Event Calculus for Run-Time reasoning (RTEC): User Manual

Alexander Artikis, Christos Vlassopoulos and Periklis Mantenoglou

Institute of Informatics & Telecommunications,
NCSR "Demokritos",
Complex Event Recognition group
cer.iit.demokritos.gr

March 28, 2023

1 Introduction

The Event Calculus for Run-Time reasoning (RTEC) is an open-source, logic programming implementation of the Event Calculus [7], optimised for computing continuous queries on data streams [3]. RTEC has been successfully used for *composite event recognition* ('event pattern matching') in various real-world application domains. 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 [8]. 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:

- Activity recognition (see [3] and `http://cer.iit.demokritos.gr/cerar`).
- City transport & traffic management [3, 4, 5].
- Maritime monitoring (see [11, 10] and `http://cer.iit.demokritos.gr/cermm`).
- Fleet management (see [13] `http://cer.iit.demokritos.gr/cerfm`).
- Multi-agent system protocols (see [9]).

The novelty of RTEC lies in the following implementation techniques:

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

This is the second version of RTEC (RTECv2). This version is more expressive as it supports:

1. *Cyclic dependencies among CEs.*

1.1 Software requirements & installation

RTEC is cross-platform and its only software requirement is a Prolog implementation. RTEC has been tested under YAP[1] and SWI[2] Prolog, operating in Ubuntu Linux. In order to download RTEC, open a terminal and type:

```
git clone https://github.com/aartikis/RTEC
```

Listing 1: Downloading RTEC.

In the following section, we will demonstrate a simple execution of RTEC on a toy example in order to illustrate the core features of RTEC. For more complex use cases, please follow the instructions in our github repository[3].

1.2 A simple example

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". RTEC computes the effects of events on such properties. In the terminology of RTEC, these properties are called "fluents".

We would like to specify the conditions under which Chris is 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 may define a happy man in our example. Winning the lottery makes someone rich and we assume that losing their wallet causes them to stop being rich.

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

```
% Rules defining output entities
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).

% Grounding of input entities
grounding(go_to(Person, Place)) :- person(Person), place(Place).
grounding(lose_wallet(Person)) :- person(Person).
grounding(win_lottery(Person)) :- person(Person).
% Grounding of output entities
grounding(location(Person)=Place) :- person(Person), place(Place).
grounding(rich(Person)=true) :- person(Person).
grounding(happy(Person)=true) :- person(Person).
```

Listing 2: Event description in RTEC.

[1] https://en.wikipedia.org/wiki/YAP_(Prolog)
[2] https://www.swi-prolog.org/
[3] https://github.com/aartikis/RTEC/blob/master/docs/contents.md

Following Prolog's convention, variables start with an upper-case letter, while predicates and constants start with a lower-case letter. The rules with heads "initiatedAt", "terminatedAt" and "holdsFor" define dependencies among events and fluents. The rules with head "grounding" define the domains of the arguments of events and fluents. We specify the entities of each domain with the code below. Create another file, say "`toy_var_domain.prolog`", and add the following code:

```
% This is our variable domain
person(chris).

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

Listing 3: Domains of entities.

The only person in our world is Chris and the places we are interested in are his home, his work and a pub. Finally, we use the following event narrative to test our event description:

```
go_to|9|9|chris|work
win_lottery|13|13|chris
go_to|17|17|chris|pub
lose_wallet|19|19|chris
go_to|21|21|chris|home
```

Listing 4: Event narrative.

According to this event narrative, Chris goes to work at 9:00 . Then, at 13:00, he finds out that he has won the lottery. Subsequently, he goes to the pub at 17:00, loses his wallet at 19:00 and, finally, returns home at 21:00.

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 instruct RTEC to use these files before execution. This may be done by editing "execution scripts/defaults.toml", the configuration file of RTEC. Update the TOML table for the toy application with the following key-value pairs:

```
# Default parameter values for the toy application
[toy]
event_description = 'path/to/toy_rules.prolog'
input_mode = "csv"
input_providers = ['path/to/toy_data.csv']
results_directory = "path/to/results/dir"
window_size = 30
step = 30
start_time = 0
end_time = 30
clock_tick = 1
goals = []
background_knowledge = ['path/to/toy_var_domain.prolog']
stream_rate = 1
dependency_graph_flag = false
dependency_graph_directory = 'path/to/graphs/dir'
include_input = false
```

Listing 5: Narrative assimilation script.

We supply our rules (see Listing 2) to RTEC with the "event_description" parameter. Moreover, we provide the domains of the entities of our event description (Listing 3) and our event narrative (Listing 4) with parameters "background_knowledge" and "input_providers", respectively. The paths of all files in the above table should point to their locations. Set the results directory to

a new folder of your choice. The full documentation of the execution parameters of RTEC is presented in Section 6.

Before execution, RTEC will consult this table of the configuration file and set its execution parameters accordingly. In order to execute RTEC, open a terminal and run the following commands: "'cd execution scripts"' and execute the command "'.
run_rtec.sh –app=toy –interactive"'.

```
cd execution scripts
./run_rtec.sh --app=toy --interactive
```

Listing 6: Run RTEC on the toy example.

The "app" flag instructs RTEC to run for the The "interactive" flag allows for the user to pose queries to the system after the execution of RTEC, by not killing the Prolog terminal. We may 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)]
```

$(14, 22)$ denotes a right-open interval stating that Chris is happy from time 14, right after he won the lottery, until time 21, when he leaves the pub. In our world, 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 = (rich(chris)=true),
I = [(14,20)] ? ;
F = (location(chris)=home),
I = [(22,inf)] ? ;
F = (location(chris)=pub),
I = [(18,22)] ? ;
F = (location(chris)=work),
I = [(10,18)] ? ;
F = (happy(chris)=true),
I = [(14,22)] ? ;
```

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

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

In this example, RTEC performed batch reasoning, i.e., the event narrative was loaded to RTEC and, then, it computed the maximal intervals of composite activities based on all input entities. This can be seen from the `window_size` parameter of the configuration file (see Listing 5), where `window_size` was set to 30 in order to span over the entire event narrative. RTEC, however, has been optimised to operate online, and process input streams in temporal windows (see Section 3.1).

In Sections 2 and 3, we describe the language and the main modules of RTEC, with a brief discussion on their algorithmic implementation. Next, Sections 4, 5 and 6 provide usage instructions for running RTEC on a custom application.

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 Event Description

RTEC employs the Event Calculus dialect described in this section. The time model 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 [12]. 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 event descriptions in RTEC.

2.1 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.

RTEC feature two special event types: `start` and `end`. These events have arity 1 and their argument a is fluent-value pair. For some fluent-value pair `F=V`, `happensAt(start(F=V), T)` (resp. `happensAt(end(F=V), T)`) is generated internally by RTEC iff T is the starting point (resp. ending point) of an interval during which `F=V` holds continuously.

2.2 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.:

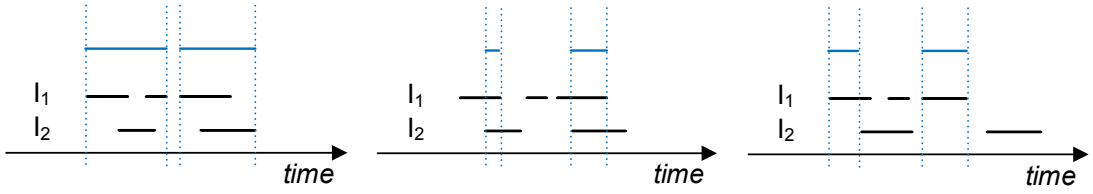
```
intersect_all([[(26,31)], [(21,26),(30,40)]], [(30,31)])
```

`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 2 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].

3 Reasoning

3.1 Windowing

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 $Q_1, Q_2, \ldots$. 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

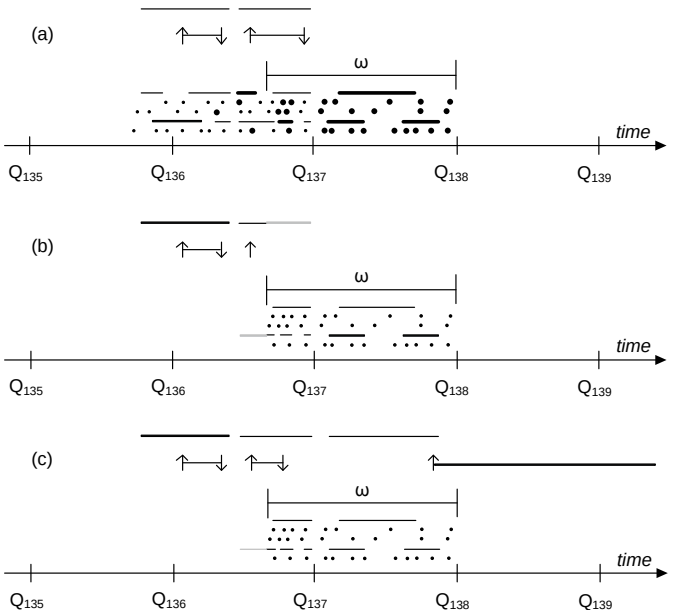


Figure 2: Windowing in RTEC.

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. The same goes for input entities that took place in $(Q_i - \omega, Q_{i-1}]$ and were subsequently revised after Q_{i-1}. Therefore, 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. Consider the following example.

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 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`.

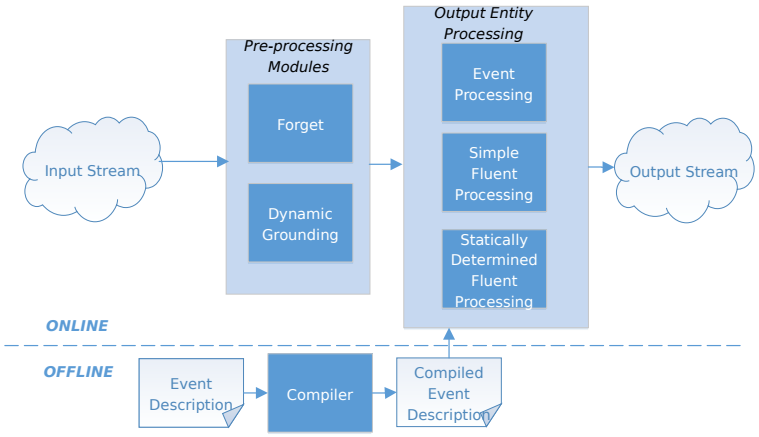


Figure 3: The architecture of RTEC.

3.2 Operations

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

3.2.1 Offline Activities

Before online execution, RTEC compiles the event description into a format that allows for more efficient reasoning. This is an offline process that is transparent to the user. The compiled event description file 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 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.

3.2.2 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). Moreover, if RTEC at least one domain of the event description has been declared as dynamic, RTEC performs an additional memory maintenance step, aimed at enhancing reasoning efficiency ('dynamic grounding' 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 hierarchy of the events and fluents in the event description is detected by the compiler and specified in the compiled event description using the `cachingOrder` predicate. 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', 'dynamic grounding', fluent and event processing.

Forgetting. 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)`.

Dynamic Grounding. After forgetting redundant input entities, RTEC proceeds to the 'dynamic grounding' pre-processing step. The purpose of this step is to identify all domain elements, e.g., persons and places in our example, which may take part in some event or fluent of the current window. All remaining domain elements, e.g., those appearing in properties computed in previous windows, are not considered during reasoning, thus avoiding redundant computations. The dynamic grounding module follows these steps:

- Fetch from memory the set *OldElements* containing all domain elements appearing in a property computed in a previous window and identify the ones which appear in some property with open intervals. These elements are stored in the set *PersistElements*.
- Find all domain elements appearing in some input entity of the current window and store them in the set *NewElements*.
- Retract all elements in the set $\mathit{OldElements} \setminus (\mathit{NewElements} \cup \mathit{PersistElements})$.
- Assert all elements in the set $(\mathit{NewElements} \cup \mathit{PersistElements}) \setminus \mathit{OldElements}$.

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 $[Q_i - \omega, Q_i]$.

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

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>Qi−ω ∧ Start=< Qi−ω then
5:   PE:=[(Start,Qi−ω+1)]
6: else
7:   PE:=[]
8: end if
9: holdsFor(SF, I)
10: assert(sdFPList(Index, SF, I, PE))
```

$Q_i{-}\omega$, 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 3.2.1 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`.)

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.

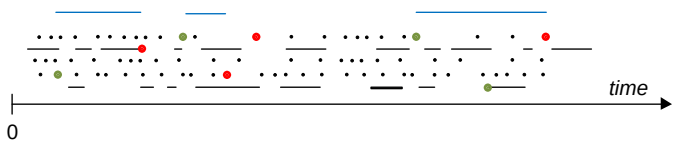


Figure 4: Maximal interval computation for simple fluents.

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 $10 < \texttt{T} \leq 25$.

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.

4 Writing an Event Description

The repository of RTEC contains event descriptions for several applications, such as maritime situational awareness and multi-agent protocols, in the `examples` folder. For instance, the file `examples/toy/resources/patterns/rules.prolog` contains the event description for the toy application (see Listing 2). In the remainder of this section, we describe the language used for writing custom event descriptions in RTEC.

4.1 Sorts

RTEC uses a many-sorted domain specification language $\mathcal{L}_{\mathsf{RTEC}}$ based on Prolog. A domain specification D, such as the specification of our toy domain (see Section 1.2), may be modelled in $\mathcal{L}_{\mathsf{RTEC}}$ as a Prolog program $\mathcal{P}_D$. $\mathcal{L}_{\mathsf{RTEC}}$ has the following domain-dependent sorts:

- **Time-points.** $\mathbb{T}_D$ contains the time-points of domain D, as well as the variables reserved for time-points. For example, we modelled our toy domain using a time-line of natural numbers, i.e., $\mathbb{T}_{toy} = \mathcal{N} \cup \mathbb{T}_{toy}^{vars}$.
- **Intervals.** $\mathbb{I}_D$ is the set of intervals of domain D. $i \in \mathbb{I}_D$ iff $i = [t_j, t_k)$, $t_j, t_k \in \mathbb{T}_D$ and $t_j \leq t_k$ or at least one endpoint is a time variable.
- **Lists of Maximal Intervals.** $\mathbb{L}_D$ is the set of lists of maximal intervals of domain D. $l = [(t_1^s, t_1^f), \ldots, (t_n^s, t_n^f)] \in \mathbb{L}_D$ iff $\forall i \in [1, n] : t_i^s, t_i^f \in \mathbb{T}_D$ and, for all time-points t_i^s, t_i^f and t_{i+1}^s that are not variables, it holds that $t_i^s \leq t_i^f < t_{i+1}^s$. In other words, lists of maximal intervals are temporally sorted and contain disjoint intervals.
- **Domain objects.** $\mathbb{O}_D$ contains the constants and variables that denote objects for domain D. For example, `chris` and `pub` are objects of our toy domain.
- **Domain object properties.** A domain may include a set of predicates $\mathbb{R}_D$ which model properties of domain objects. In our toy example, `person` $\in \mathbb{R}_{toy}$ is a unary predicate stating that an object is a person. Predicates of higher arity may be required for certain object properties. For example, `owner` $\in \mathbb{R}_{toy}$ could be a binary predicate stating by `owner(chris, pub)` that `chris` is the owner of the `pub`. The full expressive power of Prolog may be used when writing rules defining object properties.
- **Events.** $\mathbb{E}_D$ contains the events of domain D. An event follows the syntax $e(arg_1, arg_2, \ldots)$, where $e \in \mathbb{E}_D^S$ is an event type of D and $arg_i \in \mathbb{O}_D$. For example, `win_lottery, go_to` $\in \mathbb{E}_{toy}^S$, while `win_lottery`$(Person)$, `go_to`$(chris, pub) \in \mathbb{E}_{toy}$. Note that event type e may have arity 0, in which case event type e is also an event.
- **Fluent types.** $\mathbb{F}_D$ contains the fluents of domain D. A fluent follows the syntax $f(arg_1, arg_2, \ldots)$, where $f \in \mathbb{F}_D$ is a fluent type of D and $arg_i \in \mathbb{O}_D$. For example, `rich` $\in \mathbb{F}_{toy}^S$, while `rich`$(Person)$, `rich`$(chris) \in \mathbb{F}_{toy}$. Note that fluent type f may have arity 0, in which case f is also a fluent.
- **Fluent values.** For every fluent type $f \in \mathbb{F}_D^S$, $\mathbb{V}_D^f$ is the set of all possible values for fluent type f. Often $\mathbb{V}_D^f =$ `{true, false}`. For example, $\mathbb{V}_{toy}^{rich} =$ `{true, false}` and $\mathbb{V}_{toy}^{location} =$ `{work, home, pub}`.

Table 1 presented the domain independent predicates of RTEC which comprise the following sorts of $\mathcal{L}_{\mathsf{RTEC}}$:

- **Event Calculus predicates.** Set $\mathbb{R}_{EC}$ contains the core predicates of RTEC used to express event occurrence, fluent-value change and fluent-value persistence through the common sense law of inertia. These predicates are `happensAt`/2, `initially`/1, `initiatedAt`/2, `terminatedAt`/2, `holdsAt`/2 and `holdsFor`/2.
- **Interval manipulation constructs.** Set $\mathbb{R}_I$ contains the interval operations `union_all`/2, `intersect_all`/2, `relative_complement_all`/3.

4.2 Grammar

We present the Extended Affix Grammar (EAG) of $\mathcal{L}_{\mathsf{RTEC}}$, which is a context-free grammar where terminal symbols may be accompanied by a list of parameters (affixes) [6]. Like variables with the same names in Prolog, these parameters are substituted with the same non-terminal symbol for all of their instances in the same rule.

Grammar 1 presents the EAG of an event description in RTEC. An event description rule may be a definition for an output event or fluent, a grounding for the objects in an input or output entity, an index declaration or a dynamic domain declaration. Index declarations instruct the compiler of RTEC to use a specific argument of an event or fluent as its index. Dynamic domain declarations inform the compiler about the domains whose objects are determined dynamically from the input entities of the stream.

All predicates in a rule defining a simple fluent refer to the same time-point. This requirement is specified with the non-terminal symbol <time-var> in the rule for simple fluent, which is used as a parameter in the non-terminal symbols concerning the body predicates of a simple fluent definition. Moreover, the list of maximal intervals of an output statically determined fluent is computed by the last interval operation in its body (see <mil-var> in the corresponing grammar rule).

Grammar 1.

⟨*event-description*⟩ ::= ⟨*domain-rule*⟩
| ⟨*domain-rule*⟩ ⟨*event-description*⟩

⟨*domain-rule*⟩ ::= ⟨*simple-fluent-rule*⟩
| ⟨*output-sdf-rule*⟩
| ⟨*output-event-rule*⟩
| ⟨*grounding-rule*⟩
| ⟨*index-declaration*⟩
| ⟨*dynamic-domain-declaration*⟩

⟨*simple-fluent-rule*⟩ ::= ⟨*init-or-term*⟩ '(' ⟨*fluent-value-pair*⟩ ',' ⟨*time-var*⟩ ') :-' ⟨*happensAt-atom*⟩(*time-var*) ⟨*sf-body-condition-list*⟩(*time-var*)

⟨*output-sdf-rule*⟩ ::= 'holdsFor(' ⟨*fluent-value-pair*⟩ ',' ⟨*mil-var*⟩ ') :-' ⟨*holdsFor-atom*⟩ ⟨*sdf-body-condition-list*⟩[*mil-var*]

⟨*output-event-rule*⟩ ::= 'happensAt(' ⟨*event*⟩ ',' ⟨*time-var*⟩ ') :-' ⟨*happensAt-atom*⟩(*time-var*) ⟨*body-condition-list*⟩(*time-var*)

⟨*init-or-term*⟩ ::= 'initiatedAt'
| 'terminatedAt'

⟨*fluent-value-pair*⟩ ::= ⟨*fluent*⟩ '=' ⟨*value*⟩(*fluent*)

⟨*happensAt-atom*⟩(*time-var*)::= 'happensAt(' ⟨*event*⟩ ',' ⟨*time-var*⟩ ')'
| 'happensAt(start(' ⟨*fluent-value-pair*⟩ ')' ⟨*time-var*⟩ ')'
| 'happensAt(end(' ⟨*fluent-value-pair*⟩ ')' ⟨*time-var*⟩ ')'

⟨*sf-body-condition-list*⟩(*time-var*) ::= ',' ⟨*sf-body-condition*⟩(*time-var*) ⟨*sf-body-condition-list*⟩(*time-var*)
| '.'

⟨*sf-body-condition*⟩(*time-var*) ::= [‘not’] ⟨*happensAt-atom*⟩(*time-var*)
| [‘not’] ⟨*holdsAt-atom*⟩(*time-var*)
| [‘not’] ⟨*domain-object-property*⟩ ‘(’ ⟨*domain-object-list*⟩ ‘)’

⟨*sdf-body-condition-list*⟩(*mil-var*) ::= ‘,’ ⟨*sdf-body-condition*⟩ ⟨*sdf-body-condition-list*⟩
| ‘,’ ⟨*interval-operation*⟩(*mil-var*) ‘.’

⟨*sdf-body-condition*⟩ ::= ⟨*holdsFor-atom*⟩
| ⟨*interval-operation*⟩
| [‘not’] ⟨*domain-object-property-atom*⟩

⟨*sdf-body-condition*⟩(*mil-var*) ::= ⟨*interval-operation*⟩(*mil-var*)

⟨*holdsAt-atom*⟩(*time-var*) ::= ‘holdsAt(’⟨*fluent-value-pair*⟩ ‘,’ ⟨*time-var*⟩ ‘)’

⟨*holdsFor-atom*⟩ ::= ‘holdsFor(’⟨*fluent-value-pair*⟩ ‘,’ ⟨*mil-var*⟩ ‘)’

⟨*interval-operation*⟩(*mil-var*) ::= ‘union_all(’⟨*list-of-interval-lists*⟩ ‘,’ ⟨*mil-var*⟩ ‘)’
| ‘intersect_all(’⟨*list-of-interval-lists*⟩ ‘,’ ⟨*mil-var*⟩ ‘)’
| ‘relative_complement_all(’⟨*mil-var-other*⟩ ‘,[’⟨*list-of-interval-lists*⟩ ‘],’⟨*mil-var*⟩ ‘)’

⟨*interval-operation*⟩ ::= ‘union_all(’⟨*list-of-interval-lists*⟩ ‘,’ ⟨*mil-var*⟩ ‘)’
| ‘intersect_all(’⟨*list-of-interval-lists*⟩ ‘,’ ⟨*mil-var*⟩ ‘)’
| ‘relative_complement_all(’⟨*mil-var*⟩ ‘,[’⟨*list-of-interval-lists*⟩ ‘],’ ⟨*mil-var*⟩ ‘)’

⟨*grounding-rule*⟩ ::= ‘grounding(’⟨*fluent-value-pair*⟩‘):-’⟨*domain-object-property-list*⟩

⟨*index-declaration*⟩ ::= ‘index(’⟨*event*⟩‘,’⟨*domain-object*⟩‘).’
| ‘index(’⟨*fluent-value-pair*⟩‘,’⟨*domain-object*⟩‘).’

⟨*dynamic-domain-declaration*⟩ ::= ‘dynamicDomain(’⟨*domain-object-property-atom*⟩‘).’

⟨*domain-object-property-list*⟩ ::= ⟨*domain-object-property-atom*⟩‘.’
| ⟨*domain-object-property-atom*⟩‘,’⟨*domain-object-property-list*⟩

⟨*domain-object-property-atom*⟩ ::= ⟨*domain-object-property*⟩ ‘(’ ⟨*domain-object-list*⟩ ‘)’

⟨*domain-object-list*⟩ ::= ⟨*domain-object*⟩ [‘,’ ⟨*domain-object-list*⟩]

⟨*list-of-interval-lists*⟩ ::= ⟨*mil-var*⟩ [‘,’ ⟨*list-of-interval-lists*⟩]

The non-terminal symbols that are not in the head of any production rule are substituted directly with an element from the corresponding sort of RTEC, among those described in Section 4.1. For example, <fluent> is substituted by an element of set $\mathbb{F}_D$ and <mil-var> is satistituted with a variable in $\mathbb{L}_D$. Moreover, <time-var>$\in \mathbb{T}_D$, <event>$\in \mathbb{E}_D$, <value> [<f>] $\in \mathbb{V}_D^{ftype}$, <domain-object>$\in \mathbb{O}_D$, <domain-object-property>$\in \mathbb{R}_D$, <event>$\in \mathbb{E}_D$

5 Format of Input Data

RTEC processes event narratives containing information about the input events and fluents that take place at each point in time. A record of an RTEC-compatible event narrative may contain the following types of information:

1. The occurrence of an instantaneous input event.
2. The value of an input fluent at some time-point.
3. The value that an input fluent has continuously in some temporal interval.

Records begin with the corresponding event/fluent type and then contain information about the record's arrival time, the event/fluent's occurrence time and its attributes. Record fields are separated with the character '|' and, depending on the type of information they contain, have the following format:

1. *input event*:

```
EventType|ArrivalTime|OccurrTime|Attr1|...|AttrN
```

2. *input fluent value at time-point*:

```
FluentType|ArrivalTime|OccurrTime|Value|Attr1|...|AttrN
```

3. *input fluent value in interval*:

```
FluentType|ArrivalTime|OccurrStartTime|OccurrEndTime|Value|Attr1|...|AttrN
```

Notice that there is a distinction between arrival time and occurrence time; the former expresses the time at which the entity arrived at RTEC, while the latter expresses the time at which the entity was detected by the sensor. The arrival time is not translated to the Event Calculus predicates, but considered when deciding which input entities should be loaded.

As an example, the first record of Listing 4, which contains the event narrative of our toy example, is:

```
go_to|9|9|chris|work
```

The event type of this record is `go_to` and its attributes are `chris` an `work`. The arrival time and the occurrence time of this record match and are equal to 9. Therefore, this record arrived at RTEC as soon as the corresponding event took place.

As another example, consider that `working` and `break` are input statically determined fluents monitoring the activities of a person who is at work. Suppose that we have the following event narrative:

```
go_to|9|9|chris|work
working|15|10|15|true|chris
break|15|15|true|chris
break|16|16|true|chris
working|21|17|21|true|chris
go_to|21|21|chris|home
```

In this narrative, the records for the `working` fluent contain the temporal intervals during which `working=true`. For instance, the second and the fifth record of the narrative state that Chris is working in the time periods $[10, 15)$ and $[17, 21)$, respectively. The arrival times of these records are equal to the ending point of the corresponding time periods. The records for the `break` fluent report its occurrences at specific points in time. For example, the third and fourth record of the narrative describe that Chris is taking a break at time-points 15 and 16, respectively.

6 Execution parameters

RTEC supports the following execution parameters:

- `event_description`: The file containing the event description of the application. See, e.g., Listing 2 of our toy example.
- `input_mode`: RTEC may process streams of input events from csv files or named pipes (see Section 6.1). Possible values: "csv" and "fifo".
- `input_providers`: The csv files or named pipes providing input events to RTEC. See, e.g., the csv records in Listing 4 of our toy example.
- `results_directory`: The location where the intervals computed by RTEC and its execution logs will be stored.
- `window_size`: RTEC processes event streams using a sliding window (see Section 3.1). The window size is constant during execution and is equal to the value of this parameter.
- `step`: In RTEC, the step is the distance between two consecutive query times (see Section 3.1). The step is constant during execution and is equal to the value of this parameter.

- `start_time`: RTEC processes the input narratives starting from events and fluents that occur at the time-point provided with this parameter.
- `end_time`: RTEC does not process any input entities that take place after the time-point provided with this parameter.
- `clock_tick`: The temporal distance between two consecutive time-points.
- `goals`: RTEC will run all Prolog queries provided as a list in this parameter before performing narrative assimilation.
- `background_knowledge`: RTEC will consult all Prolog files provided as a list in this parameter before performing narrative assimilation.
- `stream_rate`: If the `input_mode` parameter is set to `fifo`, the value of this parameter controls the rate at which RTEC processes input records. See Section 6.1 for more information.
- `dependency_graph_flag`: If the value of this parameter is `true`, then RTEC will produce the dependency graph of the provided event description (see Section 6.2).
- `dependency_graph_directory`: If the `dependency_graph_flag` is set to `true`, the value of this parameter specifies the directory in which RTEC will store the dependency graph of the event description.
- `include_input`: If the `dependency_graph_flag` is set to `true`, then setting the value of this parameter to `true` instructs RTEC to include input entities in the dependency graph of the event description.

6.1 Reasoning over Historical Data vs a Live Stream

RTEC may process streams of input events from:

- csv files.
- named pipes.

Csv files contain historical data concerning event streams that have concluded. In contrast, named pipes are being updated in real time with new records of the input event streams. We use the execution parameter `input_mode` to inform RTEC about the type of input providers we use. When `input_mode` is "csv", RTEC reads the input event streams from csv files, whereas, when `input_mode` is "fifo", RTEC receives the input event streams incrementally as their records are written in named pipes.

RTEC supports the following options:

1. `input_mode`="csv" and the parameter `input_providers` is a list of csv files.
2. `input_mode`="fifo" and `input_providers` is a list of named pipes.
3. `input_mode`="fifo" and `input_providers` is a list of csv files.

Case 3 may be selected in order to *simulate* a live stream using historical data. To do this, the script `run_rtec.sh` opens a new named pipe for each input csv files and creates a new process that writes the records of each csv file into the corresponding named pipe incrementally, according to the timestamp of each record. Subsequently, RTEC is executed in `fifo` mode, having as input the aforementioned named pipes.

Note that each execution of RTEC uses the same file type for all input providers, i.e., it is not possible to process one live stream and one csv file with historical data in the same execution of RTEC.

In `fifo` mode, RTEC interprets timestamps in seconds, and thus waits for the input entities of the next window to arrive at the system by sleeping for a number of seconds equal to the value of the `window_size` parameter. The `stream_rate` parameter informs RTEC that the input

streams arrive faster or slower than their expected velocity, derived by interpreting time-stamps as temporal distance in seconds. For instance, if `stream_rate=2` and `window_size=10`, then RTEC waits for 5 seconds for the input entities of the next window, as they arrive at double the default velocity.

6.2 Dependency Graph of an Event Description

Before narrative assimilation, RTEC processes the provided event description offline and transforms it into a compiled version that is suitable for online reasoning. One module of the compiler of RTEC detects the dependencies among the events and the fluents of the event description and constructs the *dependency graph* of the event description. Then, RTEC determines an efficient order of processing for all events and fluents by following a topological sorting of the dependency graph.

The dependency graph of an event description provides a visual representation of all dependencies among its events and fluents, aiding debugging and paving the way towards explainability. For these reasons, the user may pass the value `true` to the parameter `dependency_graph_flag` of the configuration file in order to instruct RTEC to produce the dependency graph of the event description as a png file. This functionality requires GraphViz[4]. The parameter `dependency_graph_directory` specifies the directory in which RTEC will store the derived dependency graph. Moreover, passing the value `true` to the parameter `include_input` results in a dependency graph that contains both input and output events and fluents. Otherwise, only output entities are included in the graph.

Figure 5 present the dependency graphs of the event description used in our toy example when input entities are included or omitted.

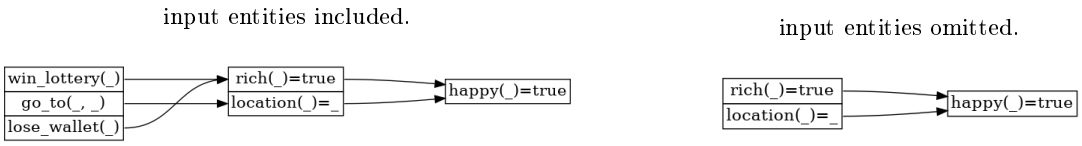


Figure 5: Dependency graphs of the "toy" event description

7 Further Information

The repository of RTEC — `https://github.com/aartikis/RTEC` — includes event descriptions of application domains, 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] `https://graphviz.org/`

[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] Cornelis H. A. Koster. Affix grammars for natural languages. In *Attribute Grammars, Applications and Systems*, 1991.

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

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

[9] Periklis Mantenoglou, Manolis Pitsikalis, and Alexander Artikis. Stream reasoning with cycles. In Gabriele Kern-Isberner, Gerhard Lakemeyer, and Thomas Meyer, editors, *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning, KR 2022, Haifa, Israel. July 31 - August 5, 2022*, 2022.

[10] 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.

[11] Manolis Pitsikalis, Alexander Artikis, Richard Dreo, Cyril Ray, Elena Camossi, and Anne-Laure Jousselme. Composite event recognition for maritime monitoring. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, DEBS 2019, Darmstadt, Germany, June 24-28, 2019.*, pages 163–174, 2019.

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

[13] Efthimis Tsilionis, Nikolaos Koutroumanis, Panagiotis Nikitopoulos, Christos Doulkeridis, and Alexander Artikis. Online event recognition from moving vehicles: Application paper. *TPLP*, 19(5-6):841–856, 2019.