Event Calculus for Run-Time reasoning (RTEC) Manual

Alexander Artikis and Christos Vlassopoulos
NCSR "Demokritos"
cer.iit.demokritos.gr

November 29, 2017

1 Introduction

The Event Calculus for Run-Time reasoning (RTEC) is a logic programming implementation of the Event Calculus [6], designed to compute continuous queries on data streams [3]. RTEC has been successfully used for *composite event recognition* ('event pattern matching'). Composite event (CE) recognition systems accept as input a stream of time-stamped simple, derived events (SDE)s. A SDE is the result of applying a computational derivation process to some other event, such as an event coming from a sensor [7]. Using SDEs as input, event recognition systems identify CEs of interest—collections of events that satisfy some pattern. The 'definition' of a CE imposes temporal and, possibly, atemporal constraints on its subevents, i.e. SDEs or other CEs. Below are a few CE recognition applications in which RTEC has been used:

- Public space surveillance [3].
- City transport & traffic management [3, 4, 5].
- Maritime monitoring [8, 1].

The novelty of RTEC lies in the following implementation techniques:

1. *Caching*, that helps in avoiding unnecessary re-computations.
2. *Interval manipulation*, that helps in expressing succinctly complex temporal phenomena.
3. *Indexing*, that makes RTEC robust to data streams that are irrelevant to the queries we want to compute.
4. *Windowing*, that supports real-time query computation.

1.1 Software requirements & installation

RTEC is cross-platform. The only software requirement is a Prolog implementation. RTEC has been tested in YAP[1] and SWI[2] Prolog. In the case of Linux, to install YAP or SWI Prolog type

```
$ sudo apt-get install yap
```

or

```
$ sudo apt-get install swi-prolog
```

To use RTEC in YAP, simply download it from `https://github.com/aartikis/RTEC`. The file `RTEC/README.txt` contains some general information about the use of RTEC.

[1]http://www.dcc.fc.up.pt/~vsc/Yap/downloads.html
[2]http://www.swi-prolog.org/Download.html

1.2 A simple example

We begin with a simple example briefly illustrating the functionality of RTEC. In the following sections we will have a closer look at the expressivity and reasoning algorithms of RTEC.

Suppose that Chris is having an all but ordinary day. He goes to work in the morning and in the afternoon he finds out that he has won the lottery. In the evening he goes to the pub, but loses his wallet. Ultimately, he goes home at night. We want to know whether Chris is happy or not, as these actions take place. Our story has three events, "go_to", "lose_wallet" and "win_lottery", and three properties, "happy", "location" and "rich". The basic concept of RTEC lies in events taking place and modifying the values of the properties. In the RTEC terminology, these properties are called "fluents".

We would like to specify the conditions that make Chris happy in our tiny world. Being rich is such a condition. Another condition could be being at the pub. Therefore, the 'union' of these two conditions can meet the needs of a happy man in our example. Winning the lottery makes someone rich. For the sake of the example, let's assume that losing your wallet causes you to stop being rich.

Now that we have designed the rules that describe our example, we are ready to express them into the logic programming language of RTEC. Use a text editor and create a new file, say "`toy_rules.prolog`", and paste the following:

```prolog
initiatedAt(rich(X)=true, T) :-
   happensAt(win_lottery(X), T).

terminatedAt(rich(X)=true, T) :-
   happensAt(lose_wallet(X), T).

initiatedAt(location(X)=Y, T) :-
   happensAt(go_to(X,Y), T).

holdsFor(happy(X)=true, I) :-
   holdsFor(rich(X)=true, I1),
   holdsFor(location(X)=pub, I2),
   union_all([I1,I2], I).
```

Listing 1: Event description in RTEC.

Following Prolog's convention, variables start with an upper-case letter, while predicates and constants start with a lower-case letter. To test the formalization above, create a new file, "`toy_declarations.prolog`", containing the following:

```prolog
% Information about all our events and fluents.
% - Is each entity an event or a fluent?
% - Is it an input or an output entity?
% - Choose an argument to be used as index for quicker access.

event(go_to(_,_)).
inputEntity(go_to(_,_)).
index(go_to(Person,_), Person).

event(lose_wallet(_)).
inputEntity(lose_wallet(_)).
index(lose_wallet(Person), Person).

event(win_lottery(_)).
inputEntity(win_lottery(_)).
index(win_lottery(Person), Person).

simpleFluent(location(_)=home).
```

```
outputEntity(location(_)=home).
index(location(Person)=home, Person).

simpleFluent(location(_)=pub).
outputEntity(location(_)=pub).
index(location(Person)=pub, Person).

simpleFluent(location(_)=work).
outputEntity(location(_)=work).
index(location(Person)=work, Person).

simpleFluent(rich(_)=true).
outputEntity(rich(_)=true).
index(rich(Person)=true, Person).

simpleFluent(rich(_)=false).
outputEntity(rich(_)=false).
index(rich(Person)=false, Person).

sDFluent(happy(_)=true).
outputEntity(happy(_)=true).
index(happy(Person)=true, Person).

sDFluent(happy(_)=false).
outputEntity(happy(_)=false).
index(happy(Person)=false, Person).

% How are the fluents grounded?
% Define the domain of the variables.

grounding(location(Person)=Place) :- person(Person), place(Place).
grounding(rich(Person)=true) :- person(Person).
grounding(rich(Person)=false) :- person(Person).
grounding(happy(Person)=true) :- person(Person).
grounding(happy(Person)=false) :- person(Person).

% In what order will the output entities be processed by RTEC?

cachingOrder(location(_)=home).
cachingOrder(location(_)=pub).
cachingOrder(location(_)=work).
cachingOrder(rich(_)=true).
cachingOrder(rich(_)=false).
cachingOrder(happy(_)=true).
cachingOrder(happy(_)=false).
```

Listing 2: Event and fluent declarations.

The above *declarations* file is a companion to the rules file. It contains information about all the events and fluents of our scenario. In the following section, we will describe the declarations language in detail.

At this point, we need to compile the rules and declarations. To do this, open a terminal, go to the `RTEC/` directory, start Prolog by typing "`yap`" and pressing ENTER, and execute the following query:

```
?- compileEventDescription('toy_declarations.prolog',
                    'toy_rules.prolog', 'toy_rules_compiled.prolog').
```

If Prolog responds with a message ending in "yes" or "true" (depending on your Prolog implementation), compilation was successful. During compilation, a new file "toy_rules_compiled.prolog" has been created. This file combines the information provided in the rules and declarations files and is in a form ready for use by RTEC.

At the next step, we must provide the domain of each variable. Create another file, say "toy_var_domain.prolog", and put the following code in it:

```
% This is our variable domain

person(chris).

place(home).
place(pub).
place(work).
```

Listing 3: Variable domain.

The contents of this file are used for *output entity* (e.g. fluent) grounding (see Listing 2, lines 49-53). Finally, to test our event description, we need an event narrative, such as the following:

```
% This is our narrative of events, given as input.

updateSDE(story, 9, 21) :-
assert(happensAtIE(go_to(chris, work), 9)),
assert(happensAtIE(win_lottery(chris), 13)),
assert(happensAtIE(go_to(chris, pub), 17)),
assert(happensAtIE(lose_wallet(chris), 19)),
assert(happensAtIE(go_to(chris, home), 21)).
```

Listing 4: Event narrative.

We make a series of assertions about what happens at each time in our scenario. happensAtIE is a compiled version of the happensAt predicate that expresses event occurrences. At 9:00 Chris goes to work. Then, at 13:00 he finds out that he has won the lottery. Subsequently, at 17:00 he goes to the pub. Afterwards, at 19:00 he loses his wallet, and finally at 21:00 he returns home. We group these assertions under the auxiliary updateSDE predicate, so that whenever we want to load this narrative we simply call this predicate.

Now we have all the necessary components for narrative assimilation—in this example, the computation of the maximal intervals of the fluents. We simply need to combine the aforementioned files and start the RTEC engine. This may be done by creating a Prolog script, "toy_queries.prolog" that contains the following:

```

:-['toy_event_stream.prolog'].
:-['../../src/RTEC.prolog'].
:-['toy_var_domain.prolog'].
:-['toy_declarations.prolog'].
:-['toy_rules_compiled.prolog'].

performER :-
        initialiseRecognition(unordered, nopreprocessing, 1),
        updateSDE(story, 9, 21),
        eventRecognition(21, 21).
```

Listing 5: Narrative assimilation script.

The code of Listing 5 accumulates the information contained in the files described above, and combines it with the main file of RTEC, namely "RTEC.prolog". Then, we define a predicate "performER" to automate narrative assimilation. At first we initialize RTEC, by setting 3 parameters. In the first parameter we state whether the input facts are temporally sorted or not (in our case they are not, therefore we use the value "`unordered`"). In the second parameter we state whether our input data need preprocessing or not (this is not the case here, so we used the value "`nopreprocessing`"). Finally, the last parameter is the distance between two consecutive timepoints in our dataset, which is 1 time unit. Then we load the event narrative using the "`updateSDE`" predicate we created in Listing 4, and finally we call the built-in `eventRecognition` predicate of RTEC for narrative assimilation. We provide 2 parameters, the current time of the query and how deep in the past will RTEC look for events and fluents in order to calculate the composite events of interest. Here, since we have a small dataset that ends at timepoint 21, we perform one query at time 21 and we take into account account all the input events that took place within the last 21 timepoints, i.e. from the beginning.

Back to Prolog now. Halt any open session with Prolog and start a new one. Then load the Prolog script:

```
?- ['toy_queries.prolog'].
```

Again, if Prolog responds with a message ending in "yes" or "true", then the file loading was successful. We are now ready for narrative assimilation, by typing the command

```
?- performER.
```

If Prolog answers "yes" or "true", that means RTEC has finished the computation of the maximal intervals of fluents. Now we can ask RTEC anything about the processed fluents. For instance, if we want to see when Chris is happy, typing "`holdsFor(happy(chris)=true,I).`" will give us the answer:

```
I = [(14,22)]
```

This means that Chris is happy from time 14, right after he won the lottery, until time 22 (not included), when he leaves the pub. A term of the form `(Ts, Te)` in RTEC represents the closed-open interval $[T_s, T_e)$. According to our example, one is happy if he is rich or at the pub. Thus, this answer seems reasonable.

To see the maximal intervals of all fluent-value pairs that have been computed by RTEC, simply type "`holdsFor(F,I).`" and press ENTER. You will receive an output that looks like this:

```
F = (location(chris)=home),
I = [(22,inf)] ? ;
F = (location(chris)=pub),
I = [(18,22)] ? ;
F = (location(chris)=work),
I = [(10,18)] ? ;
F = (rich(chris)=true),
I = [(14,20)] ? ;
F = (rich(chris)=false),
I = [] ? ;
F = (happy(chris)=true),
I = [(14,22)] ? ;
F = (happy(chris)=false),
I = []
```

In addition, we can ask what was true at a specific time-point. For instance if we ask "`holdsAt(F,16).`" we will find out what was the situation like at time-point 16. RTEC will respond:

```
F = (location(chris)=work) ? ;
F = (rich(chris)=true) ? ;
F = (happy(chris)=true)
```

So, RTEC says that at time-point 16 Chris is at work, rich, and happy.

Now that we have given a brief illustration of the basic functionality of RTEC, we can take a closer look at its language and reasoning techniques.

Table 1: Main predicates of RTEC.

Predicate	Meaning
`happensAt(E, T)`	Event `E` occurs at time `T`
`holdsAt(F=V, T)`	The value of fluent `F` is `V` at time `T`
`holdsFor(F=V, I)`	`I` is the list of the maximal intervals for which `F=V` holds continuously
`initiatedAt(F=V, T)`	At time `T` a period of time for which `F=V` is initiated
`terminatedAt(F=V, T)`	At time `T` a period of time for which `F=V` is terminated
`union_all(L, I)`	`I` is the list of maximal intervals produced by the union of the lists of maximal intervals of list `L`
`intersect_all(L, I)`	`I` is the list of maximal intervals produced by the intersection of the lists of maximal intervals of list `L`
`relative_complement_all(I′, L, I)`	`I` is the list of maximal intervals produced by the relative complement of the list of maximal intervals `I′` with respect to every list of maximal intervals of list `L`

2 The RTEC language

The time model in RTEC is linear and includes integer time-points. Where `F` is a *fluent*—a property that is allowed to have different values at different points in time—the term `F=V` denotes that fluent `F` has value `V`. Boolean fluents are a special case in which the possible values are `true` and `false`. `holdsAt(F=V, T)` represents that fluent `F` has value `V` at a particular time-point `T`. `holdsFor(F=V, I)` represents that `I` is the list of the maximal intervals for which `F=V` holds continuously. `holdsAt` and `holdsFor` are defined in such a way that, for any fluent `F`, `holdsAt(F=V, T)` if and only if `T` belongs to one of the maximal intervals of `I` for which `holdsFor(F=V, I)`.

An *event description* in RTEC includes rules that define the event instances with the use of the `happensAt` predicate, the effects of events with the use of the `initiatedAt` and `terminatedAt` predicates, and the values of the fluents with the use of the `holdsAt` and `holdsFor` predicates, as well as other, possibly atemporal, constraints. Table 1 summarises the RTEC predicates available to the event description developer.

Fluents are either simple or statically determined. In brief, simple fluents are defined by means of `initiatedAt` and `terminatedAt` rules, while statically determined fluents are defined by means of application-dependent `holdsFor` rules. More details on this distinction will be given shortly.

An event description is a (locally) stratified logic program [9]. We restrict attention to *hierarchical* event descriptions, those where it is possible to define a function *level* that maps all fluent-values `F=V` and all events to the non-negative integers as follows. Events and statically determined fluent-values `F=V` of level 0 are those whose definitions do not depend on any other events or fluents. These represent the *input entities.* There are no fluent-values `F=V` of simple fluents `F` in level 0. Events and simple fluent-values of level n ($n > 0$) are defined in terms of at least one event or fluent-value of level $n-1$ and a possibly empty set of events and fluent-values from levels lower than $n-1$. Statically determined fluent-values of level n are defined in terms of at least one fluent-value of level $n-1$ and a possibly empty set of fluent-values from levels lower than $n-1$. Events and fluent-values of level n are the *output entities.*

In the following sections we present in more detail the building blocks of RTEC.

2.1 Event Description

2.1.1 Events

Events in RTEC are instantaneous and represented with the use of the `happensAt` predicate. Our simple example has three events: `go_to`, `lose_wallet` and `win_lottery`. Input events are indicated as `happensAt` facts, i.e. they have an empty body. In contrast, output events are defined by `happensAt` rules, i.e. rules with at least one body literal.

2.1.2 Fluents

As already mentioned, fluents are either simple or statically determined.

Simple Fluents. For a simple fluent `F`, `F=V` holds at a particular time-point `T` if `F=V` has been *initiated* by an event that has occurred at some time-point earlier than `T`, and has not been *terminated* at some other time-point in the meantime. This is an implementation of the law of inertia. To compute the *intervals* `I` for which `F=V`, i.e. `holdsFor(F=V, I)`, we find all time-points `Ts` at which `F=V` is initiated, and then, for each `Ts`, we compute the first time-point `Tf` after `Ts` at which `F=V` is terminated. The time-points at which `F=V` is initiated (respectively terminated) are computed by means of domain-specific `initiatedAt` (resp. `terminatedAt`) rules.

In our example, `rich` is a simple fluent. The maximal intervals during which `rich(Person)=true` holds continuously are computed using the domain-independent implementation of `holdsFor` from the `initiatedAt` and `terminatedAt` rules defining this fluent.

In addition to constraints on events, the bodies of `initiatedAt` and `terminatedAt` rules may specify constraints on fluents by means of the `holdsAt`, `initiatedAt` and `terminatedAt` predicates.

Statically Determined Fluents. Apart from the domain-independent definition of `holdsFor`, an event description may include domain-specific `holdsFor` rules, used to define the values of a fluent `F` in terms of the values of other fluents. We call such a fluent `F` *statically determined.* `holdsFor` rules of this kind make use of interval manipulation constructs. RTEC provides three such constructs: `union_all`, `intersect_all` and `relative_complement_all`(see the last three items of Table 1). `union_all(+L, -I)` computes the list `I` of maximal intervals representing the union of maximal intervals of the lists of list `L`. For instance:

```
union_all([[(5,20), (26,30)],[(28,35)]], [(5,20), (26,35)])
```

Recall that a term of the form `(Ts, Te)` in RTEC represents the closed-open interval $[Ts, Te)$. `I` in `union_all(L, I)` is a list of maximal intervals that includes each time-point that is part of at least one list of `L`. See Figure 1(a) for a visual illustration.

`intersect_all(+L, -I)` computes the list `I` of maximal intervals such that `I` represents the intersection of maximal intervals of the lists of list `L`, as, e.g.:

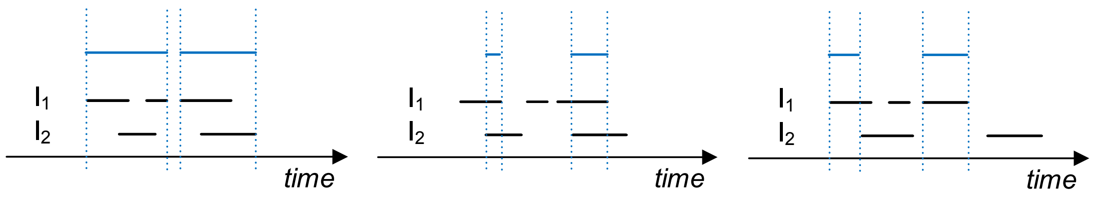
```
intersect_all([[(26,31)], [(21,26),(30,40)]], [(30,31)])
```

`I` in `intersect_all(L, I)` is a list of maximal intervals that includes each time-point that is part of all lists of `L` (see Figure 1(b)).

`relative_complement_all(+I′, +L, -I)` computes the list `I` of maximal intervals such that `I` represents the relative complements of the list of maximal intervals `I′` with respect to the maximal intervals of the lists of list `L`. Below is an example of `relative_complement_all`:

```
relative_complement_all([(5,20), (26,50)], [[(1,4),(18,22)],[(28,35)]],
                        [(5,18),(26,28),(35,50)])
```

`I` in `relative_complement_all(I′, L, I)` is a list of maximal intervals that includes each time-point of `I′` that is not part of any list of `L` (see Figure 1(c)).



(a) Union. (b) Intersection. (c) Relative Complement.

Figure 1: A visual illustration of the three interval manipulation constructs of RTEC. In this example, there are two input fluent streams, I_1 and I_2. The output of each interval manipulation construct is colored light blue.

In our example, `happy` is a statically determined fluent defined by means of `union_all`. However, this is just one way of defining happiness. For example, we could have specified that a person is happy when he is rich *and* at the pub. To specify `happy` in this way one should replace `union_all` by `intersect_all` in the `holdsFor` rule of `happy`.

The interval manipulation constructs of RTEC support the following type of definition: for all time-points `T`, `F=V` holds at `T` if and only if some Boolean combination of fluent-value pairs holds at `T`. For a wide range of fluents, this is a much more concise definition than the traditional style of Event Calculus representation, i.e. identifying the various conditions under which the fluent is initiated and terminated so that maximal intervals can then be computed using the domain-independent `holdsFor`. Compare, e.g. the statically determined fluent representation of `happy` in Listing 1 and the simple fluent representation below:

```
initiatedAt(happy(X)=true, T) :-
    initiatedAt(rich(X)=true, T).
initiatedAt(happy(X)=true, T) :-
    initiatedAt(loc(X)=pub, T).
terminatedAt(happy(X)=true, T) :-
    terminatedAt(rich(X)=true, T)
    not holdsAt(loc(X)=pub, T).
terminatedAt(happy(X)=true, T) :-
    terminatedAt(loc(X)=pub, T),
    not holdsAt(rich(X)=true, T).
```

`not` is negation by failure. The interval manipulation constructs of RTEC can also lead to much more efficient computation [3].

2.2 Declarations

The declarations of our example were presented in Listing 2. In the declarations of an event description, we first need to denote the events, simple fluents and statically determined fluents. This is done with the use of the `event`, `simpleFluent` and `sDFluent` predicates.

Each event and fluent must be declared as either `inputEntity` or `outputEntity`. As explained earlier in the section, the input entities may consist of events and/or statically determined fluents, while the output entities may comprise events, simple and/or statically determined fluents.

For each event and fluent, the user must also declare its `index`. In our example, `Person` is the index of all events and fluents. The index allows for the fast retrieval from the memory of the list of time-points, in the case of events, and maximal intervals, in the case of fluents.

To perform query computation, RTEC grounds every output entity. This process is guided by the `grounding` predicate of the declarations language of RTEC, that denotes the domain of the variables of the output entities.

The final step in the declarations is to specify the `cachingOrder`, i.e. the order in which the output entities will be processed. To take advantage of RTEC's caching technique, the output entities should be processed in a bottom-up manner. This way, when processing an output entity `U` of level n, the time-points/intervals of all entities defining `U`—these will all be in levels below n— will simply be retrieved from the cache.

Back to our example, if we look the rules, we will see that `happy` is defined in terms of `location` and `rich`. Thus, `location` and `rich` must must be processed before `happy`. `location` and `rich` are on the same level of the hierarchy and thus the order in which they are processed does not matter.

3 Reasoning in RTEC

Reasoning has to be efficient enough to support real-time decision-making, and scale to very large numbers of input and output entities. Input entities may not necessarily arrive at RTEC in a timely manner, i.e. there may be a (variable) delay between the time at which input entities take place and the time at which they arrive at RTEC. Moreover, input entities may be revised, or even completely discarded in the future, as in the case where the parameters of an input entity were originally computed erroneously and are subsequently revised, or in the case of retraction of an input entity that was reported by mistake, and the mistake was realised later.

RTEC performs narrative assimilation by computing and storing the maximal intervals of output entities, i.e. the intervals of fluents and the time-points in which events occur. Reasoning takes place at specified query times $Q_1, Q_2, \dots$. At each Q_i the input entities that fall within a specified interval — the window ω — are taken into consideration. All input entities that took place before or at $Q_i{-}\omega$ are discarded. This is to make the cost of reasoning dependent only on ω and not on the complete history. The size of ω and the temporal distance between two consecutive query times — the slide step $Q_i{-}Q_{i-1}$ — are set by the user.

At Q_i, the output entity maximal intervals computed by RTEC are those that can be derived from the input entities that occurred in the interval $(Q_i{-}\omega, Q_i]$, as recorded at time Q_i. When ω is longer than the slide step, i.e., when $Q_i{-}\omega{<}Q_{i-1}{<}Q_i$, it is possible that an input entity occurs in the interval $(Q_i{-}\omega, Q_{i-1}]$ but arrives at RTEC only after Q_{i-1}; its effects are taken into account at query time Q_i. And similarly for input entities that took place in $(Q_i{-}\omega, Q_{i-1}]$ and were subsequently revised after Q_{i-1}. In the common case that input entities arrive at RTEC with delays, or there is input entity revision, it is preferable therefore to make ω longer than the slide step. Note that information may still be lost. Any input entities arriving or revised between Q_{i-1} and Q_i are discarded at Q_i if they took place before or at $Q_i{-}\omega$. To reduce the possibility of losing information, one may increase the size of ω. Doing so, however, decreases recognition efficiency. In what follows we give an example and a detailed account of the 'windowing' algorithm of RTEC.

Figure 2 illustrates windowing in RTEC. In this example we have $\omega{>}Q_i{-}Q_{i-1}$. To avoid clutter, Figure 2 shows streams of only five input entities. These are displayed below ω, with dots for instantaneous input entities and lines for durative ones. For the sake of the example, we are interested in just two fluents:

- A simple fluent `Se`. The maximal intervals of `Se` are displayed above ω in Figure 2.
- A statically determined fluent `Std`. For the example, the maximal intervals of `Std` are defined to be the union of the maximal intervals of the two durative input entities in Figure 2. The maximal intervals of `Std` are displayed above the `Se` intervals.

For simplicity, we assume that both `Se` and `Std` are defined only in terms of input entities, i.e. they are not defined in terms of other output entities.

Figure 2 shows the steps that are followed at an arbitrary query time, say Q_{138}. Figure 2(a) shows the state of RTEC as computation begins at Q_{138}. All input entities that took place before or at $Q_{137}{-}\omega$ were retracted at Q_{137}. The thick lines and dots represent the input entities that

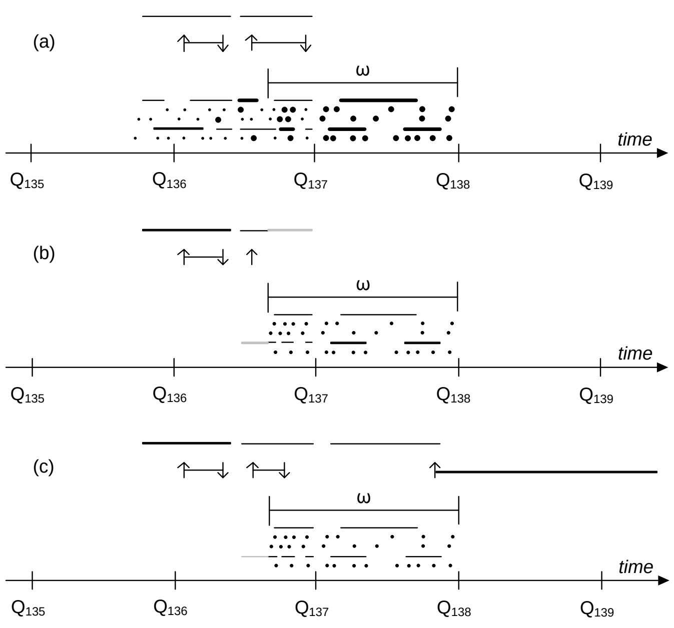


Figure 2: Windowing in RTEC.

arrived at RTEC between Q_{137} and Q_{138}; some of them took place before Q_{137}. Figure 2(a) also shows the maximal intervals for the fluents `Se` and `Std` that were computed and stored at Q_{137}.

Reasoning at Q_{138} considers the input entities that took place in $(Q_{138}{-}\omega, Q_{138}]$. All input entities that took place before or at $Q_{138}{-}\omega$ are discarded, as shown in Figure 2(b). For durative input entities that started before $Q_{138}{-}\omega$ and ended after that time, RTEC retracts the sub-interval up to and including $Q_{138}{-}\omega$. Figure 2(b) shows the interval of an input entity that is partially retracted in this way.

Now consider output entity intervals. At Q_i some of the maximal intervals computed at Q_{i-1} might have become invalid. This is because some input entities occurring in $(Q_i{-}\omega, Q_{i-1}]$ might have arrived or been revised after Q_{i-1}: their existence could not have been known at Q_{i-1}. Determining which output entity intervals should be (partly) retracted in these circumstances can be computationally very expensive [3]. We find it simpler, and more efficient, to discard all output entity intervals in $(Q_i{-}\omega, Q_i]$ and compute all intervals from scratch in that period. Output entity intervals that have ended before or at $Q_i{-}\omega$ are discarded. Depending on the user requirements, these intervals may be stored in a database for retrospective inspection of the activities of a system.

In Figure 2(b), the earlier of the two maximal intervals computed for `Std` at Q_{137} is discarded at Q_{138} since its endpoint is before $Q_{138}{-}\omega$. The later of the two intervals overlaps $Q_{138}{-}\omega$ (an interval 'overlaps' a time-point t if the interval starts before or at t and ends after or at that time) and is partly retracted at Q_{138}. Its starting point could not have been affected by input entities arriving between $Q_{138}{-}\omega$ and Q_{138} but its endpoint has to be recalculated. Accordingly, the sub-interval from $Q_{138}{-}\omega$ is retracted at Q_{138}.

In this example, the maximal intervals of `Std` are determined by computing the union of the maximal intervals of the two durative input entities shown in Figure 2. At Q_{138}, only the input entity intervals in $(Q_{138}{-}\omega, Q_{138}]$ are considered. In the example, there are two maximal intervals for `Std` in this period as can be seen in Figure 2(c). The earlier of them has its start-point at $Q_{138}{-}\omega$. Since that abuts the existing, partially retracted sub-interval for `Std` whose endpoint is $Q_{138}{-}\omega$, those two intervals are amalgamated into one continuous maximal interval as shown in Figure 2(c). In this way, the endpoint of the `Std` interval that overlapped $Q_{138}{-}\omega$ at Q_{137} is recomputed to take account of input entities available at Q_{138}. (In this particular example, it happens that the endpoint of this interval is the same as that computed at Q_{137}. That is merely a feature of this particular example. Had `Std` been defined e.g. as the *intersection* of the maximal intervals of the two durative input entities, then the intervals of `Std` would have changed in $(Q_{138}{-}\omega, Q_{137}]$.)

Figure 2 also shows how the intervals of the simple fluent `Se` are computed at Q_{138}. Arrows facing upwards (downwards) denote the starting (ending) points of the intervals of `Se`. First, in analogy with the treatment of statically determined fluents, the earlier of the two `Se` intervals in Figure 2(a), and its start and endpoints, are retracted. They occur before $Q_{138}{-}\omega$. The later of the two intervals overlaps $Q_{138}{-}\omega$. The interval is retracted, and only its starting point is kept; its new endpoint, if any, will be recomputed at Q_{138}. See Figure 2(b). For simple fluents, it is simpler, and more efficient, to retract such intervals completely and reconstruct them later from their start and endpoints by means of the domain-independent `holdsFor` rules, rather than keeping the sub-interval that takes place before $Q_{138}{-}\omega$, and possibly amalgamating it later with another interval, as we do for statically determined fluents.

The second step for `Se` at Q_{138} is to calculate its starting and ending points by evaluating the relevant `initiatedAt` and `terminatedAt` rules. For this, we only consider input entities that took place in $(Q_{138}{-}\omega, Q_{138}]$. Figure 2(c) shows the starting and ending points of `Se` in $(Q_{138}{-}\omega, Q_{138}]$. The last ending point of `Se` that was computed at Q_{137} was invalidated in the light of the new input entities that became available at Q_{138} (compare Figures 2(c)–(a)). Moreover, another ending point was computed at an earlier time.

Finally, in order to process `Se` at Q_{138} we use the domain-independent `holdsFor` to calculate the maximal intervals of `Se` given its starting and ending points. The later of the two `Se` intervals computed at Q_{137} became shorter when re-computed at Q_{138}. The second interval of `Se` at Q_{138} is open: given the input entities available at Q_{138}, we say that `Se` holds *since* time t, where t is the last starting point of `Se`.

The example used for illustration shows how RTEC performs reasoning. In the following section we have a closer look at the operation of RTEC, discussing each of its modules.

4 Operation of RTEC

Figure 3 illustrates the architecture of RTEC. In this section, we examine the modules of this architecture.

4.1 Offline Activities

Before the commencement of online activities, RTEC compiles the event description into a format that allows for more efficient reasoning. This is an offline process which is transparent to the user. The compiler is called via the predicate:

```
?- compileEventDescription(+EventDecription, +Declarations,
                           -CompiledEventDescription).
```

The input of this predicate is the event description file (such as Listing 1) and the declarations file (e.g. Listing 2). The output of this predicate is the compiled event description file which is subsequently used for online reasoning — see the bottom part ('offline') of Figure 3.

The aim of the compilation is to eliminate the number of unsuccessful evaluations of `happensAt`, `holdsFor` and `holdsAt`, and to introduce additional indexing information. These atoms are rewritten using specialised predicates, depending on whether they appear in the head or the body of a rule, whether they concern a simple or a statically determined fluent, and whether they host an input or an output entity.

When a `happensAt` predicate appears in the head of a rule in the Event Description, it is converted into `happensAtEv`. On the other hand, `happensAt` predicates that appear in the body of a

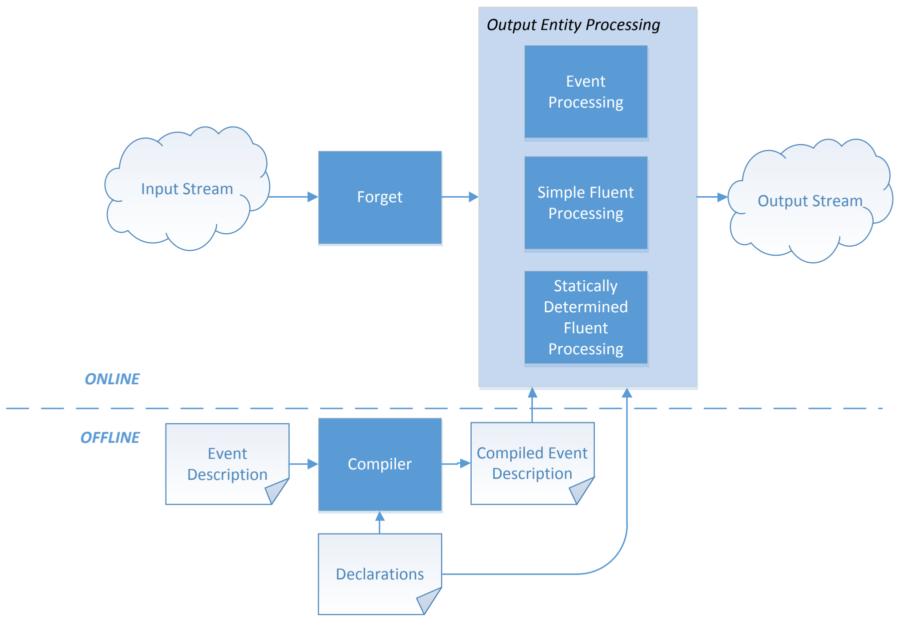


Figure 3: The architecture of RTEC.

rule are converted into `happensAtIE` (for events that are input entities) or `happensAtProcessed` (for events that are output entities).

Similarly, the `holdsFor` predicates appearing in the head of a domain-dependent rule, i.e. a rule for computing the maximal intervals of statically determined fluents, are rewritten using the predicate `holdsForSDFluent`. `holdsFor` predicates appearing in the body of a rule are translated into `holdsForProcessedSimpleFluent`, `holdsForProcessedIE` or `holdsForProcessedSDFluent` predicates, according to the fluent type they concern: simple fluents, input (statically determined) fluents, and output statically determined fluents, respectively.

In contrast to the `happensAt` and `holdsFor` predicates, `holdsAt` does not appear in the head of a rule. However, it may appear in the body of `initiatedAt` and `terminatedAt` rules; in the case of a simple fluent, the body `holdsAt` predicate is converted to a `holdsAtProcessedSimpleFluent`, whereas in the case of input or output statically determined fluent, it is converted into a `holdsAtProcessedIE` or a `holdsAtProcessedSDFluent`, respectively.

4.2 Online Activities

As already mentioned, reasoning is performed by means of continuous query processing, and concerns the computation of the maximal intervals of output entities, i.e. the intervals of simple and statically determined fluents, as well as the time-points in which events occur. At each query time Q_i, all input entities that took place before or at $Q_i{-}\omega$ are discarded/'forgotten' (see the 'forget' box in Figure 3). Then, RTEC computes and stores the intervals of each output entity (see 'output entity processing' in Figure 3). Recall that attention is restricted to hierarchical event descriptions. The form of the hierarchy is specified by the event description developer in the declarations using the `cachingOrder` predicate (see Section 2.2). RTEC adopts a caching technique where the fluents and events of the event description are processed in a bottom-up manner; this way, the intervals (resp. time-points) of the fluents (events) that are required for the processing of a fluent (event) of level n will simply be fetched from the cache without the need for re-computation. In the following sections we discuss the processes of 'forgetting', fluent and event processing.

4.2.1 Forget Mechanism

At each query time Q_i, RTEC first discards — 'forgets' — all input entities that end before or on $Q_i{-}\omega$. For each input entity available at Q_i, RTEC:

- Completely retracts the input entity if the interval attached to it ends before or on $Q_i - WM$.
- Partly retracts the interval of the input entity if it starts before or on $Q_i - WM$ and ends after that time. More precisely, RTEC retracts the input entity interval `(Start,End)` and asserts the interval `(`$Q_i{-}\omega$`,End)`.

4.2.2 Statically Determined Fluent Processing

After 'forgetting' input entities, RTEC computes and stores the intervals of each output entity. At the end of reasoning at each query time Q_i, all computed fluent intervals are stored in the computer memory as `simpleFPList` and `sdFPList` assertions. `I` in `sdFPList(Index, std, I, PE)` (resp. `simpleFPList(Index, se, I, PE)`) represents the intervals of statically determined fluent `Std` (simple fluent `Se`) starting in $(Q_i{-}\omega, Q_i]$, sorted in temporal order. `PE` stores the interval, if any, ending at $Q_i{-}\omega$. The first argument in `sdFPList` (`simpleFPList`) is an index that allows for the fast retrieval of stored intervals for a given fluent even in the presence of very large numbers of fluents. When the user queries the maximal intervals of a fluent, RTEC amalgamates `PE` with the intervals in `I`, producing a list of maximal intervals ending in $[Q_i{-}\omega, Q_i]$ and, possibly, an open interval starting in $[Q_i{-}\omega, Q_i]$.

Algorithm 1 `processSDFluent(Std,` $Q_i{-}\omega$`)`

```
1: indexOf(Std, Index)
2: retract(sdFPList(Index, Std, OldI, OldPE))
3: amalgamate(OldPE, OldI, OldList)
4: if Start,End:[Start,End) ∈ OldList ∧ End>$Q_i-\omega$ ∧ Start=< $Q_i-\omega$ then
5:     PE:=[(Start,$Q_i-\omega+1$)]
6: else
7:     PE:=[]
8: end if
9: holdsFor(SF, I)
10: assert(sdFPList(Index, SF, I, PE))
```

Listing 1 shows the pseudo-code of `processSDFluent`, the procedure for computing and storing the intervals of statically determined fluents. First, RTEC retrieves from `sdFPList` the maximal intervals of a statically determined fluent `Std` computed at Q_{i-1} and checks if there is such an interval that overlaps $Q_i{-}\omega$ (lines 1–8). In Listing 1, `OldI` represents the intervals of `Std` computed at Q_{i-1}. These intervals are temporally sorted and start in $(Q_{i-1}{-}\omega, Q_{i-1}]$. `OldPE` stores the interval, if any, ending at $Q_{i-1}{-}\omega$. RTEC amalgamates `OldPE` with the intervals in `OldI`, producing `OldList` (line 3). If there is an interval `[Start,End)` in `OldList` that overlaps $Q_i{-}\omega$, then the sub-interval `[Start,` $Q_i{-}\omega{+}1$`)` is retained. See `PE` in Listing 1. All intervals in `OldList` after $Q_i{-}\omega$ are discarded.

At the second step of `processSDFluent`, RTEC evaluates `holdsForSDFluent` rules to compute the `Std` intervals from input entities recorded as occurring in $(Q_i{-}\omega, Q_i]$ (line 9). Prior to the run-time recognition process, RTEC has transformed `holdsFor` rules concerning statically determined fluents into `holdsForSDFluent` rules, in order to avoid unnecessary `holdsFor` rule evaluations (see Section 4.1 for the compilation stage). The intervals of `Std` computed at the previous query time Q_{i-1} are not taken into consideration in the evaluation of `holdsForSDFluent` rules. The computed list of intervals `I` of `Std`, along with `PE`, are stored in `sdFPList` (line 10), replacing the intervals computed at Q_{i-1}. (Recall that, when the user queries the maximal intervals of a fluent, RTEC amalgamates `PE` with the intervals in `I`.)

4.2.3 Simple Fluent Processing

`processSimpleFluent`, the procedure for computing and storing simple fluent intervals, also has two parts. First, RTEC checks if there is a maximal interval of the fluent `Se` that overlaps $Q_i - \omega$. If there is such an interval then it will be discarded, while its starting point will be kept. Second, RTEC computes the starting points of `Se` by evaluating `initiatedAt` rules, without considering the starting points calculated at Q_{i-1}. The starting points are given to `holdsForSimpleFluent`, into which `holdsFor` calls computing the maximal intervals of simple fluents are translated at compile time. This program is defined as follows:

```
holdsForSimpleFluent(SP, Se, I) :-
     SP <> [],
     computeEndingPoints(Se, EP),
     makeintervals(SP, EP, I).
```

If the list of starting points is empty (first argument of `holdsForSimpleFluent`) then the empty list of intervals is returned. Otherwise, `holdsForSimpleFluent` computes the ending points `EP` of the fluent by evaluating `terminatedAt` rules, without considering the ending points calculated at Q_{i-1}, and then uses `makeIntervals` to compute its maximal intervals given its starting and ending points.

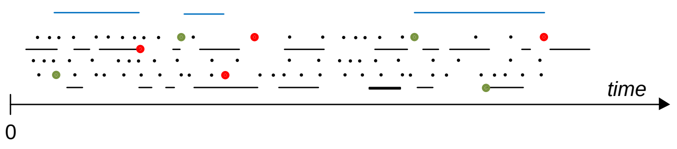


Figure 4: Maximal interval computation for simple fluents.

Figure 4 illustrates the process of `makeIntervals`. The black lines and dots indicate streams of durative and instantaneous input entities. The green dots denote starting/initiating points while the red dots indicate ending/terminating points. Note that `initiatedAt(F=V, T)` does not necessarily imply that `F<>V` at `T`. Similarly, `terminatedAt(F=V, T)` does not necessarily imply that `F=V` at `T`. `makeIntervals` finds all time-points `Ts` at which the fluent `Se` is initiated, and then, for each `Ts`, it computes the first time-point `Tf` after `Ts` at which `Se` is terminated. Suppose, for example, that `Se` is initiated at time-points 10 and 20 and terminated at time-points 25 and 30 (and at no other time-points). In that case `Se` holds at all `T` such that $10 <$ `T` ≤ 25.

4.2.4 Event Processing

`processEvents` is the procedure for computing and storing the time-points in which output events occur. In brief, `processEvents` first retracts all computed time-points of the output event in $(Q_i - \omega, Q_i]$, and then evaluates `happensAtEv` rules into which domain-dependent `happensAt` calls are translated at compile time.

5 A simplified language

Section 2 thoroughly discussed the features of the language of RTEC. In this section we will describe a simplified version of the RTEC language, and the Event Calculus, in general.

So far, in order to address an Event Recognition problem, the RTEC users need to be familiar with Prolog, and often have to take care to use several special, built-in predicates and functions – e.g.: the interval manipulation constructs – correctly. Another, simpler form of the Event Calculus, with its own compiler has been designed and developed. This simplified Event Calculus (SimplEC) aims at producing much simpler and readable Event Descriptions, even for demanding domains, and at the same time it maintains most of the expressiveness of RTEC.

The time model remains linear, with integer time points. What changes is the representation of the events and fluents, as well as the format of the rules within an Event Description.

First of all, SimplEC is not Prolog, in contrast with the classic RTEC event patterns. It is designed to resemble simple statements in English, with some pseudocode and mathematical elements. It does not contain obscure symbols like `:-`, or `\+`. Instead, it contains simple words like `if`, `or`, `not`, as well as the comma operator that indicates conjunction. There are also a few special tokens, like `happens` that stem from the classic RTEC language.

5.1 Representing events and fluents

In this simplified version of the RTEC language, there are no special predicates like `holdsAt(F=V,T)`, `holdsFor(F=V, I)`, or `happensAt(E, T)` that indicate happening, holding, initiation and termination, as we have seen on the section about RTEC. All events and fluents are represented using a compound term of the form

$$\text{functor}(\ldots \mathit{arguments} \ldots)[= \mathit{value}] \tag{1}$$

where square brackets denote the optional part of the term. This structure allows for the representation of both events and fluents. Specifically, when this term contains a value, then it is always a fluent. When the value is absent, it may either be a fluent with the default "`true`" value, or an event which, by definition, has no value. The distinction between the latter two categories is based on the context.

5.1.1 Built-in tokens

There are a handful of built-in keywords that precede the compound terms shown in 1 and help identify how the term is being used. These keywords are:

- `initiate`: Indicates that the following term is a fluent and is being initiated. Similar to classic RTEC's `initiatedAt`.
- `terminate`: Indicates that the following term is a fluent and is being terminated. Similar to classic RTEC's `terminatedAt`.
- `start`: Denotes the point in time where the following fluent starts holding. Similar to classic RTEC's `start`.
- `end`: Denotes the point in time where the following fluent ceases holding. Similar to classic RTEC's `end`.
- `happens`: Indicates that the following term is an event and expresses its happening. Similar to classic RTEC's `happensAt`.

5.1.2 Defining simple fluents

The SimplEC user can state initiation conditions for simple fluents as follows:

$$\begin{array}{l}\textbf{initiate } F_1 \textbf{ if}\\ \quad E[,\\ \quad \text{condition}]^*.\end{array} \tag{2}$$

Termination conditions are structured in the same way, the only difference being that instead of the keyword **initiate** we use the keyword **terminate**.

F_1 is a fluent, following the syntax we have seen in (1). E can be either an ordinary event structured as in (1), or one of the two special events (**start** F_2, **end** F_2), denoting the starting and ending points of the holding interval of another fluent F_2, respectively.

[, condition]* denotes an optional set of conditions surrounding the happening of E, that are significant for the initiation or termination of F_1. Such a condition could be one of the following:

- F_3, meaning that fluent F_3 must hold at that specific timepoint.
- **start** F_3, meaning that F_3 must start holding.
- **end** F_3, meaning that F_3 must cease holding.
- **not** F_3, meaning that F_3 must not hold.
- **happens** E_2, indicating the happening of event E_2.
- **not happens** E_2, indicating the absense of event E_2.
- atemporal constraint.

In the same fashion that we define the initiation and termination points of a simple fluent, we can also define the happening of events that are output entities. This can be achieved by replacing the **initiate** and **terminate** keywords with the **happens** keyword and putting an event instead of a fluent at the head of the statement. The body of the statement is structured exactly as in the initiation/termination case.

5.1.3 Defining statically determined fluents

A SimplEC statement defining a statically determined fluent consists of conjunctions, disjunctions and negations of other fluents, possibly nested within each other. The keyword **or** is used in disjunctions, the keyword **not** is used in negations, and conjunctions use the comma operator. There are no interval manipulation constructs, as the relevant information is contained in the occurrence of the respective operators and keywords.

The following example shows the definition of moving as a statically determined fluent. Two persons are considered to be moving together while they are both walking and they appear close to each other. This can be expressed in SimplEC as follows:

$$\begin{array}{l} \textit{moving}(P_1, P_2)\ \textbf{if} \\ \quad \textit{walking}(P_1), \\ \quad \textit{walking}(P_2), \\ \quad \textit{close}(P_1, P_2). \end{array} \tag{3}$$

Another, more involved example is the definition of fighting. Fighting between two persons takes place when at least one of them is moving abruptly, none of them is totally inactive, and at the same time they are very close to each other. In SimplEC, we write:

$$\begin{array}{l} \textit{fighting}(P_1, P_2)\ \textbf{if} \\ \quad (\textit{abrupt}(P_1)\ \textbf{or}\ \textit{abrupt}(P_2)), \\ \quad \textit{close}(P_1, P_2), \\ \quad \textbf{not}\ (\textit{inactive}(P_1)\ \textbf{or}\ \textit{inactive}(P_2)). \end{array} \tag{4}$$

5.2 Event Description

An Event Description in SimplEC is a set of statements that follow the above-mentioned syntax. Let's consider a sample Event Description that only contains statements (3) and (4). It would look like this:

```
moving(P1,P2) if
      walking(P1),
      walking(P2),
      close(P1,P2,34).

fighting(P1,P2) if
((      abrupt(P1) or abrupt(P2)),
      close(P1,P2)),
      not(inactive(P1) or inactive(P2)).
```
Listing 6: Event description in SimplEC.

This set of rules translates into the following RTEC Event Description:

```
holdsFor(moving(P1,P2)=true, I) :-
      holdsFor(walking(P1)=true, I1),
      holdsFor(walking(P2)=true, I2),
      holdsFor(close(P1,P2,34)=true, I3),
      intersect_all([I1,I2,I3], I).

holdsFor(fighting(X,Y)=true, I) :-
      holdsFor(active(X)=true, I1),
      holdsFor(active(X)=true, I2),
      union_all([I1,I2], I3),
      holdsFor(close(X,Y)=true, I4),
      intersect_all([I3,I4], I5),
      holdsFor(inactive(X)=true, I6),
      holdsFor(inactive(X)=true, I7),
      relative_complement_all(I5, [I6,I7], I).
```
Listing 7: Event description of Listing 6 translated into RTEC.

5.3 Declarations and Dependencies

The compiler of SimplEC parses a set of statements in the simple language and, based on its grammar, translates the statements to rules in the RTEC format. Moreover, it constructs the declarations required for computing narrative assimilation queries. The declarations distinguish, for the benefit of RTEC, between simple and statically determined fluents, and between input and output entities (events and fluents). In SimplEC statement (3), for instance, the compiler will detect three statically determined fluents, namely $moving(\_,\_)$, $walking(\_)$, and $close(\_,\_)$, of which $moving(\_,\_)$ is an output entity, as it appears in the head of the statement, and the other two are input entities, as they do not appear to be defined by other, simpler entities. This knowledge of the compiler is summarized in the following set of declarations:

$$\begin{array}{l}
\mathsf{sDFluent}(moving(\_,\_)).\\
\mathsf{sDFluent}(walking(\_)).\\
\mathsf{sDFluent}(close(\_,\_)).\\
\mathsf{outputEntity}(moving(\_,\_)).\\
\mathsf{inputEntity}(walking(\_)).\\
\mathsf{inputEntity}(close(\_,\_)).
\end{array} \tag{5}$$

The declarations also express the 'caching hierarchy', that is, the order in which fluents and events are processed. RTEC performs bottom-up processing whereby fluents and events of level 1 of a hierarchy are processed first, subsequently moving to levels 2, 3, etc. The computed intervals of each level are cached. This way, fluent and event intervals of some level n may be simply fetched from memory when required in the processing of fluents and events of some higher level m.

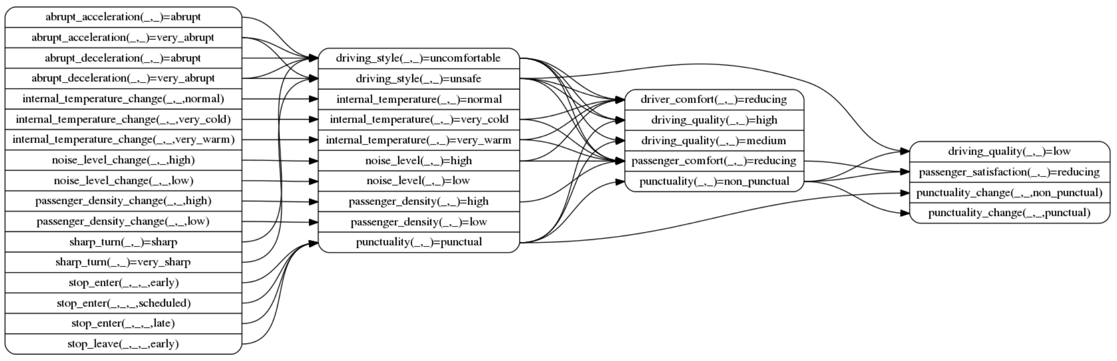

abrupt_acceleration(_,_)=abrupt
abrupt_acceleration(_,_)=very_abrupt
abrupt_deceleration(_,_)=abrupt
abrupt_deceleration(_,_)=very_abrupt
internal_temperature_change(_,_,normal)
internal_temperature_change(_,_,very_cold)
internal_temperature_change(_,_,very_warm)
noise_level_change(_,_,high)
noise_level_change(_,_,low)
passenger_density_change(_,_,high)
passenger_density_change(_,_,low)
sharp_turn(_,_)=sharp
sharp_turn(_,_)=very_sharp
stop_enter(_,_,_,early)
stop_enter(_,_,_,scheduled)
stop_enter(_,_,_,late)
stop_leave(_,_,_,early)
driving_style(_,_)=uncomfortable
driving_style(_,_)=unsafe
internal_temperature(_,_)=normal
internal_temperature(_,_)=very_cold
internal_temperature(_,_)=very_warm
noise_level(_,_)=high
noise_level(_,_)=low
passenger_density(_,_)=high
passenger_density(_,_)=low
punctuality(_,_)=punctual
driver_comfort(_,_)=reducing
driving_quality(_,_)=high
driving_quality(_,_)=medium
passenger_comfort(_,_)=reducing
punctuality(_,_)=non_punctual
driving_quality(_,_)=low
passenger_satisfaction(_,_)=reducing
punctuality_change(_,_,non_punctual)
punctuality_change(_,_,punctual)

Figure 5: Dependency graph of the city transport management event description.

To aid the user, the compiler may display the dependency graph of the event description. This is a directed graph where each vertex corresponds to a fluent or event, and for each pair of vertices (i, j) there is an edge from i to j if i appears in the body of a statement defining j. Figure 5 shows the dependency graph of an event description for city transport management. In this figure, we can observe the way in which the events and fluents affect each other. In the leftmost part of the figure there are vertices with no incoming edges. These correspond to input events and fluents that form the narrative upon which all complex activities will be recognised. In this domain, the input entities include information about the acceleration and deceleration of transport vehicles, changes in internal temperature, noise level or passenger density, as well as the time of arrival at a stop.

On the right of the bottom layer, there are two other layers of events and fluents that have both incoming and outgoing edges. These are output entities that also contribute to the definition of other output entities. On these layers we combine information from the bottom layer and produce higher-level information. For instance, we can recognise complex activities such as driving style and quality, vehicle punctuality and the passengers' comfort level. In the rightmost part of the figure, there is one last layer with no outgoing edges. These are the complex activities of the highest level—consider, for instance, passenger satisfaction.

5.4 Running the compiler

The compiler of the SimplEC language is written in Prolog and is based on Prolog's Definite Clause Grammars, which are often used for NLP purposes and building parsers, in general.

In order to use the compiler, the user must type:

```
$ swipl -l simplEC.prolog
```

And then, supposing that the user has written their set of SimplEC statements in a file named "simplec_rules.txt", type:

```
?- simplEC('simplec_rules.txt', 'event_description.prolog',
                        'declarations.prolog', 'dependency_graph.txt').
```

This will take the file with the event patterns in SimplEC and translate it to an Event description in the classic RTEC format, along with its respective declarations, as we have seen in detail, earlier in this manual. But before the Event Recognition process is started, the rules must be further compiled by the compiler of RTEC this time.

To save time and to unify the two compilation steps, there is an executable script named "compile.sh" that calls both compilers and produces the compiled event description and the declarations that are needed for RTEC to start performing Event Recognition. To call that script, the user should write in a command window:

```
$ bash compile.sh simplec_rules.txt
```

6 Further Information

The repository of RTEC — `https://github.com/aartikis/RTEC` — includes event descriptions of two application domains, activity recognition and event recognition for city transport management, as well as datasets and execution scripts for experimentation.

References

[1] Elias Alevizos, Alexander Artikis, Kostas Patroumpas, Marios Vodas, Yannis Theodoridis, and Nikos Pelekis. How not to drown in a sea of information: An event recognition approach. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 984–990, 2015.

[2] Alexander Artikis, Marek J. Sergot, and Georgios Paliouras. Run-time composite event recognition. In *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*, pages 69–80, 2012.

[3] Alexander Artikis, Marek J. Sergot, and Georgios Paliouras. An event calculus for event recognition. *IEEE Trans. Knowl. Data Eng.*, 27(4):895–908, 2015.

[4] Alexander Artikis, Matthias Weidlich, Avigdor Gal, Vana Kalogeraki, and Dimitrios Gunopulos. Self-adaptive event recognition for intelligent transport management. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 319–325, 2013.

[5] Alexander Artikis, Matthias Weidlich, François Schnitzler, Ioannis Boutsis, Thomas Liebig, Nico Piatkowski, Christian Bockermann, Katharina Morik, Vana Kalogeraki, Jakub Marecek, Avigdor Gal, Shie Mannor, Dimitrios Gunopulos, and Dermot Kinane. Heterogeneous stream processing and crowdsourcing for urban traffic management. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 712–723, 2014.

[6] Robert A. Kowalski and Marek J. Sergot. A logic-based calculus of events. *New Generation Comput.*, 4(1):67–95, 1986.

[7] D. Luckham and R. Schulte. Event processing glossary — version 1.1. Event Processing Technical Society, July 2008.

[8] Kostas Patroumpas, Alexander Artikis, Nikos Katzouris, Marios Vodas, Yannis Theodoridis, and Nikos Pelekis. Event recognition for maritime surveillance. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 629–640, 2015.

[9] T. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*. Morgan, 1987.