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

Periklis Mantenoglou ✉
Institute of Informatics & Telecommunications, NCSR “Demokritos”, Greece

Manolis Pitsikalis ✉
Institute of Informatics & Telecommunications, NCSR “Demokritos”, Greece

Alexander Artikis ✉
Institute of Informatics & Telecommunications, NCSR “Demokritos”, Greece

Contents

1 Introduction

The Event Calculus for Run-Time reasoning (RTEC) is an open-source, Prolog implementation of the Event Calculus [3], optimised for computing continuous queries on data streams [2, 5, 6, 8]. 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 [4]. 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 applications in which RTEC has been used:

- Activity recognition[1].
- City transport & traffic management [2].
- Maritime situational awareness[2].
- Fleet management[3].
- Multi-agent system protocols [6].
- Biological feedback loops [7].

The novelty of RTEC lies in the following 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 that are irrelevant to the queries we want to compute.
5. Efficient treatment of temporal specifications with *cyclic dependencies*.
6. Combination of Event Calculus-style reasoning with *Allen's interval algebra*.
7. Reasoning over streams of *events with delayed effects*.

1.1 Getting RTEC

RTEC is written in Prolog and may run in any operating system. RTEC has been tested under YAP[4] and SWI[5] Prolog in Ubuntu Linux. To download RTEC, open a terminal and type:

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

Below, we demonstrate the use of RTEC on a toy example. For detailed execution

[1] https://cer.iit.demokritos.gr/blog/applications/activity_recognition
[2] https://cer.iit.demokritos.gr/blog/applications/maritime_surveillance
[3] https://cer.iit.demokritos.gr/blog/applications/fleet_management
[4] https://en.wikipedia.org/wiki/YAP_(Prolog)
[5] https://www.swi-prolog.org/

instructions and complex use cases, please consult the documentation in the github repository[6].

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 a pub, but loses his wallet. Ultimately, he goes home at night. We want to know when Chris was happy as these events unfolded. Our story has three events, `go_to`, `lose_wallet` and `win_lottery`, and three properties, `happy`, `location` and `rich`. We call such properties “fluents”. RTEC computes the effects of events on fluents.

We assume that a person is happy when he is rich or at a pub. Moreover, winning the lottery makes you rich and losing your wallet results in losing your money. These conditions can be expressed as rules in the language of RTEC. Create a file named `rules.prolog` in directory `RTEC/examples/toy/resources/patterns` and paste the following code:

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

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

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

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

Listing 1 Event description in RTEC.

Following Prolog’s convention, variables start with an upper-case letter, while predicates and constants start with a lower-case letter. The rules with head `initiatedAt`, `terminatedAt` or `holdsFor` define dependencies among events and fluents. For example, the first `initiatedAt` rule states that, for some person `X`, `rich(X)=true` is initiated when that person wins the lottery. Moreover, the `holdsFor` rule expresses that someone is happy when he is rich or at a pub using the `union_all` predicate, which will be discussed shorty.

In order to define the domains of the variables in the arguments of events and fluents, append in `rules.prolog` the following code:

[6] `https://github.com/aartikis/RTEC/blob/master/docs/contents.md`

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

Listing 2 Domains of event and fluent arguments.

The rules with head `grounding` specify the domains of the arguments of events and fluents. The third grounding rule, e.g., states that the first argument of a `go_to` event is a person and its second argument is a location. We specify the elements of each domain with the code below. Create a file named `toy_var_domain.prolog` in `RTEC/examples/toy/dataset/auxiliary` and add the following code:

```prolog
% 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 need an event narrative to test our event description. Create a file named `toy_data.csv` in `RTEC/examples/toy/dataset/csv` and add the following:

```
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. The event narrative is in a csv format. Details about this format will be presented in Section 4.

Now we have all the necessary components for computing the maximal intervals of the fluents in our event description. We need to instruct RTEC to use the aforementioned files. This may be done by editing the file `RTEC/execution scripts/defaults.toml`, i.e., the configuration file of RTEC[7], as follows:

[7] The documentation of TOML can be found in: `https://toml.io/en/`

```
# Parameter values for the toy application
[toy]
event_description = '../examples/toy/resources/patterns/rules.prolog'
input_mode = "csv"
input_providers = ['../examples/toy/dataset/csv/toy_data.csv']
results_directory = '../examples/toy/results'
window_size = 30
step = 30
start_time = 0
end_time = 30
clock_tick = 1
background_knowledge = ['../examples/toy/dataset/auxiliary/toy_var_domain.prolog']
stream_rate = 1
dependency_graph_flag = false
dependency_graph_directory = '../examples/toy/resources/graphs'
include_input = false
```

Listing 5 Configuration file of RTEC.

The `event_description` parameter points to the file containing the rules of our event description (Listings 1 and 2). The `background_knowledge` and `input_providers` parameters point to the files containing the domain entities and the event narrative of our example (Listings 3 and 4, respectively). In Section 6, we present the full documentation of the execution parameters of RTEC.

In order to execute RTEC, open a terminal and run the following commands:

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

Listing 6 Run RTEC on the toy example.

The `app` flag instructs RTEC to run the `toy` application. In other words, RTEC will use the input files and the execution parameters provided in the table of the configuration file named `toy` (see Listing 5). The `interactive` flag allows the user to pose queries after the execution of RTEC about the fluents of our event description[8]. For instance, if we want to see when Chris is happy, we may use the following query:

```
?- holdsFor(happy(chris)=true,I).
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, type the following query:

```
?- holdsFor(F,I).
F = (rich(chris)=true),
I = [(14,20)] ? ;
```

[8] RTEC answers queries concerning the last window; details about windowing are presented in Section 3.2.

```
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, to see the values of all fluents at time-point 16, we may use the following query:

```
?- holdsAt(F,16).
F = (rich(chris)=true) ? ;
F = (location(chris)=work) ? ;
F = (happy(chris)=true)
```

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

In what follows, we describe the language and the main modules of RTEC.

2 Writing an Event Description

The time model of 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)` is true iff `T` belongs to one of the maximal intervals of `I` for which `holdsFor(F=V,I)`.

An *event description* in RTEC includes rules that specify event occurrences with the use of the `happensAt` predicate, the effects of events with the use of the `initiatedAt` and `terminatedAt` predicates, and the values of fluents with the use of the `holdsAt` and `holdsFor` predicates. Table 1 summarises the RTEC predicates available to the event description developer.

An event may be an *input entity*, i.e., its instances are reported as items of the input stream, or an *output entity*, i.e., its instances are derived based on domain-specific rules. There are two types of fluents in RTEC: *simple* and *statically determined*. Simple fluents are output entities whereas statically determined fluents may be input or output entities. Below, we present how events and fluents may be specified. For a more detailed specification of an RTEC event description, see the Appendix.

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`. Events are typically items of the input data, such as the aforementioned events, expressed by `happensAt` facts. It is also possible to define "output" events by means of `happensAt` rules of the following

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 during which `F=V` holds continuously.
`initiatedAt(F=V,T)`	At time-point `T`, a period of time during which `F=V` holds continuously is initiated.
`terminatedAt(F=V,T)`	At time-point `T`, a period of time during which `F=V` holds continuously 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`.

Table 1 Main predicates of RTEC.

form:

$$\begin{aligned}&\texttt{happensAt}(\texttt{E},\texttt{T}) \leftarrow\\&\quad \texttt{happensAt}(\texttt{E}_1,\texttt{T})[[,\\&\quad [\texttt{not}]\ \texttt{happensAt}(\texttt{E}_2,\texttt{T}),\ \ldots,[\texttt{not}]\ \texttt{happensAt}(\texttt{E}_i,\texttt{T}),\\&\quad [\texttt{not}]\ \texttt{holdsAt}(\texttt{F}_1=\texttt{V}_1,\texttt{T}),\ \ldots,[\texttt{not}]\ \texttt{holdsAt}(\texttt{F}_k=\texttt{V}_k,\texttt{T})]].\end{aligned} \tag{1}$$

The first body literal of a `happensAt` rule is a positive `happensAt` predicate; this is followed by a possibly empty set of positive/negative `happensAt` and `holdsAt` predicates denoted by "[[ ]]". `not` expresses negation-by-failure, while "[`not`]" denotes that `not` is optional. All (head and body) predicates are evaluated on the same time-point.

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

2.2 Fluents

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 *maximal intervals* `I` for which `F=V` holds continuously, i.e., `holdsFor(F=V,I)`, RTEC calculates the time-points at which `F=V` is initiated and the time-points at which `F=V` is terminated. The initiation points of `F=V` are computed by means of domain-specific `initiatedAt` rules that have the following syntax:

$$\begin{aligned}&\mathtt{initiatedAt}(\mathtt{F}=\mathtt{V},\mathtt{T}) \leftarrow\\ &\quad \mathtt{happensAt}(\mathtt{E}_1,\mathtt{T})[[,\\ &\quad [\mathtt{not}]\ \mathtt{happensAt}(\mathtt{E}_2,\mathtt{T}),\ \ldots,[\mathtt{not}]\ \mathtt{happensAt}(\mathtt{E}_i,\mathtt{T}),\\ &\quad [\mathtt{not}]\ \mathtt{holdsAt}(\mathtt{F}_1=\mathtt{V}_1,\mathtt{T}),\ \ldots,[\mathtt{not}]\ \mathtt{holdsAt}(\mathtt{F}_k=\mathtt{V}_k,\mathtt{T})]].\end{aligned} \tag{2}$$

Similarly to `happensAt` definitions (see rule (1)), the first body literal of an `initiatedAt` rule is a positive `happensAt` predicate, which is followed by a possibly empty set of positive/negative `happensAt` and `holdsAt` predicates. The syntax of `terminatedAt` rules is the same.

In our example, `rich` is a simple fluent. The maximal intervals during which `rich(Person)=true` holds continuously are computed using a domain-independent implementation of `holdsFor` that matches each initiation point of `rich(Person)=true` with the first subsequent termination point, while ignoring all intermediate initiation points.

Statically Determined Fluents. For a statically determined fluent `F`, the maximal intervals of `F=V` are computed via domain-specific `holdsFor` rules that have the following syntax:

$$\begin{aligned}&\mathtt{holdsFor}(\mathtt{F}=\mathtt{V},\mathtt{I}_{n+m}) \leftarrow\\ &\quad \mathtt{holdsFor}(\mathtt{F}_1=\mathtt{V}_1,\mathtt{I}_1)[[,\\ &\quad \mathtt{holdsFor}(\mathtt{F}_2=\mathtt{V}_2,\mathtt{I}_2),\ \ldots\mathtt{holdsFor}(\mathtt{F}_n=\mathtt{V}_n,\mathtt{I}_n),\\ &\quad \mathtt{intervalConstruct}(\mathtt{L}_1,\mathtt{I}_{n+1}),\ldots,\ \mathtt{intervalConstruct}(\mathtt{L}_m,\mathtt{I}_{n+m})]].\end{aligned} \tag{3}$$

The first body literal of a `holdsFor` rule defining `F=V` is a `holdsFor` predicate expressing the maximal intervals of a FVP other than `F=V`. This is followed by a possibly empty list of `holdsFor` predicates for other FVPs, and interval manipulation constructs, expressed by `intervalConstruct`. $\mathtt{intervalConstruct}(\mathtt{L}_j,\mathtt{I}_{n+j})$ may be $\mathtt{union\_all}(\mathtt{L}_j,\mathtt{I}_{n+j})$, $\mathtt{intersect\_all}(\mathtt{L}_j,\mathtt{I}_{n+j})$ or $\mathtt{relative\_complement\_all}(\mathtt{I}_k,\mathtt{L}_j,\mathtt{I}_{n+j})$ (see the last three items of Table 1).

`union_all(+L,-I)` computes a list of maximal intervals `I` as the union of all maximal intervals in 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 right-open interval `[Ts,Te)`. `I` in `union_all(L,I)` is a list of maximal intervals that includes the time-points that are part of at least one list in `L`.

`intersect_all(+L,-I)` computes a list of maximal intervals `I` as the intersection all maximal intervals in 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 the time-points that are part of every list in `L`.

`relative_complement_all(+I',+L,-I)` computes a list of maximal intervals `I` as the relative complement of the list of maximal intervals `I'` with respect to the maximal intervals in the lists of list `L`. As an example:

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

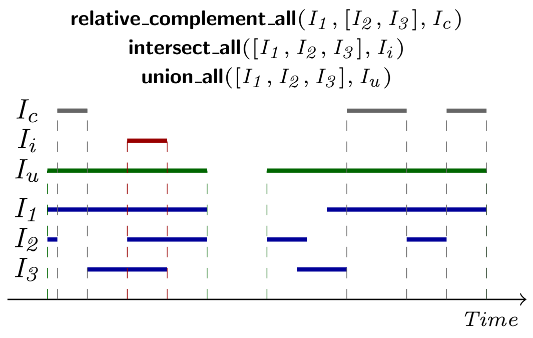

relative_complement_all(I_1, [I_2, I_3], I_c)
intersect_all([I_1, I_2, I_3], I_i)
union_all([I_1, I_2, I_3], I_u)
I_c
I_i
I_u
I_1
I_2
I_3
Time

Figure 1 Interval manipulation constructs of RTEC. I_1, I_2 and I_3 (resp. I_c, I_i and I_u) are input (output) lists of maximal intervals.

`I` in `relative_complement_all(I',L,I)` is a list of maximal intervals that includes all time-points in `I'` that are not part of any list in `L`

Figure 1 presents an example of computing `union_all`, `intersect_all` and `relative_complement_all` over the lists of maximal intervals $\mathtt{I}_1$, $\mathtt{I}_2$ and $\mathtt{I}_3$.

In our example, `happy` is a statically determined fluent defined by means of `union_all`. 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` iff 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` definition. Compare, e.g., the statically determined fluent representation of `happy` in Listing 1 with the simple fluent representation presented 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).
```

The interval manipulation constructs of RTEC can also lead to much more efficient computation—see the complexity analysis in [2].

2.3 Further Expressivity

Events with Delayed Effects. A simple fluent may be initiated at a specified time after the occurrence of a set of events. We use the predicate `fi` to specify the **f**uture **i**nitiations of a fluent-value pair (FVP). Consider, e.g., a multi-agent protocol for exchanging digital goods, and the fluent `quote(M,C,G)`, expressing that a merchant `M` has presented an offer to consumer `C` about goods `G`, and that the offer has not been accepted by `C` yet. Moreover, suppose that

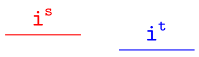
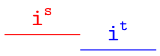
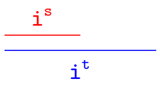
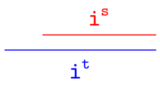
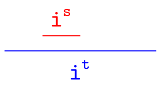
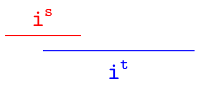
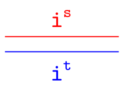
Relation	Definition	Illustration
`before`$(\mathtt{i}^s, \mathtt{i}^t)$	$\mathtt{f}(\mathtt{i}^s) < \mathtt{s}(\mathtt{i}^t)$	
`meets`$(\mathtt{i}^s, \mathtt{i}^t)$	$\mathtt{f}(\mathtt{i}^s) = \mathtt{s}(\mathtt{i}^t)$	
`starts`$(\mathtt{i}^s, \mathtt{i}^t)$	$\mathtt{s}(\mathtt{i}^s) = \mathtt{s}(\mathtt{i}^t)$, $\mathtt{f}(\mathtt{i}^s) < \mathtt{f}(\mathtt{i}^t)$	
`finishes`$(\mathtt{i}^s, \mathtt{i}^t)$	$\mathtt{s}(\mathtt{i}^s) > \mathtt{s}(\mathtt{i}^t)$, $\mathtt{f}(\mathtt{i}^s) = \mathtt{f}(\mathtt{i}^t)$	
`during`$(\mathtt{i}^s, \mathtt{i}^t)$	$\mathtt{s}(\mathtt{i}^s) > \mathtt{s}(\mathtt{i}^t)$, $\mathtt{f}(\mathtt{i}^s) < \mathtt{f}(\mathtt{i}^t)$	
`overlaps`$(\mathtt{i}^s, \mathtt{i}^t)$	$\mathtt{s}(\mathtt{i}^s) < \mathtt{s}(\mathtt{i}^t)$, $\mathtt{f}(\mathtt{i}^s) > \mathtt{s}(\mathtt{i}^t)$, $\mathtt{f}(\mathtt{i}^s) < \mathtt{f}(\mathtt{i}^t)$	
`equal`$(\mathtt{i}^s, \mathtt{i}^t)$	$\mathtt{s}(\mathtt{i}^s) = \mathtt{s}(\mathtt{i}^t)$, $\mathtt{f}(\mathtt{i}^s) = \mathtt{f}(\mathtt{i}^t)$	

Table 2 Seven relations of Allen's interval algebra.

a quote may be active for at most `R` time-points. An event `present_quote(M,C,G,P)`, stating that `M` presents an offer to `C` about `G` with price `P`, has the immediate effect of initiating `quote(M,C,G)=true`, as well as the delayed effect of initiating `quote(M,C,G)=false`, `R` time-points after the initiation of `quote(M,C,G)=true`. We may express these delayed initiations of `quote(M,C,G)=false` with the following fact:

$$\mathtt{fi}(\mathtt{quote}(\mathtt{M},\mathtt{C},\mathtt{G}) = \mathtt{true}, \mathtt{quote}(\mathtt{M},\mathtt{C},\mathtt{G}) = \mathtt{false}, \mathtt{R}). \tag{4}$$

In some cases, it may be required to express that the future initiation of an FVP is postponed, based on the events that occurred since the time of staging the future initiation. We use the predicate `p` to denote which future initiations may be postponed. For example, we may use `fi` fact (4) in conjunction with $\mathtt{p}(\mathtt{quote}(\mathtt{M},\mathtt{C},\mathtt{G}) = \mathtt{true})$, in order to express that a re-initiation of `quote(M,C,G)=true` will postpone the future initiation of `quote(M,C,G)=false`. This way, a merchant `M` may present another quote to consumer `C`, e.g., for a different price, with the aim of postponing the expiration of `quote(M,C,G)=true`.

Allen's interval algebra. RTEC supports the relations of Allen's interval algebra in statically determined fluent definitions [5]. The code of RTEC with Allen relations is publicly available[9]. Allen's interval algebra specifies thirteen jointly exhaustive and pairwise disjoint relations among intervals [1]. Table 2 presents relations `before`, `meets`, `starts`, `finishes`, `during`, `overlaps` and `equal`. The remaining six relations are the 'inverse' relations; `equal` does not have an inverse relation. The second column of Table 2 shows the conditions that must be satisfied in order to compute an Allen relation. $\mathtt{i}^s$ and $\mathtt{i}^t$ express intervals, while `s(i)` and `f(i)` denote the start and end endpoint of interval `i`, respectively.

[9] `https://github.com/aartikis/RTEC/tree/allen`

`outMode`	**Output list** `I`
`source`	$\mathtt{I} = \mathtt{S_{rel}}$
`target`	$\mathtt{I} = \mathtt{T_{rel}}$
`union`	$\mathtt{union\_all}([\mathtt{S_{rel}}, \mathtt{T_{rel}}], \mathtt{I})$
`intersect`	$\mathtt{intersect\_all}([\mathtt{S_{rel}}, \mathtt{T_{rel}}], \mathtt{I})$
`complement`	$\mathtt{relative\_complement\_all}(\mathtt{S_{rel}}, [\mathtt{T_{rel}}], \mathtt{I})$
`complement_inv`	$\mathtt{relative\_complement\_all}(\mathtt{T_{rel}}, [\mathtt{S_{rel}}], \mathtt{I})$

Table 3 Output modes of the `allen` construct.

Allen's relations have proven necessary for complex event recognition [5]. However, the interval manipulation constructs of RTEC cannot express the relations of Allen's algebra. Consider, e.g., the computation of the interval pairs $(\mathtt{i^s}, \mathtt{i^t})$, where $\mathtt{i^s} \in \mathtt{S}$ and $\mathtt{i^t} \in \mathtt{T}$, satisfying `before`. $\mathtt{intersect\_all}([\mathtt{S}, \mathtt{T}], [\,])$ states that for every interval pair $(\mathtt{i^s}, \mathtt{i^t})$, such that $\mathtt{i^s} \in \mathtt{S}$, $\mathtt{i^t} \in \mathtt{T}$, it holds that $\mathtt{i^s} \cap \mathtt{i^t} = \emptyset$. Therefore, $\mathtt{i^s}$ is `before` $\mathtt{i^t}$, or vice versa. It is impossible, however, to distinguish between the two cases.

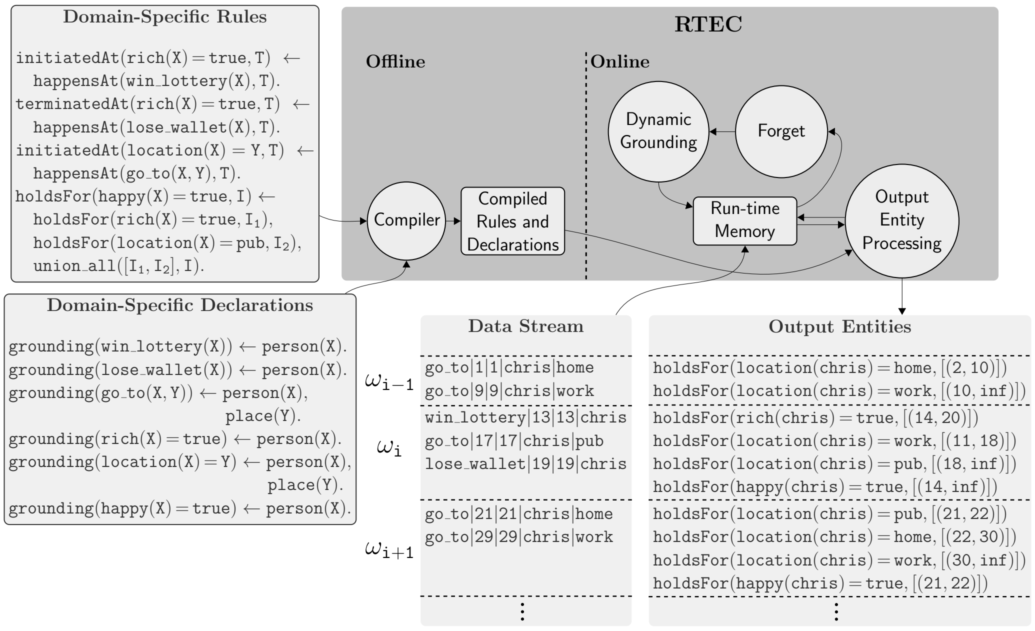
In order to express Allen relations, a `holdsFor(F=V,I)` rule defining a statically determined fluent `F` may additionally contain body predicates in the form of `allen(rel,S,T,outMode,I)`, where `rel` denotes an Allen relation, `S` and `T` are input lists of maximal intervals, `outMode` expresses how we should treat the interval pairs $(\mathtt{i^s}, \mathtt{i^t})$ satisfying `rel`, where $\mathtt{i^s} \in \mathtt{S}$ and $\mathtt{i^t} \in \mathtt{T}$, and `I` is the output list of maximal intervals.

According to `allen(rel,S,T,outMode,I)`, `I` contains the maximal intervals produced by applying `outMode` to the interval pairs of the 'source list' `S` and the 'target list' `T` satisfying `rel`, i.e., one of the Allen relations presented in Table 2. The inverse relations may be computed by reversing the order of the input lists. `outMode` is applied to the intervals of lists $\mathtt{S_{rel}} = \{\mathtt{i^s} \mid \mathtt{i^s} \in \mathtt{S} \wedge \exists \mathtt{i^t} \in \mathtt{T} : \mathtt{rel}(\mathtt{i^s}, \mathtt{i^t})\}$ and $\mathtt{T_{rel}} = \{\mathtt{i^t} \mid \mathtt{i^t} \in \mathtt{T} \wedge \exists \mathtt{i^s} \in \mathtt{S} : \mathtt{rel}(\mathtt{i^s}, \mathtt{i^t})\}$, i.e., the intervals of the source and the target lists appearing in at least one pair of intervals satisfying `rel`. The possible values of `outMode` and their meaning are presented in Table 3.

As an example, consider the maritime domain where vessels often attempt to conceal illegal activities in certain areas, such as fishing in fisheries restricted areas, by stopping transmitting their position. In order to monitor such incidents, we may use a statically determined Boolean fluent `disappearedInArea(Vl,AreaType)`, which is defined with the following rule:

$$\begin{aligned}
&\mathtt{holdsFor}(\mathtt{disappearedInArea}(\mathtt{Vl}, \mathtt{AreaType}) = \mathtt{true}, \mathtt{I_{dia}}) \leftarrow \\
&\quad \mathtt{holdsFor}(\mathtt{withinArea}(\mathtt{Vl}, \mathtt{AreaType}) = \mathtt{true}, \mathtt{S}), \\
&\quad \mathtt{holdsFor}(\mathtt{gap}(\mathtt{Vl}) = \mathtt{farFromPorts}, \mathtt{T}), \\
&\quad \mathtt{allen}(\mathtt{meets}, \mathtt{S}, \mathtt{T}, \mathtt{target}, \mathtt{I_{dia}}).
\end{aligned} \tag{5}$$

`withinArea(Vl, AreaType)=true` denotes that vessel `Vl` is in an area of type `AreaType`, while `gap(Vl)=farFromPorts` expresses that `Vl` is not transmitting its position while being in the open sea. The last condition of rule (5) expresses the `meets` Allen relation. $\mathtt{allen}(\mathtt{meets},\mathtt{S},\mathtt{T},\mathtt{target},\mathtt{I_{dia}})$ states that from the interval pairs $(\mathtt{i^s}, \mathtt{i^t})$ satisfying `meets`, where $\mathtt{i^s} \in \mathtt{S}$ and $\mathtt{i^t} \in \mathtt{T}$, we will keep in the output list $\mathtt{I_{dia}}$ the target intervals $\mathtt{i^t}$ (see the second line of Table 3). According to rule (5), therefore, a vessel `Vl` is said to disappear in an area of type `AreaType` during an interval $\mathtt{i^{dia}}$, if $\mathtt{i^{dia}}$ is an interval during which `gap(Vl)=farFromPorts`, i.e., `Vl` stopped transmitting its position while being in the open sea, and $\mathtt{i^{dia}}$ is met by an interval $\mathtt{i^s}$ during which `Vl` was within an area of type `AreaType`.



Domain-Specific Rules

```
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).
```

Domain-Specific Declarations

```
grounding(win_lottery(X)) ← person(X).
grounding(lose_wallet(X)) ← person(X).
grounding(go_to(X, Y)) ← person(X),
                         place(Y).
grounding(rich(X) = true) ← person(X).
grounding(location(X) = Y) ← person(X),
                             place(Y).
grounding(happy(X) = true) ← person(X).
```

Data Stream

```
go_to|1|1|chris|home
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
go_to|29|29|chris|work
-----------------------
⋮
```

Output Entities

```
holdsFor(location(chris) = home, [(2, 10)])
holdsFor(location(chris) = work, [(10, inf)])
-----------------------
holdsFor(rich(chris) = true, [(14, 20)])
holdsFor(location(chris) = work, [(11, 18)])
holdsFor(location(chris) = pub, [(18, inf)])
holdsFor(happy(chris) = true, [(14, inf)])
-----------------------
holdsFor(location(chris) = pub, [(21, 22)])
holdsFor(location(chris) = home, [(22, 30)])
holdsFor(location(chris) = work, [(30, inf)])
holdsFor(happy(chris) = true, [(21, 22)])
-----------------------
⋮
```

Figure 2 An illustration of the architecture of RTEC.

3 RTEC Architecture

Figure 2 illustrates the architecture of RTEC. In this section, we examine the key features of this architecture.

3.1 Rule Compilation

Before online execution, RTEC compiles the rules and declarations of an event description into a format that allows for efficient reasoning. For instance, the compiler introduces a simple indexing for efficient predicate retrieval. This is an offline process that is transparent to the user. Additionally, the compiler detects the dependencies among events and fluents of the event description and constructs its *dependency graph*. A node of this graph corresponds to an event or fluent, while its edges denote dependencies based on the rules. Figure 3, e.g., presents the dependency graph of our toy example (see Section 1.2). **`win_lottery`**, **`lose_wallet`** and **`go_to`** are input entities, and thus their nodes have no incoming edges. The node of fluent **`happy`** has two incoming edges, originating from the nodes of **`rich`** and **`location`**, because of the dependencies introduced by the **`holdsFor`** rule defining **`happy`** (see Listing 1). During online execution, RTEC processes the events and fluents of the dependency graph in a bottom-up fashion, caching all intermediate results, and thus avoids redundent computations.

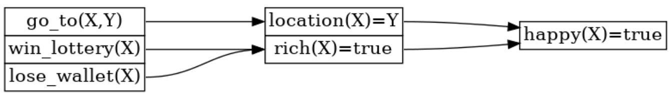


Figure 3 Dependency graph of the `toy` event description of Listing 1.

3.2 Windowing

Reasoning has to be efficient enough to support real-time decision-making, and scale to very large numbers of input and output entities, i.e., fluents and events. 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.

To address these issues, RTEC processes input data streams in windows and computes output entities, i.e., the maximal intervals of output fluents and the time-points of output events, at specified query times q_1, q_2, and so on. This is to make the cost of reasoning dependent only on the window size and not on the complete history of input and output entities. RTEC maintains a run-time memory that contains only the input and output entity instances that may be required for reasoning at the current query time (see Run-time Memory in Figure 2). At each q_i, e.g., only the input entities that fall within a specified interval—the window ω_i—are maintained in the run-time memory of RTEC. All other input entities are "forgotten" (see Forget in Figure 2). As an example, given the window $\omega_{\mathtt{i}}{=}(10, 20]$ (see $\omega_{\mathtt{i}}$ in the Data Stream of Figure 2), RTEC computes the corresponding output entities displayed in the Output Entity Stream of Figure 2.

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[10]. For more details regarding the windowing algorithm of RTEC, see [2].

3.3 Dynamic Grounding

Dynamic grounding optimises output entity derivation by discarding domain elements that do not take part in any output entity of the current window (see Dynamic Grounding in Figure 2). For example, the user may add the fact `dynamicEntity(person(_))` in Listing 2, in order to indicate that the persons participating in the output entities of our toy example are not known beforehand. Then, RTEC will derive these persons by observing incoming events. For instance, based on the input item `win_lottery|13|13|chris` in window $\omega_{\mathtt{i}}$ of Figure 2, RTEC derives that Chris is a person that may appear in an output entity. As a result, the run-time memory of RTEC contains only domain elements that take part in output events and fluents of the current window.

3.4 Output Entity Processing

RTEC computes the maximal intervals of the output entities of the current window (see Output Entity Processing in Figure 2). We employ a different module for processing each type of output entity, i.e., output statically determined fluents, simple fluents and output events.

Statically Determined Fluent Processing. For a fluent-value pair `F=V`, where `F` is a statically determined fluent, RTEC follows the steps below:

[10] See parameters `window_size` and `step` in the configuration file of Listing 5.

1. Retrieve the maximal interval of `F=V`, if any, that was derived at the previous query time and overlaps the start of the current window.
2. Compute the maximal intervals of `F=V` in the current window by evaluating its `holdsFor` definition (see rule schema (3)).
3. Amalgamate the interval derived at step 1, if any, with the intervals derived in step 2.

Simple Fluent Processing. For a fluent-value pair `F=V`, where `F` is a simple fluent, RTEC follows the steps below:

1. Retrieve the maximal interval of `F=V`, if any, that was derived at the previous query time and overlaps the start of the current window, and store its starting point.
2. Compute the remaining starting points of `F=V` using `initiatedAt` rules and, if there is at least one starting point of `F=V`, compute its ending points using `terminatedAt` rules. `initiatedAt` and `terminatedAt` rules follow schema (2).
3. Construct the maximal intervals of `F=V` by matching each one of its starting point with the first subsequent ending point, while ignoring all intermediate starting points.

Event Processing. RTEC computes output event instances by evaluating `happensAt` rules (see rule schema (1)).

4 Format of Input Data

RTEC processes narratives/streams containing information about the input events and fluents that take place as time progresses. A record of an RTEC-compatible narrative/stream may express one of the following:

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 maximal 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. 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 at time-point*:

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

3. *input fluent in interval*:

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

The arrival time expresses the time at which the entity arrived at RTEC, while the occurrence time expresses the time at which the entity was detected by the sensor. The arrival time does not appear in the arguments of Event Calculus predicates, but is 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.

5 Reasoning over Historical Data vs a Live Stream

RTEC may consume input entities from:

- files,
- named pipes, or
- a Unix domain socket

In all cases, the format of the input entities is that presented in the previous section. Files contain historical data concerning streams that have concluded. In contrast, named pipes and sockets are being updated in real time with new records. We use the execution parameter `input_mode` of the configuration file (see Listing 5) to inform RTEC about the type of input providers we use (see Section 6 for the full documentation of the execution parameters of RTEC). When `input_mode` is `csv`, RTEC consumes the input entities from files, whereas, when `input_mode` is `fifo` or `socket`, RTEC consumes the input entities incrementally as their records are written in named pipes or in a Unix domain socket, respectively. More precisely, RTEC supports the following options:

1. `input_mode=csv` and the parameter `input_providers` is a list of 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 files.
4. `input_mode=socket` and `input_providers` is a list containing a Unix domain socket.
5. `input_mode=socket` and `input_providers` is a list of files.

Cases 3 and 5 may be selected in order to *simulate* a live stream using historical data. In case 3, the script `run_rtec.sh` opens a new named pipe for each input file and creates a new process that writes the records of each file into the corresponding named pipe incrementally. Between writing consecutive input entities into the named pipe, `run_rtec.sh` waits for a number of seconds that is equal to the difference between the time-stamps of these entities. Subsequently, RTEC is executed in `fifo` mode, having as input the aforementioned named pipes. In case 5, for each input file, `run_rtec.sh` starts a new process that waits for a server socket to be constructed. Next, RTEC is executed in `socket` mode, and it creates a Unix domain socket which listens for client connection requests. Afterwards, each process generated by `run_rtec.sh` connects to the socket, and, similarly to case 3, writes the records of the corresponding file into the socket incrementally, while waiting for an amount of time dictated by the time-stamps of consecutive records.

The `stream_rate` parameter allows RTEC to replay the input stream in various velocities. For instance, when `stream_rate=2`, RTEC processes input entities that arrive at double the velocity implied by their time-stamps.

In `fifo` and `socket` mode, RTEC sleeps between consecutive query times for a number of seconds that is equal to the value of the `window_size` parameter. When processing the input data stream of Figure 2, e.g., where `window_size=10`, RTEC sleeps between query times q_{i-1} and q_i for 10 seconds, after which, provided that there are no transmission delays, all input entities that fall within ω_i have arrived at RTEC.

6 Execution parameters

The user may set custom execution parameters for RTEC by editing its configuration file (see Listing 5). These execution parameters are summarised below:

- `window_size`: RTEC processes streams using a sliding window (see Section 3.2). The window size is constant during execution and is equal to the value of this parameter.
- `step`: The step is the distance between two consecutive query times (see Section 3.2). The step is constant during execution and is equal to the value of this parameter.
- `start_time`: RTEC does not consider input entities that take place before `start_time`.
- `end_time`: RTEC does not consider input entities that take place after `end_time`.
- `clock_tick`: The temporal distance between two consecutive time-points.
- `input_mode`: RTEC may process streams of input events from files, named pipes or Unix domain sockets (see Section 5). The possible values of `input_mode` are `csv`, `fifo` and `socket`.
- `input_providers`: The files or named pipes providing input entities. See, e.g., the records in Listing 4 of our toy example.
- `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 5 for more information.
- `event_description`: The file containing the event description of the application. See, e.g., Listings 1 and 2 of our toy example.
- `background_knowledge`: A list of files containing Prolog rules and facts. RTEC may utilise this knowledge during execution. For example, the file corresponding to Listing 3, which specifies the domains of entities in our toy example, may be provided as background knowledge (see the configuration file in Listing 5).
- `output_mode`: We may write the fluent-value pair intervals computed by RTEC in a regular file or into a named pipe. The possible values of `output_mode` are `file` and `fifo`.
- `results_directory`: The location where the fluent-value pair intervals computed by RTEC and its execution logs will be stored.
- `dependency_graph_flag`: If the value of this parameter is `true`, then RTEC will produce the dependency graph of the provided event description as a png file (see Section 3.1). This functionality requires GraphViz[11].

[11] `https://graphviz.org/`

- `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 (see Figure 3). Otherwise, the graph only includes output entities.

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 J. Allen. Towards a general theory of action and time. *Artif. Intell.*, 23(2):123–154, 1984.

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

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

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

5 Periklis Mantenoglou, Dimitrios Kelesis, and Alexander Artikis. Complex event recognition with allen relations. In *KR*, pages 502–511, 2023.

6 Periklis Mantenoglou, Manolis Pitsikalis, and Alexander Artikis. Stream reasoning with cycles. In *KR*, pages 544–553, 2022.

7 Ashwin Srinivasan, Michael Bain, and A. Baskar. Learning explanations for biological feedback with delays using an event calculus. *Mach. Learn.*, 111(7):2435–2487, 2022.

8 Efthimis Tsilionis, Alexander Artikis, and Georgios Paliouras. Incremental event calculus for run-time reasoning. *J. Artif. Intell. Res.*, 73:967–1023, 2022.

A Appendix

A.1 Grammar

The extended affix grammar (EAG) of an RTEC event description is specified as follows:

▶ **Grammar 1.**

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

⟨*domain-rule*⟩ ::= ⟨*simple-fluent-rule*⟩
  | ⟨*output-sdf-rule*⟩
  | ⟨*output-event-rule*⟩
  | ⟨*fi-fact*⟩
  | ⟨*p-fact*⟩
  | ⟨*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-atom*⟩ ‘`(`’ ⟨*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*⟩ ‘)’
| ‘allen(’⟨*allen-predicate*⟩ ‘,’ ⟨*mil-var-1*⟩ ‘,’ ⟨*mil-var-2*⟩ ‘,’ ⟨*output-mode*⟩ ‘,’ ⟨*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*⟩ ‘)’
| ‘allen(’⟨*allen-predicate*⟩ ‘,’ ⟨*mil-var-1*⟩ ‘,’ ⟨*mil-var-2*⟩ ‘,’ ⟨*output-mode*⟩ ‘,’ ⟨*mil-var*⟩ ‘)’

⟨*allen-predicate*⟩ ::= ‘before’
| ‘meets’
| ‘starts’
| ‘finishes’
| ‘during’
| ‘overlaps’
| ‘equal’

⟨*output-mode*⟩ ::= ‘source’
| ‘target’
| ‘union’
| ‘intersect’
| ‘complement’
| ‘complement_inv’

⟨*fi-fact*⟩ ::= ‘fi(’ ⟨*fluent*⟩ ‘=’⟨*value*⟩ ‘,’ ⟨*fluent*⟩ ‘=’⟨*other-value*⟩ ‘,’ ⟨*time-const+*⟩ ‘).’

⟨*p-fact*⟩ ::= ‘p(’ ⟨*fluent*⟩ ‘=’⟨*value*⟩ ‘).’

⟨*grounding-rule*⟩ ::= ‘grounding(’⟨*event*⟩ ‘):-’ ⟨*domain-object-property-list*⟩ ‘.’

⟨*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*⟩]

In an EAG, terminal symbols may be accompanied by a list of parameters (affixes). Like rule variables with the same name in Prolog, these parameters are substituted with the same non-terminal symbol for all of their instances in the same rule. According to Grammar 1, 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 RTEC to use a specific argument of an event or fluent as its index when searching for instances in the run-time memory. Dynamic domain declarations inform RTEC that objects of certain domains, e.g., the ids of persons and objects appearing in a video feed, must be 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 fluents, which is used as a parameter in the non-terminal symbols concerning the body predicates of a simple fluent definition. <time-const+> signifies a positive constant from the domain of time-points, i.e., a positive integer. 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 corresponding grammar rule).

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 (see Section A.2). For example, <fluent> is substituted with a fluent type of the event description and <mil-var> is substituted with a variable corresponding to a list of maximal intervals.

A.2 Sorts

The specifications of an RTEC event description provided in Section 2 assume that the arguments of RTEC predicates, such as `happensAt` and `initiatedAt`, take values from the appropriate sorts of its language. For instance, the first argument of a `happensAt` predicate spans over event types, while its second argument spans over time-points. We suppement the specifications provided in Section 2 with the sorts of the language of RTEC. Let $\mathcal{L}_{\mathsf{RTEC}}$ be the many-sorted domain specification language of RTEC. 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. $l = [(t_1^s, t_1^f), \dots, (t_n^s, t_n^f)] \in \mathbb{L}_D$ 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 $e(arg_1, arg_2, \dots)$, 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 $f(arg_1, arg_2, \dots)$, 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 $f \in \mathbb{F}^S_D$, $\mathbb{V}^f_D$ is the set of all possible values for fluent type f. Often $\mathbb{V}^f_D = \{\texttt{true, false}\}$. For example, $\mathbb{V}^{rich}_{toy} = \{\texttt{true, false}\}$ and $\mathbb{V}^{location}_{toy} = \{\texttt{work, home, pub}\}$.

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, `holdsFor`/2, `fi`/3, and `p`/1.
- **Interval manipulation constructs.** Set $\mathbb{R}_I$ contains the interval operations `union_all`/2, `intersect_all`/2, `relative_complement_all`/3, and `allen`/5.