presented EC-TA implementation. Instead, Timed Automata have been widely used to model Real-Time systems and protocols, with the goal of formally verifying their properties [21]. Notable examples, such as model-based schedulability analysis or mutual exclusion protocols verification, can be found on the UPPAAL (i.e. one of the most widespread Time Automata Model Checker) website [3].

3 Background

This section briefly introduces to the main concepts needed to undestand this work's contribution.

3.1 Real-Time compliant Multi-Agent Systems (RTcMAS)

Having computer systems with stable and predictable behaviour is of vital importance in mission critical applications. In those applications, CPU processes and tasks must guarantee compliance to strict time constraints, which are usually expressed as deadlines for their execution times. To ensure that a set of tasks can be scheduled without having any deadline miss, a whole set of techniques have been developed [10]. Since it would be too hard to ensure compliance to these constraints if the programmer would have to take care of it, Real-Time techniques have been included at an Operating System level. In this way, the Scheduler will be the OS's component responsible to assess schedulability within certain constraints, while the programmer can focus on implementing the task's features. In order for the Real-Time techniques to work, the programmer must provide the Worst-Case Execution Time (WCET) for each task, which is usually a pessimistic estimate obtained statistically by running it many times.

In principle, these techniques can be applied in Multi-Agent Systems (MAS) as proposed in [11]. During MAS negotiation protocols, each agent accepts/refuses/delegates the execution of tasks based on their resource availability and load; so, if the hypothesis of having WCETs is applied, agent-local Real-Time scheduling policies would allow the agent to take such decision in the most rational way.

3.2 Event Calculus

Among a set of possible formalisms, the Event Calculus has been chosen as the reference one for this work, as it combines great expressiveness and flexibility with feasibility of reasoning, intuitiveness/readability and availability (implementation and literature) [5, 19, 2]. More precisely, being a logic formalism for reasoning about actions and their effects in time [17], it is a suitable tool for modeling expert systems representing the evolution in time of an entity by means of the production of events, and to be the core of intelligent agents [9, 16].

In a technical sense, EC is based on many-sorted first-order predicate calculus, known as domain-independent axioms, which are represented as normal logic programs that are executable in Prolog. The underlying time model of EC is linear. EC manipulates fluents, where a fluent represents a property that can have different values over time. The term F=V denotes that a fluent F has value V as a consequence of an action that took place at some earlier time-point and not terminated by another action in the meantime. Table 1 summarizes the main EC predicates. Predicates, functions, symbols and constants start with lowercase letter, while variables start with uppercase letter. Predicates in the text are referenced as predicate/N, where predicate is the name of the predicate and N its arity (e.g. number of arguments).

Table 1: Main Event Calculus predicates

Predicate	**Meaning**
initially(F=V)	The value of fluent F is V at time 0
holdsAt(F=V,T)	The value of fluent F is V at time T
holdsFor(F=V,[$T_{\min}$,$T_{\max}$])	The value of fluent F is V between $T_{\min}$ and $T_{\max}$
initiatesAt(F=V,T)	At time T the fluent F is initiated to have value V
terminatesAt(F=V,T)	At time T the fluent F is terminated from having value V
broken(F=V,[$T_{\min}$,$T_{\max}$])	The value of fluent F is either terminated at $T_{\max}$, or initiated to a different value than V between $T_{\min}$ and $T_{\max}$
happensAt(E,T)	An event E takes place at time T updating the state of the fluents

The domain independent axioms of EC are the following:

$$\begin{aligned}&\text{holdsAt}(F = V, 0) \leftarrow \\ &\qquad \text{initially}(F = V).\end{aligned} \tag{1}$$

$$\begin{aligned}&\text{holdsAt}(F = V, T) \leftarrow \\ &\qquad \text{initiatesAt}(F = V, T_s), \\ &\qquad T_s < T, \\ &\qquad \text{not broken}(F = V, [T_s, T]).\end{aligned} \tag{2}$$

Predicate (1) states that a fluent F holds value V at time 0, if it has been initially set to this value. For any other time T $>$ 0, the predicate (2) states that the fluent holds at time T if it has been initiated to value V at some earlier time point T_s, and it has not been broken on the meanwhile.

$$\begin{aligned}&\text{broken}(F = V, [T_{min}, T_{max}]) \leftarrow \\ &\qquad \text{terminatesAt}(F = V, T), \\ &\qquad T_{min} < T, \\ &\qquad T_{max} > T.\end{aligned} \tag{3}$$

$$\begin{aligned}&\text{broken}(F = V_1, [T_{min}, T_{max}]) \leftarrow \\ &\qquad \text{initiatesAt}(F = V_2, T), V_1 \neq V_2, \\ &\qquad T_{min} < T, \\ &\qquad T_{max} > T.\end{aligned} \tag{4}$$

Predicates (3) and (4) specify the conditions that break a fluent. Predicate (3) states that a fluent is broken between two time points Tmin and Tmax if within this interval it has been terminated to have value V. Alternatively, predicate (4) states that a fluent is broken within a time interval if it has been initiated to hold a different value.

$$\begin{aligned}&\text{holdsFor}(F = V, [T_{min}, T_{max}]) \leftarrow \\ &\qquad \text{initiatesAt}(F = V, T_{min}), \\ &\qquad \text{terminiatesAt}(F = V, T_{max}), \\ &\qquad \text{not broken}(F = V, [T_{min}, T_{max}]).\end{aligned} \tag{5}$$

$$\begin{aligned}&\text{holdsFor}(F = V, [T_{min}, +\infty]) \leftarrow \\ &\qquad \text{initiatesAt}(F = V, T_{min}), \\ &\qquad \text{not broken}(F = V, [T_{min}, +\infty]).\end{aligned} \tag{6}$$

$$\begin{aligned}&\text{holdsFor}(F = V, [-\infty, T_{max}]) \leftarrow \\ &\qquad \text{terminatesAt}(F = V, T_{max}), \\ &\qquad \text{not broken}(F = V, [-\infty, T_{max}]).\end{aligned} \tag{7}$$

Predicates (5), (6) and (7) deal with the validity intervals of fluents. In particular, predicate (5) specifies that a fluent F keeps value V for a time interval going from Tmin to Tmax if nothing happens in the middle that breaks such an interval. Predicates (6) and (7) behave in the same way, but deal with open intervals.

The domain dependent predicates in EC are typically expressed in terms of the initiatesAt/2 and terminatesAt/2 predicates. One example of a common rule for initiatesAt/2 is

$$\begin{aligned}&\text{initatesAt}(F = V, T) \leftarrow \\ &\qquad \text{happensAt}(Ev, T), \\ &\qquad Conditions[T].\end{aligned} \tag{8}$$

The above definition states that a fluent is initiated to value V at time T if an event Ev happens at this time point, and some optional conditions depending on the domain are satisfied.

3.3 JREC

Straightforward implementations of EC [17] have time and memory complexity which are not practical for developing real applications. This is due to the fact that every time the EC engine is queried, the computation starts from scratch, and all fluents validity intervals are calculated again. Cached Event Calculus (CEC), proposed by Chittaro and Montanari [13], tries instead to overcome this inefficiency by giving EC a memory mechanism, and moving computation from query time to update time.

CEC formalizes the concept of Maximal Validity Interval (MVI), that represents a time interval in which a particular fluent holds without being terminated by any event. A fluent is also associated to a list of MVIs, in order to express all the time intervals in which that fluent holds continuously.

Whenever the rule engine is updated (e.g. by inserting a new event occurrence), the fluents' MVIs are calculated, and then stored for further use, allowing incremental computation for following updates. Also, every time a new event is added to the database, CEC manages to compute MVIs only for the fluents that can vary with that event, and does not check the MVIs of those fluents that cannot possibly change, thus avoiding unnecessary computation.

jREC is a reasoning tool implemented in Java and tuProlog that is based on a lightweight version of CEC known as Reactive Event Calculus (REC) [6]. The use of Java has been an important requirement in order to ensure code portability.

jREC consists of three main components:

- The Prolog theory, which represents the actual CEC axiomatization that is loaded into tuProlog;
- The Java engine, which allows to query and update the database without having to interact directly with tuProlog, as well as adding specific domain-dependent theories;
- The Tester, which is a GUI based stand-alone tool for editing theories, visualizing fluents' MVIs and event occurrences, mainly used for prototyping and developing domain-dependent theories.

3.4 Timed Automata

Finite State Machines (FSM) are a very common tool to represent systems with relatively simple behaviours, such as vending machines, turnstiles and traffic lights. They are also useful for a variety of applications like regular expression checking, videogame AI and software engineering. In this latter field, they are expecially suited for formal verification of programs and network protocols, but they are very limited in terms of expressiveness (i.e. being unable to explicitly model time dependencies). For this reason, there is often the need to rely on more powerful techniques, such as Timed Automata (TA). They enrich the FSM semantics and synthax by providing additional constructs and mechanisms that allow to effectively model timed systems, such as:

- a finite set of Clocks. Although all clocks increase with the same speed, they can be set (or reset) individually upon state transitions.
- a finite set of Guards. They are conditions that check clock values put on state transitions, that, if not satisfied, prevent the system from going to a state from another.
- a finite set of State Invariants. Mainly used for Model Checking, their purpose is to ensure progress, by preventing an Automata to be indefinetely stuck in a certain state (reachability analysis).

Reachability analysis is one of TA's main applications. Roughly, it works by finding an Automata's the possible runs (a run is a list of $\langle StateName, ClockValue \rangle$ 2-tuples), and checking if these behave according to certain requirements. For example, they could never (or always) reach a particular state, or be locked in certain sequences of states. Finding all the possible runs is not a trivial task: since the values clocks can hold are continuous, tools such as the Regions of Equivalence are needed to manage the infinite number of possible runs. This and other techniques allow TA's verification problems to be decidable [21].

In this context, TAs have been extensively used to model and verify Real-Time systems, Network Protocols and concurrent algorithms successfully, ensuring important properties such as safety and progress [3]. Such improvements have been possible also thanks to already existing Model Checkers such as UPPAAL [4]. A relevant example can be found in [15], which shows how Schedulability Analysis can be carried out within UPPAAL, modeling Real-Time concepts such as tasks' deadlines, dependencies, periods, and WCETs.

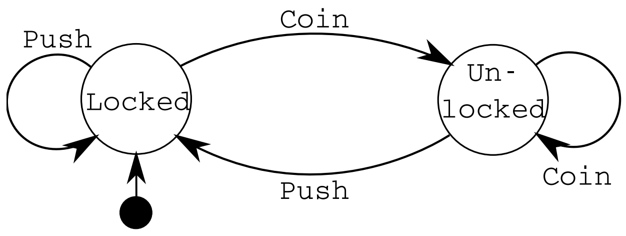

Fig. 1: A simple FSM representing a classical turnstyle mechanism. As long as a coin is provided, the turning bars will remain unlocked. When they are pushed (a person transits), the turnstyle will stay locked until a new coin is inserted.

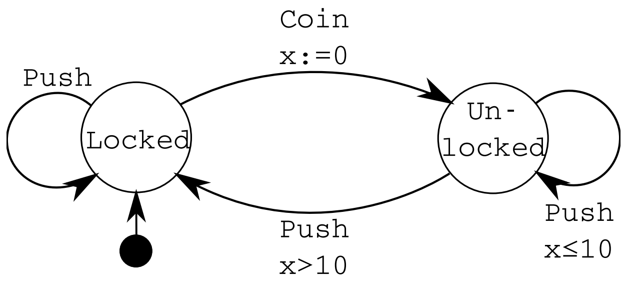

Fig. 2: A simple TA representing a variation of a classical turnstyle. When a coin is loaded, the turning bars will stay unlocked for 10 time units. If they are pushed after such time, the turnstyle will lock again, until a new coin is delivered.

4 Modeling Technique

The main contribution of this work is to propose a technique to model Timed Automata execution semantics and structure by using Event Calculus elements, such as facts, events and fluents. It is done by means of a logic theory articulated into two components:

- An automata-independent theory that contains the general machinery for the TA semantics;
- An automata-dependent theory that instantiate an actual TA, representing its graph structure, states, transitions, guard constraints and clock assignments.

This two-part design allows modularity and incremental programming: in fact, to create a new Automaton, it will be enough to write the corresponding Automata-dependent theory, without the need to modify the machinery.

4.1 Timed Automata applied to RTcMAS

The goal of translating Timed Automata into EC formulas is to establish a connection between the Real-Time part and the Knowledge Representation (KR)/Reasoning part of an agent, as shown in Figure 3. This connection would allow to have a common underlying language which is useful for both building the agent's mind (KR and reasoning) as well as to model, check and verify its Real-Time properties.

In other words, if an agent (or an agent network) is represented as a Timed Automaton (or a Timed Automata network), it would be possible to implement it in Event Calculus formulas, without having the need of third parties model checkers or ad-hoc languages. Then, once a comprehensive EC theory of such agent(s) is built, different EC reasoning techniques might be used depending on the context [19]. For TA formal verification, the most suitable reasoning technique would be Model Finding, instead, as done in [16], for an online rule-based behavior of the agent, Deduction shall be preferred choice.

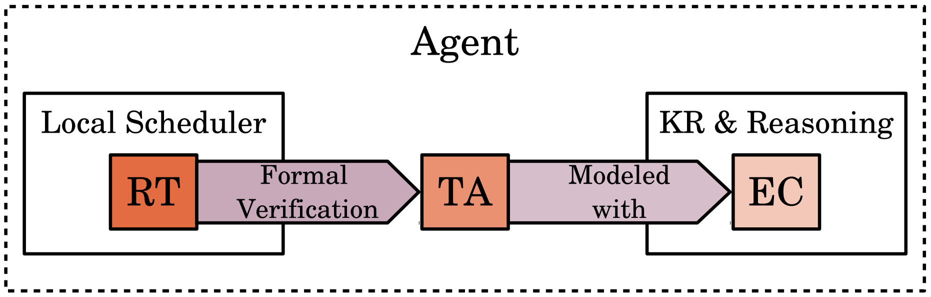

Fig. 3: Agent's components logical mapping

4.2 Automata-Independent theory

The Automata-independent theory contains the general machinery for clocks, guards, and state transition to work.

Clocks have been modeled by using integer-valued fluents named *clk*/1. Since real-valued fluents are not yet ready for practical use in the Cached/Reactive Event Calculus, only Discrete Timed Automata can be modeled. However, this should not be a limitation, since in Real-Time applications, time is usually measured by the system clock, and from a formal point of view, [18] has proven that a theory expressed in continuous time EC can be translated in discrete time EC without losing in expressiveness. As shown in listing 1.1, clock fluents are initiated and terminated by *set*/2 events, which implement the mechanism of clock setting and resetting.

Listing 1.1: Generic TA implementation: clocks set/reset

```
initiates(set(C,V),status(clk(C),V),T).

terminates(set(C,_),status(clk(C),Vold),T):-
        holds_at(status(clk(C),Vold),T).
```

When a $set(C, V)$ event happens, the clock specified in the variable C gets set to the value V. All clocks are also incremented simultaneously by *tick*/0 events, which simulate the flowing of time in the system (listing 1.2).

Listing 1.2: Generic TA implementation: flow of time

```
initiates(tick,status(clk(C),Vnew),T):-
        holds_at(status(clk(C),Vold),T),
        Vnew is Vold + 1,
        not (happens(set(C,_),T)).

terminates(tick,status(clk(C),Vold),T):-
        holds_at(status(clk(C),Vold),T).
```

It should be noticed that in order to avoid ambiguity as a double fluent initialisation, a *tick*/0 event increments a fluent only if there is not any other *set*/2 event happening simultaneously. TA's states are modeled as simple boolean fluents. Since the system can only be in one state at a time, only one state fluent (for each TA instance) can hold at a certain timepoint. State transitions are instead represented as events, that can lead to the termination or initialisation of state fluents. Such events, identified with the variable *Lab*, shall be the labels of the transitions that goes from old states *Sold* to new states *Snew*. If a *Lab* event happens at timestamp T, clauses in listing 1.3 show that state fluents *Sold* and *Snew* are terminated/initiated only (i) if there is an arc going from the old state to the new one (with label *Lab*), (ii) if the system is currently in the correct state to perform the transition, (iii) and the guard relative to such transition is satisfied at that timestamp.

Listing 1.3: Generic TA implementation: state transitions and guards

```
initiates(Lab,Snew,T):-
        arc(Lab,Sold,Snew),
        holds_at(Sold,T),
        guard(Lab,Sold,Snew,T).

terminates(Lab,Sold,T):-
        arc(Lab,Sold,Snew),
        holds_at(Sold,T),
        guard(Lab,Sold,Snew,T).
```

4.3 Automata-dependent theory

The purpose of the Automata-dependent theory is to instantiate the actual automata by defining its graph structure, labels (of states and transitions), guard conditions, clocks and clock resets. Such instantiation is traduced into writing facts and mostly ground clauses that shall unify the variables defined in the Automata-independent theory. The translation from a TA in a graphical form to an Event Calculus theory will be shown by means of an example, which will consist in modeling the TA in figure 4. It models a simple timed system, that can be thought as a lightbulb and a button. If the button is pushed while the lightbulb is switched off, it will turn on; then if the button is pressed again within a certain time window, the lightbulb will shine even brighter, otherwise it will turn off again.

The Event Calculus modeling of this TA is performed as follows:

- The initial state of the system is specified by the *initially*/2 facts. These establish the value of the clock fluent $clk(x)$ and which one of the state fluents holds at timestamp -1 (i.e. from the beginning). From the code in listing 1.4 it can be seen that the clock is initally set to 0, and the initial state of the system is the off state.

  Listing 1.4: Light control system's TA: initial state

  ```
  initially(status(clk(x),0)).

  initially(off).
  ```

- The *arc*/3 facts in listing 1.5 model the graph structure of the automata. The first argument is the transition's label, the second one is the transition's old state and the third is the transition's arrival state. For each transition in the TA's graph, one of these facts must be present in the Automata-dependent theory. States that are not connected to others by any transition cannot be represented within the current technique; this is actually an advantage, since isolated states do not make much sense in both FSMs and TAs.

Listing 1.5: Light control system's TA: graph structure

```
arc(press,off,light).
arc(press,light,off).
arc(press,bright,off).
arc(press,light,bright).
```

– Guards are implemented as shown in listing 1.6. If a transition does not have any guard, it will be enough to put a *guard*/4 term with the first three (ground) arguments being the transition's label, the transition's leaving and arrival state, and the last one being an emtpy variable (to unify any possible timestamp). If instead some constraint has to be imposed on clock values, it will be enough to put a condition on the clock fluent's value in the body of the *guard*/4 clause, using the variable T instead of the empty variable.

Listing 1.6: Light control system's TA: guards

```
guard(press,off,light,_).

guard(press,light,off,T):-
        holds_at(status(clk(x),X),T),
        X > 3.

guard(press,light,bright,T):-
        holds_at(status(clk(x),X),T),
        X =< 3.

guard(press,bright,off,_).
```

– Clock resets are instantiated by event chaining. Since clocks can be reset by launching *set*/2 events (see section 4.2), they have to be generated automatically when the appropriate transition is taken in the TA. For this particular automata (fig. 4), code in listing 1.7 shows how the $set(x,0)$ event is launched every time the TA goes from the off state to the *on* state, wrapping the *press* event that triggers the transition (effectively setting the clock x to 0).

Listing 1.7: Light control system's TA: clock reset

```
happens(set(x,0),T):-
        holds_at(off,T),
        happens(press,T).
```

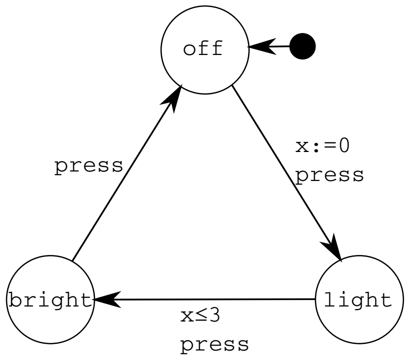

Fig. 4: The TA describing a simple light control system.

5 Tests

One of the main limitations of logic-based approaches, and specifically of the Event Calculus, is reasoning performance. Even if this work is based on jREC, which implements Reactive EC (a version of Cached EC), thus providing much lower computation times with respect to standard EC, it is still interesting to study how reasoning complexity is affected by use-case-specific parameters. [16] highlights in fact that regular EC Caching strategies are not enough to achieve reasoning feasibility in that particular use case, thus requiring solutions such as event indexing and ad-hoc performance analysis. In the present work, the number of state transitions occurrences (implemented in EC as events) and the number of TA instances (bigger TA-dependent theory) have been selected as the two main criterias to evaluate reasoning performance. They can be considered as two orthogonal dimensions, and have led to two separate tests:

1. The first fixes the number of TA instances to one, while the number of events (state transition occurrences and clock ticks) spans from 0 to 200, with a step of 40;
2. The second test fixes the number of events to 200, while the number of TA instances spans from 0 to 5, with step 1.

Within this setup, performance has been evaluated by measuring the time needed by the jREC reasoner to execute TAs runs. The reasoner is fed with a list of events, which represent the state transitions occurrences, and as the output, a list of states (with relative time references) is returned as a list of MVIs (see section 3.2). The TA's implementation chosen for the tests is the one shown in section 4.2, and the input events have been selected in such a way that every TA visits all of its three state cyclically. In addition to the state transition occurrence events, *tick* events had to be included to allow the clock fluents to work, thus modeling the flow of time in the system.

Lastly, the tests have been run on the standard jREC engine as well as on a customized jREC engine based on Red-Black Trees event indexing [16].

The machine used for such tests was a standard Ubuntu 16.04 desktop PC configuration, with 16 GBs of RAM and a i7-6700k CPU.

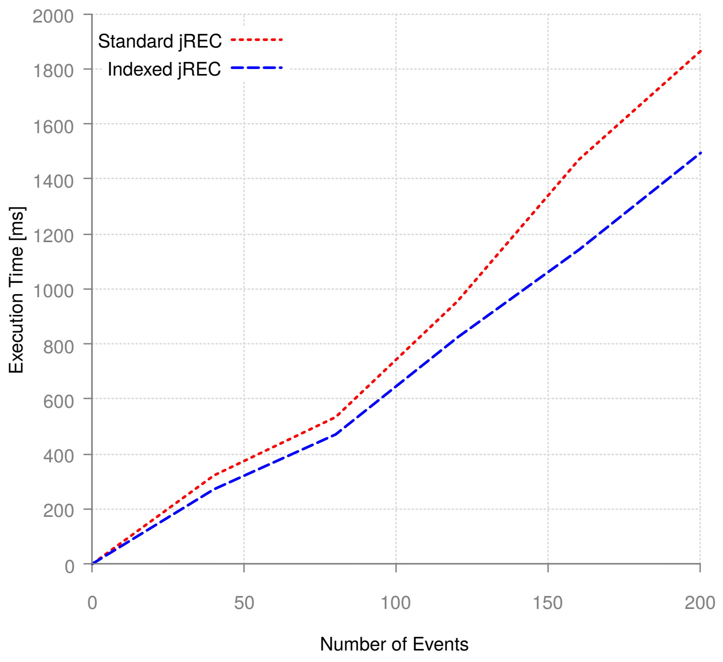

Fig. 5: Execution time for a single TA instance run, with events spanning from 0 to 200.

5.1 Results and Discussion

Plots in Fig. 5 and Fig. 6 highlight that the indexed jREC performs generally better that the standard version. This was indeed expected, given the more efficent event management accomplished by the indexing, but this difference is more noticeable in Fig. 5's plot. It can be explained by the fact that for the multiple TA instances tests (Fig. 6), the number of events is kept constant, and the indexing mechanism does not help managing the bigger TA-dependent theory. The overhead caused by more TA instances prevails over the gain obtained by the event indexing.

More importantly, the execution time trends for both tests seem to follow a linear pattern. Even though this is a desirable behaviour in terms of scalability, other factors such as TA's complexity (number of nodes, number of transitions, etc) and inter-arrival time of transition occurrences over average guards conditions' time-windows might affect this trend considerably.

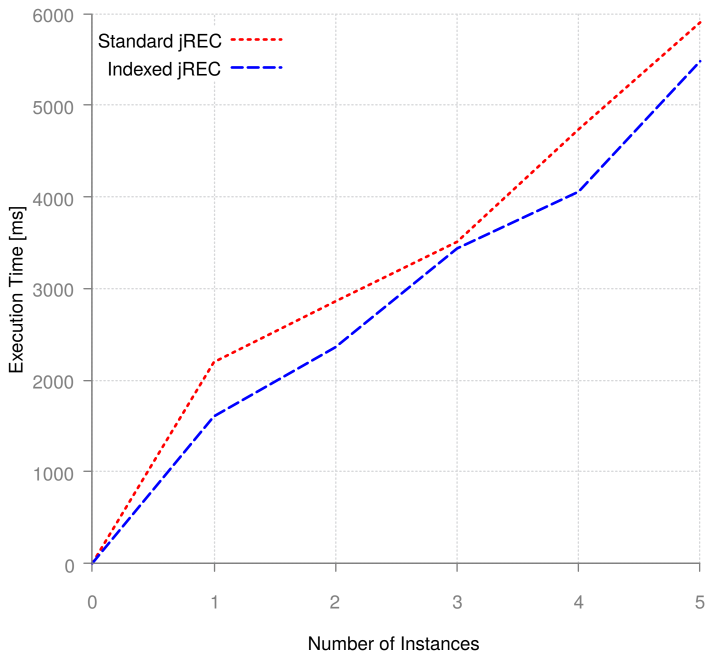

Fig. 6: Execution time for multiple TA instances runs, with number of events fixed to 200.

6 Conclusions and Future Work

In this work, a methodology to represent, code and reason on/execute Timed Automata using a Prolog implementation of the Event Calculus formalism is shown. It has been also applied to model a simple Timed Automaton as a use case, which has been also exploited to study the reasoning performance and scalability. These tests have highlighted that the desired reasoning is indeed feasible, and promising execution time trends. By now, the proposed methodology only allows to carry out forward reasoning, giving the possibility to simulate Timed Automata runs by providing a list of timed state transition occurrences. Other backwards reasoning techniques, such as abduction or postdiction proposed in [19] might be possibly involved to obtain more sophisticated model checking and formal verification capabilities, even though this would mean to abandon the Prolog implementation and go towards less efficient but more versatile engines [1]. Since the current methodology only involves TA's execution semantics, state invariants are not yet needed nor considered. Their inclusion within the TA-independent theory would be another important step towards having a complete TA representation methodology for model checking purposes. Such methodology would then enable to effectively represent one or more Real-Time agents as Timed-Automata, with the goal of formally verifying safety or reliability properties. Lastly, in order to have a more realistic figure of the reasoning system behaviour, the performance evaluation should be deepened by investigating other parameters, such as more complex TA graph structure, and transition occurrences inter-arrival time compared to guard conditions' time-window.

References

1. DECReasoner reference website. `http://decreasoner.sourceforge.net/`
2. JREC reference website. `http://www.inf.unibz.it/~montali/tools.html`
3. List of works that used UPPAAL. `http://www.it.uu.se/research/group/darts/uppaal/examples.shtml`
4. UPPAAL reference website. `http://www.uppaal.org/`
5. Artikis, A., Sergot, M., Paliouras, G.: An event calculus for event recognition. IEEE Transactions on Knowledge and Data Engineering 27(4), 895–908 (2015)
6. Bragaglia, S., Chesani, F., Mello, P., Montali, M., Torroni, P.: Reactive event calculus for monitoring global computing applications. In: Artikis, A., Craven, R., Kesim Çiçekli, N., Sadighi, B., Stathis, K. (eds.) Logic Programs, Norms and Action: Essays in Honor of Marek J. Sergot on the Occasion of His 60th Birthday, pp. 123–146. Springer Berlin Heidelberg (2012)
7. Bromuri, S., Brugues de la Torre, A., Duboisson, F., Schumacher, M.: Indexing the Event Calculus with Kd-trees to Monitor Diabetes. ArXiv e-prints (Oct 2017)
8. Brugués, A., Bromuri, S., Barry, M., del Toro, O.J., Mazurkiewicz, M.R., Kardas, P., Pegueroles, J., Schumacher, M.: Processing diabetes mellitus composite events in MAGPIE. Journal of Medical Systems 40(2), 44 (2016)
9. Brugués, A., Bromuri, S., Pegueroles-Valles, J., Schumacher, M.I.: MAGPIE: An agent platform for the development of mobile applications for pervasive healthcare. In: Proceedings of the 3rd International Workshop on Artificial Intelligence and Assistive Medicine (AI-AM/NetMed). pp. 6–10 (2014)

10. Buttazzo, G.C.: Hard real-time computing systems: predictable scheduling algorithms and applications, vol. 24. Springer Science & Business Media (2011)
11. Calvaresi, D., Schumacher, M., Marinoni, M., Hilfiker, R., Dragoni, A.F., Buttazzo, G.: Agent-based systems for telerehabilitation: strengths, limitations and future challenges. In: Proceedings of the 10th Workshop on Agents Applied in Health Care (A2HC 2017) (2017)
12. Cervesato, I., Montanari, A.: A general modal framework for the event calculus and its skeptical and credulous variants. The Journal of Logic Programming 38(2), 111 – 164 (1999), http://www.sciencedirect.com/science/article/pii/S0743106698100213
13. Chittaro, L., Montanari, A.: Efficient temporal reasoning in the cached event calculus. Computational Intelligence 12(3), 359–382 (1996), http://dx.doi.org/10.1111/j.1467-8640.1996.tb00267.x
14. Cicekli, N.K., Yildirim, Y.: Formalizing workflows using the event calculus. In: Ibrahim, M., Küng, J., Revell, N. (eds.) Database and Expert Systems Applications. pp. 222–231. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
15. David, A., Illum, J., Larsen, K.G., Skou, A.: Model-based framework for schedulability analysis using uppaal 4.1. Model-based design for embedded systems 1(1), 93–119 (2009)
16. Falcionelli, N., Sernani, P., Brugués, A., Mekuria, D.N., Calvaresi, D., Schumacher, M., Dragoni, A.F., Bromuri, S.: Event calculus agent minds applied to diabetes monitoring. In: Sukthankar, G., Rodriguez-Aguilar, J.A. (eds.) Autonomous Agents and Multiagent Systems. pp. 258–274. Springer International Publishing, Cham (2017)
17. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Generation Computing 4(1), 67–95 (1986)
18. Mueller, E.T.: Event calculus reasoning through satisfiability. J. Log. and Comput. 14(5), 703–730 (Oct 2004), http://dx.doi.org/10.1093/logcom/14.5.703
19. Mueller, E.T.: Commonsense Reasoning: An Event Calculus Based Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edn. (2015)
20. Phd, M.F.: The event calculus as a programming model for game ai
21. Waez, M.T.B., Dingel, J., Rudie, K.: A survey of timed automata for the development of real-time systems. Computer Science Review 9, 1 – 26 (2013), http://www.sciencedirect.com/science/article/pii/S1574013713000178