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

Alexander Artikis, Christos Vlassopoulos and Periklis Mantenoglou

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

October 31, 2022

1 Introduction

The Event Calculus for Run-Time reasoning (RTEC) is an open-source, logic programming implementation of the Event Calculus [6], 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 [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:

- Activity recognition (see [3] and `http://cer.iit.demokritos.gr/cerar`).

- City transport & traffic management [3, 4, 5].

- Maritime monitoring (see [10, 9] and `http://cer.iit.demokritos.gr/cermm`).

- Fleet management (see [12] `http://cer.iit.demokritos.gr/cerfm`).

- Multi-agent system protocols (see [8]).

The novelty of RTEC lies in the following implementation techniques:

1. *Interval manipulation*, that helps in expressing succinctly complex temporal phenomena (Section 2).

2. *Windowing*, that supports real-time query computation (Section 3).

3. *Caching*, that helps in avoiding unnecessary re-computations.(Section 4).

4. *Indexing*, that makes RTEC robust to data streams that are irrelevant to the queries we want to compute (Section 4).

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

1. *Cyclic dependencies among CEs* (Section 5).

1.1 Software requirements & installation

RTEC is cross-platform. The only software requirement is a Prolog implementation. RTEC has been tested under YAP[1] and SWI[2] Prolog, operating in Ubuntu Linux.

In the following section, we will demonstrate a simple execution of RTEC on a toy example. The only requirement for following these steps is to download RTEC. Simply open a terminal and type:

```
1  git clone https://github.com/aartikis/RTEC
2  cd RTEC
```

Listing 1: Downloading RTEC.

In order to run RTEC in more complex use cases, please follow the instructions in our github repository[3].

1.2 A simple example

This example illustrates the core features of RTEC. A more thorough description of the expressive power and the reasoning algorithms of RTEC is presented in later sections.

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 RTEC terminology, 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:

```
1  initiatedAt(rich(X)=true, T) :-
2      happensAt(win_lottery(X), T).
3
4  terminatedAt(rich(X)=true, T) :-
5      happensAt(lose_wallet(X), T).
6
7  initiatedAt(location(X)=Y, T) :-
8      happensAt(go_to(X,Y), T).
9
10 holdsFor(happy(X)=true, I) :-
11     holdsFor(rich(X)=true, I1),
12     holdsFor(location(X)=pub, I2),
13     union_all([I1,I2], I).
```

Listing 2: 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:

```
1  % Information about all our events and fluents.
2  % - Is each entity an event or a fluent?
```

---

[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

```
 3  % – Is it an input or an output entity?
 4  % – Choose an argument to be used as index for quicker access.
 5
 6  event(go_to(_,_)).
 7  inputEntity(go_to(_,_)).
 8  index(go_to(Person,_), Person).
 9
10  event(lose_wallet(_)).
11  inputEntity(lose_wallet(_)).
12  index(lose_wallet(Person), Person).
13
14  event(win_lottery(_)).
15  inputEntity(win_lottery(_)).
16  index(win_lottery(Person), Person).
17
18  simpleFluent(location(_)=home).
19  outputEntity(location(_)=home).
20  index(location(Person)=home, Person).
21
22  simpleFluent(location(_)=pub).
23  outputEntity(location(_)=pub).
24  index(location(Person)=pub, Person).
25
26  simpleFluent(location(_)=work).
27  outputEntity(location(_)=work).
28  index(location(Person)=work, Person).
29
30  simpleFluent(rich(_)=true).
31  outputEntity(rich(_)=true).
32  index(rich(Person)=true, Person).
33
34  simpleFluent(rich(_)=false).
35  outputEntity(rich(_)=false).
36  index(rich(Person)=false, Person).
37
38  sDFluent(happy(_)=true).
39  outputEntity(happy(_)=true).
40  index(happy(Person)=true, Person).
41
42  sDFluent(happy(_)=false).
43  outputEntity(happy(_)=false).
44  index(happy(Person)=false, Person).
45
46  % How are the fluents grounded?
47  % Define the domain of the variables.
48
49  grounding(location(Person)=Place) :- person(Person), place(Place).
50  grounding(rich(Person)=true) :- person(Person).
51  grounding(rich(Person)=false) :- person(Person).
52  grounding(happy(Person)=true) :- person(Person).
53  grounding(happy(Person)=false) :- person(Person).
54
55  % In what order will the output entities be processed by RTEC?
56
57  cachingOrder(location(_)=home).
58  cachingOrder(location(_)=pub).
59  cachingOrder(location(_)=work).
60  cachingOrder(rich(_)=true).
61  cachingOrder(rich(_)=false).
62  cachingOrder(happy(_)=true).
```

```
63  cachingOrder(happy(_)=false).
```
Listing 3: 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 sections, 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, invoke Prolog and consult the compiler as follow:

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

Then, 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:

```
1  % This is our variable domain
2
3  person(chris).
4
5  place(home).
6  place(pub).
7  place(work).
```
Listing 4: Variable domain.

The contents of this file are used for *output entity* (e.g. fluent) grounding (see Listing 3, lines 49-53). Finally, to test our event description, we need an event narrative, such as the following:

```
1  % This is our narrative of events, given as input.
2
3  updateSDE(story, 9, 21) :-
4  assert(happensAtIE(go_to(chris, work), 9)),
5  assert(happensAtIE(win_lottery(chris), 13)),
6  assert(happensAtIE(go_to(chris, pub), 17)),
7  assert(happensAtIE(lose_wallet(chris), 19)),
8  assert(happensAtIE(go_to(chris, home), 21)).
```
Listing 5: 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 afore-mentioned files and start the RTEC engine. This may be done by creating a Prolog script, "toy_queries.prolog" that contains the following:

```
1
2   :-['toy_event_stream.prolog'].
3   :-['RTEC.prolog'].
4   :-['toy_var_domain.prolog'].
5   :-['toy_declarations.prolog'].
6   :-['toy_rules_compiled.prolog'].
7
8   performER :-
9        initialiseRecognition(ordered, nodynamicgrounding, nopreprocessing, 1),
10       updateSDE(story, 9, 21),
11       eventRecognition(21, 21).
```

Listing 6: Narrative assimilation script.

The code of Listing 6 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. First, we initialize RTEC by setting four parameters. The first two parameters state that the input facts are temporally sorted and that the entities appearing in the event narrative (people and places, in our example) do not have to be grounded dynamically before CE recognition. Dynamic grounding often accelerates reasoning when processing large event narrative (see Section 4.2.2). The third parameter declares that our input data does not require some form of preprocessing, while the last parameter is the distance between two consecutive time-points, which, in our dataset, is 1 time unit. Subsequently, we load the event narrative using the "updateSDE" predicate we defined in Listing 5, and finally we call the built-in eventRecognition predicate of RTEC for narrative assimilation. We provide two 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 time-point 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 time-points, i.e., from the beginning of the stream.

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.

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 [11]. 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.

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`

2.1 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:

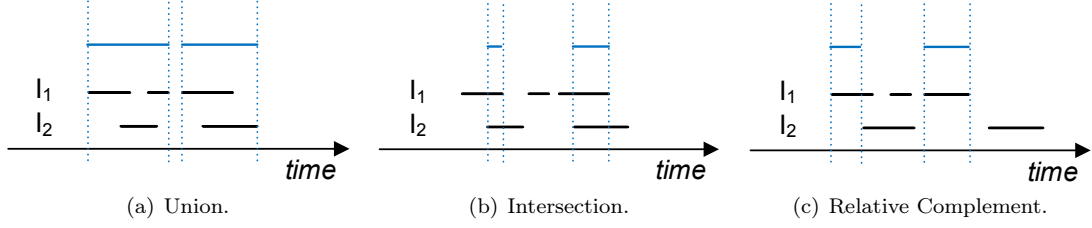(a) Union.   (b) Intersection.   (c) Relative Complement.

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.

```
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)).

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].

2.2 Declarations

The declarations of our example were presented in Listing 3. 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 $Q_1, Q_2, \ldots$ 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 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
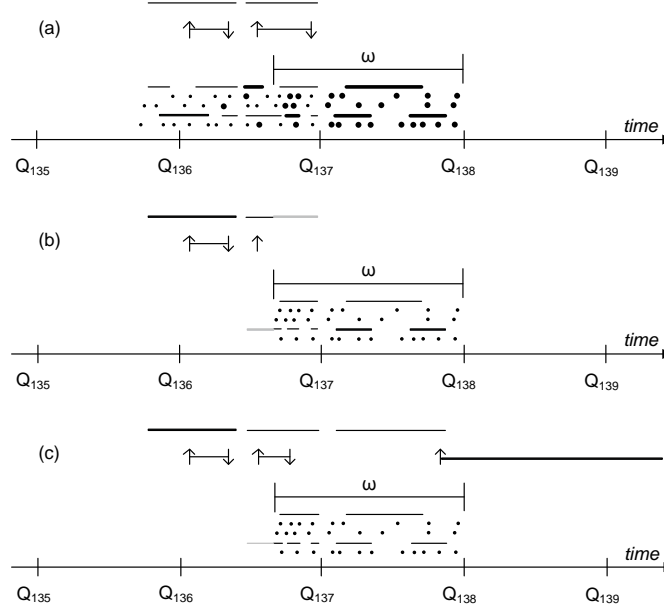
Figure 2: Windowing in RTEC.

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 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

10

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.

3.1 Running RTEC with windows

We now demonstrate how RTEC processes input events which arrive at the system in temporal windows. Consider that the event narrative of our running example arrives in three windows. Create a new Prolog file, name it "toy_event_stream_windows.prolog" and add the following updateSDE rules:

```
1  % This is our narrative of events, arriving at the system in three windows.
2
3  updateSDE(story, 8, 15) :-
4        assert(happensAtIE(go_to(chris, work), 9)),
5        assert(happensAtIE(win_lottery(chris), 13)).
6
7  updateSDE(story, 15, 19) :-
8        assert(happensAtIE(go_to(chris, pub), 17)),
9        assert(happensAtIE(lose_wallet(chris), 19)).
10
11 updateSDE(story, 19, 21) :-
12        assert(happensAtIE(go_to(chris, home), 21)).
```

Listing 7: Event narrative in three temporal windows.

In the first temporal window, spanning from 8:00 to 15:00, there are two input events. Chris goes to work at time 9 and he wins the lottery at time 13. In the next window, taking place from 15:00 to 19:00, Chris goes to the pub at time 17 and he loses his wallet at time 19. Finally, Chris goes home at time 21. This is the only event in the final window, which spans from 19:00 to 21:00. In RTEC, windows are left-open. For example, the first window includes all events taking place within (8,15].

In a dynamic environment, events should stream into the system as soon as they are available. In such cases, the temporal length of windows may vary as the stream progresses. In our example, e.g., the length of the first window is 7, since it includes the time-points {9, 10, ..., 15}. The temporal lengths of the second and third window are 4 and 2, respectively.

In order to query such a narrative, we may create a Prolog script, named "toy_queries_windows.prolog", which contains the following rules:

```
1  :-['toy_event_stream_windows.prolog'].
2  :-['RTEC.prolog'].
3  :-['toy_var_domain.prolog'].
4  :-['toy_declarations.prolog'].
5  :-['toy_rules_compiled.prolog'].
6
7  performER(window1) :-
8        initialiseRecognition(ordered, nodynamicgrounding, nopreprocessing, 1),
9        updateSDE(story, 8, 15),
10        eventRecognition(15, 7).
11
12 performER(window2) :-
13        updateSDE(story, 15, 19),
14        eventRecognition(19, 4).
15
16 performER(window3) :-
17        updateSDE(story, 19, 21),
18        eventRecognition(21, 2).
```

Listing 8: Narrative assimilation for each window.

In this case, we use one "performER" rule for each window. We only need to initiliase RTEC once, before processing the first window (see line 8). Each rule loads the event narrative which takes place within its corresponding window and, subsequently, invokes "eventRecognition" in order to perform narrative assimilation. For instance, the second "performER" rule, loads the input events within the second window, which spans from time 15 to 19, and then instructs RTEC to perform narrative assimilation at query time 19, while taking into account all events taking place at any of the last 4 time-points, including time 19, i.e., all events occurring in the interval (15,19].

Going back to Prolog, halt any open Prolog session and start a new one. Then, load the Prolog script:

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

Perform narrative assimilation for the first window by typing the command:

```
?- performER(window1).
```

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

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

RTEC has computed that Chris started working at time 10. "`inf`" denotes an open interval; the starting point of the property "`location(chris)=work`" is specified by an "`initiatedAt`" rule but no termination point for this property has been identified thereafter. In such cases, RTEC computes that the property continues to hold until further information becomes available. Similarly, Chris starts being both rich and happy at time 14 and, thankfully for Chris, no termination points for these properties have been identified since.

Perform narrative assimilation for the second window by typing the command:

```
?- performER(window2).
```

Then, see the maximal intervals of all fluent-value pairs computed by RTEC by executing "`holdsFor(F,I).`". You will receive the following output:

```
F = (location(chris)=home),
I = [] ? ;
F = (location(chris)=pub),
I = [(18,inf)] ? ;
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,inf)] ? ;
F = (happy(chris)=false),
I = []
```

Since Chris goes to the pub at time 17, which takes place within this window, RTEC has computed that the property "`location(chris)=work`" holds up until time 17, i.e., in the interval [10, 18]. Also, the property "`location(chris)=pub`" begins to hold after time 17. Since there is no information that Chris's location changes again after that event, this property is said to take place in the open interval [18,inf]. Finally, this window contains the event "`lose_wallet(chris)`" which produces a termination point for the property "`rich(chris)=true`". As a result, the open interval derived in the previous window closes in this window, by computing the interval [14, 20].

Perform narrative assimilation for the third window by typing the command:

```
?- performER(window3).
```

Then, see the maximal intervals of all fluent-value pairs computed by RTEC by executing "`holdsFor(F,I).`". You will receive the following output:

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

According to the event narrative in this window, Chris leaves the pub and goes home at time 21. As a result, the property "`location(chris)=pub`" holds up until time 21, and thus RTEC computes the interval [18,22). Afterwards, Chris is at home, and thus RTEC computes the interval [22,inf) for the property "`location(chris)=home`". The same event also terminates the property "`happy(chris)=true`", since now Chris is neither rich nor at the pub.

Notice that, after processing each window, the output of RTEC contains only the intervals starting or ending in the current window. For example, after processing the third window of our example, RTEC does not re-compute the maximal intervals [10,18) and [14,20) for the properties "`happy(chris)=true`" and "`happy(chris)=true`", respectively, as the events which started or ended these intervals are before the current window (19, 21]. This is achieved with the 'forget' mechanism of RTEC, which deletes from memory all events that are not required in the current window (Section 4.2.1). Furthermore, the 'dynamic grounding' module accelerates reasoning in each window by dynamically retracting all entities (e.g., persons) not taking part in any event of the current window, and asserting only the ones which produce some event (Section 4.2.2). In the following section we provide a more technical description of RTEC, discussing each one 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 online execution, 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(+Declarations, +EventDescription,
                              -CompiledEventDescription).
```

The input of this predicate is the event description file (such as Listing 2) and the declarations file (e.g. Listing 3). 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
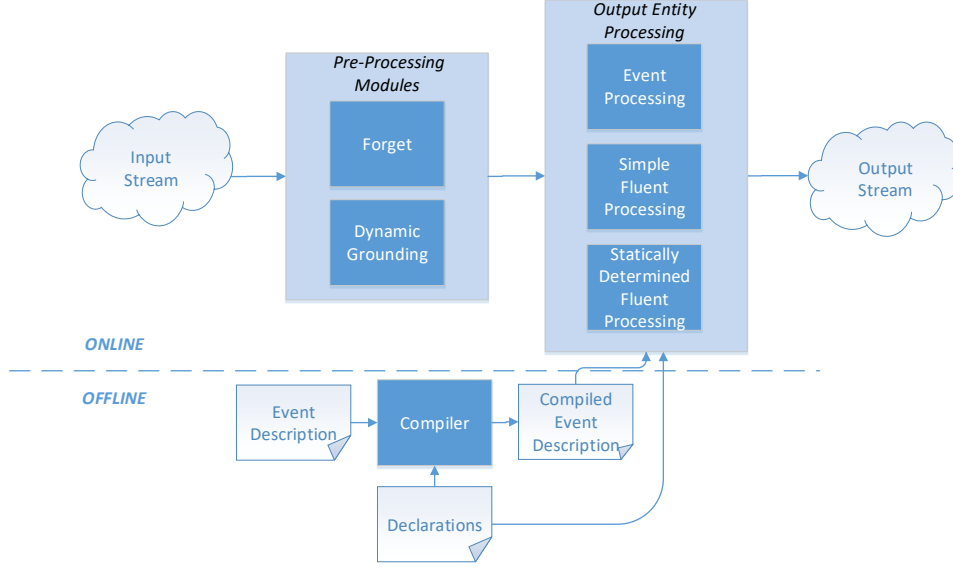
Figure 3: The architecture of RTEC.

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.

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). Moreover, if RTEC was initialised with the "`dynamicgrounding`" flag, it 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 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

15

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.

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 Dynamic Grounding Mechanism

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 *OldElements* \ (*NewElements* ∪ *PersistElements*).

- Assert all elements in the set (*NewElements* ∪ *PersistElements*) \ *OldElements*.

4.2.3 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 $Q_i - \omega$, 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.

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))
```

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.)

4.2.4 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.
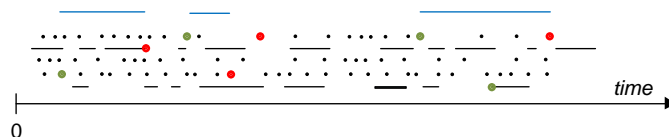

Figure 4: Maximal interval computation for simple fluents.

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 $10 < T \leq 25$.

4.2.5 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 Cyclic Dependencies

The modules of RTEC described so far are restricted to processing event descriptions whose dependency graphs are directed acyclic graphs. Temporal specifications of real applications, however, often include cyclic dependencies among properties. To address this issue, which arises in event descriptions of multi-agent system protocols and maritime situational awareness applications [8], RTEC has been extended to handle event descriptions with cyclic dependencies.

5.1 Running Example

Consider the following example:

```
1  initially(strength(X)=full).
2
3  initiatedAt(strength(X)=lowering,T):-
4      happensAt(starts_working(X),T),
5      holdsAt(strength(X)=full,T).
6
7  initiatedAt(strength(X)=tired,T):-
8      happensAt(stops_working(X),T),
9      holdsAt(strength(X)=lowering,T).
10
11  initiatedAt(strength(X)=full,T):-
12      happensAt(stops_sleeping(X),T),
13      holdsAt(strength(X)=tired,T).
```
Listing 9: Event description featuring properties with cyclic dependencies.

Listing 9 presents a set of rules describing the "strength" levels of some person X as her day progresses. Initially, i.e., before any events occur, her strength is full, which is denoted with the "initially" fact in the line 1. Next, the strength level of X drops from "full" to "lowering", after the event "starts_working" takes place. Subsequently, the value of "strength" becomes "tired" as soon as she leaves work, i.e., the event "stops_working" takes place, and finally it rises back to "full" after a good night's sleep, denoted with the event "stops_sleeping".

The modules of RTEC described so far cannot process the event description in Listing 9, because, in this case, it is impossible to specify a hierarchy of fluent-value pairs denoting their processing order. The fluent-value pair "strength(X)=full" depends on "strength(X)=tired" due to the rule starting in line 11. Next, "strength(X)=tired" requires the truth values of "strength(X)=lowering" (line 7), which, in turn, depends on "strength(X)=full" (line 3). As a result, the dependency graph corresponding to the event description in Listing 9 contains a cycle. RTEC has been modified for processing efficiently event descriptions which include cyclic dependencies, such as the one presented in Listing 9.

In order to run RTEC on the above event description, create a new Prolog file named "toy_rules_cyc.prolog" and store Listing 9. Next, store in another new file, "toy_declarations_cyc.prolog", the following declarations:

```
 1  event(starts_working(_)).
 2  inputEntity(starts_working(_)).
 3  index(starts_working(Person), Person).
 4
 5  event(stops_working(_)).
 6  inputEntity(stops_working(_)).
 7  index(stops_working(Person), Person).
 8
 9  event(stops_sleeping(_)).
10  inputEntity(stops_sleeping(_)).
11  index(stops_sleeping(Person), Person).
12
13  simpleFluent(strength(_)=full).
14  outputEntity(strength(_)=full).
15  index(strength(Person)=full, Person).
16
17  simpleFluent(strength(_)=lowering).
18  outputEntity(strength(_)=lowering).
19  index(strength(Person)=lowering, Person).
20
21  simpleFluent(strength(_)=tired).
22  outputEntity(strength(_)=tired).
23  index(strength(Person)=tired, Person).
24
25  grounding(strength(Person)=full) :- person(Person).
26  grounding(strength(Person)=lowering) :- person(Person).
27  grounding(strength(Person)=tired) :- person(Person).
28
29  cachingOrder(strength(_)=full).
30  cachingOrder(strength(_)=lowering).
31  cachingOrder(strength(_)=tired).
32
33  %%%%%%%%%%%% cycles %%%%%%%%%%%%%%
34
35  cyclic(strength(_)=full).
36  cyclic(strength(_)=tired).
37  cyclic(strength(_)=lowering).
```

Listing 10: Event and fluent declarations featuring properties with cyclic dependencies.

For event descriptions with cyclic dependencies, the user needs to add "cyclic" declarations specifying which fluent-value pairs are part of a cycle in the dependency graph. In our example, the dependencies between all three fluent-value pairs of our event description form a cycle, and are thus denoted as "cyclic" in the above file. Notice that, in this case, the order of the "cachingOrder" declarations does not matter because all three properties depend on each other. For properties in the same cycle, we can swap their caching order without any effect on reasoning efficiency.

In order to compile this event description, open a terminal, go to the RTEC/ directory, invoke Prolog and consult the compiler as follow:

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

Then, execute the following query:

```
?- compileEventDescription('toy_declarations_cyc.prolog',
                 'toy_rules_cyc.prolog', 'toy_rules_cyc_compiled.prolog').
```

The new file "toy_rules_cyc_compiled.prolog" contains the compiled event description corresponding to Listings 9 and 10.

The file "toy_var_domain_cyc.prolog" should contain the following variables:

```
1 person(mary).
```
Listing 11: Variable domain.

Next, save in file "`toy_event_stream_cyc.prolog`" the following event narrative:

```
1 updateSDE(story, 0, 30) :-
2    assert(happensAtIE(starts_working(mary), 5)),
3    assert(happensAtIE(stops_working(mary), 12)),
4    assert(happensAtIE(stops_sleeping(mary), 21)).
```
Listing 12: Event narrative.

Finally, create the Prolog script "`toy_queries_cyc.prolog`" containing the following code:

```
1  :-['toy_event_stream_cyc.prolog'].
2  :-['RTEC.prolog'].
3  :-['toy_var_domain_cyc.prolog'].
4  :-['toy_declarations_cyc.prolog'].
5  :-['toy_rules_cyc_compiled.prolog'].
6
7  performER :-
8       initialiseRecognition(ordered, nodynamicgrounding, nopreprocessing, 1),
9       updateSDE(story, 0, 30),
10      eventRecognition(30, 31).
```
Listing 13: Narrative assimilation script.

Notice that, in this case, we used a window size of 31 time-points while the query time is
30. This is done to include time-point 0 in the window, because "`initially`" is translated as
"`initiatedAt`" at time -1. Therefore, in this example, the property "`strength(mary)=full`" is
initiated at time -1, and thus starts to hold at time 0.

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

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

We are now ready for narrative assimilation. Type the command:

```
?- performER.
```

Then, we run the query "`holdsFor(F,I).`" and the output of RTEC is the following:

```
F = (strength(mary)=full),
I = [(0,6),(22,inf)] ? ;
F = (strength(mary)=lowering),
I = [(6,13)] ? ;
F = (strength(mary)=tired),
I = [(13,22)] ? ;
```

According to the event narrative in Listing 12, no event takes place until time 5. Therefore,
Mary's strength level stays full in the interval [0,6). Mary starts working at time 5 and stops
working at time 12. As a result, the property "`strength(mary)=lowering`" holds in the interval
[6,13), after which her strength level drops to "`tired`". The property "`strength(mary)=tired`"
persists until she finishes her sleep at time 21. Afterwards, from time 22 until the end of narrative,
Mary's strength level is full.

5.2   Syntax

The only additional requirement for running RTEC on event descriptions which include cyclic
dependencies is stating explicitly which fluent-value pairs take part in a cycle. This is done with
the predicate "`cyclic`" in the "declarations" file before compilation, as shown in Listing 10.

6   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] 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] 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.

[9] 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.

[10] 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.

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

[12] 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.