Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning
Applications and Systems

Stream Reasoning with Cycles

Periklis Mantenoglou[2,1] , **Manolis Pitsikalis**[3] , **Alexander Artikis**[4,1]
[1]NCSR Demokritos, Athens, Greece
[2]National and Kapodistrian University of Athens, Greece
[3]University of Liverpool, UK
[4]University of Piraeus, Greece
periklismant@di.uoa.gr, E.Pitsikalis@liverpool.ac.uk, a.artikis@iit.demokritos.gr

Abstract

Temporal specifications, such as those of multi-agent systems, often include cyclic dependencies. Moreover, there is an increasing need to evaluate such specifications in an online manner, upon streaming data. Consider, e.g., the online computation of the normative positions of the agents engaging in an e-commerce protocol. We present a formal computational framework that deals with cyclic dependencies in an efficient way. Moreover, we demonstrate the effectiveness of our framework on large synthetic and real data streams, from multi-agent systems and composite event recognition.

1 Introduction

The temporal specifications of many contemporary applications include cyclic dependencies. In multi-agent e-commerce protocols, e.g., a contract may be awarded to an agent that has not been suspended, while an agent may be suspended when not fulfilling the terms of a(nother) contract. As another example, consider the recognition of the different stages of a fishing trip, which is important for managing fishing activity and port traffic. For instance, a fishing vessel is said to be: (a) *approaching* a fishing area if it has ended another trip and goes in motion, i.e., 'under way'; (b) *trawling* when it stops the approach, i.e., it has reached the fishing area, and starts making consecutive turns; (c) *returning* when it stops trawling and goes under way; moreover, a vessel is said to have (d) *ended* its trip when it completes its return by becoming anchored or moored.

Contemporary applications also require the processing of large, evolving streams of data. Stream reasoning systems process such data streams by continuously applying temporal queries/patterns on incoming data and reporting instances of pattern satisfaction. Examples of stream reasoning systems may be found in various fields (Dell'Aglio et al. 2019). Consider, e.g., the recognition of (illegal) fishing on streams of vessel position signals. Furthermore, consider the online computation of the normative positions (Sergot 2001) of agents negotiating about a contract, given the messages exchanged between them. In all these cases, complex temporal specifications need to be evaluated with minimal latency in order to support real-time decision-making.

To deal efficiently with cyclic dependencies in temporal specifications, we present an extension of the 'Event Calculus for Run-Time reasoning' (RTEC), i.e., a logic programming implementation of the Event Calculus (Kowalski and Sergot 1986), designed to handle high-velocity data streams (Artikis, Sergot, and Paliouras 2015). The specifications in our proposed extension, RTEC$_\circ$, are locally stratified logic programs. Furthermore, RTEC$_\circ$ includes an algorithm for incremental caching, designed to avoid unnecessary re-computations when evaluating cyclic dependencies.

The contributions of the paper may be summarised as follows. First, we present RTEC$_\circ$, an open-source formal computational framework[1] for reasoning over real-world data streams and temporal specifications with cyclic dependencies. Second, we evaluate RTEC$_\circ$ by means of a complexity analysis and identify the benefits of incremental caching. Third, we present an extensive, *reproducible* empirical evaluation on large synthetic and real data streams. Moreover, we present an empirical comparison of RTEC$_\circ$ with a related computational framework and demonstrate the benefits of our approach.

We employ a voting procedure from multi-agent systems to illustrate RTEC$_\circ$. We follow the formalisation of Pitt et al. (2006), which may be summarised as follows: a committee sits and the chair opens the meeting; a member proposes a motion; another member seconds the motion; the members debate the motion; the chair calls for those in favour/against to cast their vote; finally, the motion is carried, or not, according to the standing rules of the committee.

2 Event Calculus for Run-Time Reasoning

The Event Calculus for Run-Time reasoning (RTEC) (Artikis, Sergot, and Paliouras 2015) is a logic programming implementation of the Event Calculus (Kowalski and Sergot 1986), designed for reasoning over data streams. We summarise the language of RTEC and its reasoning algorithms.

Representation. The time model is linear and includes integer time-points. Variables start with an upper-case letter, while predicates and constants start with a lower-case letter. A *fluent* is 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. The application-

[1]RTEC$_\circ$ is written in Prolog and the code is available at: https://github.com/aartikis/RTEC

specific part of a formalisation in RTEC is called *event description*.

Definition 1 (Event Description)**.** An *event description* is a set of:
1. Ground happensAt facts, expressing a stream of event instances. happensAt$(E,\ T)$ denotes that event E occurs at time-point T.
2. Rules with head initiatedAt or terminatedAt, expressing the effects of events on fluents. initiatedAt$(F=V,\ T',\ T,\ T'')$ (respectively terminatedAt$(F=V,\ T',\ T,\ T'')$) denotes that a time period during which a fluent F has value V is initiated (resp. terminated) at a time-point T, such that $T' \leq T < T''$. ■

Definition 2 (Syntax)**.** The rules defining initiatedAt predicates in RTEC have the following syntax:

$$\begin{aligned}&\text{initiatedAt}(F=V,\ T',\ T,\ T'') \leftarrow\\ &\quad \text{happensAt}(E_1,\ T),\ T' \leq T < T'',\\ &\quad [[\ [\text{not}]\ \text{happensAt}(E_2,\ T),\ \ldots,\\ &\quad\ [\text{not}]\ \text{happensAt}(E_i,\ T),\\ &\quad\ [\text{not}]\ \text{holdsAt}(F_1=V_1,\ T),\ \ldots,\\ &\quad\ [\text{not}]\ \text{holdsAt}(F_k=V_k,\ T).\ ]]\end{aligned} \tag{1}$$

The first body literal of an initiatedAt rule is a positive happensAt predicate; this is followed by a possibly empty set of positive/negative happensAt and holdsAt predicates denoted by '[[ ]]'. holdsAt$(F=V,\ T)$ expresses that fluent F has value V at a given time-point T. The evaluation of holdsAt predicates will be discussed shortly. 'not' expresses negation-by-failure (Clark 1978), while '[not]' denotes that 'not' is optional. All (head and body) predicates are evaluated on the same time-point T. T' and T'', which are added at compile-time in a process transparent to the user, specify the temporal range of T. T' and T'' are always ground in queries, allowing search optimisations through indexing. terminatedAt rules have the same form. ■

According to the common-sense law of inertia, $F=V$ holds at a particular time-point T, i.e., holdsAt$(F=V,\ T)$, if $F=V$ has been *initiated* by an event at some earlier time-point, and not *terminated* by another event in the meantime.

Example 1. Consider the following rule from voting:

$$\begin{aligned}&\text{initiatedAt}(\mathit{voted}(Ag, M)=\mathit{Vote},\ T',\ T,\ T'') \leftarrow\\ &\quad \text{happensAt}(\mathit{vote}(Ag, M, \mathit{Vote}),\ T),\ T' \leq T < T'',\\ &\quad \text{holdsAt}(\mathit{status}(M)=\mathit{voting},\ T).\end{aligned}$$

$\mathit{voted}(Ag, M)=\mathit{Vote}$ is a fluent-value pair (FVP) recording the Vote, i.e., *aye/nay*, of agent Ag on motion M. $\mathit{vote}(Ag, M, \mathit{Vote})$ expresses the act of voting, while $\mathit{status}(M)$ is a fluent recording the status of motion M, i.e., $\mathit{proposed}$, voting, voted or null. The above rule expresses the effects of casting a 'valid' vote, i.e., the vote for a motion M is recorded if the status of M is voting. ♦

Semantics. An event description in RTEC defines a *dependency graph* expressing the relationships between the FVPs of the event description.

Definition 3 (Dependency Graph)**.** The dependency graph of an event description is a directed graph such that:
1. Each vertex denotes a FVP $F=V$;
2. There exists an edge $(F_j=V_j, F_i=V_i)$ iff there is an initiatedAt or terminatedAt rule for $F_i=V_i$ having holdsAt$(F_j=V_j,\ T)$ as one of its conditions. ■

RTEC restricts attention to event descriptions defining *acyclic* dependency graphs whereby it is possible to define a function *level* that maps all FVPs $F=V$ to the non-negative integers according to the following definition.

Definition 4 (FVP Level in RTEC)**.** Given a dependency graph in RTEC, the *level* of a FVP $F=V$ is:
1. 1, if the vertex of $F=V$ has no incoming edges. In other words, $F=V$ is defined only in terms of events.
2. $n > 1$, if the vertex of $F=V$ has at least one incoming edge from a vertex of a FVP of level $n-1$, and zero or more incoming edges from vertices of FVPs of levels lower than $n-1$. ■

Proposition 1 (Semantics of RTEC)**.** An event description in RTEC is a locally stratified logic program (Przymusinski 1987). ◊

A stratification of an event description may be constructed as follows. The first stratum contains all groundings of happensAt. The remaining strata are formed by following, in a bottom-up fashion, the FVP levels of the dependency graph.

Reasoning. The key reasoning task of RTEC is the computation of the maximal intervals during which a fluent-value pair (FVP) of an event description holds continuously. To achieve this, RTEC computes the initiation points of $F=V$ by evaluating the initiatedAt rules for $F=V$. If there is at least one initiation point, then RTEC computes all time-points T at which $F=V$ is 'broken'. $F=V$ is said to be 'broken' at time-point T if $F=V$ is terminated or F is initiated with a value $V' \neq V$ at T. These are the termination points of $F=V$. Subsequently, RTEC constructs the list of maximal intervals of $F=V$ by matching each initiation point T_i of $F=V$ with the first termination point T_b after T_i, ignoring every intermediate initiation point between T_i and T_b. holdsFor$(F=V,\ I)$ denotes that $F=V$ holds continuously in the maximal intervals of the list I. holdsAt$(F=V,\ T)$ is then evaluated by checking whether T belongs to one of the maximal intervals of list I for which holdsFor$(F=V,\ I)$.

RTEC employs a simple caching mechanism to avoid unnecessary re-computations, according to which the FVPs of an event description are processed in an order specified by its dependency graph. RTEC processes FVPs in a bottom-up manner, computing and caching their intervals level-by-level. This way, the intervals of the FVPs that are required for the processing of a FVP of level n are fetched from the cache without the need for re-computation.

RTEC performs continuous query processing to compute the maximal intervals of FVPs. At each query time q_i, the events that fall within a specified sliding window ω are taken into consideration. All events that took place before or at $q_i-\omega$ are discarded/'forgotten'. This way, the cost of reasoning depends on the size of ω and not on the complete stream. The size of ω and the temporal distance between two consecutive query times, i.e., the 'step' q_i-q_{i-1}, are parameters that may be manually chosen or optimised to meet the requirements of a given application. In the case that events arrive at RTEC with delays, e.g., due to network delays, it is preferable to make ω longer than the step. This way, we may compute, at q_i, the effects of events that took place in $(q_i{-}\omega,\ q_{i-1}]$, but arrived after q_{i-1}.

3 Cyclic Dependencies

Temporal specifications, such as those found in composite event recognition (Giatrakos et al. 2020) and multi-agent systems, often include cyclic dependencies.

Example 2. Consider, e.g., the specification of the status of a motion in voting:

$$\begin{aligned}&\mathsf{initiatedAt}(status(M)=proposed,\ T',\ T,\ T'')\leftarrow\\&\quad\mathsf{happensAt}(propose(P,M),\ T),\ T'\leq T<T'',\\&\quad\mathsf{holdsAt}(status(M)=null,\ T).\end{aligned}\tag{2}$$

$$\begin{aligned}&\mathsf{initiatedAt}(status(M)=voting,\ T',\ T,\ T'')\leftarrow\\&\quad\mathsf{happensAt}(second(S,M),\ T),\ T'\leq T<T'',\\&\quad\mathsf{holdsAt}(status(M)=proposed,\ T).\end{aligned}\tag{3}$$

$$\begin{aligned}&\mathsf{initiatedAt}(status(M)=voted,\ T',\ T,\ T'')\leftarrow\\&\quad\mathsf{happensAt}(close\_ballot(C,M),T),T'\leq T<T'',\\&\quad\mathsf{holdsAt}(status(M)=voting,\ T).\end{aligned}\tag{4}$$

$$\begin{aligned}&\mathsf{initiatedAt}(status(M)=null,\ T',\ T,\ T'')\leftarrow\\&\quad\mathsf{happensAt}(declare(C,M,R),\ T),\ T'\leq T<T'',\\&\quad\mathsf{holdsAt}(status(M)=voted,\ T).\end{aligned}\tag{5}$$

The first condition of each rule expresses an agent action; $declare(C,M,R)$, e.g., expresses that agent C declared the outcome R of voting on motion M. In all actions, the first argument denotes the agent that performed the action, while the second argument denotes the motion. The formalisation above expresses the various stages of a motion M: $proposed$ (the motion needs to be seconded before voting may start), $voting$ (voters may cast their votes), $voted$ (voting has ended and the chair may declare the outcome) and $null$. The effects of an agent message on $status(M)$ depend on the value of this fluent at the time of issuing the message. A message $propose(P,M)$, e.g., from a proposer P results in $status(M)=proposed$ provided that $status(M)=null$ at the time of sending $propose(P,M)$. Performing this action when $status(M)\neq null$ has no effect on $status(M)$. The effects of the remaining actions are formalised similarly. The specification of $status$ includes an initial value for this fluent — this is omitted to simplify the presentation. ♦

The top part of Figure 1 displays the dependency graph defined by the event description of a voting protocol. This graph contains a cycle which is formed by rules (2)–(5). As another example, the bottom part of Figure 1 displays a dependency graph of NetBill, a protocol for exchanging encrypted digital goods (Sirbu 1997; Artikis and Sergot 2010), also including a cycle. In this specification, a contract concerning digital goods may be awarded to an agent that has not been suspended, while an agent may be (temporarily) suspended when not fulfilling the obligations of some other contract. The complete formalisations of voting and NetBill are publicly available[1].

RTEC cannot handle cyclic dependencies. When computing the initiation points of, e.g., $status(M)=proposed$ we cannot assume, as RTEC does, that the FVPs appearing in the body of rule (2) have been processed and thus their intervals can be fetched from the cache. $status(M)=proposed$ depends on $status(M)=null$ which in turn depends on $status(M)=proposed$.

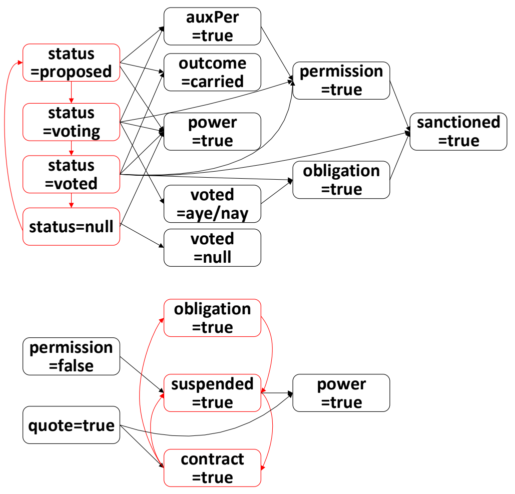

Figure 1: Dependency Graphs: Voting (top) and NetBill (bottom). We omit the arguments of fluents, and group FVPs with the same conditions into a single node, to simplify the presentation. Cycles are coloured red. Apart from the basic features of these protocols, e.g., motions in voting and quotes for digital goods in NetBill, the specifications express the normative positions of the agents, such as institutionalised power, permission and obligation, as well as sanctions/suspensions for handling non-conformance with obligations and the performance of forbidden actions.

This is not an issue, however, for other Event Calculus dialects, which can process FVPs with cyclic dependencies. Consider, e.g., the formalisation below:

$$\begin{aligned}&\mathsf{holdsAt}(F=V,\ T)\leftarrow\\&\quad\mathsf{initiatedAt}(F=V,\ q_i{-}\omega,\ T_i,\ T),\\&\quad\mathsf{not\ brokenBetween}(F=V,\ T_i,\ T).\end{aligned}\tag{6}$$

$$\begin{aligned}&\mathsf{brokenBetween}(F=V,\ T_i,\ T)\leftarrow\\&\quad\mathsf{terminatedAt}(F=V,\ T_i,\ T_b,\ T).\end{aligned}\tag{7}$$

$$\begin{aligned}&\mathsf{brokenBetween}(F=V,\ T_i,\ T)\leftarrow\\&\quad\mathsf{initiatedAt}(F=V',\ T_i,\ T_b,\ T),\ V\neq V'.\end{aligned}\tag{8}$$

Rule (6) specifies that $F{=}V$ holds at time-point T if it was initiated at some time-point T_i between the beginning of the current window $q_i{-}\omega$ and T, and has not been 'broken' between T_i and T. $F{=}V$ is broken between T_i and T if it is terminated in that interval (see rule (7)). A fluent cannot have more than one value at any time. Rule (8) captures this feature: if $F=V'$ is initiated at time $T_b\in[T_i,T)$ then $F{=}V$ is said to be broken in $[T_i,T)$, for all possible values V, other than V', of F.

Example 3. Assume that the current window $(q_i{-}\omega, q_i]$

	Predicate	Rule
1	initiatedAt$(s(m) = proposed,\ q_i{-}\omega,\ T,\ q_i)$?	(2)
2	happensAt$(propose(ag_p, m),\ t_4)$ ✓	(2), (9)
3	holdsAt$(s(m) = null,\ t_4)$?	(2), (6)
4	initiatedAt$(s(m) = null,\ q_i{-}\omega,\ T_i,\ t_4)$?	(6), (5)
5	happensAt$(declare(ag_c, m, not_carried),\ t_3)$ ✓	(5), (9)
6	holdsAt$(s(m) = voted,\ t_3)$?	(5), (6)
7	initiatedAt$(s(m) = voted,\ q_i{-}\omega,\ T_i',\ t_3)$?	(6), (4)
8	happensAt$(close_ballot(ag_c, m),\ t_2)$ ✓	(4), (9)
9	holdsAt$(s(m) = voting,\ t_2)$?	(4), (6)
10	initiatedAt$(s(m) = voting,\ q_i{-}\omega,\ T_i'',\ t_2)$?	(6), (3)
11	happensAt$(second(ag_s, m),\ t_1)$ ✓	(3), (9)
12	holdsAt$(s(m) = proposed,\ t_1)$?	(3), (6)
13	initiatedAt$(s(m) = proposed, q_i{-}\omega, T_i''', t_1)$?	(6)
14	initiatedAt$(s(m) = proposed, q_i{-}\omega, q_i{-}\omega, t_1)$ ✓	(10)
15	brokenBetween$(s(m) = proposed,\ q_i{-}\omega,\ t_1)$?	(6), (8)
16	initiatedAt$(s(m) = voting,\ q_i{-}\omega,\ T_b,\ t_1)$ ×	(8), (3)
17	initiatedAt$(s(m) = voted,\ q_i{-}\omega,\ T_b,\ t_1)$ ×	(8), (4)
18	initiatedAt$(s(m) = null,\ q_i{-}\omega,\ T_b,\ t_1)$ ×	(8), (5)
19	brokenBetween$(s(m) = proposed, q_i{-}\omega, t_1)$ ×	(8), (6)
20	holdsAt$(s(m) = proposed,\ t_1)$ ✓	(6), (3)
21	initiatedAt$(s(m) = voting,\ q_i{-}\omega,\ t_1,\ t_2)$ ✓	(3), (6)
22	brokenBetween$(s(m) = voting,\ t_1,\ t_2)$ ×	(6), (8)
23	holdsAt$(s(m) = voting,\ t_2)$ ✓	(6), (4)
24	initiatedAt$(s(m) = voted,\ q_i{-}\omega,\ t_2,\ t_3)$ ✓	(4), (6)
25	brokenBetween$(s(m) = voted,\ t_2,\ t_3)$ ×	(6), (8)
26	holdsAt$(s(m) = voted,\ t_3)$ ✓	(6), (5)
27	initiatedAt$(s(m) = null,\ q_i{-}\omega,\ t_3,\ t_4)$ ✓	(5), (6)
28	brokenBetween$(s(m) = null,\ t_3,\ t_4)$ ×	(6), (8)
29	holdsAt$(s(m) = null,\ t_4)$ ✓	(6), (2)
30	initiatedAt$(s(m) = proposed,\ q_i{-}\omega,\ t_4,\ q_i)$ ✓	(2)

Table 1: Narrative assimilation in the Event Calculus. '$status(m)$' is abbreviated as '$s(m)$' to fit the column margins. The second column shows the evaluated predicates and the third column refers to the rules used in their evaluation. We use '?' to indicate that we will illustrate predicate evaluation, '✓' to express a successful evaluation, and '×' to denote an unsuccessful evaluation. The predicates in the second column are indented to distinguish between the head and body atoms of a rule.

consists of the following event narrative/stream:

$$\begin{array}{l} \text{happensAt}(second(ag_s, m),\ t_1). \\ \text{happensAt}(close\_ballot(ag_c, m),\ t_2). \\ \text{happensAt}(declare(ag_c, m, not\_carried),\ t_3). \\ \text{happensAt}(propose(ag_p, m),\ t_4). \\ \text{where } q_i{-}\omega < t_1 < t_2 < t_3 < t_4 < q_i \end{array} \tag{9}$$

Moreover, the initial value of $status(M)$, in the current window, is set to $proposed$:

$$\begin{array}{l} \text{initiatedAt}(status(M) = proposed,\ q_i{-}\omega,\ q_i{-}\omega,\ T) \leftarrow \\ \quad T > q_i{-}\omega. \end{array} \tag{10}$$

To compute the maximal intervals for which $status(m) = proposed$ holds continuously, we first need to compute the initiation points of this FVP. Table 1 illustrates this process. To save space, we omit from this table the evaluation of the second condition (double inequality) of the initiatedAt and terminatedAt rules (see e.g. rule (2)); moreover, in the middle part of the table (see lines 14–19) we omit the presentation of the happensAt calls, while in the lower part of the table (see lines 20–30) we do not show the proof of brokenBetween. A solution to the initiatedAt call of line 1, i.e., a initiation point of $status(m) = proposed$ in $[q_i{-}\omega, q_i)$, is shown in line 30 (initiation point: t_4). ♦

Rule-set (6)–(8) constitutes a very inefficient implementation, leading to numerous unnecessary re-computations. Assume, e.g., that the event stream (9) includes an additional event:

$$\begin{array}{l} \text{happensAt}(propose(ag_p, m),\ t_5). \\ \text{where } t_4 < t_5 < q_i \end{array} \tag{11}$$

In this case, most predicate calls presented in Table 1 would have to be repeated, in order to determine whether the above event creates new initiation points for $status(m) = proposed$. More precisely, the predicate calls presented in lines 1–27 of Table 1 would be repeated, with the exception that time-point t_4 is replaced by t_5, i.e., the time-point of the $propose$ event of expression (11). Subsequently, the call to brokenBetween$(status(m) = null, t_3, t_5)$ would succeed, since $status(m) = null$ is broken at t_4, thus leading to no new initiation points for $status(m) = proposed$. Similarly, additional messages in the event stream, such as several agents proposing or seconding a motion, would increase significantly the number of unnecessary re-computations.

Furthermore, consider, e.g., the computation of the maximal intervals for which $status(M) = voting$ holds continuously, that could follow the computation of the maximal intervals for which $status(M) = proposed$. First, we would have to compute the initiation points of $status(M) = voting$, i.e.:

$$\text{initiatedAt}(status(M) = voting,\ q_i{-}\omega,\ T,\ q_i).$$

To calculate these initiation points, we would have to repeat the computations presented in lines 10–21 of Table 1, i.e., we would have to prove again that holdsAt$(status(M) = proposed, t_1)$. Similarly, to calculate the maximal intervals for which $status(M) = voted$, we would have to repeat the computations presented in lines 7–24 of Table 1, for computing the initiation points of this FVP, while to calculate the maximal intervals for which $status(M) = null$, we would have to repeat the computations presented in lines 4–27 of Table 1.

4 Efficient Treatment of Cyclic Dependencies

To address these issues, we propose RTEC$_\circ$, an extension of RTEC that computes, in an efficient way, the maximal intervals during which a FVP with cyclic dependencies holds continuously. To achieve this, RTEC$_\circ$ performs incremental caching when processing FVPs in a cycle. Below we present the semantics, reasoning algorithms and complexity of RTEC$_\circ$. The syntax of the rules in RTEC$_\circ$ is the same as that of RTEC.

Semantics. In RTEC$_\circ$, the dependency graph of an event description may include cycles. Thus, we need to modify Definition 4 of FVP level to cater for cyclic dependency graphs. The strongly connected components (SCCs) of a cyclic dependency graph include either a single vertex of a FVP with no cyclic dependencies or a set of vertices of FVPs among which there are cyclic dependencies. A dependency graph becomes acyclic by contracting its SCCs into single vertices.

Definition 5 (SCC Contracted Graph)**.** Given a directed graph G, its set of vertices V, its set of edges E and its SCCs $S_1, S_2, \ldots, S_n$, the *SCC contracted graph* $G' = (V',\ E')$ of G is defined as follows:
1. $V' = \bigcup_{1 \leq i \leq n} \{v_{S_i}\}$.
2. $e = (v_{S_i},\ v_{S_j}) \in E'$ iff $\exists v_i, v_j \in V$, such that $v_i \in S_i$, $v_j \in S_j$ and $e = (v_i,\ v_j) \in E$. ■

By construction, G' is acyclic. To construct the SCC contracted graph of voting (resp. NetBill), for example, we must merge the red nodes of the top (bottom) dependency graph shown in Figure 1 into a single node.

Definition 6 (FVP Level in RTEC$_\circ$)**.** Given a dependency graph G in RTEC$_\circ$ and the SCC contracted graph G' of G, the *level* of a FVP $F{=}V$ included in the SCC S_i of G is equal to the level of the vertex v_{S_i} of G', which is derived by following Definition 4. ■

According to Definition 6, all FVPs whose vertices are in the same cycle, and thus in the same SCC of the dependency graph, have the same level.

Proposition 2 (Semantics of RTEC$_\circ$)**.** An event description in RTEC$_\circ$ is a locally stratified logic program. ◊

Unlike RTEC, a local stratification of an event description in RTEC$_\circ$ cannot be derived solely by partitioning the groundings of its predicates in terms of the level of the FVP they concern. The ground predicates for FVPs with cyclic dependencies have to be stratified further in terms of their time-stamp. At each FVP level (with cyclic dependencies), additional strata may be introduced for each time-point of the window ω.

Reasoning. Similar to RTEC, RTEC$_\circ$ computes and caches the maximal intervals of FVPs in a bottom-up manner, following their level in the dependency graph, while the intervals of FVPs in the same level may be computed and cached in any order. In contrast to RTEC, RTEC$_\circ$ supports cyclic dependencies, and employs Algorithms 1 and 2 to evaluate the holdsAt predicates found in the bodies of initiatedAt and terminatedAt rules defining FVPs in a cycle. Algorithms 1 and 2 are an efficient implementation of rules (6)–(8), i.e., they incorporate an incremental caching technique to avoid unnecessary re-computations, such as those mentioned in Section 3.

To compute holdsAt$(F{=}V,\ T)$, RTEC$_\circ$ first checks whether $F{=}V$ has been processed at the current query-time q_i (see line 1 of Algorithm 1), i.e., whether the maximal intervals for which $F{=}V$ holds continuously have been computed. If they have been computed, then RTEC$_\circ$ fetches them from the cache (line 2) and checks whether T belongs in these intervals (line 3). If the intervals have not been computed yet, then RTEC$_\circ$ checks whether some time-points, before or at T, for which $F{=}V$ holds or not have already been cached (see cachedLEQ in line 4). If that is the

Algorithm 1 holdsAt$(F = V,\ T)$

```
1: if processed($F = V$) then
2:     holdsFor($F = V,\ I$)
3:     if $T \in I$ then return true
4: else if cachedLEQ($T,\ F = V$) $\neq$ [] then
5:     $last$(cachedLEQ($T,\ F = V$), ($T_{lCP},\ Tval_{lCP}$))
6:     if $T == T_{lCP}$ then
7:         if $Tval_{lCP} == +$ then return true
8:     else if holdsAtEC($F = V,\ T_{lCP},\ T,\ Tval_{lCP}$) then
9:         updateCache($F = V,\ T,\ +$)
10:        return true
11:    else updateCache($F = V,\ T,\ -$)
12: else
13:    if holdsAtEC($F = V,\ q_i - \omega,\ T,\ -$) then
14:        updateCache($F = V,\ T,\ +$)
15:        return true
16:    else updateCache($F = V,\ T,\ -$)
17: return false
```

Algorithm 2 holdsAtEC$(F = V,\ T_{lCP},\ T,\ Tval_{lCP})$

```
1: if $T_{lCP} < T$ then
2:     if $Tval_{lCP} == +$ then
3:         if brokenAt($F = V,\ T_{lCP},\ T_b,\ T$) then
4:             if holdsAtEC($F = V,\ T_b + 1,\ T,\ -$) then
5:                 return true
6:         else return true
7:     else
8:         if initiatedAt($F = V,\ T_{lCP},\ T_i,\ T$) then
9:             if holdsAtEC($F = V,\ T_i + 1,\ T,\ +$) then
10:                return true
11: return false
```

case (lines 4–11), then RTEC$_\circ$ retrieves the cached time-point T_{lCP} closer to T along with the truth value $Tval_{lCP}$ of $F{=}V$ at T_{lCP} (line 5). If T_{lCP} coincides with T (line 6), then its truth value is returned; '$+$' denotes that $F{=}V$ holds and '$-$' that it does not. Otherwise, i.e., if T_{lCP} does not coincide with T (lines 8–11), RTEC$_\circ$ computes whether $F{=}V$ holds at T restricting the evaluation in $[T_{lCP},\ T)$; the evaluation is performed by means of holdsAtEC, which is presented in Algorithm 2. Moreover, the outcome of the evaluation is cached (lines 9 and 11). Finally, if no time-points for $F{=}V$ have been cached (lines 12–16), then RTEC$_\circ$ computes whether $F{=}V$ holds at T performing the evaluation in $[q_i{-}\omega,\ T)$ (line 13) and caching the outcome.

Algorithm 2 presents the steps for evaluating holdsAtEC$(F = V,\ T_{lCP},\ T,\ Tval_{lCP})$, i.e., calculating whether $F{=}V$ holds at T restricting the search in $[T_{lCP},\ T)$ and taking into consideration $Tval_{lCP}$, i.e., the truth value of $F{=}V$ at T_{lCP}. All arguments of holdsAtEC are ground. First, if $F{=}V$ holds at T_{lCP} (see lines 2–6 of Algorithm 2), then we check whether $F{=}V$ is broken at some time $T_b \in [T_{lCP},\ T)$. brokenAt is a simple variation of brokenBetween (see rules (7) and (8)) returning the time-point at which a FVP is broken. If $F{=}V$ is not broken in $[T_{lCP},\ T)$, then RTEC$_\circ$ returns that $F{=}V$ holds at T. Otherwise, i.e., if $F{=}V$ is broken at some $T_b \in [T_{lCP},\ T)$, RTEC$_\circ$ calls recursively holdsAtEC, this time restricting the

	Predicate	Rule/Alg.
1	$\mathsf{initiatedAt}(s(m)\!=\!proposed,\ q_i\!-\!\omega,\ T,\ q_i)$?	(2)
2	$\mathsf{happensAt}(propose(ag_p, m),\ t_5)$ ✓	(2), (11)
3	$\mathsf{holdsAt}(s(m)\!=\!null,\ t_5)$?	(2), a1
4	$\mathsf{processed}(s(m)\!=\!null)$ ×	a1
5	$last(\mathsf{cachedLEQ}(t_5,\ s(m)\!=\!null),\ (t_4, +))$ ✓	a1, (12)
6	$\mathsf{holdsAtEC}(s(m)\!=\!null,\ t_4,\ t_5,\ +)$?	a1, a2
7	$\mathsf{brokenAt}(s(m)\!=\!null,\ t_4,\ t_4,\ t_5)$ ✓	a2
8	$\mathsf{holdsAtEC}(s(m)\!=\!null,\ t_4\!+\!1,\ t_5,\ -)$ ×	a2
9	$\mathsf{holdsAtEC}(s(m)\!=\!null,\ t_4,\ t_5,\ +)$ ×	a2, a1
10	$\mathsf{updateCache}(s(m)\!=\!null,\ t_5,\ -)$ ✓	a1
11	$\mathsf{holdsAt}(s(m)\!=\!null,\ t_5)$ ×	a1, (2)
12	$\mathsf{initiatedAt}(s(m)\!=\!proposed,\ q_i\!-\!\omega,\ T,\ q_i)$ ×	(2)
13	$\mathsf{initiatedAt}(s(m)\!=\!voting,\ q_i\!-\!\omega,\ T,\ q_i)$?	(3)
14	$\mathsf{happensAt}(second(ag_s, m),\ t_1)$ ✓	(3), (9)
15	$\mathsf{holdsAt}(s(m)\!=\!proposed,\ t_1)$?	(3), a1
16	$\mathsf{processed}(s(m)\!=\!proposed)$ ✓	a1
17	$\mathsf{holdsFor}(s(m)\!=\!proposed,\ I),\ t_1 \in I$ ✓	a1
18	$\mathsf{holdsAt}(s(m)\!=\!proposed,\ t_1)$ ✓	a1, (3)
19	$\mathsf{initiatedAt}(s(m)\!=\!voting,\ q_i\!-\!\omega,\ t_1,\ q_i)$ ✓	(3)
20	$\mathsf{initiatedAt}(s(m)\!=\!null,\ q_i\!-\!\omega,\ T,\ q_i)$?	(5)
21	$\mathsf{happensAt}(declare(ag_c, m),\ t_3)$ ✓	(5), (9)
22	$\mathsf{holdsAt}(s(m)\!=\!voted,\ t_3)$?	(5), a1
23	$\mathsf{processed}(s(m)\!=\!voted)$ ×	a1
24	$last(\mathsf{cachedLEQ}(t_3,\ s(m)\!=\!voted),\ (t_3, +))$ ✓	a1, (12)
25	$\mathsf{holdsAt}(s(m)\!=\!proposed,\ t_3)$ ✓	a1, (5′)
26	$\mathsf{initiatedAt}(s(m)\!=\!null,\ q_i\!-\!\omega,\ t_3,\ q_i)$ ✓	(5)

Table 2: Stream reasoning with RTEC$_\circ$. '$status(m)$' is abbreviated as '$s(m)$'. a1–a2 refer to Algorithms 1–2.

evaluation interval to $[T_b\!+\!1,\ T)$, where $T_b\!+\!1$ denotes the next time-point of T_b, and starting from the negative truth value of $F\!=\!V$ at T_b. Second, if $F\!=\!V$ does not hold at T_{lCP} (see lines 7–10), then RTEC$_\circ$ determines whether $F\!=\!V$ is initiated at some $T_i \in [T_{lCP},\ T)$. If $F\!=\!V$ is initiated in this interval, RTEC$_\circ$ calls $\mathsf{holdsAtEC}$ with the evaluation interval $[T_i\!+\!1,\ T)$ and the initial truth value of $F\!=\!V$ being positive. The example below illustrates the incremental caching of RTEC$_\circ$.

Example 4. Consider the stream consisting of events (9) and (11), while the initial value of $status(m)$ in the current window is $proposed$ (see formula (10)). The computation of the initiation points of $status(m)\!=\!proposed$ commences with the evaluation presented in Table 1, with some additional calls, e.g., to $\mathsf{processed}$ (see Algorithm 1), to check for cached intervals and time-points. Furthermore, RTEC$_\circ$ caches all computed time-points for which a FVP holds. In this example, RTEC$_\circ$ caches the following:

$$\begin{aligned}&\mathsf{holdsAt}(status(m)\!=\!proposed,\ t_1).\\&\mathsf{holdsAt}(status(m)\!=\!voting,\ t_2).\\&\mathsf{holdsAt}(status(m)\!=\!voted,\ t_3).\\&\mathsf{holdsAt}(status(m)\!=\!null,\ t_4).\end{aligned} \tag{12}$$

The top part of Table 2 (lines 1–12) shows the remaining evaluation concerning the initiation points of $status(m)\!=\!proposed$, and, in particular, the processing of the event $propose(ag_p, m)$ at t_5. As in Table 1, we show a subset of the predicate calls to simplify the presentation. To prove whether $status(m)\!=\!null$ holds at t_5, as required by rule (2), RTEC$_\circ$ follows Algorithm 1 and consults the cache, first by determining if the intervals of this FVP have already been computed—at this stage they have not—and then by looking for cached time-points. The closest cached time-point to t_5 is t_4, while $status(m)\!=\!null$ holds at t_4 (see cache (12)). Subsequently, RTEC$_\circ$ follows Algorithm 2 to prove whether $status(m)\!=\!null$ still holds at t_5. This is not the case, since $status(m)\!=\!null$ is broken at t_4 (by the occurrence of a $propose$ event at that time) and not re-initiated in the meantime. Consequently, no new initiation points are computed for $status(m)\!=\!proposed$.

With the use of caching, RTEC$_\circ$ can restrict attention to the events that have not been processed so far, avoiding unnecessary re-computations. In this example, without the cached time-points of $status$ we would have to repeat most of the steps presented in Example 3, only to compute again the values of $status$ before t_5.

The middle part of Table 2 (lines 13–19) shows the computation of the initiation points of $status(m)\!=\!voting$. At this stage, the intervals I for which $status(m)\!=\!proposed$ holds continuously have been computed and cached, and are $(q_i\!-\!\omega,\ t_1]$ and $(t_4,\ \infty)$. In other words, m is said to be 'proposed' from the beginning of the current window until t_1, and since t_4. As can be seen from Table 2, the computation of the initiation points of $status(m)\!=\!voting$ is very efficient. RTEC$_\circ$ quickly computes that $status(m)\!=\!proposed$ holds at t_1, as required by rule (3), since the intervals of this FVP may be fetched from the cache. This way, the unnecessary re-computations discussed after Example 3 are avoided.

The bottom part of Table 2 (lines 20–26) presents another illustration of the effects of incremental caching, by showing the computation of the initiation points of $status(m)\!=\!null$. Again, reasoning is very efficient: we fetch from the cache (12) time-point t_3 for which $status(m)\!=\!voted$ holds, and directly prove rule (5) expressing the conditions in which $status(m)\!=\!null$ is initiated. ⧫

Complexity. We present the worst-case complexity of Algorithms 1 and 2, and compare it against the complexity of reasoning with cycles in the absence of incremental caching. According to Algorithms 1 and 2, RTEC$_\circ$ consults its cache to evaluate $\mathsf{holdsAt}(F = V,\ T)$, and when reasoning is necessary, it is restricted to intervals that have not been explored so far. Moreover, after the end of the evaluation of $\mathsf{holdsAt}(F = V,\ T)$ the cache is updated. Therefore, in the worst-case, RTEC$_\circ$ will have to evaluate each $\mathsf{initiatedAt}/\mathsf{terminatedAt}$ rule for $F\!=\!V$ at every-time point of ω, but no more than that. In other words, the worst-case complexity is

$$\mathcal{O}(\omega k), \tag{13}$$

where ω denotes the window size and k is the cost of computing whether a FVP is initiated or terminated at a given time-point (see (Artikis, Sergot, and Paliouras 2015) for an estimation of k).

In the absence of incremental caching, we do not mark the intervals that have been explored so far, and thus we may repeat evaluations that have already been performed. In the worst-case, the evaluation of all earlier initiation and termination points of FVPs in a cycle has to repeated ω times.

Thus, the worst-case complexity is

$$\mathcal{O}(\omega^2 rk), \tag{14}$$

where r is the number of FVPs in a cycle.

In real-world applications, ω, i.e., the window size, can be large. Therefore, the use of the caching mechanism of RTEC$_\circ$ leads to significant performance gains (compare expressions (13) and (14)). This is key difference of our proposed computational framework from related work.

5 Experimental Analysis

5.1 Experimental Setup

We evaluated RTEC$_\circ$ on two multi-agent systems (MAS) protocols, formalising a voting procedure and NetBill, an e-commerce procedure, as well as on composite event recognition for maritime situational awareness. The complete event descriptions of all applications, including the corresponding datasets, are available with the code of RTEC$_\circ$[1], allowing the *reproducibility* of our empirical analysis.

Voting & NetBill. The voting protocol is our running example (Pitt et al. 2006). Agents occupy the roles of proposer, seconder, voter and/or chair, and deliberate over various motions over time. NetBill includes consumers and merchants negotiating over digital goods (Sirbu 1997). Consumers request quotes for goods and the interested merchants reply with the quotes, or even proactively advertise their goods. A timely acceptance of a quote leads to a contract, which defines the processes of sending the digital goods and payment (Artikis and Sergot 2010). In both protocols, a set of normative positions, such as institutionalised power, permission and obligation, guide the behaviour of the agents (Sergot 2001). Moreover, sanctions deal with the performance of forbidden actions and non-compliance with obligations. Figure 1 displays the dependency graphs of these protocols.

Synthetic data generators produced the event streams of the two protocols. In the case of voting, multiple agents and motions were generated, and the agents were assigned roles. Then, agents proposed motions, some of which were subsequently seconded and voted for. In the case of NetBill, quotes were progressively requested, presented, accepted and sometimes fulfilled. To simulate realistic MAS, the data generators produced events which do not comply with the rules, e.g. closing the ballot before all votes are cast, and not complying with the terms of a contract in NetBill. RTEC$_\circ$ was instructed to compute, in an online fashion, the maximal intervals of the FVPs presented in Figure 1, e.g., compute the maximal intervals in which an agent is permitted to perform an action.

Maritime Situational Awareness. In this application, the input events are Automatic Identification System (AIS) position signals emitted by vessels, including information about their heading, speed and navigational status. FVPs express various types of dangerous, suspicious and illegal vessel activity, such as ship-to-ship transfer of goods in the open sea, that must be detected in real-time. We extended the event description of Pitsikalis et al. (2019) with cyclic dependencies among FVPs, as required by domain experts, in order to capture more accurately the maritime behaviours of interest, such as fishing. We evaluated RTEC$_\circ$ on a publicly available dataset[2] of 18M AIS position signals, emitted from 5K vessels sailing in the Atlantic Ocean around the port of Brest, France, between October 2015–March 2016. Moreover, we employed a much larger dataset, made available to us by IMIS Global[3], containing 55M position signals from 34K vessels sailing in the European seas between January 1-30, 2016. A description of both datasets may be found in (Pitsikalis et al. 2019).

RTEC$_\circ$ is written in Prolog[1]. Our experiments were conducted using YAP-6.3 Prolog, on a single node of a desktop PC running Ubuntu 20.04, with Intel Core i7-4770 CPU @ 3.40GHz and 16GB RAM.

5.2 Experimental Results

We begin with a set of experiments which demonstrate the benefits of our caching mechanism for processing FVPs with cyclic dependencies. For this purpose, we employed a downgraded version of RTEC$_\circ$, named RTEC$_\circ$-naive, which processes FVPs with cyclic dependencies without a caching mechanism. Figure 2a shows the experimental results in voting and NetBill. The presented reasoning times are an average of 100 windows, while RTEC$_\circ$ and RTEC$_\circ$-naive always produced the same FVP intervals. We used a maximum response time of 10 seconds per window, i.e., when the reasoning time exceeded this threshold, then we stopped the execution. The reasoning times of RTEC$_\circ$-naive, using windows of 80 time-points, exceeded this threshold for both MAS protocols and, therefore, are not presented in Figure 2a. Our experimental results show that, as we increase the window size, the reasoning times of RTEC$_\circ$-naive increase exponentially, while RTEC$_\circ$ scales much better. These results are consistent with our complexity analysis, which indicated that the absence of the caching mechanism of RTEC$_\circ$ results in unnecessary re-computations, that increase with the window size.

In the next set of experiments, we compared RTEC$_\circ$ against jREC, a Java-Prolog implementation of the 'Reactive Event Calculus' (Chesani et al. 2010; Montali et al. 2013). jREC has been evaluated in several application domains (Bragaglia et al. 2012; Chesani et al. 2013; Loreti et al. 2019). Moreover, it is an open-source implementation[4], which facilitates the comparison against RTEC$_\circ$. Figure 2b shows the results of the comparison. Both systems were evaluated on a fragment of the event description of the voting protocol. We restricted attention to the $status$ fluent, the specification of which creates a cycle in the dependency graph (see Figure 1). The task, therefore, was to compute the maximal intervals for which $status$ has some value ($proposed$, $voting$, $voted$, $null$) continuously. We used a window size of 10 time-points for RTEC$_\circ$ and instructed jREC to evaluate the trace of input events every 10 time-points. We performed experiments for windows with 800–6,400 events, and made sure that both systems computed the same FVP intervals. Figure 2b shows that the use of RTEC$_\circ$ leads to significant performance gain. Moving

[2]https://zenodo.org/record/1167595

[3]https://imisglobal.com/

[4]https://www.inf.unibz.it/~montali/tools.html

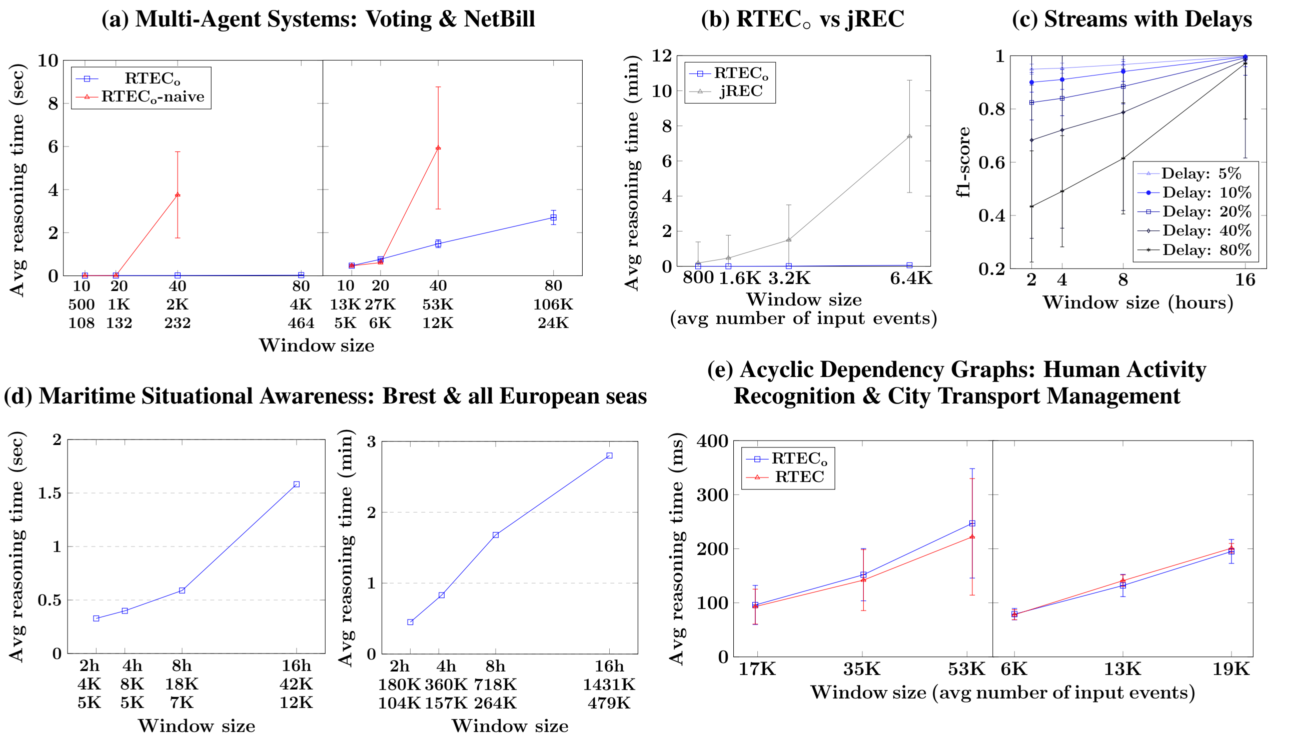

Figure 2: (a) The reasoning times of RTEC$_\circ$ and RTEC$_\circ$-naive in voting (left) and NetBill (right). The horizontal axis denotes window size in terms of time-points (top), average number of input entities (middle) and average number of computed FVP intervals (bottom) per window. (b) RTEC$_\circ$ and jREC computing the maximal intervals of the *status* fluent of the voting protocol. (c) The predictive accuracy of RTEC$_\circ$ in the presence of delayed events on the dataset of Brest. (d) RTEC$_\circ$ for maritime situational awareness on the datasets of Brest (left) and Europe (right). The window size is presented as in figure (a). (e) The reasoning times of RTEC$_\circ$ and RTEC in temporal specifications without cyclic dependencies: human activity recognition (left) and city transport management (right).

from 800 events to 6,400 events barely affects the performance of RTEC$_\circ$. In contrast, jREC requires considerably more time to process the increasing number of input events.

In addition to MAS, we evaluated RTEC$_\circ$ on composite event recognition for maritime situational awareness. We relied on real data, i.e., position signals from thousands of vessels, producing millions of input events. Real data streams often include delayed events. In the maritime domain, vessel position signals may arrive with a delay that exceeds 6 hours, especially when such signals are relayed through satellites. This issue may be addressed by longer, overlapping windows (see Section 2). To simulate realistic scenarios, we introduced delays into the maritime datasets. The temporal extent of the delay was set using a Gamma distribution. The percentage of delayed events, which were chosen uniformly, ranges from 5% to 80%. Figure 2c shows the predictive accuracy of RTEC$_\circ$ when processing data streams with delayed events. The step, i.e., the temporal distance between two consecutive query times, was set to 2 hours. We varied the size of the windows from 2 hours to 16 hours. The f1-scores were derived by comparing the intervals computed by RTEC$_\circ$ on data streams with delayed events to the intervals computed by RTEC$_\circ$ on the respective data streams without delayed events. Figure 2c demonstrates the necessity of longer windows in the maritime domain.

Figure 2d displays the average reasoning times of RTEC$_\circ$ per window size when processing real maritime data. In the dataset of Brest, the 2-hour window includes on average 4K events/vessel position signals, while the 16-hour window includes approximately 42K events. The computed FVP intervals range from 5K, in the 2-hour window, to 12K in the 16-hour window. In the significantly larger dataset concerning all European seas, the windows include 180K–1,431K events, while the computed FVP intervals range between 104K and 479K. We introduced delays to 40% of the inputs events in the data streams. Figure 2d shows that RTEC$_\circ$ is capable of real-time stream reasoning in real-world applications. In the dataset of Brest, e.g., RTEC$_\circ$ reasons about the events of a 16-hour window in just over 1.5 seconds, in order to recognise a wide range of maritime activities of interest, such as fishing, tugging, etc., while in the dataset of all European seas, RTEC$_\circ$ requires just below 3 minutes to reason about a 16-hour window.

For completeness, we compared RTEC$_\circ$ and RTEC in temporal specifications without cyclic dependencies, i.e., the specifications used for human activity recognition and city transport management in (Artikis, Sergot, and Paliouras 2015). Figure 2e displays the average reasoning times. In both applications, the reasoning times of RTEC$_\circ$ are comparable to those of RTEC. In other words, the mechanism of RTEC$_\circ$ for handling FVPs with cyclic dependencies does not impose a computational overhead in applications without such dependencies.

6 Related Work

Numerous computational frameworks based on the Event Calculus have been proposed in the literature. The 'Macro-Event Calculus' (Cervesato and Montanari 2000), e.g., leverages the concept of 'macro-event' to support composite/macro event operators such as sequence, disjunction, parallelism and iteration. The 'Interval-based Event Calculus' (Paschke and Bichler 2008) supports the representation of durative events and includes various event operators, such as sequence, concurrency and negation. The F2LP system translates a reformulation of the Event Calculus into answer set programs, so that the efficient answer set programming solvers may be used for reasoning (Lee and Palla 2012). The 'Reactive Event Calculus' (REC) (Chesani et al. 2010; Montali et al. 2013) adopts a lightweight version of the *update-time* reasoning of the 'Cached Event Calculus' (Chittaro and Montanari 1996), which retrieves and revises the necessary fluent intervals upon the arrival of (delayed) events.

None of these frameworks handles effectively cyclic dependencies. RTEC$_\circ$ includes an incremental caching technique for dealing with such dependencies efficiently, and consequently may scale to high-velocity data streams. Our complexity analysis showed the performance gains of RTEC$_\circ$. Furthermore, our extensive empirical analysis, and comparison with jREC, i.e., the Java-Prolog implementation of REC, demonstrated the effectiveness of our approach.

Various frameworks for reasoning over streams have been proposed in the literature (Dell'Aglio et al. 2017; Dell'Aglio et al. 2019). Ronca (2020), e.g., provides tight complexity bounds for a temporal extension of negation-free Datalog. LARS is a well-known stream reasoning language featuring built-in window constructs (Beck, Dao-Tran, and Eiter 2018). Laser (Bazoobandi, Beck, and Urbani 2017) is a reasoner that employs a fixed-point materialisation of restricted LARS formulas to handle data streams. The empirical analysis of Beck et al. (2018) showed that Laser outperforms other related reasoners (Beck, Eiter, and Folie 2017; Le-Phuoc et al. 2011; Barbieri et al. 2010). Laser, however, cannot express the event descriptions of RTEC$_\circ$ since it is restricted to stratified negation.

Eiter et al. (2019) presented a distributed architecture using 'stream stratification' (Beck, Dao-Tran, and Eiter 2018) in order to decompose LARS programs into sub-programs that may be evaluated in parallel, while Dodaro et al. (2020) presented an approach handling constraint satisfaction problems in a streaming setting by means of reinforcement learning. These issues are orthogonal to our work; e.g., we aim to develop distributed reasoning techniques (Giatrakos et al. 2020) for RTEC$_\circ$ as part of our future work.

In the field of composite event recognition, one of the best known logic-based stream reasoning systems is the Chronicle Recognition System (Dousson and Maigat 2007). TESLA (Cugola and Margara 2010) is an event pattern language that supports content and temporal filters, negation, timers, aggregates and customisable selection and consumption policies. Neither of these frameworks supports cyclic dependencies. ETALIS (Anicic et al. 2012) is a logic programming framework that aims to support event recognition by combining reasoning over streams and background knowledge. ETALIS does not allow for the explicit representation of time, complicating the specification of fluent value changes, including the formalisation of the commonsense law of inertia. Moreover, it is unclear how one may model cyclic dependencies, such as those of the applications presented in this paper, in the language of ETALIS.

A key difference between our work and the aforementioned stream reasoning frameworks lies in the use of the Event Calculus, which allows us to develop expressive temporal specifications for a wide range of applications, such as MAS and composite event recognition. At the same time, the built-in representation of inertia allows us to develop succinct specifications, supporting code maintenance and reasoning efficiency. With the use of the Event Calculus, one may develop intuitive specifications, facilitating the interaction between data scientist and domain (e.g. maritime) expert, and pave the way for explainability. Furthermore, we may develop custom optimisation techniques, such as those presented in this paper, to meet the requirements of contemporary applications concerning latency.

Wan (2009) presented a framework for belief logic programming that eliminates cyclic dependencies by introducing auxiliary rules and atoms representing intermediate results. The number of these auxiliary clauses increases with the length of the cycle. On the contrary, RTEC$_\circ$ handles cycles by utilising the Event Calculus, e.g., the formalisation of the law of inertia, and does not require auxiliary clauses. Moura and Damásio (2015) presented an approach for modular logic programming that supports positive cycles, i.e., cycles without negation. In contrast, RTEC$_\circ$ is not restricted to positive cycles. Moreover, we presented reasoning algorithms for handling cycles in an efficient manner.

7 Summary & Further Work

We presented RTEC$_\circ$, a formal computational framework for reasoning over real-world streams and temporal specifications including cyclic dependencies. The event descriptions in RTEC$_\circ$ are locally stratified logic programs. RTEC$_\circ$ includes a novel incremental caching mechanism, which avoids unnecessary re-computations and thus optimises stream reasoning. We evaluated RTEC$_\circ$ by means of a complexity analysis and showed its benefits. Furthermore, we presented an extensive, reproducible empirical evaluation, on both synthetic and real data streams, including millions of events. For further work, we will investigate the use of distributed reasoning techniques in RTEC$_\circ$ (Eiter, Ogris, and Schekotihin 2019; Giatrakos et al. 2020).

Acknowledgements

This work was supported by the INFORE project, which has received funding from the European Union's Horizon 2020 research and innovation programme, under grant agreement No 825070.

References

Anicic, D.; Rudolph, S.; Fodor, P.; and Stojanovic, N. 2012. Real-time complex event recognition and reasoning-a logic programming approach. *Applied Artificial Intelligence* 26(1-2):6–57.

Artikis, A., and Sergot, M. 2010. Executable specification of open multi-agent systems. *Logic Journal of the IGPL* 18(1):31–65.

Artikis, A.; Sergot, M.; and Paliouras, G. 2015. An event calculus for event recognition. *IEEE TKDE* 27(4):895–908.

Barbieri, D. F.; Braga, D.; Ceri, S.; Valle, E. D.; and Grossniklaus, M. 2010. C-SPARQL: a continuous query language for RDF data streams. *Int. J. Semantic Comput.* 4(1):3–25.

Bazoobandi, H. R.; Beck, H.; and Urbani, J. 2017. Expressive stream reasoning with laser. In *ISWC*, volume 10587, 87–103.

Beck, H.; Dao-Tran, M.; and Eiter, T. 2018. LARS: A logic-based framework for analytic reasoning over streams. *Artif. Intell.* 261:16–70.

Beck, H.; Eiter, T.; and Folie, C. 2017. Ticker: A system for incremental asp-based stream reasoning. *Theory Pract. Log. Program.* 17(5-6):744–763.

Bragaglia, S.; Chesani, F.; Mello, P.; Montali, M.; and Torroni, P. 2012. Reactive event calculus for monitoring global computing applications. In *Logic Programs, Norms and Action*, volume 7360, 123–146.

Cervesato, I., and Montanari, A. 2000. A calculus of macro-events: Progress report. In *TIME*, 47–58.

Chesani, F.; Mello, P.; Montali, M.; and Torroni, P. 2010. A logic-based, reactive calculus of events. *Fundam. Informaticae* 105(1-2):135–161.

Chesani, F.; Mello, P.; Montali, M.; and Torroni, P. 2013. Representing and monitoring social commitments using the event calculus. *Auton. Agents Multi Agent Syst.* 27(1):85–130.

Chittaro, L., and Montanari, A. 1996. Efficient temporal reasoning in the cached event calculus. *Computational Intelligence* 12(3):359–382.

Clark, K. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logic and Databases*. 293–322.

Cugola, G., and Margara, A. 2010. Tesla: a formally defined event specification language. In *DEBS*, 50.

Dell'Aglio, D.; Valle, E. D.; van Harmelen, F.; and Bernstein, A. 2017. Stream reasoning: A survey and outlook. *Data Sci.* 1(1-2):59–83.

Dell'Aglio, D.; Eiter, T.; Heintz, F.; and Phuoc, D. L. 2019. Special issue on stream reasoning. *Semantic Web* 10(3):453–455.

Dodaro, C.; Eiter, T.; Ogris, P.; and Schekotihin, K. 2020. Managing caching strategies for stream reasoning with reinforcement learning. *TPLP* 20(5):625–640.

Dousson, C., and Maigat, P. L. 2007. Chronicle recognition improvement using temporal focusing and hierarchisation. In *IJCAI*, 324–329.

Eiter, T.; Ogris, P.; and Schekotihin, K. 2019. A distributed approach to LARS stream reasoning (system paper). *TPLP* 19(5-6):974–989.

Giatrakos, N.; Alevizos, E.; Artikis, A.; Deligiannakis, A.; and Garofalakis, M. N. 2020. Complex event recognition in the big data era: a survey. *VLDB J.* 29(1):313–352.

Kowalski, R., and Sergot, M. 1986. A logic-based calculus of events. *New Generation Computing* 4(1):67–96.

Le-Phuoc, D.; Dao-Tran, M.; Xavier Parreira, J.; and Hauswirth, M. 2011. A native and adaptive approach for unified processing of linked streams and linked data. In *The Semantic Web – ISWC*, 370–388.

Lee, J., and Palla, R. 2012. Reformulating the situation calculus and the event calculus in the general theory of stable models and in answer set programming. *J. Artif. Intell. Res.* 43:571–620.

Loreti, D.; Chesani, F.; Mello, P.; Roffia, L.; Antoniazzi, F.; Cinotti, T. S.; Paolini, G.; Masotti, D.; and Costanzo, A. 2019. Complex reactive event processing for assisted living: The habitat project case study. *Expert Systems with Applications* 126:200–217.

Montali, M.; Maggi, F. M.; Chesani, F.; Mello, P.; and van der Aalst, W. M. P. 2013. Monitoring business constraints with the event calculus. *ACM Trans. Intell. Syst. Technol.* 5(1):17:1–17:30.

Moura, J., and Damásio, C. V. 2015. Allowing cyclic dependencies in modular logic programming. In *Progress in Artificial Intelligence*, 363–375.

Paschke, A., and Bichler, M. 2008. Knowledge representation concepts for automated SLA management. *Decis. Support Syst.* 46(1):187–205.

Pitsikalis, M.; Artikis, A.; Dreo, R.; Ray, C.; Camossi, E.; and Jousselme, A. 2019. Composite event recognition for maritime monitoring. In *DEBS*, 163–174.

Pitt, J.; Kamara, L.; Sergot, M.; and Artikis, A. 2006. Voting in multi-agent systems. *Computer Journal* 49(2):156–170.

Przymusinski, T. 1987. On the declarate semantics of stratified deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*.

Ronca, A. 2020. *Rule-based stream reasoning*. Ph.D. Dissertation, University of Oxford, UK.

Sergot, M. 2001. A computational theory of normative positions. *ACM Transactions on Computational Logic* 2(4):522–581.

Sirbu, M. 1997. Credits and debits on the Internet. *IEEE Spectrum* 34(2):23–29.

Wan, H. 2009. Belief logic programming with cyclic dependencies. In *Web Reasoning and Rule Systems*, volume 5837, 150–165.