NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Reasoning over Complex Temporal Specifications and Noisy Data Streams

Periklis Mantenoglou

ATHENS

JULY 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Συλλογιστική πάνω σε Πολύπλοκες Χρονικές Προδιαγραφές και Θορυβώδεις Ροές Δεδομένων

Περικλής Μαντένογλου

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2024

PhD THESIS

Reasoning over Complex Temporal Specifications and Noisy Data Streams

Periklis Mantenoglou

SUPERVISOR: Panagiotis Stamatopoulos, Assistant Professor, NKUA

THREE-MEMBER ADVISORY COMMITTEE:

Panagiotis Stamatopoulos, Assistant Professor, NKUA

Alexander Artikis, Associate Professor, University of Piraeus

Georgios Paliouras, Researcher A, NCSR "Demokritos"

SEVEN-MEMBER EXAMINATION COMMITTEE

Panagiotis Stamatopoulos,
Assistant Professor, NKUA

Alexander Artikis,
Associate Professor, University of Piraeus

Georgios Paliouras,
Researcher A, NCSR "Demokritos"

Dimitrios Gunopulos,
Professor, NKUA

Manolis Koubarakis,
Professor, NKUA

Panagiotis Rondogiannis,
Professor, NKUA

Kostas Stathis,
Professor, Royal Holloway University of London

Examination Date: July 31, 2024

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Συλλογιστική πάνω σε Πολύπλοκες Χρονικές Προδιαγραφές και Θορυβώδεις Ροές Δεδομένων

Περικλής Μαντένογλου

Ημερομηνία Εξέτασης: 31 Ιουλίου 2024

ABSTRACT

Contemporary applications commonly demand the detection of ‘situations of interest’ in real-time and with minimal latency. In maritime situational awareness, e.g., it is crucial to identify and report vessel behaviours that may indicate dangerous, illegal or environmentally hazardous activities in a timely manner. Monitoring situations of interest is challenging, as large, high-velocity streams of data are being generated continuously, and it is not feasible to store these streams in memory and process them at a later time. Stream reasoning involves the real-time detection of critical situations by reasoning over large volumes of incrementally-arriving, symbolic data, typically termed as ‘events’. In order to support contemporary applications, a stream reasoning system needs to employ efficient reasoning algorithms that are tailored for the streaming setting, a highly expressive pattern specification language with a formal and declarative semantics, as well as techniques for handling noise and uncertainty. This thesis is motivated by the absence of a framework with all the aforementioned features, and aims at extending state-of-the-art computational frameworks in order to support stream reasoning effectively. We focus on frameworks that employ the Event Calculus, a logic programming formalism for representing events and reasoning about their effects over time. The Event Calculus exhibits a formal, declarative semantics, while supporting both instantaneous and durative events, hierarchical and relational event specifications, temporal persistence and background knowledge. RTEC is a computational framework that is based on the Event Calculus and includes several optimisation techniques for stream reasoning. We propose three extensions of RTEC, i.e., $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A}, supporting, respectively, temporal specifications with cyclic dependencies, events with delayed effects and situations specified using the relations of Allen’s interval algebra. Moreover, we study PIEC, i.e., an Event Calculus-based system for probabilistic event recognition, and present oPIEC, i.e., an extension of PIEC for reasoning over noisy event streams. In order to support large data streams, we propose two bounded-memory versions of oPIEC. All of our proposed frameworks have a formal semantics and feature optimised stream reasoning algorithms that have been assessed both theoretically and empirically, using large, real and synthetic data streams, in experiments that included comparisons with state-of-the-art frameworks. Our analysis demonstrates that $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A} are suitable for stream reasoning with complex temporal specifications, while the variants of oPIEC exhibit high predictive accuracy and reasoning efficiency when reasoning over noisy data streams.

SUBJECT AREA: Knowledge Representation and Reasoning

KEYWORDS: stream reasoning, Event Calculus, logic programming, probabilistic logic programming, uncertainty, temporal pattern matching, composite event recognition

ΠΕΡΙΛΗΨΗ

Οι σύγχρονες εφαρμογές απαιτούν την ανίχνευση «καταστάσεων ενδιαφέροντος» σε πραγματικό χρόνο και με ελάχιστη καθυστέρηση. Στην επιτήρηση ναυτιλιακών δραστηριοτήτων, π.χ., είναι σημαντικό να εντοπίζονται και να αναφέρονται εγκαίρως συμπεριφορές πλοίων που μπορεί να υποδεικνύουν επικίνδυνες, παράνομες ή περιβαλλοντικά επιβλαβείς δραστηριότητες. Η επιτήρηση καταστάσεων ενδιαφέροντος αποτελεί πρόκληση, καθώς μεγάλες ροές δεδομένων υψηλής ταχύτητας παράγονται συνεχώς, ενώ δεν είναι εφικτό να αποθηκεύσουμε αυτά τα δεδομένα και να τα επεξεργαστούμε σε επόμενο χρόνο. Η συλλογιστική πάνω σε ροές δεδομένων περιλαμβάνει την ανίχνευση κρίσιμων καταστάσεων σε πραγματικό χρόνο, εφαρμόζοντας τεχνικές συλλογιστικής πάνω σε μεγάλους όγκους από σταδιακά παραγόμενα, συμβολικά δεδομένα, που συνήθως ονομάζονται «γεγονότα». Προκειμένου να υποστηρίζει σύγχρονες εφαρμογές, ένα σύστημα συλλογιστικής σε ροές δεδομένων πρέπει να χρησιμοποιεί αποτελεσματικούς αλγόριθμους συλλογιστικής που είναι προσαρμοσμένοι σε περιβάλλοντα ροών δεδομένων, μια εξαιρετικά εκφραστική γλώσσα προδιαγραφών προτύπων με τυπική και δηλωτική σημασιολογία, καθώς και τεχνικές για τη διαχείριση του θορύβου και της αβεβαιότητας. Αυτή η διατριβή παρακινείται από την απουσία ενός συστήματος που να διαθέτει όλα τα προαναφερθέντα χαρακτηριστικά και στοχεύει στην επέκταση σύγχρονων υπολογιστικών συστημάτων προκειμένου να χειρίζονται αποτελεσματικά εφαρμογές συλλογιστικής σε ροές δεδομένων. Εστιάζουμε σε συστήματα που χρησιμοποιούν τον Λογισμό Γεγονότων, έναν φορμαλισμό βασισμένο στο λογικό προγραμματισμό για την αναπαράσταση γεγονότων και το συμπερασμό των συνεπειών τους με την πάροδο του χρόνου. Ο Λογισμός Γεγονότων παρουσιάζει μια τυπική, δηλωτική σημασιολογία, ενώ υποστηρίζει στιγμιαία και διαρκή γεγονότα, ιεραρχικές και σχεσιακές προδιαγραφές γεγονότων, τη χρονική ανθεκτικότητα των συνεπειών των γεγονότων, καθώς και βοηθητικές βάσεις γνώσης που προδιαγράφουν τον τομέα εφαρμογής. Το RTEC είναι ένα σύστημα που βασίζεται στον Λογισμό Γεγονότων και περιλαμβάνει πολλές τεχνικές βελτιστοποίησης για τη συλλογιστική σε ροές δεδομένων. Προτείνουμε τρεις επεκτάσεις του RTEC, δηλαδή τα συστήματα $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ και RTEC_{A}, τα οποία υποστηρίζουν, αντιστοίχως, χρονικές προδιαγραφές με κυκλικές εξαρτήσεις, γεγονότα με μελλοντικές συνέπειες και καταστάσεις που ορίζονται μέσω των σχέσεων της άλγεβρας διαστημάτων του Allen. Επιπλέον, μελετάμε το PIEC, δηλαδή ένα σύστημα βασισμένο στο Λογισμό Γεγονότων για πιθανοτική αναγνώριση γεγονότων, και προτείνουμε το σύστημα oPIEC, δηλαδή μια επέκταση του PIEC για συλλογιστική πάνω σε θορυβώδεις ροές δεδομενών. Για να υποστηρίξουμε μεγάλες ροές δεδομένων, προτείνουμε δύο εκδόσεις του oPIEC με περιορισμένη μνήμη. Όλα τα συστήματα που αναπτύξαμε έχουν τυπική σημασιολογία και διαθέτουν βελτιστοποιημένους αλγόριθμους συλλογιστικής σε ροές δεδομένων που έχουν αξιολογηθεί τόσο θεωρητικά όσο και εμπειρικά, χρησιμοποιώντας μεγάλες, πραγματικές και συνθετικές ροές δεδομένων, σε πειράματα που περιλαμβάνουν συγκρίσεις με σύγχρονα συστήματα. Η ανάλυσή μας φανερώνει πως τα συστήματα

$\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ και RTEC_{A} είναι κατάλληλα για συλλογιστική πάνω σε ροές δεδομένων με πολύπλοκες χρονικές προδιαγραφές, ενώ οι εκδόσεις του oPIEC επιτυγχάνουν υψηλή ακρίβεια πρόβλεψης και ταχύτητα συλλογισμού όταν λειτουργούν πάνω σε θορυβώδεις ροές δεδομένων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αναπαράσταση Γνώσης και Συλλογιστική

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: συλλογιστική πάνω σε ροές δεδομένων, Λογισμός Γεγονότων, λογικός προγραμματισμός, πιθανοτικός λογικός προγραμματισμός, ταίριασμα χρονικών μοτίβων, αναγνώριση σύνθετων γεγονότων

ACKNOWLEDGEMENTS

Completing a PhD thesis may be a long and arduous task. At times, the final goal seems to be miles ahead, creating anxiety and desperation in one's mind. In the case of this thesis, I was very lucky to have the constant support of my supervisors, colleagues, friends and family. Their positive attitude, words of encouragement and irreplaceable company made the task of completing this thesis not only doable, but also an exciting journey that made up some of the most fruitful and cherished years of my life.

First and foremost, I would like to wholeheartedly thank my PhD supervisors. Dr. Alexander Artikis is the head of the complex event recognition group at NCSR "Demokritos", an Assistant Professor at the University of Piraeus and the main supervisor of this thesis. Alexander has in-depth knowledge on the field of artificial intelligent and a unique talent of apprehending and distinguishing the key aspects of an idea, based on my (sometimes messy and incomplete) descriptions, in order to provide vital insights for the progress of my work. Without his feedback and expertise, it would be impossible to complete this thesis. Moreover, Alexander was extremely helpful with the preparation of the research papers and presentations related to this thesis. His comments were always clear and instructive, instilling a well-balanced amount of feedback, which, I believe, enhanced my creativity and helped me improve. Alexander was my 'academic teacher', for which I am grateful.

Dr. Giorgos Paliouras is the head of the AI Lab SKEL at NCSR "Demokritos". Giorgos and I had valuable discussions regarding the research directions of this thesis, in which Giorgos's comprehensive knowledge regarding statistical-relational AI and composite event recognition, and his willingness to share his knowledge, were crucial. Giorgos was always supportive at every step of my research, taking the time to read my paper drafts, while being patient and considerate with my mistakes and providing acute feedback, which proved indispensable for the progress of my work.

I would also like to thank Dr. Panagiotis Stamatopoulos (Associate Professor at the National and Kapodistrian University of Athens) for making this PhD thesis possible. While working on my thesis, I had to opportunity to participate as an assistant in the Logic Programming course given by Panagiotis Stamatopoulos, a rewarding experience that was beneficial for my progress, as logic programming is at the core of my thesis. Panagiotis was always eager to help me in my role as a course assistant and very supportive regarding all sorts of issues that came up as I progressed with my work.

I am also grateful to Professors Dimitrios Gunopulos, Manolis Koubarakis and Panagiotis Rondogiannis of the National and Kapodistrian University of Athens and Professor Kostas Stathis of Royal Holloway University of London for agreeing to take part in the seven-member examination committee of this dissertation.

PhD theses typically require several years of work. Everyday life in the workplace naturally affects the progress of a thesis, enhancing or hindering creativity and productivity, depending on the quality of the workplace. I was very lucky to be part of a positive and encouraging working environment. My colleagues and I had interesting discussions concerning research ideas, as well as conversations that took our minds off work, which was vital for a stress-free and creative PhD experience. I am very thankful to Manolis Pitsikalis, Elisavet Keramioti, Vasilis Stavropoulos, Manos Ntoulias, Nefeli Prokopaki-Kostopoulou, Evangelia Santorinaiou, Alexandros Ntogramatzis, Alex Troupiotis, Giannis Fikioris, Alexia Atsidakou and Leonidas Tsekouras. They were some of my colleagues and friends whose company was irreplaceable.

My friends have been very supportive and encouraging in the last few years, always taking the time to ask me how my dissertation is going. At times of paper rejections or research complications, the kind words of my friends helped me get back on my feet and almost single-handedly cured my anxiety and glass-half-empty attitude. Having good friends is really an indispensable part of life. This is one of the main lessons I grasped in the years of working towards this thesis.

Last, but certainly not least, I would like to thank, from the bottom of my heart, my parents, Έφη and Γιάννης. My parents have been encouraging me to pursue my dreams ever since my early school years, working hard and making sacrifices their whole lives to produce an environment where this was even possible. Their warm, comforting words were essential to keep me going, even when making progress seemed like a tremendous task in my mind. For this reason, this dissertation is dedicated to my parents.

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Συνοπτική περιγραφή του διδακτορικού (5–8 σελίδες).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Stream reasoning involves temporal pattern matching over large volumes of incrementally-arriving *events*, i.e., symbolic data that are associated with a time of occurrence, which may be produced by applying a computational process on raw data [Fragkoulis et al., 2024]. In the maritime domain, e.g., a collection of position and velocity signals emitted from a vessel may be translated into a symbolic event indicating a change in the speed or the heading of the vessel by means of 'critical point annotation' [Fikioris et al., 2023]. The goal of temporal pattern matching over streams of events is the detection of *situations of interest*, i.e., spatio-temporal combinations of events, often incorporating background knowledge [Dell'Aglio et al., 2017]. For instance, in the maritime domain, 'trawling', i.e., a type of fishing activity which involves several consecutive turns, can be expressed as a sequence of 'change in heading' events.

A stream reasoning system is equipped with a *temporal specification language*, i.e., a formalism defining a set of operators between events. In order to define a trawling activity in the maritime domain, e.g., a temporal specification language may employ a 'sequence' operator and define '$trawling(V)$', i.e., the situation indicating that a vessel V is trawling, as a sequence of '$change\_in\_heading(V)$' events, where the temporal distance between two consecutive $change\_in\_heading(V)$ events does not exceed 10 minutes [Pitsikalis et al., 2019]. Such patterns may be given by domain experts [Skarlatidis et al., 2015a] or learned from data [Katzouris and Artikis, 2020; Michelioudakis et al., 2024].

Numerous frameworks employ logic-based languages [Artikis et al., 2012b; Giatrakos et al., 2020]. Logic-based formalisms offer several advantages, as they exhibit a formal and declarative semantics, allow for a compact and intuitive representation of spatio-temporal patterns, as the user need only specify *what* is to be computed, and not necessarily *how* it is to be computed, and incorporate background knowledge. Thus, logic-based formalisms may serve as the basis of a temporal specification language for stream reasoning. The temporal pattern matching task is assigned to an automated reasoner for the selected logical formalism, deriving situations of interest based on the provided declarative specifications.

Temporal pattern matching in a streaming setting introduces several challenges compared to the batch setting. The large, high-velocity and theoretically unbounded data streams that we find in contemporary applications cannot be maintained in memory, while it is not computationally feasible to reason over the entire history of the input data for each new event arrival. To address these issues, reasoning is often performed over windows, i.e., partial and bounded views of the stream [Verwiebe et al., 2023]. At each query time, the system reasons over the events that fall within the current window, ignoring events with earlier time-stamps, which are typically discarded from memory. To ensure correctness, the temporal extent of the window must be selected in such a way, so that the derivations of the system concerning situations inside the window are the same as they would have been if the entire history of events were available [Ronca et al., 2022].

Reasoning over large data streams requires highly efficient and optimised algorithms, ideally performing one pass over the input data. To avoid repeating earlier computations, stream reasoning systems often cache and re-use some of their previous derivations. Such caching techniques often benefit from compositional specifications that form large hierarchies of sub-patterns [White et al., 2007]. In this way, reasoning algorithms operate following the dependencies in a hierarchy, while caching intermediate results, paving the way for efficient reasoning [Chittaro and Montanari, 1996; Montali et al., 2013; Artikis et al., 2015].

Another issue that we encounter in stream reasoning is data uncertainty [Dell'Aglio et al., 2017]; erroneous or incomplete events may lead to the detection of incorrect instances of pattern satisfaction. Consider, e.g., the task of detecting composite human activities based on symbolic representations of video feeds. Gaps in sensor data are common in this scenario, as a result of object occlusion in videos, and may lead to the detection of incorrect activities [Singh and Vishwakarma, 2019]. Similarly, in the maritime domain, the malfunction of a signal transmitter may lead to a communication gap concerning the whereabouts of a vessel [Pitsikalis et al., 2019]. A catalogue of the sources of such uncertainty/noise may be found in [Alevizos et al., 2017]. In order to express noisy input events, the systems generating input events from raw data may associate a probability value with each generated event, serving as a confidence estimate [Kimmig et al., 2011; Bellodi et al., 2020]. A probabilistic reasoning framework may then reason over such probabilistic events in order to derive the probabilities of situations of interest at each point in time.

1.1 Motivation

Stream reasoning frameworks are often assessed in terms of the range of temporal specifications they support, the efficiency of their reasoning algorithms, as well as the ability to handle uncertainty [Dell'Aglio et al., 2017]. The potential for extending the range of temporal specifications supported by such frameworks, while maintaining their high reasoning efficiency, and the absence of frameworks handling noisy data streams motivate this thesis. We focus our attention on the Event Calculus, a logic programming formalism for representing events and reasoning about their effects over time [Kowalski and Sergot, 1986]. In order to compactly represent the effects of events, the Event Calculus employs the commonsense law of inertia, according to which a situation of interest is currently in effect, if it has been initiated by an event at some earlier time, and not terminated by another event in the meantime. The Event Calculus exhibits a formal, declarative semantics, while supporting non-monotonic reasoning, hierarchical specifications, i.e., situations of interest defined in terms of other situations, relational specifications, i.e., situations involving multiple entities of the domain, and background knowledge. Moreover, the Event Calculus has been employed in numerous settings, including mobility assistance [Bromuri et al., 2010], reactive and proactive health monitoring [Chaudet, 2006; Kafali et al., 2017], e-commerce with negotiation agents [Alrayes et al., 2018; Hopkins et al., 2019], simulations with cognitive agents [Shahid et al., 2021; Shahid et al., 2023], computational ethics [Berreby et al., 2018], biological feedback processes [Srinivasan et al., 2022], as well as the online detection of composite activities in shipping [Montali et al., 2013] and public space monitoring [Skarlatidis et al., 2015a].

We discuss three types of temporal specifications—cyclic dependencies, events with delayed effects and Allen's interval relations—and noisy data streams. These features are not supported by Event Calculus-based frameworks for reasoning over event streams, which serves as the main motivation for this thesis. Subsequently, we outline applications that require some of the aforementioned types of specifications and/or feature noisy data streams.

1.1.1 Stream Reasoning over Complex Temporal Specifications

The 'Run-Time Event Calculus' (RTEC) is a formal computational framework for stream reasoning [Artikis et al., 2015]. RTEC is a logic programming implementation of the Event Calculus. The language of RTEC has a formal, declarative semantics; temporal specifications in RTEC are locally stratified logic programs [Przymusinski, 1988]. RTEC extends the Event Calculus with optimisation techniques for handling data streams. To avoid redundant computations, RTEC processes hierarchical patterns using bottom-up reasoning and caching. Moreover, RTEC employs a sliding window, thus reasoning over a small part of the input data, while discarding events taking place before the window. RTEC has proven highly efficient in numerous applications, such as city transport management [Artikis et al., 2015], maritime situational awareness [Pitsikalis et al., 2019] and commercial fleet management [Tsilionis et al., 2022].

The language of RTEC, however, is not expressive enough to capture some types of temporal specifications that we find in contemporary applications.

The specifications of multi-agent e-commerce protocols, e.g., often include *cyclic dependencies*. For instance, 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. Stream reasoning frameworks do not typically support patterns with cyclic dependencies. RTEC, e.g., is restricted to acyclic knowledge bases [Artikis et al., 2015]. Moreover, frameworks that do support patterns with cyclic dependencies often evaluate these patterns in an inefficient manner. For instance, jREC [Montali et al., 2013] may perform several redundant computations when processing a pattern with cyclic dependencies.

Apart from immediate effects, events often have *delayed effects*. In multi-agent e-commerce protocols, e.g., establishing a contract obliges the consumer to pay the agreed price, and the merchant to deliver the goods, within a specified time [Chopra et al., 2020]. In simulations of biological systems, the activation of a gene may affect the concentration of a protein after a time delay [Srinivasan et al., 2022]. In the maritime domain, a trawling activity is said to end 10 minutes after a 'change in heading' event, provided that no other event of this type has taken place in the meantime [Pitsikalis et al., 2019]. Several formalisms have been proposed for handling events with delayed effects [Karlsson et al., 1998; Marín and Sartor, 1999; Miller and Shanahan, 2002; Hindriks and Riemsdijk, 2013; Demolombe, 2014; Chopra et al., 2020]. These approaches, however, cannot handle continuous queries over large, evolving data streams and complex temporal specifications that need to be computed with minimal latency. Moreover, the stream reasoning systems that we find in the literature do not support events with delayed effects. In RTEC, e.g., an event may have immediate effects, i.e., 'initiating'/'terminating' a situation of interest at the time of its occurrence, but not delayed effects.

Situations of interest are typically durative, and thus should be expressed using temporal intervals. Such a treatment allows the detection of ongoing phenomena, while circumventing the unintended semantics of using time-points for representing durative properties [Galton and Augusto, 2002; Paschke, 2005; White et al., 2007]. In order to produce more accurate patterns, temporal intervals may be combined using the relations of *Allen's interval algebra*, i.e., thirteen jointly exhaustive and pairwise disjoint relations among intervals [Allen, 1984]. Consider, e.g., the detection of a 'suspicious rendez-vous' of two vessels in the maritime domain, where one of the vessels turns off signal transmissions, while being close to the other vessel. This phenomenon can be expressed with the 'during' relation of Allen's algebra, while it cannot be captured by common interval operators, such as union and intersection. The use of Allen's algebra has proven quintessential for defining such phenomena and is supported by several frameworks [Brendel et al., 2011; Anicic et al., 2012; Körber et al., 2021; Awad et al., 2022]. The languages of these frameworks, however, do not allow for the explicit representation of time, complicating the specification of persistence and changes in the detected situations over time. In contrast, such specifications are innately supported in Event Calculus dialects, like RTEC. The language of RTEC, however, does not capture the relations of Allen's interval algebra.

1.1.2 Reasoning over Noisy Data Streams

Event Calculus-based frameworks that handle uncertainty do not support stream reasoning [Thon et al., 2011; Skarlatidis et al., 2015b; D'Asaro et al., 2017; McAreavey et al., 2017; Artikis et al., 2021]. The 'Probabilistic Interval-based Event Calculus' (PIEC) is a formal computational framework that processes events with attached probability values and deduces the (probabilities of the) maximal intervals of situations of interest by means of Event Calculus-style reasoning [Artikis et al., 2021]. It has been shown that interval-based recognition improves upon the predictive accuracy of point-based recognition in the presence of data uncertainty. PIEC requires the entire dataset of event probabilities in order to guarantee correct interval computation. This is not feasible in a streaming setting, where only a very small and bounded portion of the input can be stored in memory and processed at a later time. Moreover, for each new event arrival, PIEC reconstructs the output intervals from scratch, while each event commonly induces only minor or no changes to the previously computed intervals. This sort of re-computation leads to significant reasoning inefficiencies.

1.1.3 Applications

There are numerous real-world applications that require efficient reasoning over complex temporal specifications and large, high-velocity streams of (noisy) events. This is the main motivation for this thesis. We discuss applications from the fields of composite event recognition [Giatrakos et al., 2020], multi-agent systems [Wooldridge, 2009] and biological feedback processes [Thomas and d'Ari, 1990].

1.1.3.1 Composite Event Recognition

We outline two composite event recognition applications, i.e., maritime situational awareness and human activity recognition.

Maritime Situational Awareness. Maritime activities need to be monitored in real time, in order to detect dangerous, illegal and environmentally hazardous activities with minimal latency, thus ensuring safe shipping. Automatic Identification System (AIS) position signals are continuously being emitted by vessels, and contain information about their heading, speed and navigational status [Artikis and Zissis, 2021]. Streams of symbolic events can be produced based on AIS signals by means of trajectory simplification, generating 'critical' events to signify major changes in a vessel's behaviour, such as a stop, a turn or a gap in signal transmissions [Patroumpas et al., 2015; Fikioris et al., 2023], and spatial processing, annotating spatial relations between vessels and maritime areas, ports or other vessels [Santipantakis et al., 2018]. A computational framework may process such streams of symbolic events and detect composite maritime activities that require special attention, such as ship-to-ship transfers of goods in the open sea, search and rescue operations, etc., by means of temporal pattern matching. Composite maritime activities may be defined in terms of cyclic dependencies (e.g., 'fishing trip'), events with delayed effects (e.g., 'trawling') and Allen relations (e.g., 'suspicious rendez-vous'). Moreover, AIS signals may include gaps and erroneous indications, leading to noisy input events [Bereta et al., 2021].

Human Activity Recognition. The timely detection of composite human activities in public spaces may prevent theft and physical harm, paving the way for a secure environment. In this application, the input streams are symbolic representations of video feeds, containing events concerning simple activities performed by one person and identified on individual video frames, such as 'walking' and 'running', as well as the coordinates and the orientation of the tracked people. The temporal pattern matching task is to detect situations of interest expressing composite human activities, i.e., 'meeting', 'moving together' and 'fighting'. Symbolic representations of video feeds often include erroneous indications, because, e.g., of temporary failures in the object tracker or object occlusion, leading to noisy event streams [Singh and Vishwakarma, 2019]. Therefore, human activity recognition requires a framework that handles noisy event streams.

1.1.3.2 Multi-Agent Systems

We describe two multi-agent systems protocols, formalising a voting procedure and the e-commerce protocol NetBill.

Voting. Our voting protocol 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 [Pitt et al., 2006]. Agents occupy the roles of proposer, seconder, voter and/or chair, and deliberate over various motions over time.

NetBill is an e-commerce protocol that 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 [Yolum and Singh, 2002; 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 [Kowalski and Sergot, 1985; Jones and Sergot, 1996; Sergot, 2001; Artikis and Pitt, 2008]. Moreover, sanctions deal with the performance of forbidden actions and non-compliance with obligations. The temporal pattern matching task is to detect the normative positions and the suspension periods of agents over time by reasoning over streams of agent actions. The specifications of voting and NetBill include cyclic dependencies and events with delayed effects.

1.1.3.3 Biological Feedback Processes

Biological processes exhibiting multi-stationary or oscillatory behaviour can often be specified via *feedback loops*. Biological feedback loops model control mechanisms expressing that a change in the value of some biological variable, e.g., the concentration of an amino acid, results in a (re-)adjustment of its value after some time [Srinivasan et al., 2022]. In homeostasis, e.g., the production of an amino acid is regulated based on its concentration. If its concentration is initially high, but decreasing, then its production is initially inhibited and later activated, once its concentration levels fall below a threshold. As a result, the concentration of the amino acid oscillates around a specified value. The Generalised Kinetic Logic models feedback loops in terms of asynchronous automata, supporting state changes with time delays [Thomas, 1991]. Figure 1.1, e.g., depicts two biological feedback loops in the Generalised Kinetic Logic, modelling immune system response in vertebrates and the invasion of a phage in a bacterial cell. We describe these two feedback loops, following [Srinivasan et al., 2022].

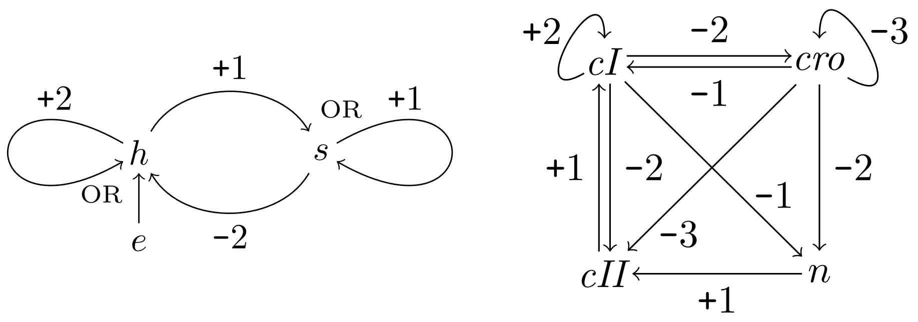


Figure 1.1: The feedback loops of immune response (left) and phage infection (right), after [Srinivasan et al., 2022]. An edge from ‘x’ to ‘y’ with label ‘+n’ (resp. ‘-n’), where n is a positive integer, expresses a positive (negative) interaction, i.e., the value of y increases (decreases) by 1 at a specified time after the value of x becomes greater or equal to n.

Immune Response. The role of the immune system is to produce antibodies in order to mitigate invaders called antigens. Immune response mainly involves two categories of cells. B-lymphocytes generate antibodies, while T-lymphocytes regulate their production. T-lymphocytes consist of h-cells and s-cells, each specialised to the production of antibodies that deal with different parts of the antigen. Figure 1.1 (left) depicts a simplified model of the feedback loop for immune response. Variables h and s denote the concentrations of h-cells and s-cells. h and s have three possible values, while e is a Boolean variable denoting the presence of an antigen. According to the edges connecting h and s, if $h \geq 1$, then s increases by 1, while $s \geq 2$ results in the decrease of h by 1. These value changes take place after some specified time delays d_s^+ and d_h^-, respectively. Therefore, an increase in the concentration of h-cells induces the secretion of s-cells, which in turn suppress the production of h-cells.

Phage Infection. The invasion of a phage (virus) into a host bacterium may result in the integration of the phage’s DNA into the host’s DNA, leading to immunity for the host. The combined DNA produces a repressor protein cI that prevents the transcription of two groups of viral genes, called N and Cro, which would otherwise produce proteins n and cro, resulting in the death of the host. Moreover, the combined DNA produces protein cII, which promotes the activation of repressor gene cI. Figure 1.1 (right) presents a simplified model of the feedback loop for phage infection. Variables cI, n, cro and cII denote the concentrations of the corresponding proteins. According to the edge from cII to cI, e.g., variable cI increases by 1 at a specified time after the value of variable cII becomes greater or equal to 1.

We study simulations of the biological processes for immune response and phage infection. In both cases, the task is to compute the values of all variables in the corresponding feedback loop as a simulation progresses. Both biological processes demand reasoning over cyclic dependencies and events with delayed effects.

1.2 Contributions

The contributions of this thesis can be divided into two main parts. In the first part, we extend RTEC to three orthogonal directions and propose three systems: $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A}, supporting, respectively, temporal specifications with cyclic dependencies, events with delayed effects and Allen relations. In the second part, we propose oPIEC, a novel framework that handles noisy data streams.

1.2.1 Stream Reasoning over Complex Temporal Specifications

Below, we highlight the key features of $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A}.

$\text{RTEC}_{\circ}$ is an extension of RTEC for reasoning over temporal patterns with cyclic dependencies. $\text{RTEC}_{\circ}$ employs a set of novel reasoning algorithms, featuring an incremental caching technique, in order to avoid redundant computations.

$\text{RTEC}^{\rightarrow}$ supports reasoning over streams of events with delayed effects. $\text{RTEC}^{\rightarrow}$ extends the language of RTEC with two new types of language constructs. The former type designates the domain properties that are subject to change based on the delayed effects of a specified set of events, as well as the temporal extent of the time delay of these effects. The latter type of construct specifies the delayed effects that may be postponed by subsequent event occurrences. At compile-time, $\text{RTEC}^{\rightarrow}$ establishes the optimal processing order of a given temporal specification. At run-time, $\text{RTEC}^{\rightarrow}$ reasons over events with delayed effects incrementally, while caching intermediate computations, thus avoiding re-computations.

RTEC_{A} extends the language of RTEC with a novel language construct, allowing for Allen relations in temporal patterns. RTEC_{A} computes Allen relation satisfaction with one pass over the input intervals, while ensuring correct Allen relation computation with windowing.

For each of the proposed frameworks, i.e., $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A}, we prove that the programs supported by its temporal specification language belong in the class of locally stratified logic programs. Moreover, we prove the correctness of the novel reasoning algorithms of each framework, and outline their worst-case time complexity, demonstrating that their costs are bound by a small part of the stream with constant size.

We conducted extensive empirical analyses using large, synthetic and real data streams for composite event recognition, multi-agent systems and biological feedback processes. Our evaluation included empirical comparisons of $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A} with state-of-the-art stream reasoning frameworks. Our results demonstrate that $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A} are highly efficient, verifying our complexity analyses, and outperform the state of the art by orders of magnitude. $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A} are open-source frameworks; the code, the domain specifications and the data streams we used for our experiments are publicly available[1]. Thus, our experiments are *reproducible*.

[1] *https://github.com/aartikis/rtec*

1.2.2 Reasoning over Noisy Data Streams

We propose **oPIEC**, i.e., an extension of the probabilistic event recognition framework PIEC that is designed for reasoning over data streams, as opposed to the batch processing of PIEC. First, we propose a technique for identifying the minimal set of time-points that need to be cached, in a memory structure called 'support set', in order to guarantee correct interval computation for durative phenomena. Second, we present a formal analysis of oPIEC, where we prove its correctness, i.e., we show that oPIEC computes the same intervals as PIEC, and then present its worst-case time complexity, which is significantly lower than that of PIEC. Third, we propose two ways to further reduce the cached time-points, supporting highly efficient reasoning, while at the same time minimising the effects on correctness. Our first solution is oPIEC^{b}, i.e., an extension of oPIEC with a bounded support set and an algorithm to decide which elements of the support set should deleted, in order to make room for new ones. Our second solution is oPIEC^{bd}, i.e., another bounded-memory extension of oPIEC that leverages interval duration statistics to resolve memory conflicts. In cases where the provided statistics are not sufficient to make enough room in the support set, oPIEC^{bd} invokes the memory maintenance algorithm of oPIEC^{b} to delete further elements. Fourth, we present complexity analyses for oPIEC^{b} and oPIEC^{bd}, demonstrating that their memory maintenance algorithms do not introduce any delays in reasoning. Fifth, we present a thorough empirical evaluation on a benchmark dataset for human activity recognition, as well as real data streams from the field of maritime situational awareness. Our experimental evaluation shows that oPIEC^{b} and oPIEC^{bd} may achieve comparable predictive accuracy to batch reasoning, avoiding the prohibitive cost of such reasoning. Moreover, our evaluation demonstrates that oPIEC^{bd} is the preferred option for stream reasoning, striking the best balance between predictive accuracy and computational efficiency. oPIEC^{b} and oPIEC^{bd} are open source frameworks[2], allowing for the reproducibility of our experiments.

1.2.3 Publications

This thesis is based on the following publications:

Journal Publications:

- Mantenoglou P., Pitsikalis M., Artikis A., *Reasoning over Streams of Events with Delayed Effects.*
  In *Transactions on Knowledge and Data Engineering (TKDE)*, **under review**.
- Mantenoglou P., Artikis A., Paliouras G., *Online Event Recognition over Noisy Data Streams.*
  In *International Journal of Approximate Reasoning (IJAR)*, 161, 2023.
  DOI: *https://doi.org/10.1016/j.ijar.2023.108993*

Conference Publications:

[2] *https://github.com/periklismant/opiec*

- Mantenoglou P., Kelesis D., Artikis A., *Complex Event Recognition with Allen Relations.*
  In *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 502–511, 2023.
  DOI: *https://doi.org/10.24963/kr.2023/49*
- Mantenoglou P., Pitsikalis M., Artikis A., *Stream Reasoning with Cycles.*
  In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 533–553, 2022.
  DOI: *https://doi.org/10.24963/kr.2022/56*
- Mantenoglou P., Artikis A., Paliouras G., *Online Probabilistic Interval-based Event Calculus.*
  In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI)*, pp. 2624–2631, 2020.
  DOI: *https://doi.org/10.3233/FAIA200399*

Peripheral Publication:

- Andrienko N., Andrienko G., Artikis A., Mantenoglou P., Rinzivillo S., *Human-in-the-Loop: Visual Analytics for Building Models Recognising Behavioural Patterns in Time Series.*
  In *IEEE Computer Graphics and Applications (CG&A)*, pp. 1–15, 2024.

1.3 Thesis Outline

This thesis is structured as follows. In Chapter 2, we outline the background material that is necessary to describe $\mathrm{RTEC}_{\circ}$, $\mathrm{RTEC}^{\rightarrow}$ and $\mathrm{RTEC}_{\mathrm{A}}$. Particularly, we discuss logic programming, the Event Calculus, the computational framework RTEC and Allen's interval algebra. Afterwards, we provide a literature review on frameworks that are relevant to $\mathrm{RTEC}_{\circ}$, $\mathrm{RTEC}^{\rightarrow}$ and $\mathrm{RTEC}_{\mathrm{A}}$. In Chapter 3, we propose $\mathrm{RTEC}_{\circ}$, $\mathrm{RTEC}^{\rightarrow}$ and $\mathrm{RTEC}_{\mathrm{A}}$, while, in Chapter 4, we present an empirical evaluation of these systems, demonstrating their benefits compared to related frameworks. Next, we move to the second part of the thesis, where we introduce oPIEC, i.e., a computational framework handling noisy data streams. In Chapter 5, we provide the necessary background material for oPIEC, which comprises probabilistic logic programming, Prob-EC, i.e., a probabilistic extension of the Event Calculus, and PIEC, i.e., an interval-based extension of Prob-EC. Moreover, we provide a literature review on frameworks handling uncertainty. Chapter 6 presents oPIEC, as well as two bounded-memory versions of oPIEC, i.e., oPIEC^{b} and oPIEC^{bd}. Afterwards, in Chapter 7, we provide an experimental evaluation of oPIEC^{b} and oPIEC^{bd}, including a comparison with related frameworks. Chapter 8 summarises our work and proposes further research directions.

2. BACKGROUND: STREAM REASONING

We provide the necessary background material for the specification of RTEC$_\circ$, RTEC$^\rightarrow$ and RTEC$_\mathsf{A}$; our work on reasoning over noisy data streams, i.e., oPIEC, is presented in Chapters 5–7. First, we describe some key concepts of logic programming. We focus on the semantics of logic programming and, in particular, establish an intended meaning for locally stratified logic programs, i.e., a class of programs whose variable-free incarnations do not include cyclic dependencies with negation. Our proposed frameworks restrict attention to locally stratified logic programs. Second, we provide an overview of the Event Calculus, i.e., a formalism that extends logic programming with a representation of time, events, and the effects of events over time. The Event Calculus serves as the basis of all of our proposed frameworks, i.e., RTEC$_\circ$, RTEC$^\rightarrow$, RTEC$_\mathsf{A}$ and oPIEC, paving the way for a unified Event Calculus-based framework for temporal pattern matching over event streams. Third, we discuss previous work on RTEC, highlighting the syntax and the key reasoning algorithms of the system. Fourth, we describe the relations of Allen's interval algebra; we introduce these relations in the language of RTEC in Section 3.4. Fifth, we present a literature review on stream reasoning frameworks.

2.1 Logic Programming Concepts

The basis of a stream reasoning framework needs to be a formal and declarative formalism. Without a declarative semantics, it may become difficult to trace unintended behaviours of the framework, complicating error detection. Moreover, using a declarative formalism, the user need only specify the situations of interest of the domain by describing the conditions under which these situations arise, without the need to write an algorithm for computing each one of these situations. Logic programming is a declarative formalism with a formal semantics, allowing for a compact and intuitive representation of the patterns of a domain, while seamlessly incorporating background domain knowledge. Thus, logic programming can serve as the foundation of a specification language for temporal pattern matching.

Logic programming is a subset of first-order logic where well-formed formulas are in clausal form. We summarise the necessary aspects of logic programming for this thesis, following [Lloyd, 1987].

The alphabet consists of variables, constants, predicate symbols and connectives, i.e., ',' for conjunction, '$\leftarrow$' for implication, and 'not' for negation-by-failure [Clark, 1977]. Variables start with an upper-case letter, while constants and predicate symbols start with a lower-case letter. A *term* is a variable or a constant. An *atom* is an expression of the form '$p(t_1, \dots, t_n)$', where p is a predicate symbol with arity n and $t_1, \dots, t_n$ are terms. An atom is called *ground* if it contains no variables. A *literal* is an atom or a negated atom, i.e., an expression of the form 'not $p(t_1, \dots, t_n)$'. A *program* is a set of clauses, i.e., formulas following the syntax '$a \leftarrow b_1, \dots, b_n$', where a is an atom and $b_1, \dots, b_n$ are literals. We call a the head of the clause, while $b_1, \ldots, b_n$ comprise the body of the clause. Due to the syntactical form of clauses, we refer to them as *rules* throughout. All variables that appear in the body of a rule are assumed to be universally quantified. A rule with an empty body is called a *fact*. A *definite rule* is a rule whose body literals, if any, are all positive, i.e., they are all atoms. A program is called *definite* if it contains only definite rules, and *normal* otherwise. A *ground rule* contains only ground atoms, and a *ground program* contains only ground rules.

In logic programming, a program constitutes a declarative description of what is true in the world. A semantics is used to map each possible program to its intended meaning. Given a program P, an *interpretation* of P consists of (i) a domain D, (ii) a mapping of each constant in P to an element of D, and (iii) a mapping of each n-ary predicate symbol in P to a relation on D^n. A *model* of a program P is an interpretation of P where all rules in P are true statements. A ground atom a is a *logical consequence* of a program P if, for every interpretation I, I is a model of P implies that $a \in I$. Given a program P, the *Herbrand universe* U_P of P is the set of all constants that appear in P, and the *Herbrand base* B_P of P is the set of all ground atoms $p(t_1, \ldots, t_n)$, where p is a predicate symbol of P with arity n and $t_1, \ldots, t_n \in U_P$. An *Herbrand interpretation* of a program P is an interpretation of P such that (i) the domain is the Herbrand universe U_P of P and (ii) all constants in P are mapped to themselves in U_P. Since, for Herbrand interpretations, the mapping of constants is fixed, we often identify an Herbrand interpretation with the subset of the Herbrand base that contains the ground atoms that are true with respect to the interpretation. An *Herbrand model* of a program P is an Herbrand interpretation of P that is a model of P. The role of a semantics is to identify the model of the program that expresses its *intended interpretation*, i.e., the meaning of the program.

Intuitively, the intended interpretation of a program should consist of the ground atoms that are logical consequences of the program, and no other ground atom. For a definite program P, i.e., a program without negation, the logical consequences of P can be specified without ambiguity by its model M_P, which is the intersection of all Herbrand models of P (see Theorem 6.2 in [Lloyd, 1987]). M_P is the least Herbrand model of P, i.e., for every Herbrand model $M \neq M_P$ of P, it holds that $M_P \subset M$. M_P is the intended interpretation of P; the other models of P contain at least one ground atom that is not a logical consequence of P, and thus do not capture the intended meaning of P.

Unfortunately, the above result cannot be generalised to the broader class of normal logic programs, i.e., programs with negation, because these programs may have multiple minimal Herbrand models. The problem of identifying the intended interpretation of a normal logic program has been studied thoroughly, and many semantics have been proposed [Apt and Bol, 1994]; notably, the stable model semantics [Gelfond and Lifschitz, 1988] and the well-founded semantics [van Gelder et al., 1991]. For some normal logic programs, there is a natural choice of a model that should be the intended interpretation of the program, in the sense that the model is compliant with the 'closed world assumption' (CWA) [Reiter, 1977], i.e., an inference rule stating that 'not a' is true if a cannot be finitely proven. Consider, e.g., a program P that contains the fact 'b' and the rule '$a \leftarrow b,$ not c'. P has two models: $M_1 = \{b, a\}$ and $M_2 = \{b, c\}$. While both M_1 and M_2 are minimal Herbrand models of P, only model M_1 captures the intended meaning of P. The use of negation-by-failure imposes a type of default preference among the negated premise and the consequent of rule '$a \leftarrow b,$ not c', according to which c has a higher 'priority for minimisation' than a. This type of priority is related to the CWA, i.e., we apply the CWA rule to c first, deducing that it is false on the grounds that c cannot be proven based on the rules of the program, and only then try to prove a. We define a *priority relation* $<_m$ on the elements of the Herbrand base of a program, and the auxiliary relation $\leq_m$, as follows:

Definition 1 (Priority Relation (after [Przymusinski, 1988])): Given a program P, the relations $<_m$ and $\leq_m$ are defined as follows:

- $a <_m c$, if c is a negative body literal and a is the head of a ground instance of a rule in P.
- $a \leq_m b$, if b is a positive body literal and a is the head of a ground instance of a rule in P.
- if $a \leq_m b$ and $b \leq_m c$, then $a \leq_m c$.
- if $a <_m b$ and $b \leq_m c$, then $a <_m c$.
- if $a \leq_m b$ and $b <_m c$, then $a <_m c$.
- if $a <_m b$, then $a \leq_m b$.
- nothing else satisfies $<_m$ or $\leq_m$. ■

Given two models M and N of a program P, it is possible to determine which model is *preferable* based on the relative priorities of the ground atoms they contain. If there is no model of P that is preferable to some model N of P, then N is a *perfect model* of P.

Definition 2 (Preference Relation and Perfect Models (after [Przymusinski, 1988])): Suppose that M and N are two distinct models of a program P. We say that N is preferable to M, i.e., $N \ll M$, if, for every ground atom a in $N \setminus M$, there is a ground atom b in $M \setminus N$, such that $a <_m b$. We say that a model N of P is perfect if there are no models of P that are preferable to N. We call $\ll$ the preference relation on models. ■

For the program with fact 'b' and rule '$a \leftarrow b,$not c', e.g., whose models are $M_1 = \{b, a\}$ and $M_2 = \{b, c\}$, we have $a <_m c$, and thus $M_1 \ll M_2$. Therefore, M_1 is a perfect model of the program, while M_2 is not a perfect model of the program.

Unfortunately, an arbitrary normal logic program may not admit a perfect model. For example, a program P containing the rules '$a \leftarrow$ not b' and '$b \leftarrow$ not a' has two models $M_a = \{a\}$ and $M_b = \{b\}$. It holds that $b <_m a$ and $a <_m b$, and thus we have $M_a \ll M_b$ and $M_b \ll M_a$. Therefore, none of the models M_a and M_b is perfect. This type of semantic ambiguities arise in programs that contain cyclic dependencies with negation. In order to avoid these issues, we restrict our attention to programs that are *locally stratified*.

Definition 3 (Locally Stratified Logic Program (after [Przymusinski, 1988])): A normal logic program P with a finite Herbrand base B_P is called locally stratified if it is possible to decompose B_P into disjoint sets (strata) $P_0, \ldots, P_n$, so that, for every ground instance of a rule '$a \leftarrow b_1, \ldots, b_m,$not $c_1, \ldots,$not c_k' in P, we have that, if $a \in P_i$, where $1 \leq i \leq n$, then:

- $b_1, \ldots, b_m \in P_j$, where $j \leq i$, and
- $c_1, \ldots, c_k \in P_j$, where $j < i$.

A decomposition $P_0, \ldots, P_n$ of P that satisfies the above conditions is called a local stratification of P. ■

A locally stratified logic program has a unique perfect model (see Theorem 4 in [Przymusinski, 1988]). In what follows, the intended interpretation of the logic programs we present coincides with their unique perfect model.

Logic programming is a natural candidate formalism for serving as the basis of a specification language for temporal pattern matching, due to its formal and declarative semantics and the ability to express both temporal and atemporal knowledge [Chesani et al., 2009; Paschke and Kozlenkov, 2009; Falcionelli et al., 2019; Baumgartner, 2021b]. In the following section, we present the Event Calculus, i.e., a logic programming formalism for representing event occurrences over time and reasoning about the changes in domain properties that are induced by these events.

2.2 The Event Calculus

Temporal pattern matching over data streams requires a formalism that captures situations of interest that evolve over time; logic programming is not sufficient for expressing such temporal specifications. Consider, e.g., the domain of human activity recognition and the task of detecting composite human activities, where we may need to represent the event that a person p_1 is walking. In logic programming, we can represent this event as a fact: '$walking(p_1)$', where $walking$ is a predicate and p_1 is a constant corresponding to some person p_1. $walking(p_1)$, however, does not contain any information about the temporal aspects of the activity, such as its time of occurrence. This type of information is necessary for specifying composite human activities. For instance, we may want to infer that two people, p_1 and p_2, are moving together when both of them are walking at the same time. The facts $walking(p_1)$ and $walking(p_2)$ are not sufficient to prove that p_1 and p_2 are moving together because these facts do not express the time frames during which each person is walking.

We employ the Event Calculus, i.e., a logic programming formalism with a time sort and a set of predicates for representing events and their effects on domain properties [Kowalski and Sergot, 1986]. Since the inception of the Event Calculus, many dialects have been put forward, including formulations in (variants of) first-order logic and as logic programs [Shanahan, 1999; Cervesato et al., 2000; Miller and Shanahan, 2002; Mueller, 2008; Mueller, 2015]. Event Calculus dialects are many-sorted, and include sorts for representing time, events and 'fluents', i.e., properties that may have different values at different points in time. Event occurrences may change the values of fluents. Thus, fluent values reflect the effects of events on the world over time. Given a fluent F, the term $F = V$ denotes that F has value V. Boolean fluents are a special case in which the possible values are true and false.

Table 2.1: Main predicates of the Event Calculus.

Predicate	**Meaning**
happensAt($E,\ T$)	Event E occurs at time T
initiatedAt($F=V,\ T$)	At time T, a period of time during which fluent F has value V is initiated
terminatedAt($F=V,\ T$)	At time T, a period of time during which fluent F has value V is terminated
initially($F=V$)	The value of fluent F is V at time 0
holdsAt($F=V,\ T$)	The value of fluent F is V at time T

We describe a logic programming implementation of the Event Calculus where time is linear, consisting of non-negative integer time-points [Artikis et al., 2010]. happensAt($E,\ T$) denotes that an instantaneous event E occurs at time-point T. initiatedAt($F=V,\ T$) (resp. terminatedAt($F=V, T$)) denotes that a time period during which a fluent F has the value V continuously is initiated (terminated) at time-point T. initially($F=V$) expresses that fluent F has the value V at time-point 0, while holdsAt($F=V,\ T$) states that fluent F has value V at time-point T. Table 2.1 summarises the main predicates of this Event Calculus dialect.

A key feature of the Event Calculus is the built-in representation of the common-sense law of inertia, according to which $F=V$ holds at a time-point, if $F=V$ has been 'initiated' by an event at some earlier time-point, and not 'terminated' by another event in the meantime. Consider the following *domain-independent* axiomatisation:

$$\begin{aligned}&\text{holdsAt}(F=V,\ T) \leftarrow\\ &\quad \text{initially}(F=V),\\ &\quad \text{not broken}(F=V,\ 0,\ T).\end{aligned} \tag{2.1}$$

$$\begin{aligned}&\text{holdsAt}(F=V,\ T) \leftarrow\\ &\quad \text{initiatedAt}(F=V,\ T_s),\ T_s < T,\\ &\quad \text{not broken}(F=V,\ T_s,\ T).\end{aligned} \tag{2.2}$$

$$\begin{aligned}&\text{broken}(F=V,\ T_s,\ T) \leftarrow\\ &\quad \text{terminatedAt}(F=V,\ T_f),\\ &\quad T_s < T_f < T.\end{aligned} \tag{2.3}$$

$$\begin{aligned}&\text{broken}(F=V,\ T_s,\ T) \leftarrow\\ &\quad \text{initiatedAt}(F=V',\ T_f),\ V \neq V',\\ &\quad T_s < T_f < T.\end{aligned} \tag{2.4}$$

broken($F=V,\ T_s,\ T$) is an auxiliary predicate that checks whether $F=V$ is terminated, or F is initiated with a value other than V, between time-points T_s and T. F, V, T_s and T are ground at the time when broken is called. Rules (2.1) and (2.2) imply that $F=V$ holds at some time-point T if it held initially, i.e., at time-point 0, or it has been initiated by an event at a time-point before T, and has not been 'broken' in the meantime. $F = V$ is broken between time-points T_s and T if, at an intermediate time-point T_f, $F = V$ is terminated (rule (2.3)) or $F = V'$ is initiated, for some $V' \neq V$ (rule (2.4)).

Rule 2.4 suggests that a fluent cannot have more than one value at any time; initiatedAt($F = V,\ T$) implies that $F = V'$ is broken at T, for all $V' \neq V$. Moreover, a fluent may not have a value at some time-point(s). For instance, it is not the same to initiate $F =$ true and terminate $F =$ false; the former implies, but is not implied by, the latter.

initiatedAt($F = V,\ T$) and terminatedAt($F = V,\ T$) are defined by means of *domain-dependent* rules. Consider the example below:

Example 1 (Human Activity Recognition): Suppose that we have a symbolic representation of video content for human activity recognition. The representation includes simple human activities, e.g., walking or being 'active', i.e., making non-abrupt body movements in the same position, as well as position and orientation information for the tracked people. We may employ the fluent-value pair (FVP) '$moving(P_1, P_2) =$ true' to denote that two people P_1 and P_2 are said to be moving together, and specify the conditions under which $moving(P_1, P_2) =$ true is initiated or terminated using the following rules:

$$\begin{aligned}
&\text{initiatedAt}(moving(P_1,P_2)=\text{true},\ T) \leftarrow\\
&\quad \text{happensAt}(walking(P_1),\ T),\\
&\quad \text{happensAt}(walking(P_2),\ T),\\
&\quad \text{holdsAt}(close(P_1,P_2)=\text{true},\ T),\\
&\quad \text{holdsAt}(similarOrientation(P_1,P_2)=\text{true},\ T).
\end{aligned} \tag{2.5}$$

$$\begin{aligned}
&\text{terminatedAt}(moving(P_1,P_2)=\text{true},\ T) \leftarrow\\
&\quad \text{happensAt}(walking(P_1),\ T),\\
&\quad \text{holdsAt}(close(P_1,P_2)=\text{false},\ T).
\end{aligned} \tag{2.6}$$

$$\begin{aligned}
&\text{terminatedAt}(moving(P_1,P_2)=\text{true},\ T) \leftarrow\\
&\quad \text{happensAt}(active(P_1),\ T),\\
&\quad \text{happensAt}(active(P_2),\ T).
\end{aligned} \tag{2.7}$$

Rule (2.5) suggests that if P_1 and P_2 are walking close to each other with a similar orientation, then P_1 and P_2 start moving together. Rules (2.6)–(2.7) express some termination conditions for $moving(P_1, P_2) =$ true. For example, rule (2.6) states that $moving(P_1, P_2) =$ true is terminated when P_1 walks away from P_2, and P_1 and P_2 are no longer close to each other. $\Diamond$

In order to compute whether $moving(P_1, P_2) =$ true holds at a given time-point, the domain-independent axiomatisation of inertia, i.e., rules (2.1)–(2.4), must be combined with the domain-specific definition of $moving(P_1, P_2) =$ true, i.e., rules (2.5)–(2.7). Note that $moving(P_1, P_2) =$ true may have multiple initiations as P_1 and P_2 may be interacting for several video frames. Similarly, $moving(P_1, P_2) =$ true may have multiple terminations. In this formulation of the Event Calculus, initiatedAt($F = V,\ T$) does not necessarily imply that $F \neq V$ at T. Similarly, terminatedAt($F = V,\ T$) does not necessarily imply that $F = V$ at T. Suppose that $F = V$ is initiated at time-points 50 and

65 and terminated at time-points 75 and 86, and at no other time-points. In that case, $F=V$ holds at all T such that $50 < T \leq 75$.

Note that rules (2.1)–(2.7) do not strictly conform with the syntax for logic programming rules we presented in Section 2.1, according to which every atom appearing in a rule has the form '$p(t_1, \ldots, t_n)$', $t_1, \ldots, t_n$ being constants or variables. In rule (2.5), e.g., the first arguments of happensAt($walking(P_1),\ T$) and initiatedAt($moving(P_1, P_2) =$ true, T) are $walking(P_1)$ and $moving(P_1, P_2) =$ true, which are not constants or variables. We use *reification*, i.e., treat an expression of a first-order language, like event $walking(P_1)$ or fluent $moving(P_1, P_2)$, as a concrete object of the domain (see Section 2.4 in [Mueller, 2015]). In the case of an FVP, such as $moving(P_1, P_2) =$ true, we apply reification one step further, by constructing a domain object corresponding to the FVP. As a result, events and FVPs are treated as terms, and thus our Event Calculus formulation does not violate the syntax of logic programming.

The Event Calculus is a logic programming formalism that allows us to monitor the events that take place over time and the changes in the values of fluents that come as a result of these events. Thus, the Event Calculus constitutes a suitable formalism for temporal pattern matching over streams. However, reasoning directly with the Event Calculus can be inefficient. In the next section, we highlight some inefficiencies of the Event Calculus, and describe an Event Calculus dialect that is optimised for stream reasoning.

2.3 The Run-Time Event Calculus (RTEC)

The Event Calculus dialect we described in Section 2.2 is not suitable for reasoning over data streams. For example, the Event Calculus may need to reason over the entire history of the stream for each new event arrival, which is not feasible in a streaming setting. In rule (2.2), e.g., in order to derive holdsAt($F=V$, T), a reasoner may attempt to prove initiatedAt($F=V$, T_s) at every time-point T_s that is before T, despite the fact that the only initiation of $F=V$ that is required to prove holdsAt($F=V$, T) is the one that is the closest to T. The redundant computations of initiatedAt($F=V$, T_s) queries at every time-point prior to T may lead to an unmanageable computational overhead. Moreover, the Event Calculus may perform several redundant computations in the common cases where multiple domain-specific rules share common premises. Suppose, e.g., that the condition holdsAt($close(P_1, P_2) =$ true, T) of rule (2.5) appears as a condition of multiple initiatedAt/terminatedAt rules, resulting in the re-evaluation of holdsAt($close(P_1, P_2) =$ true, T) every time such a rule is invoked.

To address these issues, the Run-Time Event Calculus (RTEC) extends the Event Calculus with optimisation techniques for stream reasoning [Artikis et al., 2015]. We motivate RTEC and provide an informal description of its key features. (See Section 3.1 for a formal account of RTEC.)

RTEC is interval-based, i.e., it computes the maximal intervals during which FVPs hold continuously, instead of employing point-based reasoning, which is costly over long time-lines. In order to handle large volumes of events that may span over a long period of time, RTEC employs 'windowing' and 'forgetting', isolating parts of the stream (windows) and reasoning over the information in each window independently. Moreover, in order to avoid re-computations, RTEC caches the maximal intervals of FVPs that were computed at previous steps. We describe the key features of RTEC, following [Artikis et al., 2015].

Representation. RTEC derives the list of maximal intervals I during which an FVP $F=V$ holds continuously. RTEC employs a new predicate called holdsFor, where holdsFor$(F=V, I)$ denotes that $F=V$ holds continuously in the maximal intervals of list I.

RTEC features two types of fluents: *simple* and *statically determined*. Simple fluents are specified based on domain-dependent initiatedAt and terminatedAt rules (see, e.g., rules (2.5)–(2.7)), and follow the commonsense law of inertia, as specified by rules (2.1)–(2.4). In other words, the simple fluents of RTEC are defined in the same way as the fluents of our Event Calculus dialect. In order to compute the maximal intervals of an FVP $F=V$, where F is a simple fluent, RTEC employs a domain-independent definition of holdsFor. According to this definition, RTEC first computes the *initiation points* of $F=V$ by evaluating the initiatedAt rules for $F=V$. Then, RTEC computes all time-points T at which $F=V$ is 'broken', i.e., $F=V$ is terminated or F is initiated with a value other than V. These are the *termination points* of $F=V$. Subsequently, RTEC constructs the list of maximal intervals I of $F=V$ by pairing each initiation point T_s of $F=V$ with the first termination point T_f after T_s, ignoring every intermediate initiation point between T_s and T_f.

The maximal intervals of a statically determined fluent are specified based on a domain-dependent holdsFor rule. Consider the following example from human activity recognition:

Example 2 ($moving$ as a Statically Determined Fluent (after [Artikis et al., 2015])): The following rule is an alternative definition for the FVP '$moving(P_1, P_2)=\text{true}$' as a statically determined fluent:

$$\begin{aligned}
&\text{holdsFor}(moving(P_1, P_2)=\text{true}, I) \leftarrow \\
&\quad \text{holdsFor}(walking(P_1)=\text{true}, I_1), \\
&\quad \text{holdsFor}(walking(P_2)=\text{true}, I_2), \\
&\quad \text{holdsFor}(close(P_1, P_2)=\text{true}, I_3), \\
&\quad \text{intersect\_all}([I_1, I_2, I_3], I).
\end{aligned} \tag{2.8}$$

Lists I_1, I_2 and I_3 contain, respectively, the maximal intervals during which P_1 is walking, P_2 is walking, and P_1 and P_2 are close to each other. Rule (2.8) states that list I, containing the maximal intervals during which $moving(P_1, P_2)=\text{true}$ holds continuously, is computed as the intersection of the maximal intervals in I_1, I_2 and I_3. $\Diamond$

In Example 2, the definition of $moving(P_1, P_2)$ includes the predicate intersect_all, i.e., an interval manipulation construct of RTEC. The language of RTEC provides three interval manipulation constructs: union_all, intersect_all and relative_complement_all. union_all(L, I) (resp. intersect_all(L, I)) computes the list of maximal intervals I as the union (intersection) of all lists of maximal intervals of list L. relative_complement_all(I', L, I) computes the list of maximal intervals I by removing from the maximal intervals of list I' all time-points included in some list of maximal intervals of list L. Figure 2.1 presents an illustration of

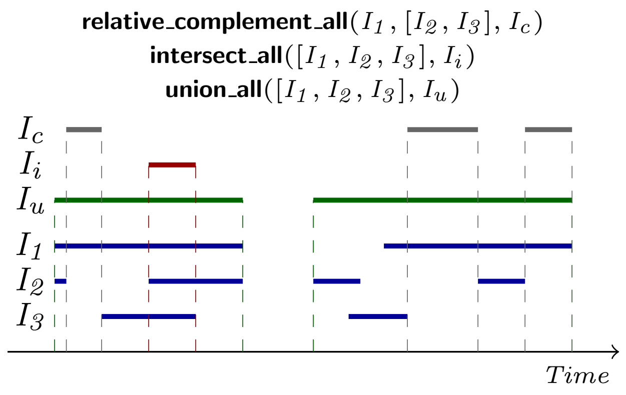

relative_complement_all$(I_1, [I_2, I_3], I_c)$
intersect_all$([I_1, I_2, I_3], I_i)$
union_all$([I_1, I_2, I_3], I_u)$
I_c
I_i
I_u
I_1
I_2
I_3
$Time$

Figure 2.1: 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.

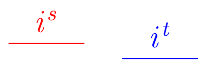
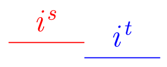
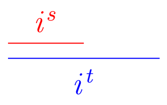
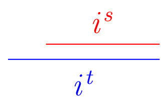
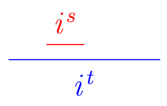
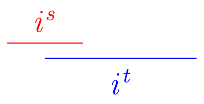
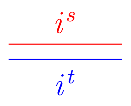
the interval manipulation constructs of RTEC.

Reasoning. RTEC employs a simple caching mechanism to avoid unnecessary re-computations. This caching mechanism leverages the hierarchical organisation of the rules defining FVPs in an RTEC program. RTEC processes such FVP hierarchies in a bottom-up manner, computing and caching the intervals of FVPs as the execution moves up in the hierarchy. This way, the intervals of the FVPs that are required for the processing of another FVP 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 $(q_i{-}\omega, q_i]$ are taken into consideration, while all other events are discarded/'forgotten'. This ensures that the cost of reasoning depends on the size of the window ω and not on the complete stream. 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 out-of-order, e.g., due to variable time lags in network transmissions, 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}.

RTEC captures several operators that are commonly required to express complex temporal patterns, such as conjunction, disjunction, negation and temporal persistence. Towards high efficiency and minimal reasoning latency, RTEC employs bottom-up caching for hierarchical patterns and windowing. Moreover, RTEC has proven highly efficient in challenging, real-world applications, such as maritime situational awareness, where RTEC was able to reason over windows spanning 16 hours and including, on average, over 3 million events, in order to compute the maximal intervals of composite maritime activities, in approximately 10 minutes [Pitsikalis et al., 2019], and commercial fleet management, where RTEC managed to reason over 16-hour windows containing, on average, 1.5 million events in approximately 5 minutes [Tsilionis et al., 2022]. However, there are several

Table 2.2: Seven relations of Allen's interval algebra.

Relation	**Definition**	**Illustration**
before(i^s, i^t)	$f(i^s) < s(i^t)$	
meets(i^s, i^t)	$f(i^s) = s(i^t)$	
starts(i^s, i^t)	$s(i^s) = s(i^t)$, $f(i^s) < f(i^t)$	
finishes(i^s, i^t)	$s(i^s) > s(i^t)$, $f(i^s) = f(i^t)$	
during(i^s, i^t)	$s(i^s) > s(i^t)$, $f(i^s) < f(i^t)$	
overlaps(i^s, i^t)	$s(i^s) < s(i^t)$, $f(i^s) > s(i^t)$, $f(i^s) < f(i^t)$	
equal(i^s, i^t)	$s(i^s) = s(i^t)$, $f(i^s) = f(i^t)$	

types of temporal patterns that are not supported by the language of RTEC. In Chapter 3, we propose three extensions to the language of RTEC, broadening the set of supported temporal specifications, while maintaining high reasoning efficiency. One of these extensions concerns the relations of Allen's interval algebra, which are not supported by RTEC. In the next section, we outline the relations of Allen's algebra.

2.4 Allen's Interval Algebra

Allen's interval algebra specifies thirteen jointly exhaustive and pairwise disjoint relations among ordered pairs of intervals [Allen, 1984]. These relations are qualitative, in the sense that they are defined based on the relative temporal position of the two intervals, and not their numeric values. Given an ordered pair of intervals (i^s, i^t), we can determine the relation it satisfies by applying a set of arithmetic comparisons on the endpoints of intervals i^s and i^t. Table 2.2 presents the Allen 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 2.2 shows the conditions under which each Allen relation is satisfied. i^s and i^t express intervals, while $s(i)$ and $f(i)$ denote the start and end endpoint of interval i, respectively.

Allen relations are often required to express the temporal specifications found in modern applications [Körber et al., 2021; Awad et al., 2022]. RTEC, however, does not support

Allen relations, as it is not possible to express these relations using the interval manipulation constructs of RTEC, i.e., union_all, intersect_all, and relative_complement_all. Consider, e.g., the lists of intervals $\mathcal{S}$ and $\mathcal{T}$, and the computation of the interval pairs (i^s, i^t), where $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, satisfying the before relation. intersect_all$([\mathcal{S}, \mathcal{T}], [\,])$ states that for every interval pair (i^s, i^t), such that $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, it holds that $i^s \cap i^t = \emptyset$. Therefore, i^s is before i^t, or vice versa. It is impossible, however, to distinguish between the two cases.

In Section 3.4, we propose an extension to the language of RTEC in order to support Allen relations, and provide the necessary reasoning algorithms for computing Allen relations over data streams. In the next section, we provide a literature review on stream reasoning systems.

2.5 Literature Review

We review frameworks that perform temporal pattern matching without noisy events; we discuss frameworks handling noisy events in Section 5.4. We focus on frameworks that employ a logic-based formalism, possibly incorporating the Event Calculus, and/or support interval-based reasoning. Approaches that update temporal patterns by means of machine learning techniques (e.g., [Katzouris and Artikis, 2020; Katzouris et al., 2023]), are complementary to our work, and thus not discussed here. Table 2.3 highlights the most prominent frameworks of our discussion, in the sense that they fulfil several of the requirements we outlined in Chapter 1, and compares them in terms of their underlying formalism and temporal model, as well as their support for background knowledge, negation, hierarchical patterns, stream reasoning, cyclic dependencies, events with delayed effects and Allen relations.

The literature contains numerous automata-based frameworks for temporal pattern matching over streams [Wu et al., 2006; Demers et al., 2007; Poppe et al., 2019; Zhao et al., 2021; Alevizos et al., 2022]. CORE is a recently proposed automata-based engine that employs a compact data structure for maintaining previous derivations, leading to low reasoning complexity, and has proven to outperform other automata-based engines by orders of magnitude [Bucchi et al., 2022]. The language of CORE exhibits a formal semantics and supports an interval-based representation for situations of interest. However, CORE has limited expressivity, as it supports only unary relations, applied only to the last event read. Thus, CORE cannot express, e.g., the pattern of 'moving together' from the human activity recognition domain (see (2.5)–(2.7)), which involves two entities. Moreover, although exhibiting a compositional semantics, CORE does not support hierarchies, in the sense that patterns are defined only over instantaneous properties. A comparison of automata-based and logic-based frameworks, including RTEC, may be found in the survey of Giatrakos et al. [2020].

Several approaches employ extensions of SQL for handling data streams [Terry et al., 1992; Arasu et al., 2006; Bai et al., 2006; Körber et al., 2021; Zervoudakis et al., 2021; Awad et al., 2022]. The StreamMill system [Laptev et al., 2016] is based on ESL, i.e., a minimal extension of SQL that supports stream reasoning [Law et al., 2004]. ESL is re-

Table 2.3: A comparison of state-of-the-art frameworks. B.K.: Background Knowledge, N.: Negation, H.: Hierarchies, S.R.: Stream Reasoning, C.: Cycles, D.E.: Events with Delayed Effects, A.: Allen Relations, ✓*: Although lacking a built-in represenation and optimised treatment, the underlying language is expressive enough to capture this feature.

Approach	**Formalism**	**Temporal Model**	**B.K.**	**N.**	**H.**	**S.R.**	**C.**	**D.E.**	**A.**
RTEC	Logic programming	Intervals, Explicit, Event Calculus.	✓	✓	✓	✓			
CORE	Complex Event Automata	Intervals, Implicit.				✓			
Ticker	LARS	Points, Explicit.	✓	✓		✓			
GKL-EC	Logic programming	Points, Explicit, Event Calculus.	✓	✓			✓	✓	
s(CASP)	Answer set programming	Points, Explicit, Event Calculus.	✓	✓			✓*	✓	
Fusemate	Logic programming and description logic $\mathcal{ALCIF}$	Points, Explicit, Event Calculus.	✓	✓	✓	✓	✓*		
jREC^{fi}	Logic programming	Intervals, Explicit, Event Calculus.	✓	✓	✓	✓	✓*	✓	
jREC^{rbt}	Logic programming	Intervals, Explicit, Event Calculus.	✓	✓	✓	✓	✓*		
D^{2}IA	Interval-based extension of CQL.	Intervals, Implicit.		✓	✓	✓			✓
TPStream	SQL-like query language.	Intervals, Implicit.				✓			✓
ETALIS	Rule-based language translated into logic programming.	Intervals, Implicit.	✓	✓		✓			✓
Approach	**Formalism**	**Temporal Model**	**B.K.**	**N.**	**H.**	**S.R.**	**C.**	**D.E.**	**A.**

stricted to non-blocking queries, i.e., queries that can be answered incrementally, as new stream items arrive, without having to wait for an arbitrary amount of time on new item arrivals [Zaniolo, 2012]. Although SQL-based formalisms are expressive enough to capture complex temporal constraints, the resulting expressions are commonly highly complicated, and hard to understand and optimise (see, e.g., Examples 2 and 3 in [Aghasadeghi et al., 2024]).

There are also logic-based stream reasoning frameworks. The Chronicle Recognition System (CRS) represents situations of interest as sets of events that are associated with time constraints [Dousson and Maigat, 2007]. The language of CRS supports several operators, including sequencing, iteration and negation. TESLA provides a logic-based event specification language that supports negation, aggregates, iteration and event hierarchies [Cugola and Margara, 2010]. Contrary to logic programming-based approaches, CRS and TESLA do not support background knowledge. See [Artikis et al., 2012b] for a tutorial on logic-based event recognition.

Some logic-based systems restrict attention to (variants of) Datalog. MeTeoR [Walega et al., 2023] extends a fragment of DatalogMTL [Walega et al., 2019] with windowing. MeTeoR does not support negation, meaning that several features of RTEC, such as default persistence based on the law of inertia and the relative_complement_all operator, cannot be expressed in MeTeoR. Logica is a Datalog-like programming language that compiles into SQL and runs on BigQuery[1]. In Chapter 4, we present an experimental comparison of Logica with other frameworks. DDlog [Ryzhyk and Budiu, 2019] is an incremental version of Datalog that handles continuously-arriving updates of input facts. DCDatalog [Wu et al., 2022] and BigDatalog [Shkapsky et al., 2016] are optimised with parallelisation techniques for recursive queries. DDLog, DCDatalog and BigDatalog reason over atemporal data, and thus cannot be employed directly on temporal patterns.

One of the most prominent logic-based languages is LARS, i.e., an extension of Answer Set Programming (ASP), featuring built-in window constructs for stream reasoning [Beck et al., 2018]. Fragments of the LARS language are supported by a family of stream reasoners, including Ticker [Beck et al., 2017] and Laser [Bazoobandi et al., 2017]. Laser 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 [Barbieri et al., 2010; Phuoc et al., 2011], including Ticker. Moreover, Eiter et al. [2019] presented a distributed architecture using 'stream stratification' [Beck et al., 2018] in order to decompose LARS programs into sub-programs that may be evaluated in parallel. Laser and the distributed LARS framework, however, are restricted to globally stratified logic programs, i.e., stratified programs where all ground atoms of the Herbrand base that have the same predicate are in the same stratum. RTEC event descriptions are locally stratified logic programs, which comprise a broader class than globally stratified logic programs [Przymusinski, 1988]. In the language of Ticker, we were able to express an event description that corresponds to a locally stratified logic program in RTEC, and, as a result, were able to include Ticker in our empirical comparisons (see Chapter 4). More-

[1] *https://github.com/EvgSkv/logica*

over, although the LARS language can express rules with interval-based derivations, such rules are not included in the fragments of LARS supported by the aforementioned LARS-based reasoners [Beck et al., 2018].

A key difference between our work and the aforementioned frameworks lies in the use of the Event Calculus, which allows us to develop expressive temporal specifications for a wide range of stream reasoning applications, such composite event recognition and multi-agent systems. 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 introduce extensions to the language of the Event Calculus with optimised reasoning algorithms, like the language constructs and reasoning algorithms that we present in Chapter 3, in order to meet the requirements of contemporary applications concerning expressivity and reasoning efficiency.

Various 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 efficient answer set programming solvers may be used for reasoning [Lee and Palla, 2012].

Srinivasan et al. [2022] proposed an Event Calculus dialect with an integrated domain specification for ‘biological feedback loops’ [Thomas and d’Ari, 1990], which follows the Generalised Kinetic Logic [Thomas, 1991], in order to predict the evolution of such loops. We call the resulting logic program GKL-EC. GKL-EC employs a logic programming implementation of the Event Calculus, and thus seamlessly supports background knowledge and negation-by-failure. The value of a variable participating in a feedback loop typically depends on the value of the same variable at an earlier time-point. Moreover, the value of a variable changes with a specified time delay after the emission of a set of specified biological signals. To address these issues, GKL-EC was designed to handle cyclic dependencies and events with delayed effects. Contrary to RTEC, GKL-EC does not feature windowing or optimised processing for hierarchical event description, and thus does not support stream reasoning.

s(CASP) is a query-driven execution model for ASP with constraints, supporting Event Calculus-based reasoning [Arias et al., 2022]. Commonly, ASP solvers ground the input program, which requires that the domains of the variables of the program are discrete [Lee and Palla, 2012; Gebser et al., 2019]. In contrast, s(CASP) evaluates ASP programs without grounding them, and thus seamlessly supports dense domains. Moreover, s(CASP) is able to compute answer sets containing non-ground variables and constraints, as well as partial models containing only the fragment of a stable model that is necessary to answer a given query. The Event Calculus dialect adopted by s(CASP) includes the trajectory(F_1, T_1, F_2, T_2) predicate, expressing that if a fluent F_1 is initiated at time T_1 and continues to hold until time T_1+T_2, then fluent F_2 holds at T_1+T_2 [Miller and Shanahan, 2002]. Using this predicate, s(CASP) may express events with delayed effects. Contrary to RTEC, s(CASP) does not include techniques for handling data streams, such as windowing and optimised reasoning for compositional/hierarchical patterns. Moreover, s(CASP) employs a point-based representation for its derivations, as opposed to the interval-based representation of RTEC, leading to slower reasoning (see our empirical comparisons in Section 4.2).

The aforementioned Event Calculus-based frameworks are not suitable for stream reasoning. Fusemate is a logic programming framework for reasoning over data streams [Baumgartner, 2021b]. Fusemate integrates Event Calculus-based reasoning with a description logic, thus treating ABoxes as Event Calculus fluents and modelling their persistence through time [Baumgartner, 2021a]. By taking advantage of the language of logic programming, Fusemate seamlessly supports background knowledge and negation-by-failure, while the use of the Event Calculus allows for an explicit representation of time and offers an intuitive representation of time-varying properties. Moreover, the language of Fusemate is restricted to a class of stratified programs, where cyclic dependencies with negation are allowed, provided that, as we follow the induced dependencies, the values of the time arguments of the predicates we encounter are decreasing. This way, Fusemate can evaluate a program by following one of its stratifications bottom-up, while avoiding re-computations, paving the way for the efficient processing of hierarchical patterns. Contrary to RTEC, Fusemate adopts a point-based Event Calculus dialect, and performs time-point-based query evaluation, leading to a larger number of computations and higher reasoning times than RTEC (see Section 4.2).

The ‘Reactive Event Calculus’ (REC) [Chesani et al., 2009; Chesani et al., 2010] adopts a lightweight version of the *update-time* reasoning of the ‘Cached Event Calculus’ [Chittaro and Montanari, 1996]. This way, REC caches and revises fluent intervals incrementally, upon the arrival of (delayed) events, paving the way for stream reasoning. There are several frameworks that use REC [Chesani et al., 2013; Montali et al., 2013; Falcionelli et al., 2019], that are typically based on a Java-Prolog implementation of REC called jREC [Bragaglia et al., 2012]. jREC is based on logic programming, and supports background knowledge and negation. Moreover, jREC computes situations of interest in temporal intervals, and employs an incremental caching and updating schema for derived situations, providing support for event hierarchies. We focus on two versions of jREC, i.e., jREC^{fi} and jREC^{rbt}. jREC^{fi} supports events with delayed effects and has been used for monitoring the effects of such events in the maritime domain [Montali et al., 2013]. jREC^{rbt} improves upon the efficiency of retrieving and updating fluent intervals in REC via an indexing schema that uses red-black trees [Falcionelli et al., 2019]. jREC^{rbt} is more efficient than jREC^{fi}; however, jREC^{rbt} does not support events with delayed effects. In Chapter 4, we compare our extensions of RTEC against jREC^{fi} and jREC^{rbt}.

One of the key advantages of RTEC and jREC is their interval-based semantics. In stream reasoning and composite event recognition applications, situations of interest are typically durative and naturally represented using temporal intervals [Giatrakos et al., 2020].

Using sets of individuals time-points to represent occurrences of durative situations fails to capture ongoing activities and leads to semantic ambiguities in cases of compositional/hierarchical patterns [Galton and Augusto, 2002; Paschke, 2005; White et al., 2007]. In the following, we discuss frameworks that model durative situations using temporal intervals.

D^2IA augments the Big Data stream processing engine Flink with interval-based semantics [Awad et al., 2022]. The language of D^2IA extends CQL [Arasu et al., 2006] with a set of temporal operators for mapping streams of instantaneous events to streams of durative events, and vice versa, as well as reasoning over durative events with Allen relations. Moreover, D^2IA employs windowing, in order to support streaming applications. Compared to RTEC, the language of D^2IA has several limitations. In D^2IA, e.g., it is not straightforward to combine instantaneous and durative events in the conditions of a pattern, while negation is limited to the absence of a specified type of instantaneous event between the occurrences of two other events, and background knowledge is not supported.

TPStream [Körber et al., 2021] transforms instantaneous events into durative situations of interest, and computes temporal patterns, including Allen relations, over the intervals of the derived situations. In TPStream, situations cannot be defined in terms of multiple event types or background knowledge. For example, it is not possible to express the ‘moving together’ activity from the human activity recognition domain (see rules (2.5)–(2.7)). ISEQ [Li et al., 2011] is a temporal operator that processes streams of durative events using temporal windows and outputs sequences of specified event types that satisfy a set of Allen relations. ISEQ does not allow for the derivation of durative situations from instantaneous events. Unlike RTEC, neither ISEQ nor TPStream supports relational patterns, such as ‘moving together’, which involves two entities of the domain. This is a significant limitation for modern applications. ETALIS [Anicic et al., 2012] is an event-driven stream reasoning system that supports Allen relations and incorporates background knowledge in temporal patterns. ETALIS does not allow for the explicit representation of time, complicating the specification of fluent value changes, including the formalisation of the common-sense law of inertia.

RTEC exhibits many desirable features for stream reasoning, such as an expressive, formal temporal specification language, an efficient treatment of compositional patterns, an interval-based semantics, and an optimised processing of the input stream with a bounded memory using windowing. Fluent-value pair (FVP) hierarchies, i.e., FVPs appearing in the definitions of other FVPs, allow for structured, succinct representations, and thus code maintenance. A hierarchy of FVPs may exhibit a structure where FVPs participate in the definition of multiple other FVPs. RTEC can naturally handle hierarchies, paving the way for caching, and thus avoiding unnecessary re-computations. Point-based reasoning, as opposed to interval-based reasoning—see, e.g., ASTRO [Das et al., 2018] and the LARS reasoners Ticker and Laser [Beck et al., 2018]—leads to unintended semantics when reasoning over hierarchies [Galton and Augusto, 2002; Paschke, 2005; White et al., 2007]. Moreover, RTEC has proven highly efficient in contemporary applications with complex temporal specifications and large, real data streams [Artikis et al., 2015; Pitsikalis et al., 2019; Tsilionis et al., 2022].

In a parallel line of work, Tsilionis et al. [2022] presented an extension of RTEC that optimises reasoning when input events arrive with a variable delay, as well as when input events are revised or retracted. This extension does not broaden the set of temporal patterns supported by RTEC, and thus it is orthogonal to the three extensions to the language of RTEC that we propose in Chapter 3.

Based on our literature review, RTEC is one of the few frameworks that support stream reasoning with an Event Calculus-style knowledge representation, an explicit time model including both points and intervals, and a language with a formal and declarative semantics, supporting background knowledge, negation and pattern hierarchies. Moreover, as we demonstrate empirically in Section 4.2, RTEC is the most efficient among the Event Calculus-based frameworks found in the literature. For these reasons, we chose to extend the range of temporal specifications supported by RTEC, paving the way for expressive stream reasoning with the Event Calculus. In the next chapter, we present our three extensions of RTEC.

3. STREAM REASONING OVER COMPLEX TEMPORAL SPECIFICATIONS

RTEC is a formal computational framework that extends the Event Calculus with optimisation techniques, aiming at highly efficient stream reasoning. We propose RTEC$_\circ$, RTEC$^\rightarrow$ and RTEC$_\mathsf{A}$, i.e., three frameworks that are built by extending RTEC, in order to support cyclic dependencies, events with delayed effects and Allen relations, respectively. RTEC$_\circ$, RTEC$^\rightarrow$ and RTEC$_\mathsf{A}$ broaden the set of temporal patterns supported by RTEC, while supporting highly efficient reasoning over data streams. Section 3.1 provides a formal account of RTEC. Afterwards, Sections 3.2, 3.3 and 3.4 present RTEC$_\circ$, RTEC$^\rightarrow$ and RTEC$_\mathsf{A}$.

3.1 A Formal Account of RTEC

First, we provide a formal description of the language of RTEC. Afterwards, we present a structure called ‘dependency graph’, i.e., a directed acyclic graph modelling the dependencies among the fluent-value pairs (FVP)s in the specifications of a domain. With the help of the dependency graph, we identify the semantics of an RTEC program.

Syntax. RTEC features events, simple fluents and statically determined fluents. The temporal specifications of a domain in RTEC, containing a simple or a statically determined fluent definition for each situation of interest of the domain, form an *event description*.

Definition 4 (Event Description in RTEC): An event description in RTEC is a set of:

1. Ground happensAt facts, expressing a stream of event instances.
2. Rules with head initiatedAt$(F = V, T_1, T, T_2)$ or terminatedAt$(F = V, T_1, T, T_2)$, expressing the effects of events on an FVP $F = V$ of a simple fluent F.
3. Rules with head holdsFor$(F = V, I)$, defining an FVP $F = V$ of a statically determined fluent F in terms of other FVPs. ■

Definition 5 (Syntax of Simple Fluent Definitions in RTEC): The rules defining initiatedAt predicates in RTEC have the following syntax:

$$\begin{aligned}
&\mathsf{initiatedAt}(F = V,\ T',\ T,\ T'') \leftarrow \\
&\quad \mathsf{happensAt}(E_1,\ T),\ T' \leq T < T''[[, \\
&\quad [\mathsf{not}]\ \mathsf{happensAt}(E_2,\ T),\ \ldots, \\
&\quad [\mathsf{not}]\ \mathsf{happensAt}(E_i,\ T), \\
&\quad [\mathsf{not}]\ \mathsf{holdsAt}(F_1 = V_1,\ T),\ \ldots, \\
&\quad [\mathsf{not}]\ \mathsf{holdsAt}(F_k = V_k,\ T), \\
&\quad \textit{atemporal\_constraints}]].
\end{aligned} \tag{3.1}$$

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, and *atemporal_constraints*, i.e., a conjunction of atemporal predicates expressing background knowledge, denoted by '[[ ]]'. 'not' expresses negation-by-failure, while '[not]' denotes that 'not' is optional. All (head and body) predicates that have a time argument 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. ■

Definition 6 (Syntax of Statically Determined Fluent Definitions in RTEC): The rules defining statically determined fluents in RTEC have the following syntax:

$$\begin{aligned}
&\text{holdsFor}(F = V,\ I_{n+m}) \leftarrow \\
&\quad \text{holdsFor}(F_1 = V_1,\ I_1)[[, \\
&\quad \text{holdsFor}(F_2 = V_2,\ I_2),\ \ldots \\
&\quad \text{holdsFor}(F_n = V_n,\ I_n), \\
&\quad \text{intervalConstruct}(L_1,\ I_{n+1}),\ \ldots \\
&\quad \text{intervalConstruct}(L_m,\ I_{n+m}), \\
&\quad \textit{atemporal\_constraints}]].
\end{aligned} \tag{3.2}$$

The first body literal of a holdsFor rule defining $F = V$ is a holdsFor predicate expressing the maximal intervals of an FVP other than $F = V$. This is followed by a possibly empty list, denoted by '[[ ]]', of holdsFor predicates for other FVPs, interval manipulation constructs, expressed by intervalConstruct in formulation (3.2), and atemporal constraints expressing background knowledge. intervalConstruct(L_j, I_{n+j}) may be union_all(L_j, I_{n+j}), intersect_all(L_j, I_{n+j}) or relative_complement_all(I_k, L_j, I_{n+j}). I_k, where $k < n + j$, is a list of maximal intervals appearing earlier in the body of the rule, and list L_j contains a subset of such lists. The output list I_{n+m} contains the maximal intervals during which $F = V$ holds continuously. ■

While statically determined fluent definitions are less expressive than simple fluent definitions, they have the advantages of being more compact and less costly to reason with. A statically determined fluent definition contains one rule, constructing the maximal intervals of an FVP based on the maximal intervals of other FVPs, using the interval manipulation constructs of RTEC. In contrast, in order to construct a simple fluent definition, we have to identify the various conditions under which a situation of interest is initiated and terminated, so that its maximal intervals can then be computed by the domain-independent holdsFor predicate of RTEC. Moreover, based on the worst-case time complexity analysis presented in [Artikis et al., 2015], the cost of deriving the maximal intervals of a statically determined fluent is linear to the window size ω, while maximal interval computation for a simple fluent is quadratic to ω. Therefore, whenever possible, a situation of interest should be defined using a statically determined fluent.

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

Definition 7 (Dependency Graph in RTEC): The dependency graph of an event description in RTEC is a directed graph $G = (\mathcal{V}, \mathcal{E})$, where:

1. $\mathcal{V}$ contains one vertex $v_{F=V}$ for each FVP $F = V$.

2. $\mathcal{E}$ contains an edge $(v_{F_j\,=\,V_j}, v_{F_i\,=\,V_i})$ 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, or
    - there is a holdsFor rule for $F_i = V_i$ having holdsFor$(F_j = V_j, I)$ 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 8 (Vertex Level and FVP Level in RTEC): Given a directed acyclic graph, the level of a vertex v is equal to:

1. 1, if v has no incoming edges.
2. n, where $n > 1$, if v has at least one incoming edge from a vertex of level $n{-}1$, and zero or more incoming edges from vertices of levels lower than $n{-}1$.

Suppose that an event description in RTEC defines an acyclic dependency graph $G = (\mathcal{V}, \mathcal{E})$ and that $v_{F\,=\,V} \in \mathcal{V}$ is the vertex corresponding to FVP $F = V$. The level of $F = V$ is equal to the level of vertex $v_{F\,=\,V}$. ■

Proposition 1 (Semantics of RTEC): An event description in RTEC that defines an acyclic dependency graph is a locally stratified logic program [Artikis et al., 2015]. ▲

A local 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. Since an RTEC event description that define an acyclic dependency graph is a locally stratified logic program, it admits a unique perfect model (see Section 2.1).

We propose three extensions of RTEC. In Section 3.2, we relax the constraint that the dependency graph must be acyclic. In Section 3.3, we introduce a way to express the *delayed* effects of events. In Section 3.4, we extend statically determined fluent definitions (see formulation (3.2)), in order to support Allen relations as body conditions and interval constructors.

3.2 RTEC$_\circ$: Reasoning over Specifications with Cyclic Dependencies

We propose RTEC$_\circ$, i.e., an extended version of RTEC that supports event descriptions with cyclic dependencies. We motivate the need for handling cyclic dependencies, and study the semantics of RTEC$_\circ$. Subsequently, we present the reasoning algorithms of RTEC$_\circ$ for handling cyclic dependencies. We prove the correctness of the proposed algorithms, and outline their complexity.

3.2.1 Motivation

Temporal specifications, such as those found in composite event recognition and multi-agent systems, often include cyclic dependencies.

Example 3 (Cyclic Dependencies in Temporal Specifications): Consider, e.g., the specification of the status of a motion in a multi-agent voting protocol (see Section 1.1.3):

$$\begin{aligned}&\text{initiatedAt}(status(M) = proposed,\ T',\ T,\ T'') \leftarrow\\&\quad \text{happensAt}(propose(P, M),\ T),\ T' \leq T < T'',\\&\quad \text{holdsAt}(status(M) = null,\ T).\end{aligned} \tag{3.3}$$

$$\begin{aligned}&\text{initiatedAt}(status(M) = voting,\ T',\ T,\ T'') \leftarrow\\&\quad \text{happensAt}(second(S, M),\ T),\ T' \leq T < T'',\\&\quad \text{holdsAt}(status(M) = proposed,\ T).\end{aligned} \tag{3.4}$$

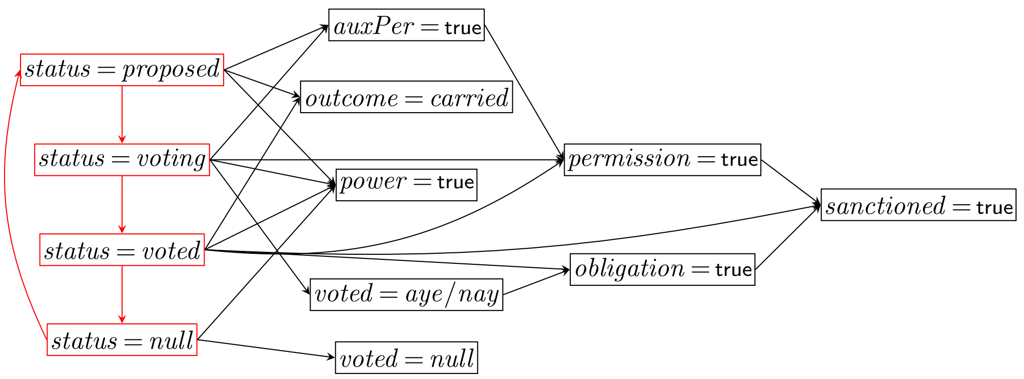
$$\begin{aligned}&\text{initiatedAt}(status(M) = voted,\ T',\ T,\ T'') \leftarrow\\&\quad \text{happensAt}(close\_ballot(C, M), T), T' \leq T < T'',\\&\quad \text{holdsAt}(status(M) = voting,\ T).\end{aligned} \tag{3.5}$$

$$\begin{aligned}&\text{initiatedAt}(status(M) = null,\ T',\ T,\ T'') \leftarrow\\&\quad \text{happensAt}(declare(C, M, R),\ T),\ T' \leq T < T'',\\&\quad \text{holdsAt}(status(M) = voted,\ T).\end{aligned} \tag{3.6}$$

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. $\diamondsuit$

The top part of Figure 3.1 displays the dependency graph defined by the event description of a voting protocol. This graph contains a cycle which is formed by rules (3.3)–(3.6). As another example, the bottom part of Figure 3.1 displays a dependency graph of NetBill, a protocol we described in Section 1.1.3 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.

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 (3.3) have been processed and thus their intervals can be fetched from



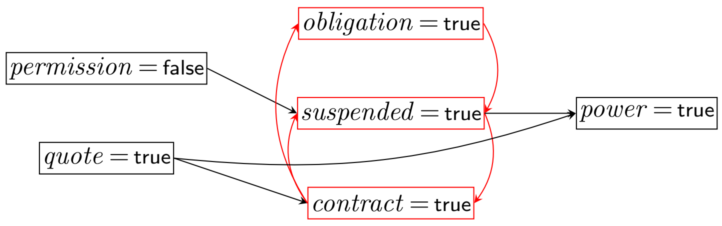


Figure 3.1: Dependency Graphs: voting (top) and NetBill (bottom). In order to simplify the presentation, we omit the arguments of fluents, group FVPs with the same conditions into a single node, and display a vertex v_j as j. Cycles are coloured red. Apart from the basic features of these protocols, 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 (see Section 1.1.3).

the cache. $status(M)=proposed$ depends on $status(M)=null$ which in turn depends on $status(M)=proposed$.

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

$$\begin{aligned}&\text{holdsAt}(F=V,\ T) \leftarrow\\ &\quad \text{initiatedAt}(F=V,\ q_i{-}\omega,\ T_i,\ T),\\ &\quad \text{not brokenBetween}(F=V,\ T_i,\ T).\end{aligned} \tag{3.7}$$

$$\begin{aligned}&\text{brokenBetween}(F=V,\ T_i,\ T) \leftarrow\\ &\quad \text{terminatedAt}(F=V,\ T_i,\ T_b,\ T).\end{aligned} \tag{3.8}$$

$$\begin{aligned}&\text{brokenBetween}(F=V,\ T_i,\ T) \leftarrow\\ &\quad \text{initiatedAt}(F=V',\ T_i,\ T_b,\ T),\ V{\neq}V'.\end{aligned} \tag{3.9}$$

Rule (3.7) 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 or F is initiated with a different value in that interval (see rule (3.8) and (3.9)).

Table 3.1: Stream reasoning with the Event Calculus. 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, '$\checkmark$' to express a successful evaluation, and '$\times$' to denote an unsuccessful evaluation. The predicates in the second column are indented to distinguish between the head and body atoms of a rule.

	Predicate	Rule
1	initiatedAt$(status(m)=proposed,\ q_i{-}\omega,\ T,\ q_i)$?	(3.3)
2	happensAt$(propose(ag_p,m),\ t_4)$ ✓	(3.3), (3.10)
3	holdsAt$(status(m)=null,\ t_4)$?	(3.3), (3.7)
4	initiatedAt$(status(m)=null,\ q_i{-}\omega,\ T_i,\ t_4)$?	(3.7), (3.6)
5	happensAt$(declare(ag_c,m,not_carried),\ t_3)$ ✓	(3.6), (3.10)
6	holdsAt$(status(m)=voted,\ t_3)$?	(3.6), (3.7)
7	initiatedAt$(status(m)=voted,\ q_i{-}\omega,\ T_i',\ t_3)$?	(3.7), (3.5)
8	happensAt$(close_ballot(ag_c,m),\ t_2)$ ✓	(3.5), (3.10)
9	holdsAt$(status(m)=voting,\ t_2)$?	(3.5), (3.7)
10	initiatedAt$(status(m)=voting,\ q_i{-}\omega,\ T_i'',\ t_2)$?	(3.7), (3.4)
11	happensAt$(second(ag_s,m),\ t_1)$ ✓	(3.4), (3.10)
12	holdsAt$(status(m)=proposed,\ t_1)$?	(3.4), (3.7)
13	initiatedAt$(status(m)=proposed,q_i{-}\omega,T_i''',t_1)$?	(3.7)
14	initiatedAt$(status(m)=proposed,q_i{-}\omega,q_i{-}\omega,t_1)$ ✓	(3.11)
15	brokenBetween$(status(m)=proposed,\ q_i{-}\omega,\ t_1)$?	(3.7), (3.9)
16	initiatedAt$(status(m)=voting,\ q_i{-}\omega,\ T_b,\ t_1)$ ×	(3.9), (3.4)
17	initiatedAt$(status(m)=voted,\ q_i{-}\omega,\ T_b,\ t_1)$ ×	(3.9), (3.5)
18	initiatedAt$(status(m)=null,\ q_i{-}\omega,\ T_b,\ t_1)$ ×	(3.9), (3.6)
19	brokenBetween$(status(m)=proposed,q_i{-}\omega,t_1)$ ×	(3.9), (3.7)
20	holdsAt$(status(m)=proposed,\ t_1)$ ✓	(3.7), (3.4)
21	initiatedAt$(status(m)=voting,\ q_i{-}\omega,\ t_1,\ t_2)$ ✓	(3.4), (3.7)
22	brokenBetween$(status(m)=voting,\ t_1,\ t_2)$ ×	(3.7), (3.9)
23	holdsAt$(status(m)=voting,\ t_2)$ ✓	(3.7), (3.5)
24	initiatedAt$(status(m)=voted,\ q_i{-}\omega,\ t_2,\ t_3)$ ✓	(3.5), (3.7)
25	brokenBetween$(status(m)=voted,\ t_2,\ t_3)$ ×	(3.7), (3.9)
26	holdsAt$(status(m)=voted,\ t_3)$ ✓	(3.7), (3.6)
27	initiatedAt$(status(m)=null,\ q_i{-}\omega,\ t_3,\ t_4)$ ✓	(3.6), (3.7)
28	brokenBetween$(status(m)=null,\ t_3,\ t_4)$ ×	(3.7), (3.9)
29	holdsAt$(status(m)=null,\ t_4)$ ✓	(3.7), (3.3)
30	initiatedAt$(status(m)=proposed,\ q_i{-}\omega,\ t_4,\ q_i)$ ✓	(3.3)

We may use the domain-independent rules (3.7)–(3.9) in conjunction with the definition of the $status$ fluent of rules (3.3)–(3.6) for processing streams of agent actions in a voting procedure and computing the values of the $status$ fluent at each time-point. Consider the following example:

Example 4 (Processing Cyclic Dependencies with the Event Calculus): Assume that the current window $(q_i{-}\omega, q_i]$ consists of the following event stream:

$$\begin{aligned}
&\text{happensAt}(second(ag_s,m),\ t_1).\\
&\text{happensAt}(close\_ballot(ag_c,m),\ t_2).\\
&\text{happensAt}(declare(ag_c,m,not\_carried),\ t_3).\\
&\text{happensAt}(propose(ag_p,m),\ t_4).\\
&\text{where } q_i{-}\omega < t_1 < t_2 < t_3 < t_4 < q_i
\end{aligned} \tag{3.10}$$

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

$$\begin{aligned}&\mathsf{initiatedAt}(status(M) = proposed,\ q_i{-}\omega,\ q_i{-}\omega,\ T) \leftarrow\\ &\quad T > q_i{-}\omega.\end{aligned} \tag{3.11}$$

To compute the maximal intervals for which $status(m) = proposed$ holds continuously, we first need to compute the initiation points of this FVP. Table 3.1 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 (3.3)); 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). $\lozenge$

Rule-set (3.7)–(3.9) constitutes a very inefficient implementation, leading to numerous unnecessary re-computations. Assume, e.g., that the event stream (3.10) includes an additional event:

$$\begin{aligned}&\mathsf{happensAt}(propose(ag_p, m),\ t_5).\\ &\text{where } t_4 < t_5 < q_i\end{aligned} \tag{3.12}$$

In this case, most predicate calls presented in Table 3.1 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 3.1 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 (3.12). Subsequently, the call to $\mathsf{brokenBetween}(status(m) = null, t_3, t_5)$ 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.:

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

To calculate these initiation points, we would have to repeat the computations presented in lines 10–21 of Table 3.1, i.e., we would have to prove again that $\mathsf{holdsAt}(status(M) = proposed, t_1)$. 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 3.1, 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 3.1.

To address these issues, we propose RTEC$_\circ$, an extended version 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 and the reasoning algorithms of RTEC$_\circ$. Afterwards, we prove the correctness of RTEC$_\circ$, and outline its complexity.

3.2.2 Semantics

In RTEC$_\circ$, the dependency graph of an event description may include cycles (see, e.g., Figure 3.1). Thus, we need to modify Definition 8 of FVP level to cater for dependency graphs with cycles. The strongly connected components (SCCs) of a cyclic dependency graph include either a single vertex, corresponding to an 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 9 (SCC Contracted Graph): Given a directed graph $G = (\mathcal{V}, \mathcal{E})$ and its SCCs $S_1, S_2, \dots, S_n$, the *SCC contracted graph* $G' = (\mathcal{V}', \mathcal{E}')$ of G is defined as follows:

1. $\mathcal{V}' = \bigcup_{1 \leq i \leq n} \{v_{S_i}\}$.
2. $e = (v_{S_i}, v_{S_j}) \in \mathcal{E}'$ iff $\exists v_i, v_j \in \mathcal{V}$, such that $v_i \in S_i$, $v_j \in S_j$, where $S_i \neq S_j$, and $e = (v_i, v_j) \in \mathcal{E}$. ■

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 3.1 into a single node.

Definition 10 (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 8. ■

According to Definition 10, all FVPs whose vertices are in the same cycle, and thus in the same SCC of the dependency graph, have the same level.

Nomikos et al. [2005] have proposed the cycle-sum test, i.e., a procedure for deciding if a normal logic program (without functions) where predicates have a time argument is locally stratified. The temporal specifications that can be expressed as logic programs in RTEC$_\circ$ pass the cycle-sum test, and thus are locally stratified.

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

Proof. See Appendix A. □

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.

3.2.3 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 simple FVPs in a cycle (RTEC$_\circ$ does not support cyclic dependencies with statically determined fluents). Algorithms 1 and 2 are an efficient implementation of rules (3.7)–(3.9), i.e., they incorporate an incremental caching technique to avoid unnecessary re-computations, such as those mentioned in Section 3.2.1.

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 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 2–5 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 (3.8) and (3.9)) 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 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 6–9), 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 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 5 (Processing Cyclic Dependencies with RTEC$_\circ$): Consider the stream consisting of events (3.10) and (3.12), while the initial value of $status(m)$ in the current window is $proposed$ (see formula (3.11)). The computation of the initiation points of

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
```

$status(m) = proposed$ commences with the evaluation presented in Table 3.1, with some additional calls, e.g., to 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:

$$\begin{array}{l} \text{holdsAt}(status(m) = proposed,\ t_1). \\ \text{holdsAt}(status(m) = voting,\ t_2). \\ \text{holdsAt}(status(m) = voted,\ t_3). \\ \text{holdsAt}(status(m) = null,\ t_4). \end{array} \tag{3.13}$$

The top part of Table 3.2 (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 3.1, 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 (3.3), 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 (3.13)). 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 4, only to compute again the values of $status$ before t_5.

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

The middle part of Table 3.2 (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 3.2, 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 (3.4), since the intervals of this FVP may be fetched from the cache. This way, the unnecessary re-computations discussed after Example 4 are avoided.

The bottom part of Table 3.2 (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 (3.13) time-point t_3 for which $status(m)=voted$ holds, and directly prove rule (3.6) expressing the conditions in which $status(m)=null$ is initiated. ◊

3.2.4 Correctness and Complexity

We prove the correctness of RTEC$_\circ$ and present its complexity for processing an event description with cyclic dependencies.

Proposition 3 (Correctness of RTEC$_\circ$): RTEC$_\circ$ computes all maximal intervals of the FVPs of an event description with cyclic dependencies, and no other interval. ▲

Proof Sketch. First, we demonstrate that proving the correctness of maximal interval computation for an FVP $F=V$ that is in a cycle reduces to proving the correctness of computing holdsAt($F=V,T$). Second, we prove that holdsAt($F=V,T$) is evaluated correctly at every time-point T of the first window of the stream. To do this, we use an inductive proof on T. Third, we generalise the proof for all subsequent windows. We use an inductive proof on the sequence of the windows of the stream. As a result, we have proven that holdsAt($F=V,T$), where $F=V$ is part of a cycle, is evaluated correctly at every

Table 3.2: Stream reasoning with RTEC. a1–a2 refer to Algorithms 1–2.

	Predicate	Rule/Alg.
1	initiatedAt$(status(m)=proposed,\ q_i{-}\omega,\ T,\ q_i)$?	(3.3)
2	happensAt$(propose(ag_p, m),\ t_5)$ ✓	(3.3), (3.12)
3	holdsAt$(status(m)=null,\ t_5)$?	(3.3), a1
4	processed$(status(m)=null)$ $\times$	a1
5	$last($cachedLEQ$(t_5,\ status(m)=null),\ (t_4,+))$ ✓	a1, (3.13)
6	holdsAtEC$(status(m)=null,\ t_4,\ t_5,\ +)$?	a1, a2
7	brokenAt$(status(m)=null,\ t_4,\ t_4,\ t_5)$ ✓	a2
8	holdsAtEC$(status(m)=null,\ t_4{+}1,\ t_5,\ -)$ $\times$	a2
9	holdsAtEC$(status(m)=null,\ t_4,\ t_5,\ +)$ $\times$	a2, a1
10	updateCache$(status(m)=null,\ t_5,\ -)$ ✓	a1
11	holdsAt$(status(m)=null,\ t_5)$ $\times$	a1, (3.3)
12	initiatedAt$(status(m)=proposed,\ q_i{-}\omega,\ T,\ q_i)$ $\times$	(3.3)
13	initiatedAt$(status(m)=voting,\ q_i{-}\omega,\ T,\ q_i)$?	(3.4)
14	happensAt$(second(ag_s, m),\ t_1)$ ✓	(3.4), (3.10)
15	holdsAt$(status(m)=proposed,\ t_1)$?	(3.4), a1
16	processed$(status(m)=proposed)$ ✓	a1
17	holdsFor$(status(m)=proposed,\ I),\ t_1 \in I$ ✓	a1
18	holdsAt$(status(m)=proposed,\ t_1)$ ✓	a1, (3.4)
19	initiatedAt$(status(m)=voting,\ q_i{-}\omega,\ t_1,\ q_i)$ ✓	(3.4)
20	initiatedAt$(status(m)=null,\ q_i{-}\omega,\ T,\ q_i)$?	(3.6)
21	happensAt$(declare(ag_c, m),\ t_3)$ ✓	(3.6), (3.10)
22	holdsAt$(status(m)=voted,\ t_3)$?	(3.6), a1
23	processed$(status(m)=voted)$ $\times$	a1
24	$last($cachedLEQ$(t_3,\ status(m)=voted),\ (t_3,+))$ ✓	a1, (3.13)
25	holdsAt$(status(m)=proposed,\ t_3)$ ✓	a1, (3.6′)
26	initiatedAt$(status(m)=null,\ q_i{-}\omega,\ t_3,\ q_i)$ ✓	(3.6)

time-point of the stream. Moreover, based on the aforementioned reduction, it follows that RTEC$_\circ$ computes all maximal intervals of the FVPs of an event description with cyclic dependencies, and no other interval.

We provide a sketch for the second part of the proof. In order to prove the inductive step, i.e., prove that holdsAt$(F = V, T)$ is evaluated correctly, provided that holdsAt$(F' = V', T')$, where $T' < T$ and $F' = V'$ is in some cycle that contains $F = V$, is evaluated correctly, we follow three steps. First, we demonstrate that the cached tuple $(T_{lCP}, Tval_{lCP})$ used by Algorithm 1 to prove holdsAt$(F = V, T)$ is correct, i.e., $Tval_{lCP} = +$ if $F = V$ holds at T_{lCP}, and $Tval_{lCP} = -$ if $F = V$ does not hold at T_{lCP}. Second, we prove that Algorithm 2, which is used by RTEC$_\circ$ to decide whether $F = V$ holds at T given the cached tuple $(T_{lCP}, Tval_{lCP})$, operates according to the law of inertia in the Event Calculus. Third, we demonstrate that the initiations and terminations of $F = V$ that are required by Algorithm 2 are evaluated correctly. The full proof is provided in Appendix B. □

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 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 holdsAt$(F = V, T)$ the cache is updated.

Therefore, in the worst-case, $\text{RTEC}_\circ$ will have to evaluate each initiatedAt/terminatedAt rule for $F = V$ at every time-point of the window, but no more than that.

Proposition 4 (Complexity of $\text{RTEC}_\circ$): The cost of evaluating the maximal intervals of an FVP whose definition includes cyclic dependencies is $\mathcal{O}(\omega k)$, where ω is the size of the window and k is the cost of computing whether an FVP is initiated or terminated at a given time-point (see [Artikis et al., 2015] for an estimation of k). ▲

Proof. See Appendix B. □

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, given that we follow an increasing temporal order, 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 $\mathcal{O}(\omega^2 rk)$, 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 $\text{RTEC}_\circ$ leads to significant performance gains (compare the complexity expression presented in Proposition 4 against the cost of processing cyclic dependencies without incremental caching that we outlined above). This optimised processing with caching is the key difference of our proposed computational framework from related work (see our empirical comparison of RTEC with jREC in Section 4.2.1).

3.2.5 Discussion

Several stream reasoning frameworks, such as RTEC, CORE and D^2IA, do not support temporal specifications with cyclic dependencies (see Table 2.3). Moreover, many Event Calculus-based frameworks—e.g., s(CASP), Fusemate, jREC^{fi}, and jREC^{rbt}—are not designed for processing cyclic dependencies, and thus do not feature efficient algorithms for reasoning over such dependencies in a streaming setting. GKL-EC is built specifically for the case of biological feedback loops, and cannot reason with the event descriptions of other applications, such as maritime situational awareness and multi-agent systems. Moreover, GKL-EC is not designed for reasoning over data streams.

Our complexity analysis showed the performance gains of $\text{RTEC}_\circ$. In Chapter 4, we compare $\text{RTEC}_\circ$ with an implementation of jREC, demonstrating that $\text{RTEC}_\circ$ outperforms jREC by about one order of magnitude when reasoning over cyclic dependencies. This result highlights the benefits of the caching mechanism of $\text{RTEC}_\circ$ compared to an Event Calculus-based reasoner without a caching mechanism that is tailored specifically for cyclic dependencies

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, $\text{RTEC}_\circ$ 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.

Aghasadeghi et al. [2024] proposed two incremental algorithms for monitoring the satisfaction of (possibly cyclic) temporal patterns over time-evolving graphs using timed automata. In this context, cyclic patterns comprise sequences of events with time constraints that are mapped onto edges of the graph that form a cycle. In contrast, RTEC$_\circ$ evaluates FVPs whose definitions include cyclic dependencies, i.e., the satisfaction of the FVP at time-point T depends on its truth value at some time-point(s) earlier than T. Moreover, in [Aghasadeghi et al., 2024], the satisfaction of a temporal pattern does not lead to the generation of a new event/situation, and thus hierarchies are not supported.

3.3 RTEC$^\rightarrow$: Reasoning over Streams of Events with Delayed Effects

We propose RTEC$^\rightarrow$, i.e., an extension of RTEC that supports events with delayed effects. We describe the novel language constructs of RTEC$^\rightarrow$, and subsequently study its semantics. Afterwards, we present the reasoning algorithms for handling events with delayed effects. We prove the correctness of the proposed algorithms and outline their complexity.

3.3.1 Motivation

Temporal specifications often include events with delayed effects. In simulations of biological systems, a signal may dictate that the functions of a gene will be switched off at a specified time after the signal took place [Srinivasan et al., 2022]. In multi-agent voting protocols, the chair may be permitted to close the ballot of a motion only after a specified time since the start of the voting procedure [Pitt et al., 2006]. In e-commerce, a contract dictates a set of normative positions with deadlines for the corresponding parties [Hindriks and Riemsdijk, 2013]. In the legal domain, a law may become applicable after a specified time period has passed from its publication [Marín and Sartor, 1999]. In maritime situational awareness, the navigational status of a passenger ship is expected to change from ‘under way’ to ‘moored’ at most by a specified time after the ship started sailing [Montali et al., 2013]. To capture such phenomena, we present RTEC$^\rightarrow$, an extension of RTEC that supports events with delayed effects.

3.3.2 Representation

A key feature of RTEC$^\rightarrow$ is the support of events that may change the values of fluents in the future.

Definition 11 (Event Description in RTEC$^\rightarrow$): An event description in RTEC$^\rightarrow$ extends RTEC event descriptions with the following types of facts:

- fi$(F=V, F=V', R)$, where $F=V$ and $F=V'$ are FVPs with the same fluent, $V \neq V'$, and R is a positive integer, expressing that the initiation of $F=V$ at a time-point T leads to the ***future initiation*** (fi) of $F=V'$ at time-point $T{+}R$, provided that $F=V$ is not 'broken' between T and $T{+}R$. We call R the *delay* of the effects of the event(s) initiating $F=V$. Moreover, if $F=V$ is 'broken' between T and $T{+}R$, then we say that the future initiation of $F=V'$ is *cancelled*.
- p$(F=V)$, expressing that the future initiation specified by fi$(F=V, F=V', R)$ may be *postponed*. For instance, assuming an initiation of $F=V$ at T, a re-initiation of $F=V$ at some time-point T', where $T{<}T'{<}T{+}R$, moves the future initiation of $F=V'$ from $T{+}R$ to $T'{+}R$, provided that $F=V$ is not 'broken' between T' and $T'{+}R$. ■

According to the above definition, in addition to immediate effects, events may have *delayed effects*. Consider a set of events $E_{F=V}$ that appear with positive happensAt predicates in the body of an initiatedAt rule for $F=V$ (rule (3.1)). An immediate effect of the events in $E_{F=V}$ is the initiation of $F=V$, provided that all other conditions, if any, of the initiatedAt rule are satisfied. Moreover, assuming that fi$(F=V, F=V', R)$, the events in $E_{F=V}$ may have delayed effects, in the sense that $F=V'$, where $V' \neq V$, may be initiated in the future.

Example 6 (Expiring Quotes): Consider NetBill, i.e., the e-commerce protocol we described in Section 1.1.3. A quote may be defined with the following rules[1]:

$$\begin{aligned}&\text{initiatedAt}(quote(M, C, G)=\text{true}, T) \leftarrow\\ &\quad \text{happensAt}(present\_quote(M, C, G, P), T).\end{aligned} \tag{3.14}$$

$$\begin{aligned}&\text{initiatedAt}(quote(M, C, G)=\text{false}, T) \leftarrow\\ &\quad \text{happensAt}(accept\_quote(C, M, G), T).\end{aligned} \tag{3.15}$$

In some cases, it may be required that quotes expire after a certain time—expired quotes may not lead to a contract. Moreover, it may be useful to denote when an active quote is close to expiring, in order to warn the consumer that the offer will soon be unavailable. To address these requirements, we may augment rules (3.14) and (3.15) with facts fi$(quote(M, C, G)=\text{true}, quote(M, C, G)=expiring, r_e)$ and fi$(quote(M, C, G)=expiring, quote(M, C, G)=\text{false}, r_f)$. The former fi fact expresses that $present\_quote(M, C, G, P)$, i.e., the event that immediately initiates $quote(M, C, G)=\text{true}$ (see rule (3.14)), leads to a future initiation of $quote(M, C, G)=expiring$, after a delay r_e, unless $quote(M, C, G)=\text{true}$ is 'broken' in the meantime (an $accept\_quote(C, M, G)$ event may 'break' $quote(M, C, G)=\text{true}$ (see rule (3.15))). Based on the latter fi fact, an initiation of $quote(M, C, G)=expiring$ leads to a future initiation of $quote(M, C, G)=\text{false}$ after a delay r_f, unless $quote(M, C, G)=expiring$ is 'broken' in the meantime. In other words, the occurrence of $present\_quote(M, C, G, P)$ at T has the following effects, provided that no future initiation is cancelled: (a) $quote(M, C, G)=\text{true}$ holds in $(T, T{+}r_e]$, (b) $quote(M, C, G)=expiring$ holds in $(T{+}r_e, T{+}r_e{+}r_f]$, and (c) $quote(M, C, G)=\text{false}$ holds after $T{+}r_e{+}r_f$. $\Diamond$

[1]We will omit the second and the fourth arguments of initiatedAt/terminatedAt predicates to simplify the presentation.

A future initiation of $F = V'$ at time-point $T{+}R$, based, e.g., on fi$(F = V, F = V', R)$, implies that $F = V$ is 'broken' at $T{+}R$. In RTEC$^{\rightarrow}$, it is also possible to express the future termination of an FVP $F = V$ without making any assertions about the future value of F, i.e., we become agnostic about the value of F in the future. This is a simpler case than future initiations, and thus not discussed here.

An event description may contain an arbitrary set of fi$(F = V, F = V', R)$ facts. However, it is not meaningful to state in the same event description both fi$(F = V, F = V', R)$ and fi$(F = V, F = V'', R')$, where $V \neq V' \neq V''$. Suppose, e.g., that $F = V$ is initiated at time-point T, $R < R'$ and $F = V$ is not cancelled between T and $T{+}R$. Then, $F = V'$ will be initiated at $T{+}R$, and thus $T{+}R$ will be a termination point of $F = V$. As a result, the future initiation of $F = V''$ at $T{+}R'$ will be cancelled. Therefore, in the case of fi$(F = V, F = V', R)$ and fi$(F = V, F = V'', R')$, only the future initiation with the shorter delay may take place.

In some cases, future initiations may be postponed.

Example 7 (Prolonged Quotes): In NetBill, we may allow for the possibility that a merchant M extends the interval during which a consumer C may accept a quote, and thus establish a contract between M and C. For instance, when $quote(M, C, G) =$ true, M may present another quote to consumer C, e.g., for a different price, with the aim of postponing the expiration of $quote(M, C, G)$. To cater for this possibility, we may add p$(quote(M, C, G) =$ true$)$ in the event description. This way, a re-initiation of $quote(M, C, G) =$ true will postpone its future termination. $\lozenge$

Example 8 (Streams of Events with Delayed Effects): Figure 3.2 visualises streams of events with delayed effects. In examples (a)(ii) and (b)(ii), we have a stream of three events that lead to immediate initiations of an FVP $F = V$. Suppose that these events take place at time-points T_1, T_2 and T_3, where $T_3 - T_1 < R$. According to fi$(F = V, F = V', R)$, there may be future initiations of $F = V'$ at time-points $T_1{+}R$, $T_2{+}R$ and $T_3{+}R$. If p$(F = V)$ is not in the event description, i.e., the future initiations of $F = V'$ may not be postponed (see example (a)(ii)), then we have a future initiation of $F = V'$ at $T_1{+}R$, since $F = V$ is not 'broken' between T_1 and $T_1{+}R$. As a result of this future initiation of $F = V'$, $F = V$ is 'broken' at $T_1{+}R$, and thus all subsequent future initiations of $F = V'$ are cancelled. In contrast, if p$(F = V)$ is in the event description, i.e., the future initiations of $F = V'$ may be postponed (example (b)(ii)), the re-initiation of $F = V$ at T_2 moves the future initiation of $F = V'$ from $T_1{+}R$ to $T_2{+}R$, and the re-initiation of $F = V$ at T_3 moves the future initiation of $F = V'$ from $T_2{+}R$ to $T_3{+}R$. Two further illustrations of streams of events with delayed effects are presented in examples (a)(iii) and (b)(iii), and (a)(iv) and (b)(iv). Examples (a)(i) and (b)(i), and (a)(v) and (b)(v), concern windowing and will be discussed shortly. $\lozenge$

3.3.3 Semantics

fi facts introduce additional dependencies among FVPs. According to fi$(F = V, F = V', R)$, a future initiation of $F = V'$ depends on the initiation of $F = V$. We update the definition of

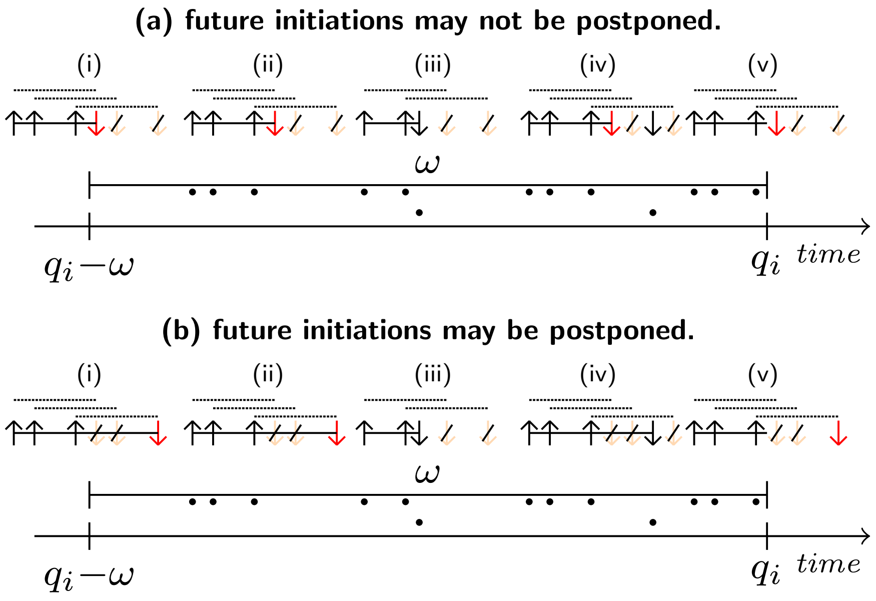


Figure 3.2: Future initiations expressed by fi$(F = V, F = V', R)$ in a streaming setting. ω and q_i denote the window size and the i-th query time. The black points below ω represent the events of the stream. The events in the top (resp. bottom) row lead to immediate initiations (terminations) of $F = V$ (e.g., rules (3.14) and (3.15)). Arrows facing upwards (downwards) indicate initiations (terminations) of $F = V$. Red downward arrows express future terminations of $F = V$, which are the result of future initiations of $F = V'$, while black downward arrows indicate immediate terminations of $F = V$. Faded, crossed-out arrows denote cancelled future terminations of $F = V$, which are due to cancelled future initiations of $F = V'$. The dotted lines above the arrows express the delay R. Solid lines between arrows express the maximal intervals of $F = V$. In (a)(v) and (b)(v), the displayed intervals end at q_i to indicate that they are open, i.e., $F = V$ is not terminated within the current window.

a dependency graph as follows.

Definition 12 (Dependency Graph in RTEC$^{\rightarrow}$): The dependency graph of an event description in RTEC$^{\rightarrow}$ is a directed graph $G = (\mathcal{V}, \mathcal{E}_i \cup \mathcal{E}_f)$, where:

- $\mathcal{V}$ contains one vertex $v_{F=V}$ for each FVP $F = V$.
- $\mathcal{E}_i$ contains an 'i-edge' $(v_{F=V}, v_{F'=V'})$ iff there is an initiatedAt/terminatedAt rule for $F' = V'$ having holdsAt$(F = V, T)$ as one of its conditions.
- $\mathcal{E}_f$ contains an 'f-edge' $(v_{F=V}, v_{F=V'})$ iff there is an fi$(F = V, F = V', R)$ fact, and for every fi$(F = V, F = V'', R')$ fact, where $V' \neq V''$, if any, it holds that $R < R'$. ■

In other words, the edges of RTEC dependency graphs without future initiations (see Definition 7), i.e., the edges expressing immediate initiations/terminations, are now called 'i-edges', in order to distinguish them from the edges expressing future initiations/terminations, i.e., 'f-edges'.

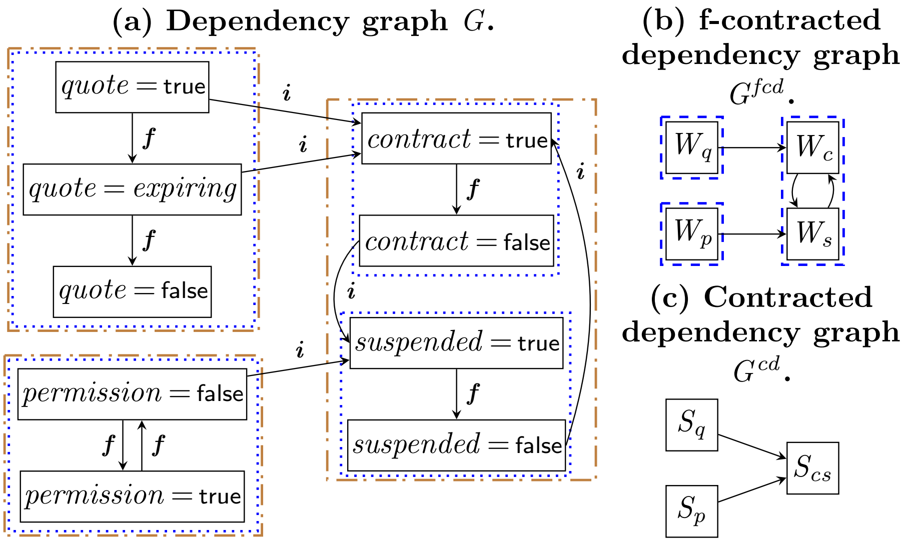


Figure 3.3: (a) The dependency graph G, (b) the f-contracted dependency graph G^{fcd} and (c) the contracted dependency graph G^{cd} of NetBill. For simplicity, we omitted the arguments of fluents, while a vertex v_j is displayed as j. In diagram (a), the blue dotted rectangles denote the weakly connected components of the i-reduced dependency graph G^{ird} of G. The brown dotted-dashed rectangles denote the cd-components of G. In diagram (b), the blue dashed rectangles denote the strongly connected components of G^{fcd}. The vertices of W_q, W_p, W_c and W_s are the result of contracting the weakly connected components of G^{ird}, concerning resp. the FVPs of $quote$, $permission$, $contract$ and $suspended$. In diagram (c), the vertices of S_q, S_p and S_{cs} are produced by contracting the strongly connected components of G^{fcd}, concerning W_q, W_p, and W_c and W_s.

Example 9 (Dependency Graph with Future Initiations): Figure 3.3(a) presents a fragment of the dependency graph of an event description for NetBill. i-edges denote FVP dependencies based on initiatedAt/terminatedAt rules. The i-edges pointing to the vertex of $contract(M, C, G) =$ true, e.g., express that $contract(M, C, G) =$ true may be initiated, i.e., a contract between a merchant M and a consumer C about goods G may come into effect, when $quote(M, C, G) =$ true or $quote(M, C, G) = expiring$, i.e., the quote is still active. (Recall that events, e.g., $accept\_quote(C, M, G)$ that may initiate $contract(M, C, G)$, are not included in dependency graphs.) f-edges model FVP dependencies based on fi facts. The f-edge $(v_{quote(M,C,G)=\text{true}}, v_{quote(M,C,G)=expiring})$, e.g., is due to fi$(quote(M, C, G) =$ true$, quote(M, C, G) = expiring, r_e)$ (Example 6). $\Diamond$

A dependency graph in RTEC$^{\rightarrow}$ may include an f-edge $(v_{F=V}, v_{F=V'})$, according to which, following Definition 8, $F = V'$ has a higher level than $F = V$. However, $F = V$ and $F = V'$ depend on each other, i.e., the initiations of $F = V'$ terminate $F = V$, and vice versa. Therefore, $F = V$ and $F = V'$ should remain in the same level. Moreover, a dependency graph in RTEC$^{\rightarrow}$ may include cycles (see, e.g., Figure 3.3(a)). To address these issues, we have to extend the definition of 'FVP level' (Definition 8), which is necessary for stratifying RTEC$^{\rightarrow}$ programs, and guides the reasoning algorithms. In what follows, we present the steps that lead to the new definition of FVP level. Subsequently, we present the semantics of RTEC$^{\rightarrow}$, and in the following section, we present the reasoning algorithms.

In order to define the level of FVPs in an RTEC$^{\rightarrow}$ event description, we first transform its dependency graph G into the ***f-contracted*** dependency graph G^{fcd}, where the vertices connected by f-edges have been contracted into a single vertex. To do this, we construct the ***i-reduced*** dependency graph G^{ird} of G, which is generated by removing all i-edges from G, and contract the vertices that are connected in G^{ird}.

Definition 13 (f-Contracted Dependency Graph): Given a dependency graph $G=(\mathcal{V},\mathcal{E}_i \cup \mathcal{E}_f)$ of an event description in RTEC$^{\rightarrow}$, the i-reduced dependency graph $G^{ird}=(\mathcal{V},\mathcal{E}_f)$ of G, and the weakly connected components (WCC)s $W_1, W_2, \dots, W_n$ of G^{ird}, the f-contracted dependency graph $G^{fcd}=(\mathcal{V}^{fcd},\mathcal{E}^{fcd})$ of G is defined as follows:

- $\mathcal{V}^{fcd}=\bigcup_{1\leq j\leq n}\{v_{W_j}\}$.
- $(v_{W_j},\ v_{W_k}) \in \mathcal{E}^{fcd}$ iff $\exists v_j, v_k \in \mathcal{V}$, such that $v_j \in W_j$, $v_k \in W_k$ and $(v_j, v_k) \in \mathcal{E}_i$. ■

Example 10 (f-Contracted Dependency Graph): Figure 3.3(b) presents the f-contracted dependency graph of NetBill. The i-reduced dependency graph is the same as the graph of Figure 3.3(a) without the i-edges. The vertices of FVPs of $quote$ and $contract$, e.g., constitute WCCs of the i-reduced dependency graph, denoted by W_q and W_c, and thus have been contracted into vertices v_{W_q} and v_{W_c}, respectively. Moreover, there is an edge (v_{W_q}, v_{W_c}) in the f-contracted dependency graph because the dependency graph includes i-edges from the vertices of $quote$ to a vertex of $contract$. ◊

An f-contracted dependency graph may include cycles (see, e.g., Figure 3.3(b)). To define the FVP level in the presence of cycles, we contract G^{fcd} based on its strongly connected components (SCC)s, generating the ***contracted*** dependency graph G^{cd} of G.

Definition 14 (Contracted Dependency Graph): Given the f-contracted dependency graph $G^{fcd}=(\mathcal{V}^{fcd},\mathcal{E}^{fcd})$ of a dependency graph G in RTEC$^{\rightarrow}$, and the SCCs $S_1, S_2, \dots, S_n$ of G^{fcd}, the contracted dependency graph $G^{cd}=(\mathcal{V}^{cd},\mathcal{E}^{cd})$ of G is defined as follows:

1. $\mathcal{V}^{cd}=\bigcup_{1\leq j\leq n}\{v_{S_j}\}$.
2. $(v_{S_j}, v_{S_k}) \in \mathcal{E}^{cd}$ iff $\exists v_j, v_k \in \mathcal{V}^{fcd}$, such that $v_j \in S_j$, $v_k \in S_k$, where $S_j \neq S_k$, and $(v_j,\ v_k) \in \mathcal{E}^{fcd}$. ■

By construction, G^{cd} is acyclic.

Example 11 (Contracted Dependency Graph): Figure 3.3(c) presents the contracted dependency graph of NetBill. Vertices v_{W_c} and v_{W_s} of the f-contracted dependency graph constitute a SCC, and thus have been contracted into a single vertex $v_{S_{cs}}$, in order to produce the contracted dependency graph. The edges $(v_{S_q}, v_{S_{cs}})$ and $(v_{S_p}, v_{S_{cs}})$ in the contracted dependency graph are due to the edges (v_{W_q}, v_{W_c}) and (v_{W_p}, v_{W_s}) in the f-contracted graph. ◊

We say that the vertices of FVPs that are mapped onto the same vertex in G^{cd} comprise a *cd-component* of G. We call the union of the cd-components of G its *cd-partition*.

Definition 15 (cd-partition and cd-component): Consider a dependency graph G in RTEC$^{\rightarrow}$, the i-reduced dependency graph G^{ird} of G, and the f-contracted dependency graph G^{fcd} of G, where $S_1, \dots, S_n$ are the SCCs of G^{fcd}. $G = G_{S_1} \cup \dots \cup G_{S_n}$ is the cd-partition of G iff, for each $F = V$ such that $v_{F=V} \in G_{S_i}$, $v_{F=V}$ is included in a WCC W of G^{ird} such that v_W is included in SCC S_i of G^{fcd}. We call G_{S_i} a cd-component of G. ■

According to Definition 15, there is a one-to-one mapping between the cd-components of G and the vertices of G^{cd}.

Example 12 (cd-components): Figure 3.3(a) displays the cd-components of the dependency graph of NetBill. We have three cd-components G_{S_q}, G_{S_p} and $G_{S_{cs}}$, containing, respectively, the vertices of $quote$, the vertices of $permission$, and the vertices of $contract$ and $suspended$. ◊

Definition 16 (FVP Level in RTEC$^{\rightarrow}$): Given a dependency graph G in RTEC$^{\rightarrow}$, its cd-components $G_{S_1}, \dots, G_{S_n}$ and the contracted dependency graph G^{cd} of G, the level of an FVP $F = V$, such that $v_{F=V} \in G_{S_i}$, is defined as the level of the vertex v_{S_i} in G^{cd}. ■

Example 13 (FVP Level with Future Initiations): In the contracted dependency graph of NetBill (Figure 3.3(c)), the vertices v_{S_q} and v_{S_p} have level 1, while $v_{S_{sc}}$ has level 2. Therefore, the FVPs of $quote$ and $permission$ have level 1, and the FVPs of $contract$ and $suspended$ have level 2. ◊

The semantics of RTEC$^{\rightarrow}$ remains a locally stratified logic program. The definition of FVP level allows us to arrange the initiatedAt/terminatedAt predicates of FVPs into strata in ascending FVP level order. For FVPs of the same cd-component, and thus the same level, a stratification may be constructed by introducing an additional stratum for each time-point T of the window, in ascending temporal order.

Proposition 5 (Semantics of RTEC$^{\rightarrow}$): An event description in RTEC$^{\rightarrow}$ is a locally stratified logic program. ▲

Proof. See Appendix A. □

3.3.4 Reasoning

One way to handle events with delayed effects would be to employ the following rules:

$$\begin{array}{l} \text{initiatedAt}(F = V',\, T{+}R) \leftarrow \\ \quad \text{fi}(F = V, F = V', R), \\ \quad \text{initiatedAt}(F = V,\, T), \\ \quad \text{not cancelled}(F = V,\, T,\, T{+}R). \end{array} \tag{3.16}$$

$$\begin{array}{l} \text{cancelled}(F = V,\, T,\, T{+}R) \leftarrow \\ \quad \text{brokenBetween}(F = V,\, T,\, T{+}R). \end{array} \tag{3.17}$$

$$\begin{aligned}&\mathsf{cancelled}(F=V,\,T,\,T{+}R) \leftarrow\\ &\quad \mathsf{p}(F=V),\\ &\quad \mathsf{initiatedBetween}(F=V,\,T,\,T{+}R).\end{aligned} \tag{3.18}$$

According to rule (3.16), $F=V'$ will be initiated at $T{+}R$ if an initiation of $F=V$ leads to an initiation of $F=V'$ with delay R, indicated by fi$(F=V, F=V', R)$, $F=V$ is indeed initiated at T, and the future initiation is not 'cancelled' between T and $T{+}R$. Rule (3.17) expresses that the future initiation of $F=V'$ will be cancelled if $F=V$ is 'broken' between T and $T{+}R$, i.e., $F=V$ is terminated or $F=V''$, where $V''\neq V$, is initiated. Rule (3.18) concerns future initiations that may be postponed, as indicated by p$(F=V)$, and expresses that the future initiation of $F=V'$ will be cancelled if $F=V$ is initiated between T and $T{+}R$. In other words, the future initiation of $F=V'$ may be postponed by re-initiating $F=V$.

Unfortunately, rules (3.16)–(3.18) are not suitable for reasoning over data streams. The evaluation of these rules may lead to the computation of initiation points that have already been computed. See, e.g., Figure 3.2(a)(ii), where we have three initiations of $F=V$ at time-points T_1, T_2 and T_3, and a future initiation of $F=V'$ at $T_1{+}R$. When computing the effects of the initiations of $F=V$ at T_2 and T_3, we would have to prove twice more the future initiation of $F=V'$ at $T_1{+}R$, only to cancel the future initiations of $F=V'$ at $T_2{+}R$ and $T_3{+}R$. Furthermore, rules (3.16)–(3.18) are not designed to operate in the presence of windowing, which is quintessential for streaming applications. See, e.g., Figure 3.2(a)(i), where the events initiating $F=V$ take place before the start of the current window, while the future initiations are said to take place after the start of the window.

For these reasons, rules (3.16)–(3.18) are not part of RTEC$^{\rightarrow}$. To address the aforementioned issues, we have developed a caching algorithm that avoids unnecessary re-computations when reasoning over events with delayed effects, and an algorithm identifying the minimal information that needs to be transferred between sliding windows in order to guarantee correctness. These algorithms are presented in Section 3.3.4. Below, in Section 3.3.4, we present a compile-time process computing the optimal FVP processing order in the presence of an arbitrary set of fi relations.

Off-line Reasoning

Based on Definition 16, RTEC$^{\rightarrow}$ can process the FVPs of an event description in a bottom-up fashion, according to their level, caching the FVP intervals derived at each level. This way, when processing an FVP $F=V$ of level m, the intervals of the FVPs of levels lower than m that participate in the definition of $F=V$, if any, may be retrieved from the cache without the need for re-computation. This processing order, however, is not applicable to FVPs of the same level, e.g., FVPs with fi relations.

Example 14 (Processing Quotes): Consider the top-left cd-component G_{S_q} of the dependency graph of NetBill (Figure 3.3(a)). Suppose that $quote(M,C,G)=$true is initiated at time-points T_1 and T_2, and that future initiations are not cancelled. Based on the fi facts (see Example 6), $quote(M,C,G)=expiring$ is initiated at $T_1{+}r_e$ and

$T_2{+}r_e$, and $quote(M, C, G) = $false is initiated at $T_1{+}r_e{+}r_f$ and $T_2{+}r_e{+}r_f$. Processing $quote(M, C, G) = $false first leads to redundant computations. In order to compute the future initiations of $quote(M, C, G) = $false that fall within the window $(q_i{-}\omega, q_i]$, we need to evaluate whether $quote(M, C, G) = expiring$ is initiated at each time-point in $(q_i{-}\omega, q_i{-}r_f]$. In turn, in order to compute the future initiations of $quote(M, C, G) = expiring$ in $(q_i{-}\omega, q_i{-}r_f]$, we need to evaluate the initiations of $quote(M, C, G) = $true at each time-point in $(q_i{-}\omega, q_i{-}r_f{-}r_e]$.

We can avoid redundant computations at time-points where we may not have future initiations by processing the FVPs of G_{S_q} in an order induced by the f-edges of G_{S_q}. This way, we first compute the initiation points T_1 and T_2 of $quote(M, C, G) = $true by evaluating initiatedAt($quote(M, C, G) = $true, T) at each time-point of the window. Next, given the initiation points T_1 and T_2 of $quote(M, C, G) = $true, we can compute the initiations of $quote(M, C, G) = expiring$ at $T_1{+}r_e$ and $T_2{+}r_e$, without considering the remaining time-points of the window. Similarly, given the initiation points $T_1{+}r_e$ and $T_2{+}r_e$ of $quote(M, C, G) = expiring$, we can compute the initiations of $quote(M, C, G) = $false at $T_1{+}r_e{+}r_f$ and $T_2{+}r_e{+}r_f$, without any redundant evaluations. $\diamondsuit$

In order to avoid unnecessary computations, RTEC$^{\rightarrow}$ evaluates and caches the initiations of the FVPs in a cd-component in an order induced by the f-edges.

Definition 17 (FVP Order Relation): Consider the dependency graph G of an event description in RTEC$^{\rightarrow}$ and the cd-components $G_{S_1}, \dots, G_{S_n}$ of G. For a cd-component G_{S_i}, we define the FVP order relation $\prec_{G_{S_i}}$ on the FVPs whose vertices are in G_{S_i} as follows:

- If G_{S_i} is acyclic, then $F = V \prec_{G_{S_i}} F' = V'$ iff there is a path in G_{S_i} that starts from $v_{F\,=\,V}$ and ends at $v_{F'\,=\,V'}$.
- If G_{S_i} contains a cycle, then relation $\prec_{G_{S_i}}$ is empty. ■

In the cd-component G_{S_q} of NetBill, we have $quote = $true $\prec_{G_{S_q}}$ $quote = expiring$ $\prec_{G_{S_q}}$ $quote = $false.

The FVP order relation $\prec_{G_{S_i}}$ is a strict partial order, meaning that some FVPs may not be comparable with $\prec_{G_{S_i}}$, even in an acyclic cd-component.

Example 15 (Partial FVP Order Relation): Consider an event description with fi($F = V_1, F = V_3, R_1$) and fi($F = V_2, F = V_3, R_2$). The dependency graph has a cd-component G_{S_i}, containing the vertices of $F = V_1$, $F = V_2$ and $F = V_3$. Based on f-edges $(v_{F\,=\,V_1}, v_{F\,=\,V_3})$ and $(v_{F\,=\,V_2}, v_{F\,=\,V_3})$, we have $F = V_1 \prec_{G_{S_i}} F = V_3$ and $F = V_2 \prec_{G_{S_i}} F = V_3$. However, there is no directed path connecting vertices $v_{F\,=\,V_1}$ and $v_{F\,=\,V_2}$, and thus $F = V_1$ and $F = V_2$ are not ordered with relation $\prec_{G_{S_i}}$. $\diamondsuit$

Based on $\prec_{G_{S_i}}$, we define a total processing order of FVPs as follows:

Definition 18 (Processing Order of FVPs): Given the FVP order relation $\prec_{G_{S_i}}$ of a cd-component G_{S_i} of a dependency graph in RTEC$^{\rightarrow}$, a processing order of FVPs is any strict total order $\prec^*_{G_{S_i}}$ that is a linear extension of $\prec_{G_{S_i}}$. ■

Example 16 (Processing order of FVPs): For the event description of Example 15, $\prec^*_{G_{S_i}}$

Algorithm 3 processCDComponent

```
    Input: cd-component $G_{S_i}$, cached intervals $I$.
    Output: $I$, including the intervals of the FVPs in $G_{S_i}$.
 1: if $G_{S_i}$ does not contain a cycle then
 2:     for each $v_{F=V}$ in $G_{S_i}$ do $IP[F=V] \leftarrow [\,]$
 3:     for each $v_{F=V}$ in processingOrder($G_{S_i}$) do
 4:         $IP[F=V]$.add(RTEC.evalIP($F=V, I$))
 5:         $TP[F=V] \leftarrow$RTEC.evalTP($F=V, I$)
 6:         if fi($F=V, F=V', R$) then
 7:             $FtIP[F=V'] \leftarrow$evalFI($IP[F=V], TP[F=V], V', R$)
 8:             $TP[F=V]$.add($FtIP[F=V']$)
 9:             $IP[F=V']$.add($FtIP[F=V']$)
10:         $I[F=V] \leftarrow$RTEC.mi($IP[F=V], TP[F=V]$)
11: else if $G_{S_i}$ does not contain an f-edge then
12:     $I$.updateIntervals(RTEC$_\circ$($G_{S_i}, I$))
13: else $I$.updateIntervals(cyclicFI($G_{S_i}, I$))
14: return $I$
```

is consistent with $\prec_{G_{S_i}}$, i.e., $F=V_1 \prec^*_{G_{S_i}} F=V_3$ and $F=V_2 \prec^*_{G_{S_i}} F=V_3$. Moreover, since $F=V_1$ and $F=V_2$ are not comparable with $\prec_{G_{S_i}}$, $\prec^*_{G_{S_i}}$ extends $\prec_{G_{S_i}}$ with $F=V_1 \prec^*_{G_{S_i}} F=V_2$ or $F=V_2 \prec^*_{G_{S_i}} F=V_1$. ◊

The processing order of FVPs is derived at compile-time, in a process transparent to the event description developer. During run-time operations, RTEC$^{\rightarrow}$ utilises this processing order to guide reasoning over FVPs of the same level.

Run-Time Reasoning

RTEC$^{\rightarrow}$ processes the cd-components of a dependency graph level-by-level, and, for each cd-component G_{S_i}, RTEC$^{\rightarrow}$ computes and caches the future initiations and the maximal intervals of the FVPs whose vertices are in G_{S_i}, by following processing order $\prec^*_{G_{S_i}}$. For the dependency graph of NetBill (Figure 3.3(a)), e.g., RTEC$^{\rightarrow}$ processes G_{S_q} or G_{S_p} first, and $G_{S_{cs}}$ last, because the FVPs in G_{S_q} and G_{S_p} have level 1, and the FVPs in $G_{S_{cs}}$ have level 2. Moreover, for cd-component G_{S_q}, RTEC$^{\rightarrow}$ follows $\prec^*_{G_{S_q}}$, computing and caching the future initiations and the intervals of the FVPs in the following order: $quote(M, C, G) = \text{true}$, $quote(M, C, G) = expiring$, and, finally, $quote(M, C, G) = \text{false}$. This way, RTEC$^{\rightarrow}$ does not perform any redundant computation.

RTEC$^{\rightarrow}$ also identifies the attempts of initiating an FVP at some future time-point that need to be transferred between sliding windows, in order to guarantee correctness. Below, we present the key reasoning algorithms of RTEC$^{\rightarrow}$.

Algorithm 3 presents the steps for deriving the maximal intervals of the FVPs in a cd-

Algorithm 4 evalFI

```
    Input: $IP[F=V]$, $TP[F=V]$, $V'$, $R$.
    Global: $q_i, \omega$.
    Output: The future initiations of $F=V'$.
1: $FtIP[F=V'] \leftarrow [\,]$
2: $T_{att} \leftarrow$ findAttempt$(F=V, R)$
3: if $T_{att} \neq null$ and not cancelled$(q_i - \omega,\ T_{att},\ IP[F=V],\ TP[F=V])$ then
4:     $FtIP[F=V']$.add$(T_{att})$
5: for each $T$ in $IP[F=V]$ do
6:     if $T+R \leq q_i$ and not cancelled$(T,\ T+R,\ IP[F=V],\ TP[F=V] \cup FtIP[F=V'])$ then
7:         $FtIP[F=V']$.add$(T+R)$
8: computeAttempts$(F=V, R)$
9: return $FtIP[F=V']$
```

component G_{S_i}. I contains the intervals of FVPs in previously processed cd-components. We start with the case where there is no cycle in G_{S_i} (lines 1–10). RTEC$^{\rightarrow}$ processes the FVPs in G_{S_i} following processing order $\prec^*_{G_{S_i}}$ (line 3), and, for each FVP $F=V$, RTEC$^{\rightarrow}$ employs RTEC (see Section 2.3) to evaluate the initiatedAt and terminatedAt rules of $F=V$ (lines 4–5). The derived initiation points of $F=V$ are added in list $IP[F=V]$, while list $TP[F=V]$ contains the derived termination points. Next, based on fi$(F=V, F=V', R)$, RTEC$^{\rightarrow}$ evaluates the future initiations of $F=V'$, as they constitute terminations of $F=V$ (line 7). The details of this process will be presented shortly. RTEC$^{\rightarrow}$ stores the time-points of these future initiations in the auxiliary list $FtIP[F=V']$, and adds the items of $FtIP[F=V']$ to $TP[F=V]$. Moreover, RTEC$^{\rightarrow}$ adds the items of $FtIP[F=V']$ to $IP[F=V']$; this way, the future initiations of $F=V'$ will be available to RTEC$^{\rightarrow}$ when processing $F=V'$. At this point, lists $IP[F=V]$ and $TP[F=V]$ contain all initiation and termination points of $F=V$, and no other time-points (see Section 3.3.5 for the correctness proof). RTEC$^{\rightarrow}$ is then able to construct the maximal intervals of $F=V$ (see RTEC.mi in line 10), by pairing initiation and termination points (see Section 2.3).

A cd-component G_{S_i} may contain cycles (see, e.g., G_{S_p} and $G_{S_{cs}}$ in Figure 3.3(a)). If G_{S_i} contains cycles without f-edges, then RTEC$^{\rightarrow}$ employs RTEC$_{\circ}$ (see Section 3.2) to compute the intervals of FVPs in G_{S_i} (see line 12 of Algorithm 3). For cycles with at least one f-edge (line 13), RTEC$^{\rightarrow}$ employs an extended version of the algorithms of RTEC$_{\circ}$, handling future initiations in the presence of cycles (see Algorithm 13 in Appendix B).

In the following, we describe the steps of RTEC$^{\rightarrow}$ for computing the future initiations of $F=V'$, that lead to terminations of $F=V$. Due to windowing, it is possible for an initiation of $F=V$ to take place at a time-point T of the current window, while the corresponding future initiation of $F=V'$ at $T+R$ falls outside the window. In such cases, RTEC$^{\rightarrow}$ generates and caches an attempt event att that may mark a future initiation $F=V'$ at $T+R$. At each query time, RTEC$^{\rightarrow}$ evaluates the future initiations of $F=V'$ based on the cached attempt events, and the events that fall within the window. Algorithm 4 presents the reasoning process. First, RTEC$^{\rightarrow}$ processes the attempt events cached at the previous query

Algorithm 5 findAttempt

```
    Input: $F = V, R$
    Global: $q_i, \omega, Attempts[F = V], I_{q_{i-1}}[F = V]$
    Output: $T_{att}$
 1: $T_{att} \leftarrow null$
 2: if $q_i{-}\omega{+}1$ in $I_{q_{i-1}}[F = V]$ then
 3:    if p$(F = V)$ then
 4:       for each $T$ in $Attempts[F = V]$ do
 5:          if $T > q_i{-}\omega$ and $T{-}R \leq q_i{-}\omega$ then $T_{att} \leftarrow T$
 6:          else if $T{-}R > q_i{-}\omega$ then return $T_{att}$
 7:    else
 8:       $[T_s, T_f) \leftarrow getIntervalContainingTimepoint(I_{q_{i-1}}[F = V], q_i{-}\omega{+}1)$
 9:       for each $T_{att}$ in $Attempts[F = V]$ do
10:          if $T_{att} > q_i{-}\omega$ and $T_{att}{-}R == T_s$ then return $T_{att}$
11: return $T_{att}$
```

time, in order to determine which one of them, if any, may mark a future initiation of $F = V'$ at the current query time (line 2 of Algorithm 4, and Algorithm 5). Second, RTEC$^{\rightarrow}$ computes the future initiations of $F = V'$, based on the selected attempt event, if any, and the events that took place inside the window (lines 3–7 of Algorithm 4). Third, RTEC$^{\rightarrow}$ computes and caches attempt events that may mark future initiations of $F = V'$ at a future query time (line 8 of Algorithm 4, and Algorithm 6). In what follows, we present each of these three steps.

Processing attempt events from the previous query time. RTEC$^{\rightarrow}$ decides which one of the attempt events that were computed and cached at the previous query time q_{i-1}, if any, may mark a future initiation of $F = V'$ at q_i. First, we make sure that the future initiation of $F = V'$ was not cancelled before the current window by checking whether $F = V$ holds at the start of the window. This is done by verifying that the list of intervals computed for $F = V$ at the previous query time q_{i-1} contains the start of the window (line 2 of Algorithm 5). If it holds, then we keep an attempt event such that the corresponding initiation of $F = V$ takes place before $q_i{-}\omega$. If p$(F = V)$ is in the event description, then we keep the most recent such event (lines 3–6). Otherwise, we choose the earliest one (lines 7–10). Note that it is not possible to determine at q_{i-1} which attempt event will be used at q_i, because we make no assumptions about the size of the window at q_i.

Example 17 (Processing attempt events from the previous query time): Consider Figures 3.2(a)(i) and (b)(i). For each initiation of $F = V$ at T, that took place before the current window, with $T{+}R$ falling within the window, we have a cached attempt event at $T{+}R$. These events are displayed by red and faded arrows. The red arrow denotes the attempt event att that is kept, while the faded arrows denote the attempt events that are discarded. In (a)(i), future initiations of $F = V'$ may not be postponed, and thus att takes place R time-points after the first initiation of $F = V$. In (b)(i), att takes place R time-points after the last initiation of $F = V$, since future initiations of $F = V'$ may be postponed. $\diamondsuit$

Computing future initiations. After determining which cached attempt event att, if any, may mark a future initiation of $F = V'$, RTEC$^{\rightarrow}$ computes the future initiations of $F = V'$ that fall within the current window. First, RTEC$^{\rightarrow}$ evaluates whether att marks a future initiation of $F = V'$ by checking whether the future initiation of $F = V'$ is cancelled between the start of the window and the time-stamp T_{att} of att (see lines 3–4 of Algorithm 4). Second, for each initiation of $F = V$ at T, RTEC$^{\rightarrow}$ computes a future initiation of $F = V'$ at $T{+}R$, provided that $T{+}R$ falls inside the window and the future initiation of $F = V'$ is not cancelled between T and $T{+}R$ (lines 5–7). $IP[F = V]$, like all lists in RTEC$^{\rightarrow}$, is temporally sorted; thus, RTEC$^{\rightarrow}$ examines the initiations of $F = V$ in ascending temporal order. Lines 5–7 are an efficient implementation of rules (3.16)–(3.18). In order to identify an initiation of $F = V$ that may lead to a future initiation of $F = V'$, RTEC$^{\rightarrow}$ examines the list of cached initiation points of $F = V$, without requiring any rule computations (contrast line 5 of Algorithm 4 with the second condition of rule (3.16)). Moreover, in order to evaluate whether a future initiation of $F = V'$ is cancelled between T and $T{+}R$, RTEC$^{\rightarrow}$ checks if there is a cached initiation or termination of $F = V$ between T and $T{+}R$ that cancels the future initiation of $F = V'$. Instead, rules (3.17)–(3.18) evaluate initiations and terminations of $F = V$ that may have been previously computed.

Example 18 (Future initiations within window): Consider example (a)(ii) of Figure 3.2. RTEC$^{\rightarrow}$ computes and caches the initiations of $F = V$ at T_1, T_2 and T_3. RTEC$^{\rightarrow}$ then determines whether we have future initiations of $F = V'$ at $T_1{+}R$, $T_2{+}R$ and $T_3{+}R$ (see lines 5–7 of Algorithm 4). RTEC$^{\rightarrow}$ starts with the initiation point of $F = V$ at T_1, and computes a future initiation of $F = V'$ at $T_1{+}R$, because, according to the cached points, the future initiation of $F = V'$ is not cancelled between T_1 and $T_1{+}R$. As a result, $T_1{+}R$ is cached in list $FtIP[F = V']$. Next, RTEC$^{\rightarrow}$ determines that the future initiations of $F = V'$ at $T_2{+}R$ and $T_3{+}R$ are cancelled, by retrieving $T_1{+}R$ from $FtIP[F = V']$.

In example (a)(iii), $F = V$ is initiated twice, at time-points T_1 and T_2, and terminated once, at T_3. For the initiation of $F = V$ at T_1 (resp. T_2), RTEC$^{\rightarrow}$ determines that the future initiation of $F = V'$ at $T_1{+}R$ ($T_2{+}R$) is cancelled by verifying that the cached termination point T_3 of $F = V$ is between T_1 (T_2) and $T_1{+}R$ ($T_2{+}R$) (see lines 5–7 of Algorithm 4). Example (a)(iv) is similar to (a)(iii), with the exception that the termination point of $F = V$ takes place between the first and the last future initiation of $F = V'$. In this case, RTEC$^{\rightarrow}$ detects that only the first future initiation of $F = V'$ takes place, as in (a)(ii). In examples (b)(ii)–(iv), the future initiations of $F = V'$ may be postponed by a re-initiation of $F = V$. In (b)(ii), e.g., RTEC$^{\rightarrow}$ determines that the future initiations of $F = V'$ at $T_1{+}R$ and $T_2{+}R$ are cancelled, because they are postponed by the cached initiation T_3 of $F = V$ (lines 5–7 of Algorithm 4). Thus, we have a future initiation of $F = V'$ at $T_3{+}R$. $\Diamond$

Computing attempt events for the next query time. In the last step at q_i, RTEC$^{\rightarrow}$ derives the attempt events that may lead to future initiations of $F = V'$ at the next query time q_{i+1}. Recall that an attempt event takes place R time-points after the initiation of $F = V$. Moreover, we make no assumptions about the size of the window at q_{i+1}. If future initiations of $F = V'$ may be postponed, then every attempt event may mark an initiation of $F = V'$ at q_{i+1}, and is therefore cached (lines 2–4 of Algorithm 6). Otherwise, if future initiations of $F = V'$ may not be postponed, then we cache only the attempt events that take place R

Algorithm 6 computeAttempts

```
   Input: $F = V, R$
   Global: $I[F = V], IP[F = V]$
   Output: $Attempts[F = V]$
1: $Attempts[F = V] \leftarrow [\,]$
2: if p$(F = V)$ then
3:    for each $T$ in $IP[F = V]$ do
4:       $Attempts[F = V]$.add($T{+}R$)
5: else
6:    for each $[T_s, T_f)$ in $I[F = V]$ do
7:       $Attempts[F = V]$.add($T_s{+}R$)
8: cache($Attempts[F = V]$)
```

time-points after the starting point of a maximal interval of $F = V$ (lines 5–7).

Example 19 (Computing attempt events for the next query time): Consider examples (a)(v) and (b)(v) of Figure 3.2. In the case of (a)(v), only the attempt event indicated by the red arrow will be cached, while in (b)(v) the attempt events denoted by the red arrow and the faded ones will be cached. If we assume non-overlapping windows in the case of (b)(v), i.e., the next window starting at q_i, then the attempt events of the faded arrows will be discarded at q_{i+1}. However, if the next window starts between two initiations denoted by upward arrows, we will discard all attempt events corresponding to initiations that fall within the next window, because these initiations may be invalidated in the presence of the events that arrive in $(q_i, q_{i+1}]$ and have a time-stamp in $(q_{i+1}{-}\omega, q_i]$. $\Diamond$

3.3.5 Correctness and Complexity

RTEC$^{\rightarrow}$ guarantees correct FVP interval computation provided that the window size is sufficient to tolerate the lags, if any, in the arrival of events. In other words, for each event E with time-stamp T, there is a query time q_i, such that E has arrived by q_i and $T \in (q_i{-}\omega, q_i]$.

Proposition 6 (Correctness of RTEC$^{\rightarrow}$): RTEC$^{\rightarrow}$ computes all maximal intervals of the FVPs of an event description, and no other interval. ▲

Proof Sketch. We present a sketch of the proof, focusing on acyclic dependency graphs. The complete proof, including cyclic graphs, may be found in Appendix B. Assume an event description with fi$(F = V, F = V', R)$. The proof consists of three steps. First, we prove that RTEC$^{\rightarrow}$ transfers the attempt events between windows that are required for correct FVP interval computation, and no other attempt event. Second, we show that RTEC$^{\rightarrow}$ computes all future initiations of $F = V'$, and no other future initiation. Third, at the time of computing the maximal intervals of an FVP, the cache contains all necessary initiations and terminations of the FVP, and no other initiation/termination.

We present the proof of the second step. At the start of future initiation computation, the cache contains all initiations and immediate terminations of $F=V$, and no other initiation or immediate termination of $F=V$. Suppose that $T_1,\dots,T_k$ are the temporally sorted initiations of $F=V$. We show that RTEC$^{\rightarrow}$ computes a future initiation of $F=V'$ at $T_i{+}R$, where $1\leq i\leq k$, iff the future initiation of $F=V'$ is not cancelled between T_i and $T_i{+}R$. For the base case, according to lines 5–7 of Algorithm 4, RTEC$^{\rightarrow}$ computes a future initiation of $F=V'$ at $T_1{+}R$ iff there is no cancellation point between T_1 and $T_1{+}R$. Note that there are no initiations of $F=V$ before T_1, and thus no future initiations of $F=V'$ before $T_1{+}R$. In other words, the cache has all cancellation points of the future initiation of $F=V'$ between T_1 and $T_1{+}R$. Next, assume that RTEC$^{\rightarrow}$ has evaluated correctly the future initiations of $F=V'$ up to $T_{n-1}{+}R$, where $1<n\leq k$. Since RTEC$^{\rightarrow}$ caches the future initiations that it derives (line 7 of Algorithm 4), the cache contains all cancellation points of the future initiations of $F=V'$ up to $T_{n-1}{+}R$. As a result, RTEC$^{\rightarrow}$ computes a future initiation of $F=V'$ at $T_n{+}R$ iff there is no cancellation point between T_n and $T_n{+}R$.

The above proof may be generalised to a cd-component with an arbitrary set of f-edges. There are two cases. First, we may have a 'chain' of n f-edges $(v_{F=V_1},v_{F=V_2}),(v_{F=V_2},v_{F=V_3}),\dots,(v_{F=V_n},v_{F=V_{n+1}})$. Second, we may have a vertex $v_{F=V_m}$ with n incoming f-edges $(v_{F=V_1},v_{F=V_m}),\dots,(v_{F=V_n},v_{F=V_m})$ (see Example 15). More composite cd-components may be built by combining these two structures. In both cases, at the time of evaluating the future initiation of an FVP, the cache contains all the cancellation points that are necessary for correct computation. In the former case, based on Definition 18, RTEC$^{\rightarrow}$ processes the FVPs in the following order: $F=V_1, F=V_2,\dots,F=V_{n+1}$. As a result, when processing $F=V_k$, where $1<k\leq n{+}1$, the cache of RTEC$^{\rightarrow}$ contains all initiations and immediate terminations of $F=V_{k-1}$. At this point, the future initiations of $F=V_{k+j}$, where $1\leq j\leq n{-}k{+}1$, are not required, because they do not terminate $F=V_{k-1}$, and thus do not cancel the future initiations of $F=V_k$. In general, a future initiation of $F=V_{k+j}$, based on f-edge $(v_{F=V_{k+j-1}},v_{F=V_{k+j}})$, terminates $F=V_{k+j-1}$, and no other FVP. In the latter case, RTEC$^{\rightarrow}$ processes FVPs $F=V_1,\dots,F=V_n$ before $F=V_m$. As a result, when processing $F=V_m$, the cache of RTEC$^{\rightarrow}$ contains all the initiations and immediate terminations of FVPs $F=V_1,\dots,F=V_n$. Therefore, in both cases, the cache of RTEC$^{\rightarrow}$ contains the necessary information for correct future initiation computation. □

Recall that RTEC$^{\rightarrow}$ processes the cd-components of a dependency graph level-by-level. The proposition below presents the complexity of processing a cd-component.

Proposition 7 (Complexity of RTEC$^{\rightarrow}$): The cost of evaluating the future initiations of the FVPs in a cd-component is $\mathcal{O}(n_v(\omega{-}R)log(\omega))$, where n_v is the number of possible values of the FVPs, ω is the window size and R is the delay of a future initiation. ▲

Proof. See Appendix B. □

Proposition 7 presents the cost of RTEC$^{\rightarrow}$ in the worst-case, where we have an initiation/termination of an FVP at each time-point of the window. In practice, the number k of initiations and terminations of an FVP is much smaller than the window size ω, and thus the cost of RTEC$^{\rightarrow}$, i.e., $\mathcal{O}(n_v klog(k))$, is significantly smaller. Note that the benefits of our approach, as compared to frameworks that lack the caching techniques of RTEC$^{\rightarrow}$, are considerable. The cost, e.g., of reasoning with rules (3.16)–(3.18) is exponential in $\omega - R$. We illustrate the benefits of RTEC$^{\rightarrow}$ compared to frameworks without caching in our empirical analysis (see Chapter 4).

3.3.6 Discussion

Table 2.3 outlined several stream reasoning frameworks, most of which do not support events with delayed effects. GKL-EC and s(CASP) may express events with delayed effects using the trajectory predicate. trajectory is restricted to non-inertial fluents and does not capture future effects that may be postponed [Miller and Shanahan, 2002]. In the case of jREC^{fi}, the algorithms used for handling events with delayed effects do not utilise an optimised processing order for FVPs, and do not feature caching, leading to reasoning inefficiencies. We present an empirical comparison of RTEC$^{\rightarrow}$ against GKL-EC, s(CASP) and jREC^{fi} on event descriptions with events with delayed effects in Chapter 4.

RTEC$^{\rightarrow}$ is motivated by the work of Miller and Shanahan [2002], where a first-order logic Event Calculus dialect was extended to support events with delayed effects. Several other formalisms support such events, including the action language presented in [Karlsson et al., 1998], an Event Calculus-based formalism [Marín and Sartor, 1999], and multi-agent system specification languages expressing normative positions with deadlines [Hindriks and Riemsdijk, 2013; Demolombe, 2014; Chopra et al., 2020]. There are several differences between RTEC$^{\rightarrow}$ and these approaches. Delayed effects that may be postponed, e.g., are not supported in [Karlsson et al., 1998; Marín and Sartor, 1999; Miller and Shanahan, 2002; Demolombe, 2014; Chopra et al., 2020], while the formalisms presented in [Marín and Sartor, 1999; Miller and Shanahan, 2002; Hindriks and Riemsdijk, 2013; Demolombe, 2014; Chopra et al., 2020] do not support delayed effects that may lead to further delayed effects (see Example 6). Moreover, none the aforementioned approaches features reasoning techniques capable of handling streaming applications.

3.4 RTEC$_A$: Reasoning over Specifications with Allen Relations

We present RTEC$_A$, i.e., an extension of RTEC that supports event descriptions containing the relations of Allen's Interval Algebra. We describe the language constructs of RTEC$_A$ and outline its semantics. Moreover, we provide reasoning algorithms that handle temporal patterns with Allen relations, demonstrate the correctness of our proposed algorithms and outline their complexity.

Table 3.3: Output modes of the allen construct.

outMode	**Output list I**
source	$I = \mathcal{S}_{\mathsf{rel}}$
target	$I = \mathcal{T}_{\mathsf{rel}}$
union	union_all$([\mathcal{S}_{\mathsf{rel}}, \mathcal{T}_{\mathsf{rel}}], I)$
intersect	intersect_all$([\mathcal{S}_{\mathsf{rel}}, \mathcal{T}_{\mathsf{rel}}], I)$
complement	relative_complement_all$(\mathcal{S}_{\mathsf{rel}}, [\mathcal{T}_{\mathsf{rel}}], I)$
complement_inv	relative_complement_all$(\mathcal{T}_{\mathsf{rel}}, [\mathcal{S}_{\mathsf{rel}}], I)$

3.4.1 Motivation

Allen's relations have proven necessary for expressing the temporal specifications of contemporary applications [Körber et al., 2021; Awad et al., 2022]. However, as mentioned in Section 2.3, the interval manipulation constructs of RTEC cannot express the relations of Allen's algebra. Consider, e.g., the computation of the interval pairs (i^s, i^t), where $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, satisfying before. intersect_all$([\mathcal{S}, \mathcal{T}], [\,])$ states that for every interval pair (i^s, i^t), such that $i^s \in \mathcal{S}$, $i^t \in \mathcal{T}$, it holds that $i^s \cap i^t = \emptyset$. Therefore, i^s is before i^t, or vice versa. It is impossible, however, to distinguish between the two cases. To address this issue, we present an extension of RTEC that supports temporal specifications with Allen relations.

3.4.2 Representation

We cannot express Allen relations in RTEC without extending its expressive power. Simple fluent definitions evaluate fluent initiation and termination conditions on a particular time instant, and thus do not support interval endpoint comparisons. Moreover, as already mentioned, the interval manipulation constructs in statically determined fluent definitions cannot express Allen relations. To address this issue, we extend the statically determined fluent definitions.

Definition 19 (Syntax of statically determined fluent definitions in RTEC$_{\mathbf{A}}$): A holdsFor$(F = V, I)$ rule defining a statically determined fluent F may additionally contain body predicates in the form of allen$(\mathsf{rel}, \mathcal{S}, \mathcal{T}, \mathsf{outMode}, I)$, where rel denotes an Allen relation, $\mathcal{S}$ and $\mathcal{T}$ are input lists of maximal intervals, outMode expresses how we should treat the interval pairs (i^s, i^t) satisfying rel, where $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, and I is the output list of maximal intervals. ■

According to allen$(\mathsf{rel}, \mathcal{S}, \mathcal{T}, \mathsf{outMode}, I)$, I contains the maximal intervals produced by applying outMode to the interval pairs of the 'source list' $\mathcal{S}$ and the 'target list' $\mathcal{T}$ satisfying rel, i.e., one of the Allen relations presented in Table 2.2. The inverse relations may be computed by reversing the order of the input lists. outMode is applied to the intervals of lists $\mathcal{S}_{\mathsf{rel}} = \{i^s \mid i^s \in \mathcal{S} \wedge \exists i^t \in \mathcal{T} : \mathsf{rel}(i^s, i^t)\}$ and $\mathcal{T}_{\mathsf{rel}} = \{i^t \mid i^t \in \mathcal{T} \wedge \exists i^s \in \mathcal{S} : \mathsf{rel}(i^s, i^t)\}$, i.e., the

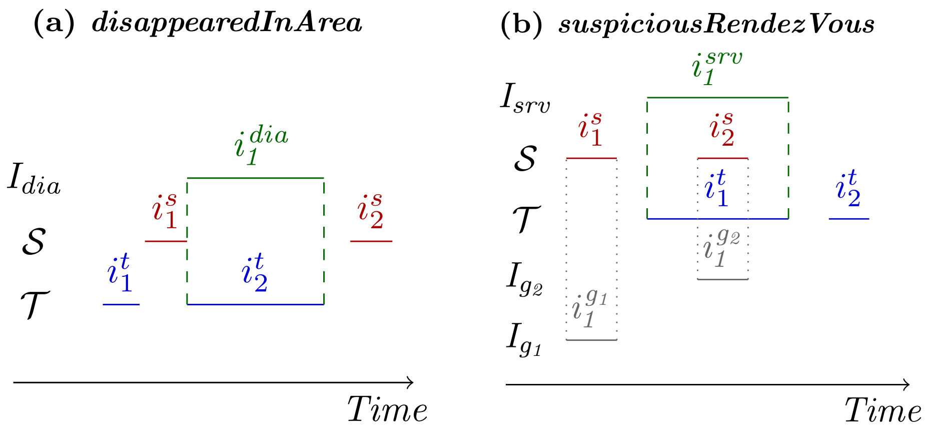


Figure 3.4: Maximal interval computation with the allen construct.

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 3.3. Below, we illustrate the use of the allen predicate in fluent definitions for maritime situational awareness (see Section 1.1.3 for a description of this application).

Example 20 (Allen relations for maritime situational awareness): Vessels often attempt to conceal illegal activities in certain areas, such as fishing in fisheries restricted areas, by stopping transmitting their position. See the rule below:

$$\begin{aligned}&\mathsf{holdsFor}(\mathit{disappearedInArea}(\mathit{Vl}, \mathit{AreaType}) = \mathsf{true}, I_{dia}) \leftarrow \\ &\quad \mathsf{holdsFor}(\mathit{withinArea}(\mathit{Vl}, \mathit{AreaType}) = \mathsf{true}, \mathcal{S}), \\ &\quad \mathsf{holdsFor}(\mathit{gap}(\mathit{Vl}) = \mathit{farFromPorts}, \mathcal{T}), \\ &\quad \mathsf{allen}(\mathsf{meets}, \mathcal{S}, \mathcal{T}, \mathsf{target}, I_{dia}).\end{aligned} \tag{3.19}$$

$\mathit{disappearedInArea}(\mathit{Vl}, \mathit{AreaType})$ is a statically determined Boolean fluent defined in terms of $\mathit{withinArea}(\mathit{Vl}, \mathit{AreaType})$, i.e., a simple fluent expressing that vessel Vl is in an area of type $\mathit{AreaType}$, and $\mathit{gap}(\mathit{Vl})$, i.e., a multi-valued fluent expressing the intervals during which vessel Vl stopped transmitting its position. The specification of gap is available with the complete event description of maritime situational awareness. The last condition of rule (3.19) expresses the meets Allen relation. $\mathsf{allen}(\mathsf{meets}, \mathcal{S}, \mathcal{T}, \mathsf{target}, I_{dia})$ states that from the interval pairs (i^s, i^t) satisfying meets, where $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, we will keep in the output list I_{dia} the target intervals i^t (see the second line of Table 3.3). According to rule (3.19), therefore, a vessel Vl is said to disappear in an area of $\mathit{AreaType}$ during an interval i^{dia}, if i^{dia} is an interval during which $\mathit{gap}(\mathit{Vl}) = \mathit{farFromPorts}$, i.e., Vl stopped transmitting its position while being in the open sea, and i^{dia} is met by an interval i^s during which Vl was within an area of $\mathit{AreaType}$. Figure 3.4(a) provides a graphical illustration. In this illustration, the only interval pair satisfying meets is (i_1^s, i_2^t), and thus $I_{dia} = [i_2^t]$. If we wanted to include i_1^s in the output I_{dia}, then we would have replaced target by union in the last condition of rule (3.19) (see the third line of Table 3.3). This way, i_1^s would be amalgamated with i_2^t producing a single interval, i.e., $I_{dia} = [i_1^s \cup i_2^t]$. In any case, the interval manipulation constructs of RTEC cannot express $\mathit{disappearedInArea}$. For instance, relative_complement_all$(\mathcal{T}, [\mathcal{S}], I)$ would discard the common time-point of i^s_1 and i^t_2, and would include i^t_1, which does not satisfy meets. Similarly, union_all$([\mathcal{S}, \mathcal{T}], I_{dia})$ would include all intervals of $\mathcal{S}$ and $\mathcal{T}$, which is incorrect.

Proximate vessels may stop transmitting their position to conduct illegal activities, such as an illegal cargo transfer. Consider the formalisation below:

$$\begin{aligned}
&\mathsf{holdsFor}(suspiciousRendezVous(Vl_1, Vl_2)=\mathsf{true}, I_{srv}) \leftarrow \\
&\quad \mathsf{holdsFor}(gap(Vl_1)=farFromPorts, I_{g_1}), \\
&\quad \mathsf{holdsFor}(gap(Vl_2)=farFromPorts, I_{g_2}), \\
&\quad \mathsf{holdsFor}(proximity(Vl_1, Vl_2)=\mathsf{true}, \mathcal{T}), \\
&\quad \mathsf{union\_all}([I_{g_1}, I_{g_2}], \mathcal{S}), \mathsf{allen}(\mathsf{during}, \mathcal{S}, \mathcal{T}, \mathsf{target}, I_{srv}).
\end{aligned} \tag{3.20}$$

$suspiciousRendezVous(Vl_1, Vl_2)$ is a statically determined fluent, and $proximity(Vl_1, Vl_2)$ is a Boolean fluent denoting whether two vessels, Vl_1 and Vl_2, are close to each other. $\mathcal{T}$, i.e., the list of maximal intervals during which two vessels are close to each other, is derived by an online spatial processing technique on vessel positional signals, which is robust to interim signal gaps [Santipantakis et al., 2018]. union_all in rule (3.20) derives $\mathcal{S}$, i.e., the list of maximal intervals during which $gap(Vl_1)=farFromPorts$ or $gap(Vl_2)=farFromPorts$. Then, allen$(\mathsf{during}, \mathcal{S}, \mathcal{T}, \mathsf{target}, I_{srv})$ identifies the maximal intervals of $\mathcal{T}$ that contain an interval of $\mathcal{S}$ and stores them in list I_{srv}. Therefore, rule (3.20) specifies that vessels Vl_1 and Vl_2 may be conducting an illegal activity during an interval i^{srv}, if i^{srv} is an interval during which Vl_1 and Vl_2 are close to each other, and i^{srv} contains an interval i^s during which at least one of the vessels stops transmitting its position. Figure 3.4(b) displays an illustration of rule (3.20). union_all constructs list $\mathcal{S}$ as the union of the intervals in lists I_{g_1} and I_{g_2}. Among the intervals of $\mathcal{S}$ and $\mathcal{T}$, during is only satisfied for the interval pair (i^s_2, i^t_1), resulting in $I_{srv}=[i^t_1]$. Note that I_{srv} cannot be derived by the interval manipulation constructs of RTEC. union_all$([\mathcal{S}, \mathcal{T}], I)$, e.g., would compute list $I=[i^s_1, i^t_1, i^t_2]$, including intervals i^s_1 and i^t_2 that do not satisfy during. $\diamondsuit$

As mentioned earlier, Table 3.3 lists the possible values of outMode and their meaning. Consider the computation of allen$(\mathsf{meets}, \mathcal{S}, \mathcal{T}, \mathsf{outMode}, I_{dia})$ in the example of Figure 3.4(a), where the only interval pair satisfying meets is (i^s_1, i^t_2). If outMode was complement or complement_inv, we would apply the relative_complement_all construct on i^s_1 and i^t_2 (complement_inv reverses the order of operants in relative_complement_all) and compute, respectively, list I_{dia} as $[i^s_1 \setminus i^t_2]$ or $[i^t_2 \setminus i^s_1]$. Note that some combinations of rel and outMode are equivalent. For instance, if an interval pair (i^s, i^t) satisfies during, then i^s is a sub-interval of i^t. Therefore, allen$(\mathsf{during}, \mathcal{S}, \mathcal{T}, \mathsf{union}, I)$ and allen$(\mathsf{during}, \mathcal{S}, \mathcal{T}, \mathsf{target}, I)$ produce the same output list.

3.4.3 Semantics

An event description in RTEC$_\mathrm{A}$ defines a dependency graph (see Definition 7). According to Definition 7, the addition of allen constructs in statically determined fluents definitions does not introduce additional dependencies among FVPs. Therefore, our extension of RTEC does not affect its semantics.

Algorithm 7 allen(rel, $\mathcal{S}$, $\mathcal{T}$, outMode, I)

```
1: $\mathcal{S}^c, \mathcal{T}^c \leftarrow retrieveCachedIntervals(\ )$
2: $\mathcal{S} \leftarrow append(\mathcal{S}^c, \mathcal{S}),\quad \mathcal{T} \leftarrow append(\mathcal{T}^c, \mathcal{T})$
3: $pairs \leftarrow compute\_allen\_relation(\mathcal{S}, \mathcal{T}, \mathsf{rel})$
4: $\mathcal{S}_{\mathsf{rel}}, \mathcal{T}_{\mathsf{rel}} \leftarrow getSourceTargetIntervals(pairs)$
5: $I \leftarrow applyOutMode(\mathcal{S}_{\mathsf{rel}}, \mathcal{T}_{\mathsf{rel}}, \mathsf{outMode})$
6: $windowing(\mathcal{S}, \mathcal{T}, \mathsf{rel})$
```

Algorithm 8 $compute\_allen\_relation(\mathcal{S}, \mathcal{T}, \mathsf{rel})$

```
1: $p_s \leftarrow 1$, $p_t \leftarrow 1$, $pairs \leftarrow [\,]$
2: while $p_s \leq length(\mathcal{S})$ and $p_t \leq length(\mathcal{T})$ do
3:     $i^s \leftarrow \mathcal{S}[p_s]$, $i^t \leftarrow \mathcal{T}[p_t]$
4:     if $\mathsf{rel} = \mathsf{before}$ then
5:         if $\mathsf{before}(i^s, i^t)$ then
6:             $pairs.add((i^s, [i^t, \ldots, \mathcal{T}[length(\mathcal{T})]]))$
7:         else if $f(i^s) \leq f(i^t)$ then
8:             $pairs.add((i^s, [\mathcal{T}[p_t{+}1], \ldots, \mathcal{T}[length(\mathcal{T})]]))$
9:     else if $\mathsf{rel}(i^s, i^t)$ then $pairs.add((i^s, [i^t]))$
10:    if $f(i^s) \leq f(i^t)$ or $(s(i^s) \leq f(i^t)$ and $\mathsf{rel} \in \{\mathsf{starts}, \mathsf{finishes}, \mathsf{during}, \mathsf{equal}\})$ then
11:        $p_s \leftarrow p_s + 1$
12:    if $f(i^s) \geq f(i^t)$ or $(f(i^s) {\geq} s(i^t)$ and $\mathsf{rel} \in \{\mathsf{before}, \mathsf{meets}, \mathsf{starts}, \mathsf{overlaps}, \mathsf{equal}\})$ then
13:        $p_t \leftarrow p_t + 1$
14: return $pairs$
```

Proposition 8 (Semantics of RTEC$_{\mathsf{A}}$): An event description in RTEC$_{\mathsf{A}}$ is a locally stratified logic program. ▲

Proof. See Appendix A. □

3.4.4 Reasoning

We extend the process of statically determined fluent evaluation of RTEC with algorithms computing Allen relations. Algorithm 7 presents the steps of the evaluation of allen(rel, $\mathcal{S}$, $\mathcal{T}$, outMode, I). This algorithm derives all interval pairs (i^s, i^t), such that $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, satisfying Allen relation rel, and stores them in list $pairs$ (line 3). We then compute lists $\mathcal{S}_{\mathsf{rel}}$ and $\mathcal{T}_{\mathsf{rel}}$ containing, respectively, the source and the target intervals appearing in list $pairs$ at least once (line 4). This way, we may apply outMode to lists $\mathcal{S}_{\mathsf{rel}}$ and $\mathcal{T}_{\mathsf{rel}}$, as specified in Table 3.3, in order to compute the output list I (line 5). In what follows, we present the algorithm computing the interval pairs of $\mathcal{S}$ and $\mathcal{T}$ satisfying an Allen relation, and the bookkeeping operations which are necessary for correct Allen relation computation in a streaming setting (see lines 1, 2 and 6 of Algorithm 7).

Allen Relation Computation. Algorithm 8 computes the interval pairs of $\mathcal{S}$ and $\mathcal{T}$ satisfying an Allen relation rel and stores them in list $pairs$. I in holdsFor$(F=V, I)$ is a *sorted* list of maximal intervals (even if the items of the stream are not sorted) [Artikis et al., 2015]. Therefore, $\mathcal{S}$ and $\mathcal{T}$ are also sorted lists of maximal intervals (see Definition 19). We evaluate rel by means of interval endpoint comparisons, following the corresponding definition in Table 2.2. An element of $pairs$ is a tuple of the form $(i^s, \mathcal{T}')$, denoting that the source interval i^s satisfies rel with every interval in the list of target intervals $\mathcal{T}' \subseteq \mathcal{T}$. Using this compact representation, we avoid enumerating all computed interval pairs, without information loss. For example, the tuple $(i^s_1, [i^t_1, i^t_2])$ for a relation rel denotes that rel(i^s_1, i^t_1) and rel(i^s_1, i^t_2) hold.

Algorithm 8 uses two pointers, p_s and p_t, to traverse $\mathcal{S}$ and $\mathcal{T}$. If rel is before and, indeed, before(i^s, i^t) holds, we add the tuple $(i^s, [i^t, \dots, \mathcal{T}[length(\mathcal{T})]])$ to $pairs$, where $\mathcal{T}[length(\mathcal{T})]$ denotes the last interval of $\mathcal{T}$ (line 6). If before(i^s, i^t) does not hold and $f(i^s) \leq f(i^t)$, i.e., the source interval does not end after the target interval, then i^s is before all target intervals after i^t. Consequently, we add the tuple $(i^s, [\mathcal{T}[p_t{+}1], \dots, \mathcal{T}[length(\mathcal{T})]])$ to $pairs$ (line 8). If rel is not before and rel(i^s, i^t) holds, we simply add (i^s, i^t) to $pairs$ (line 9).

Afterwards, Algorithm 8 increments pointer p_s and/or pointer p_t. p_s (resp. p_t) may be incremented only if the current source (target) interval i^s (i^t) cannot satisfy rel with any subsequent target (source) interval. Since $\mathcal{S}$ and $\mathcal{T}$ are sorted lists of maximal intervals, we can check this based on the relative positions of i^s and i^t, and the given relation rel, without iterating over any subsequent interval of $\mathcal{S}$ and $\mathcal{T}$. The conditions in which pointers p_s and p_t may be incremented, while guaranteeing the correct computation of interval pairs satisfying rel, are presented in lines 10 and 12.

Example 21 (Allen relation computation): In the example of Figure 3.4(a), allen(meets, $\mathcal{S}, \mathcal{T}$, target, I_{dia}) is used to compute the maximal intervals of $disappearedInArea$ in list I_{dia}. In order to derive these intervals, RTEC$_\mathsf{A}$ computes all interval pairs in lists $\mathcal{S}$ and $\mathcal{T}$ satisfying meets (see line 3 of Algorithm 7). This is achieved with Algorithm 8. In this example, the source list is $\mathcal{S} = [i^s_1, i^s_2]$, the target list is $\mathcal{T} = [i^t_1, i^t_2]$. Initially, p_s points to i^s_1 and p_t points to i^t_1. Algorithm 8 verifies that meets does not hold for the interval pair (i^s_1, i^t_1) in line 9. Next, we check whether pointer p_s needs to be incremented. Since $f(i^s_1) > f(i^t_1)$, the condition in line 10 fails, and thus we do not increment p_s. In contrast, the condition in line 12 succeeds because $f(i^s_1) > f(i^t_1)$. Therefore, we increment p_t (line 13). The interval pair of the next iteration is (i^s_1, i^t_2). meets(i^s_1, i^t_2) is satisfied and thus we compute the pair $(i^s_1, [i^t_2])$. i^s_1 and i^t_2 cannot satisfy meets with any future interval; consequently, we increment both p_s and p_t. There is no target interval after i^t_2. Thus, Algorithm 8 terminates and returns $(i^s_1, [i^t_2])$. $\lozenge$

Windowing. In order to handle streaming data, stream reasoning systems often employ windowing techniques. Recall that, at each query time q_j, RTEC reasons over the items of an input stream that fall within a specified sliding window $w_j = (q_j{-}\omega, q_j]$, where ω is the size of the window, while all elements of the stream that took place before or at $q_j{-}\omega$ are discarded/'forgotten'. In the common case that the elements of a stream arrive with delays, e.g., due to network delays, it is preferable to make ω longer than the step. This

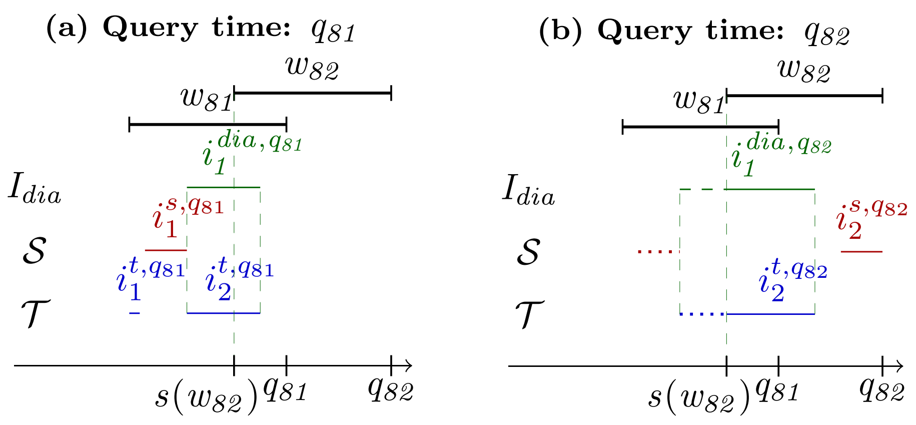


Figure 3.5: Online maximal interval computation.

way, we may reason, at q_j, over the stream elements that took place in $(q_j{-}\omega, q_{j-1}]$, but arrived after q_{j-1}. As an example, Figure 3.5 shows the intervals of $\mathcal{S}$ and $\mathcal{T}$ of Figure 3.4(a) as they are available at query times q_{81} and q_{82}. The corresponding windows w_{81} and w_{82} are overlapping in order to accommodate, at query time q_{82}, stream elements that took place in $(q_{82}{-}\omega, q_{81}]$, but arrived after q_{81}.

RTEC$_\text{A}$ follows RTEC and reasons over streams using sliding windows. We make the following assumptions. First, the window size and the step remain constant. Thus, we can always derive the endpoints of the next window based on the current query time. Second, the delays in the stream may be tolerated by the window size. In other words, at query time q_j, the intervals taking place before the current window w_j are not revised. In contrast, the intervals that were available or derived at q_{j-1} and take place within w_j may be revised at q_j. RTEC$_\text{A}$ guarantees correct reasoning by computing, at q_j, all interval pairs (i^s, i^t) satisfying Allen relation rel, such that at least one of i^s and i^t intersects with window w_j. The proof of correctness is presented in the following section. To compute all such interval pairs, we cache at each query time the intervals that may be required in the future (line 6 of Algorithm 7). This way, we may retrieve at q_j the intervals cached at q_{j-1} (see lines 1–2) that allow us to perform correct Allen relation computation. In the following example, we motivate our caching technique.

Example 22 (Windowing): Figure 3.5(a) illustrates the computation of $\text{allen}(\text{meets}, \mathcal{S}, \mathcal{T}, \text{target}, I_{dia})$ at query time q_{81}, where $\mathcal{S}$ and $\mathcal{T}$ contain only the intervals that fall within window w_{81}. Contrast these intervals with the ones depicted in Figure 3.4(a). Interval $i_1^{t,q_{81}}$, e.g., is shorter than the interval i_1^t of Figure 3.4(a) because a segment of i_1^t falls outside w_{81}. Moreover, the events leading to the extension of $i_2^{t,q_{81}}$ up to q_{81} have been delayed and are not available at q_{81}, and thus $i_2^{t,q_{81}}$ ends earlier than q_{81}. Based on the intervals in w_{81}, we compute that $\text{meets}(i_1^{s,q_{81}}, i_2^{t,q_{81}})$ holds at q_{81} and derive the output interval $i_1^{dia,q_{81}}$, which matches $i_2^{t,q_{81}}$.

The intervals of $\mathcal{S}$ and $\mathcal{T}$ available at the next query time, q_{82}, i.e., $i_2^{s,q_{82}}$ and $i_2^{t,q_{82}}$, are displayed in Figure 3.5(b). The first segment of $i_2^{t,q_{81}}$, i.e., $[s(i_2^{t,q_{81}}), s(w_{82})]$ is missing, because it is outside the current window, while its final segment $(s(w_{82}), f(i_2^{t,q_{81}})]$ has been

Algorithm 9 $windowing(\mathcal{S}, \mathcal{T}, \textsf{rel})$

```
1: $s(w_{j+1}) = q_j + step - \omega$
2: $\mathcal{S}_< \leftarrow getIntervalsBeforeTimepoint(\mathcal{S}, s(w_{j+1}))$
3: $i^s_* \leftarrow getIntervalContainingTimepoint(\mathcal{S}, s(w_{j+1}))$
4: $i^t_* \leftarrow getIntervalContainingTimepoint(\mathcal{T}, s(w_{j+1}))$
5: **if** $i^s_* \neq null$ **then**
6:    **if** rel $\in$ {meets, overlaps, before} **or** ($i^t_* \neq null$ **and**
       ((rel $\in$ {starts, equal} **and** $s(i^s_*) = s(i^t_*)$) **or**
        (rel $\in$ {finishes, during} **and** $s(i^s_*) > s(i^t_*)$))) **then**
7:       $cache([s(i^s_*), s(w_{j+1})])$
8: **if** $i^t_* \neq null$ **then**
9:    **if** rel $\in$ {meets, starts, overlaps} **and** $\exists i^s \in \mathcal{S}_<$ : rel($i^s, i^t_*$) **then**
10:      $cache([s(i^t_*), s(w_{j+1})])$, $cache(i^s)$
11:   **else if** rel $\in$ {before} **and** $\exists i^s \in \mathcal{S}_<$ : (before($i^s, i^t_*$)
          **and** $f(i^s) \geq s(w_{j+1}) - mem$) **then**
12:      $cache([s(i^t_*), s(w_{j+1})])$
13:   **else if** rel $\in$ {finishes, during} **or** ($i^s_* \neq null$ **and**
          ((rel $\in$ {starts, equal} **and** $s(i^s_*) = s(i^t_*)$) **or**
           (rel $\in$ {overlaps} **and** $s(i^s_*) < s(i^t_*)$))) **then**
14:      $cache([s(i^t_*), s(w_{j+1})])$
15:   **if** rel $\in$ {during} **then**
16:      **for** $i^s \in \mathcal{S}_<$ : rel($i^s, i^t_*$) **do** $cache(i^s)$
17: **if** rel $\in$ {before} **then**
18:   **for** $i^s \in \mathcal{S}_<$ : $f(i^s) \geq s(w_{j+1}) - mem$ **do** $cache(i^s)$
```

extended, given the events that arrived after q_{81}. Considering the intervals i^s_1 and i^t_2 of Figure 3.4(a), it is not possible to compute meets(i^s_1, i^t_2) at q_{82} because i^s_1 and part of i^t_2 take place before w_{82}. To address this issue, we cache, at q_{81}, $i^{s,q_{81}}_1$ and the segment of $i^{t,q_{81}}_2$ that is before time-point $s(w_{82})$. Figure 3.5(b) depicts these cached (segments of) intervals with dotted lines. At q_{82}, $i^{s,q_{81}}_1$, which matches i^s_1, is added to $\mathcal{S}$ and the cached segment of $i^{t,q_{81}}_2$ is amalgamated with $i^{t,q_{82}}_2$, forming an interval that matches i^t_2. As a result, we compute that meets(i^s_1, i^t_2) holds and the output interval $i^{dia,q_{82}}_1$ at q_{82}. In Figure 3.5(b), the dashed segment of $i^{dia,q_{82}}_1$ denotes the interval part that falls outside w_{82}. In contrast to $i^{dia,q_{81}}_1$, $i^{dia,q_{82}}_1$ matches i^{dia}_1, i.e., the output interval displayed in Figure 3.4(a). $\Diamond$

Example 22 demonstrates that the prefix of a target interval intersecting with the next window, and a source interval ending before the next window, may need to be cached to guarantee correct reasoning. Target intervals ending before the next window are not cached because they cannot satisfy any Allen relation with a source interval ending in the future. Algorithm 9 presents our caching procedure. First, we compute the start endpoint of the next window $s(w_{j+1})$ (line 1), and identify the list of source intervals $\mathcal{S}_<$ taking place before $s(w_{j+1})$, as well as the source and target intervals i^s_* and i^t_*, if any, containing

$s(w_{j+1})$ (lines 2–4). For example, in Figure 3.5(a), $\mathcal{S}_< =[i_1^{s,q_{81}}]$, $i_*^s = null$ and $i_*^t = i_2^{t,q_{81}}$. The segments of i_*^s and i_*^t that are before $s(w_{j+1})$ may need to be cached (see $i_2^{t,q_{81}}$ in Example 22). The conditions for caching $[s(i_*^s), s(w_{j+1})]$ and $[s(i_*^t), s(w_{j+1})]$ are in lines 5–7 and 8–14, respectively. Moreover, we may need to cache a subset of the intervals in $\mathcal{S}_<$. The conditions for caching such intervals are presented in lines 8–10 and 15–18.

In the case of before, it is impossible to guarantee correct reasoning without caching every source interval. For example, interval $i_1^{s,q_{81}}$ of Figure 3.5(a) will satisfy before with all target intervals after $i_2^{t,q_{81}}$ that arrive in the future. Thus, we need to always keep $i_1^{s,q_{81}}$ in memory to ensure correctness. In order to maintain a balance between efficiency and correctness in the case of before, we use a memory threshold mem and cache, at query time q_j, all source intervals ending in $[s(w_{j+1}){-}mem, s(w_{j+1})]$ (see lines 17–18 of Algorithm 9). If at least one of these intervals is before i_*^t, i.e., the target interval containing $s(w_{j+1})$, then we also cache $[s(i_*^t), s(w_{j+1})]$ (see lines 11–12). This way, we may compute, at q_{j+1}, the interval pairs (i^s, i^t) satisfying before, such that i^t intersects with window w_{j+1} and i^s ends in $[s(w_{j+1}){-}mem, s(w_{j+1})]$.

3.4.5 Correctness and Complexity

We prove the correctness of RTEC$_A$ and present its complexity, with respect to Allen relation computation for stream reasoning.

Proposition 9 (Correctness of RTEC$_A$): RTEC$_A$ computes all maximal intervals of a statically determined fluent defined in terms of an Allen relation, and no other interval. ▲

As expected, RTEC$_A$ is correct provided that interval delays, if any, can be tolerated by the window size. In other words, all intervals occurring before query time q_j that were not available at q_j, take place after $s(w_k)$, where $k > j$, and will be available by query time q_k. For the case of before, we additionally permit delayed source intervals taking place in $[s(w_k){-}mem, s(w_k)]$ and arriving by q_k.

To prove the correctness of RTEC$_A$, we first show that Algorithm 8 computes all interval pairs (i^s, i^t), where $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, satisfying an Allen relation, and no other interval pair. Then, we show that Algorithm 9 caches all intervals that may be required by Algorithm 8 in the future for correct Allen relation computation, and no other interval.

Lemma 1: Algorithm 8 computes all interval pairs (i^s, i^t), where $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, satisfying an Allen relation, and no other interval pair. ▲

Proof Sketch. We present the proof for meets; the proofs for the remaining relations are similar and may be found in Appendix B. Algorithm 8 is sound because, according to line 9, it may only compute an interval pair (i^s, i^t), such that $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, if meets(i^s, i^t) holds. Towards proving completeness, suppose that meets(i^s, i^t) holds and Algorithm 8 does not compute (i^s, i^t). In this case, according to line 9, there is no iteration of the *while* loop of Algorithm 8 such that pointer p_s points to i^s and p_t points to i^t. The condition of line 2 states that Algorithm 8 iterates over all items in at least one of its input lists. Suppose that, in the current iteration, p_s points to i^s when p_t points to an interval i_b^t that is before i^t.

By the definition of meets, we have $f(i^s) = s(i^t)$, while it holds that $s(i^t) > f(i^t_b)$, because $\mathcal{T}$ is a sorted list of maximal intervals. Therefore, it holds that $f(i^s) > f(i^t_b)$, and thus we only increment pointer p_t (see lines 10–13). This condition continues to hold for all target intevals until p_t points to i^t. Similarly, assume that p_t points to i^t when p_s points to an interval i^s_b that is before i^s. $s(i^t) = f(i^s) > f(i^s_b)$ holds, and thus Algorithm 8 increments p_s until it points to i^s. In both cases, we reach an iteration of the *while* loop where p_s points to i^s and p_t points to i^t, which is a contradiction. Thus, if meets(i^s, i^t) holds, then Algorithm 8 computes (i^s, i^t). □

Lemma 2: Algorithm 9 caches all intervals that may satisfy an Allen relation with an interval arriving in the future, and no other interval. ▲

Proof Sketch. Suppose that there is a source interval i^s_* containing the start of the next window $s(w_{j+1})$. We will prove that Algorithm 9 caches $[s(i^s_*), s(w_{j+1})]$ iff i^s_* may satisfy an Allen relation with a target interval i^t arriving in the future. meets/overlaps/before: if i^t occurs in the next window w_{j+1}, it holds that $s(i^t) > s(i^s_*)$, and thus i^s_* may satisfy meets, overlaps or before with i^t. Therefore, Algorithm 9 caches $[s(i^s_*), s(w_{j+1})]$ (line 6). starts/equal: starts(i^s_*, i^t) and equal(i^s_*, i^t) may hold only if $s(i^s_*) = s(i^t)$ and $f(i^s_*) \leq f(i^t)$, in which case i^t also contains $s(w_{j+1})$. Thus, we cache $[s(i^s_*), s(w_{j+1})]$ iff there is a target interval starting at $s(i^s_*)$ and containing $s(w_{j+1})$ (see the conditions for starts and equal in line 6). finishes/during: finishes(i^s_*, i^t) and during(i^s_*, i^t) may hold only if $s(i^s_*) > s(i^t)$ and $f(i^s_*) \leq f(i^t)$. Therefore, we cache $[s(i^s_*), s(w_{j+1})]$ iff there is a target interval starting before $s(i^s_*)$ and containing $s(w_{j+1})$ (see the conditions for finishes and during in line 6). The proofs for caching a target interval containing $s(w_{j+1})$ and source intervals ending before $s(w_{j+1})$ are provided in Appendix B.

Proposition 10 (Complexity of RTEC$_\mathbf{A}$): The cost of computing the maximal intervals of a statically determined fluent defined in terms of an Allen relation is $\mathcal{O}(n)$, where n is the number of input intervals. ▲

Proof. See Appendix B. □

We identified the conditions according to which a source (resp. target) interval cannot satisfy an Allen relation with any future target (source) interval. Algorithm 8 leverages these conditions (lines 10 and 12) in order to compute all interval pairs satisfying an Allen relation in a *single pass* over the input intervals. Moreover, we specified the conditions that allow us to detect in *linear time* the intervals that need to be cached (see lines 5–18 of Algorithm 9). In practice, the number of cached intervals is negligible. Thus the cost of computing Allen relations remains constant as the stream progresses, and is bound by the size of the window.

3.4.6 Discussion

We have identified stream reasoning frameworks that support Allen relations (see Table 2.3). These systems have some shortcomings compared to our proposed framework. In

D^{2}IA and ETALIS, Allen relation satisfaction generates the union of the intervals satisfying the relation. Thus, these systems do not support other output interval construction modes (see, e.g., the output modes in Table 3.3). ETALIS and ISEQ do not take advantage of the common assumption that the intervals of situations of interest are maximal. As a result, Allen relation computation requires more complicated reasoning algorithms that yield higher reasoning times compared to our approach. TPStream evaluates an Allen relation by performing a binary search over the sorted list of target intervals for each source interval. Thus, the cost of Allen relation computation in TPStream is $\mathcal{O}(nlogn)$, where n is the number of input intervals, which is higher than the complexity of our framework (see Proposition 10).

The literature contains several plane-sweeping-based algorithms that compute Allen relations. For example, Piatov et al. [2021] developed a family of interval join algorithms, including Allen relations. These algorithms have log-linear time complexity, and thus higher cost than RTEC$_\mathsf{A}$. Chekol et al. [2019] extended SPARQL with plane-sweeping-based algorithms for Allen relation computation. Contrary to RTEC$_\mathsf{A}$, this work does not support streaming data. The aforementioned approaches do not operate on maximal intervals. In this context, Allen relation computation becomes a harder task that leads to higher worst-case reasoning times compared to the case of having maximal intervals. Moreover, plane-sweeping-based approaches iterate over every endpoint of the intervals in the input lists, whereas, in RTEC$_\mathsf{A}$, we have identified the conditions under which we may increment both pointers in the same iteration (see Algorithm 8), while preserving result completeness (see Lemma 1).

Havelund et al. [2021] proposed an extension of Allen's algebra featuring quantification over intervals. This work focuses on relations before, during and overlaps, omitting the remaining relations. Unlike RTEC$_\mathsf{A}$, this approach does not support the construction of intervals by means of (arbitrary conditions on) concurrent events. nfer [Kauffman et al., 2018] is a rule-based system transforming instantaneous event streams into interval-based, hierarchical abstractions, possibly using Allen relations. nfer does not include optimisations for Allen relation computation and does not guarantee correct reasoning in a streaming setting.

TPStream and the system of Chawda et al. [2014] support distributed Allen relation computation. Pilourdault et al. [2016] compute approximate incarnations of Allen relations. RTEC$_\mathsf{A}$ does not support approximate Allen relation and lacks distribution techniques; extensions of RTEC$_\mathsf{A}$ with these features are considered as future work directions.

4. EXPERIMENTAL EVALUATION OF RTEC$_\circ$, RTEC$^\rightarrow$ AND RTEC$_A$

We present an empirical evaluation of RTEC$_\circ$, RTEC$^\rightarrow$ and RTEC$_A$. In Section 4.1, we describe the applications and the computational frameworks that we employed in our experimental evaluation. Section 4.2 presents our experiments, while Section 4.3 summarises our results.

4.1 Experimental Setup

We evaluated RTEC$_\circ$, RTEC$^\rightarrow$ and RTEC$_A$ on (fragments of) event descriptions from several applications that require stream reasoning. Our evaluation included empirical comparisons of the reasoning efficiency of RTEC$_\circ$, RTEC$^\rightarrow$ and RTEC$_A$ against related event processing frameworks. Below, we briefly describe the applications and the datasets we employed. Afterwards, we list the frameworks that we included in our empirical evaluation.

4.1.1 Applications

Our evaluation involves the applications that we outlined in Section 1.1.3, as well as city transport management, which we describe below. We briefly discuss the event descriptions and the datasets we employed for these applications.

Maritime Situational Awareness. In this application, the input events concern simple maritime activities, which are generated based on AIS position signals emitted by vessels. Spatio-temporal combinations of these simple activities form composite patterns, signifying various types of dangerous, suspicious and illegal vessel activities, such as ship-to-ship transfer of goods in the open sea, that must be detected in real-time. For some of our experiments, we extended the event description for the maritime domain proposed by Pitsikalis et al. [2019] with cyclic dependencies among FVPs, in order to capture more accurately the maritime behaviours of interest, such as fishing. Moreover, in some of our experiments, we introduced composite maritime activities defined in terms of Allen relations (see, e.g., rule (3.19)). We employed a publicly available dataset[1] 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[2], 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]. The task was to compute the maximal intervals of FVPs expressing composite maritime activities, such as trawling and ship-to-ship transfer.

Human Activity Recognition. We detected instances of composite human activities by

[1] *https://zenodo.org/record/1167595*
[2] *https://imisglobal.com/*

reasoning over input event streams consisting of simple activities performed by one person, such as ‘walking’ and ‘running’, as well as the coordinates and the orientation of the tracked people. FVPs are used to model spatial relations between the entities appearing in video frames, such as two people being close to each other, and composite human activities that involve two people, like ‘meeting’ and ‘fighting’. We employed CAVIAR[3], a benchmark activity recognition dataset that contains 28 videos, adding up to 26,419 video frames. In CAVIAR, the input events that occur at each video frame have been manually annotated by the creators of the dataset. The task was to compute the maximal intervals of FVPs expressing composite human activities.

City Transport Management This application involves the real-time monitoring of qualitative parameters regarding the operation of city transport vehicles, such as punctuality, driving quality and passenger comfort, aiming towards the improvement of city transport. The input events we used for this application were derived from sensors equipped to public transport vehicles (buses and trams) and concern changes in the position and acceleration of the vehicle, in-vehicle temperature, noise level and passenger density. FVPs are related to the punctuality of a public transport vehicle, driving style and quality, passenger and driver comfort, and passenger satisfaction. We employed a dataset that was supplied by the PRONTO project [Artikis et al., 2012a], and contains real data. We used a version of the dataset that was enriched with synthetic data, in order to include instances of event types that were missing from the dataset, since the corresponding event processing components were not yet functional at the time of data collection. The task was to compute the maximal intervals of FVPs describing the quality of city transportation.

Multi-Agent Systems. We employed a simple voting protocol and an e-commerce protocol called NetBill. Figure 3.1 displays the dependency graphs of voting and NetBill. We used synthetic data generators to produce 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 multi-agent systems, 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. The task was to compute the maximal intervals of (subsets of) the FVPs presented in the dependency graphs of Figure 3.1.

Biological Feedback Processes. Our experiments included simplified models of feedback loops for immune response and phage infection (see Figure 1.1). We express feedback loops in RTEC$^{\rightarrow}$ by modelling biological variables, such as the concentration of h-cells in immune system response, with FVPs and the value changes of these variables with future FVP initiations. The feedback loops we studied evolve based solely on the initial values of their variables and the delays that guide the changes in the values of these variables. In other words, there are no incoming streams of events in this application; the input data concern the initial values of FVPs and the delays of future initiations. The task

[3] *https://groups.inf.ed.ac.uk/vision/DATASETS/CAVIAR/CAVIARDATA1/*

was to compute the maximal intervals of FVPs, expressing the values of the variables in the feedback loops of Figure 1.1, as simulations progressed.

4.1.2 Competing Frameworks

We compared $\text{RTEC}_{\circ}$ against the following frameworks that support cyclic dependencies: $\text{RTEC}_{\circ}$-naive, i.e., an implementation of $\text{RTEC}_{\circ}$ that processes cyclic dependencies using rules (3.7)–(3.9) instead of the algorithms presented in Sections 3.2.3; and jREC, i.e., 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 $\text{RTEC}_{\circ}$.

We compared $\text{RTEC}^{\rightarrow}$ against $\text{RTEC}^{\rightarrow}$-naive, i.e., an implementation of $\text{RTEC}^{\rightarrow}$ that employs rules (3.16)–(3.18) instead of the algorithms presented in Sections 3.3.4; s(CASP), i.e., a query-driven execution model for Answer Set Programming with constraints, that has been extended with Event Calculus-based reasoning [Arias et al., 2018; Arias et al., 2022]; jREC^{fi}, i.e., an implementation of jREC that supports future initiations [Montali et al., 2013]; and GLK-EC, i.e., a framework that integrates the Event Calculus with a sequential logic specifying asynchronous automata in order to process biological feedback loops [Srinivasan et al., 2022]. Moreover, we compared $\text{RTEC}^{\rightarrow}$ against frameworks that do not support events with delayed effects. Our comparison included Fusemate, i.e., a logic programming system that integrates the Event Calculus with description logics [Baumgartner, 2021a]; Logica[5], i.e., a Datalog-like programming language that compiles into SQL and runs on BigQuery[6]; jREC^{rbt}, i.e., a version of jREC that employs red-black trees as an indexing method for manipulating lists of events and FVP intervals efficiently [Falcionelli et al., 2019]; and Ticker, i.e., a stream reasoning system that supports a fragment of the logic-based language LARS [Beck et al., 2017; Beck et al., 2018]. Ticker is the only LARS-based framework in which we were able to express a meaningful subset of the event descriptions of our applications (see Section 2.5).

We compared RTEC_{A} with the following frameworks that support Allen relations: AEGLE [Georgala et al., 2016], i.e., a state-of-the-art system for Allen relation computation that has been integrated in the link discovery framework LIMES [Ngomo et al., 2021]; and D^2IA, i.e., an extension of the Big Data stream processing engine Flink with interval-based semantics that has been used to detect the maximal intervals of situations of interest [Awad et al., 2022].

$\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A} are written and executed in Prolog. For the remaining frameworks, we used the programming language (version) recommended by its developers. Moreover, we made sure that, in each experiment, all systems produced the same results. We used a single core of a desktop PC running Ubuntu 20.04, with Intel Core

[4] *https://www.inf.unibz.it/~montali/tools.html*
[5] *https://github.com/EvgSkv/logica*
[6] *https://cloud.google.com/bigquery*

i7-4770 CPU @3.40GHz and 16GB RAM. The code of RTEC$_\circ$, RTEC$^\rightarrow$ and RTEC$_A$, as well as the complete event descriptions and the datasets we employed, are available in the repository of our frameworks[7].

4.2 Experimental Results

We present an empirical evaluation of the extensions of RTEC that we proposed. In Section 4.2.1, we provide an evaluation of RTEC$_\circ$. We demonstrate that RTEC$_\circ$ is much more efficient than RTEC$_\circ$-naive, verifying our complexity analysis of RTEC$_\circ$. Afterwards, we present a comparison of RTEC$_\circ$ against jREC on an event description with cycles, demonstrating that RTEC$_\circ$ outperforms jREC by orders of magnitude. Moreover, our evaluation shows that RTEC$_\circ$ supports reasoning over large, real data streams for maritime situational awareness, and that our extension does not compromise the efficiency of RTEC (see Section 2.3) on event descriptions that do not include cyclic dependencies. Section 4.2.2 presents an evaluation of RTEC$^\rightarrow$, demonstrating that RTEC$^\rightarrow$ does not perform redundant computations, and outperforms state-of-the-art frameworks, often by several orders of magnitude. Moreover, RTEC$^\rightarrow$ reasons over large data streams with events with delayed effects from the maritime domain very efficiently, and does not compromise the performance of RTEC when these events are omitted. In Section 4.2.3, we provide an evaluation of RTEC$_A$. We demonstrate that RTEC$_A$ outperforms state-of-the-art frameworks on the tasks of Allen relation computation in batch and windowing mode, and reasoning over large data streams and temporal specifications with Allen relations.

4.2.1 Stream Reasoning with RTEC$_\circ$

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 compared RTEC$_\circ$ against RTEC$_\circ$-naive. Figure 4.1 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. The step, i.e., the temporal distance between two consecutive query times, was set to 10 time-points. 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 multi-agent systems protocols and, therefore, are not presented in Figure 4.1. 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. Figure 4.2 shows the

[7] *https://github.com/aartikis/rtec*

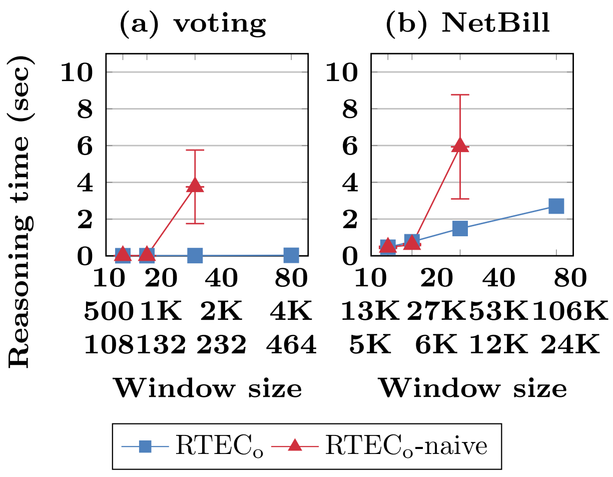


Figure 4.1: The reasoning times of RTEC$_\circ$ and RTEC$_\circ$-naive in (a) voting and (b) NetBill. The horizontal axes denote window size in terms of time-points (top), average number of input entities (middle) and average number of computed FVP intervals (bottom) per window.

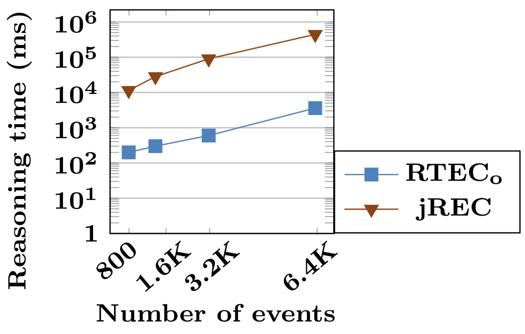


Figure 4.2: RTEC$_\circ$ and jREC computing the maximal intervals of the $status$ fluent of the voting protocol.

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 3.1). 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 4.2 shows that the use of RTEC$_\circ$ leads to significant performance gain. In the case of 6,400 events, RTEC$_\circ$ computes the maximal intervals of $status$ in less than 5 seconds, while jREC requires about 8 minutes to derive the same intervals.

In addition to multi-agent systems, we evaluated RTEC$_\circ$ on composite event recognition for maritime situational awareness. We relied on real data, i.e., position signals from

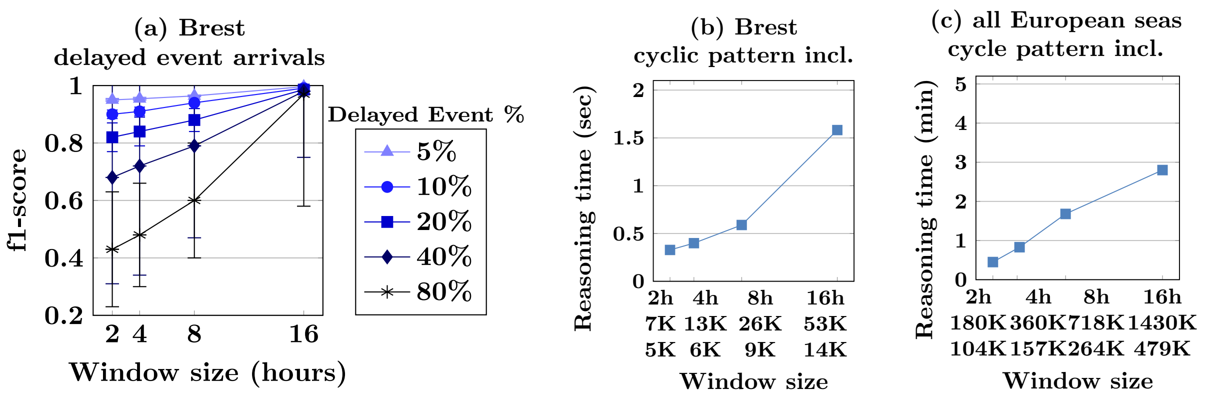


Figure 4.3: (a) The predictive accuracy of RTEC$_\circ$ in the presence of delayed event arrivals on the dataset of Brest. (b)–(c) RTEC$_\circ$ for maritime situational awareness on the datasets of (b) Brest and (c) Europe. The horizontal axes denote window size in terms of hours (top), average number of input entities (middle) and average number of computed FVP intervals (bottom) per window.

thousands of vessels, producing millions of input events. Real data streams often include delayed event arrivals. In the maritime domain, vessel position signals may arrive with a delay that exceeds 8 hours, especially when such signals are relayed through satellites. This issue may be addressed by longer, overlapping windows (see Section 2.3). 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 event arrivals, which were chosen uniformly, ranges from 5% to 80%. Figure 4.3 (a) shows the predictive accuracy of RTEC$_\circ$ when processing data streams with delayed event arrivals. The step 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 event arrivals to the intervals computed by RTEC$_\circ$ on the respective data streams without delayed event arrivals. Figure 4.3 (a) demonstrates the necessity of longer windows in the maritime domain.

Figures 4.3 (b) and (c) 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 7K events/vessel position signals, while the 16-hour window includes approximately 53K events. The computed FVP intervals range from 5K, in the 2-hour window, to 14K 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 input events in the data streams. Figures 4.3 (b) and (c) 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

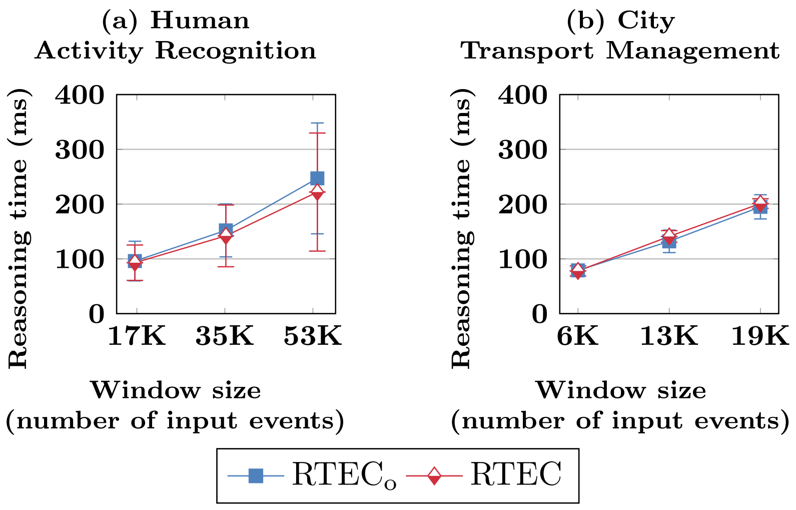


Figure 4.4: The reasoning times of RTEC$_\circ$ and RTEC in temporal specifications without cyclic dependencies: (a) human activity recognition and (b) city transport management.

cyclic dependencies, i.e., the specifications used for human activity recognition and city transport management in [Artikis et al., 2015]. Figure 4.4 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.

4.2.2 Stream Reasoning with RTEC$^\rightarrow$

We present the experimental results of our evaluation of RTEC$^\rightarrow$. For our evaluation on feedback loops, we performed an experiment for each possible configuration of initial variable values. For the remaining experiments, each result is the average of at least 30 queries. We report the standard deviations only when they are not negligible. Furthermore, we terminated all experiments that exceeded 15 minutes; we do not report the reasoning times of these experiments.

In the first set of experiments, we compared RTEC$^\rightarrow$ against RTEC$^\rightarrow$-naive. Figure 4.5 shows the results of the comparison on voting and NetBill. Note that the event description of voting does not include future initiations that may be postponed. The window size ranges from 10 to 80 time-points, while the 'step', i.e., the distance between two consecutive query times, was set to 10 time-points. In NetBill, e.g., the windows contain, on average, 6K to 44K input events. Our results show that, as the window size increased, RTEC$^\rightarrow$-naive required a significantly larger number of rule computations to derive the same FVP intervals as RTEC$^\rightarrow$. The absence of a caching mechanism resulted in redundant rule computations, verifying our complexity analysis.

In the second set of experiments, we compared RTEC$^\rightarrow$ with other systems that sup-

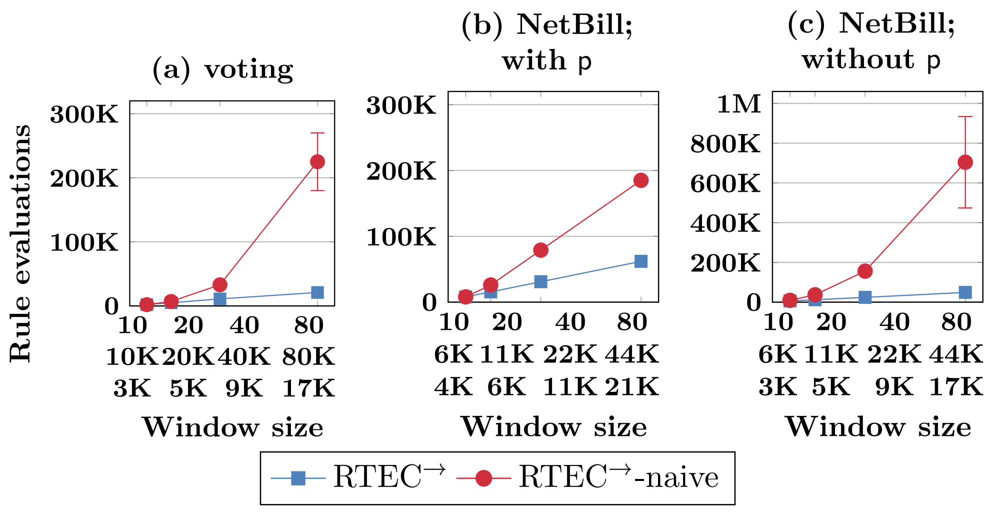


Figure 4.5: Stream reasoning in (a) voting with delayed effects that may not be postponed, and NetBill with delayed effects that (b) may and (c) may not be postponed. The window size is presented as in Figure 4.1. The vertical axes show the avg. number of FVP rule evaluations.

port events with delayed effects. These systems, i.e., s(CASP), jREC^{fi} and GKL-EC, do not support delayed effects that may postponed. First, we compared RTEC$^{\rightarrow}$ with s(CASP) and jREC^{fi} on small subsets of voting and NetBill. The left diagram of Figure 4.6 shows the results of the comparison on voting; the results on NetBill are similar, and thus omitted. The window size and the step of RTEC$^{\rightarrow}$ were set to 10 time-points. s(CASP) and jREC^{fi} do not use windows; they compute the values of fluents incrementally at each time-point. Figure 4.6 (left) presents the total reasoning times of RTEC$^{\rightarrow}$, s(CASP) and jREC^{fi} for streams of increasing size. We observe that RTEC$^{\rightarrow}$ scales much better than s(CASP) and jREC^{fi}. s(CASP) performs point-based reasoning, as opposed to interval-based reasoning, which proved to be computationally expensive. jREC^{fi} and s(CASP) evaluated future initiations without caching previous derivations, which led to unnecessary re-computations. Moreover, jREC^{fi} and s(CASP) do not have a mechanism for 'forgetting' earlier events of the data stream, which makes them inappropriate for streaming applications. These performance issues did not allow us to perform larger-scale comparisons.

GKL-EC was developed specifically for processing biological feedback loops, and thus we assessed it only on this domain. The comparison concerns two feedback loops expressing immune response in vertebrates and the invasion of a phage in a bacteria cell [Srinivasan et al., 2022]. In immune response, the presence of an antigen may lead to an increase in the production of antibodies after a certain delay. In phage invasion, the activation of a bacterial gene produces, after a delay, a protein that suppresses viral genes. We instructed RTEC$^{\rightarrow}$ and GKL-EC to compute the values of all variables in these loops as time progressed. The window size and the step of RTEC$^{\rightarrow}$ were set to 10 time-points. GLK-EC does not use windows. Figures 4.6 (middle and right) show the reasoning times of RTEC$^{\rightarrow}$ and GKL-EC for computing the values of loop variables over streams of increasing size. Our results demonstrate that RTEC$^{\rightarrow}$ scales significantly better than GKL-EC. GKL-

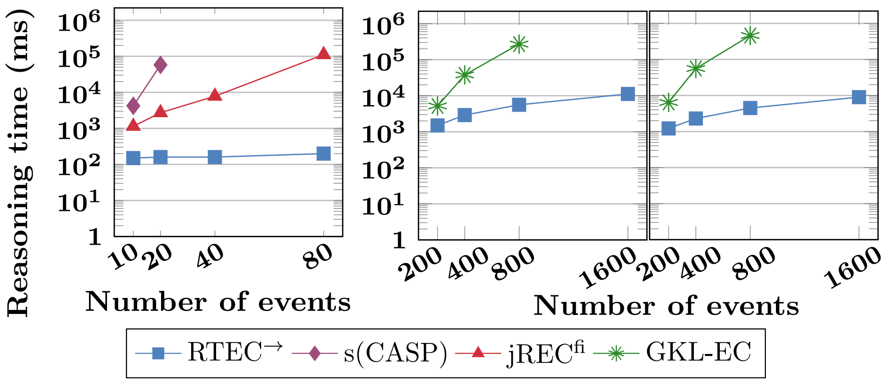


Figure 4.6: Stream reasoning in a fragment of voting (left), and feedback loops for immune response (middle) and phage infection (right). The horizontal axes show the stream size; the vertical axes show the avg. of the total execution times.

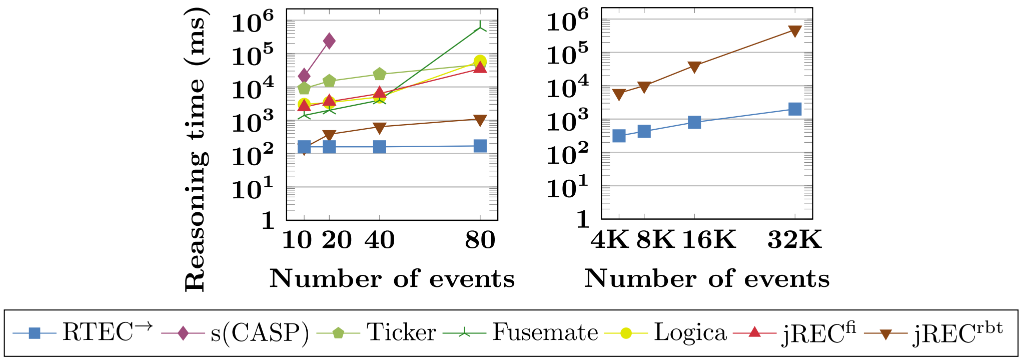


Figure 4.7: Stream reasoning for computing $quote$.

EC performs point-based reasoning without 'forgetting', which hindered its performance.

In the third set of experiments, we compared RTEC$^{\rightarrow}$ with related systems on a task that does not require future initiation computation. We instructed all systems to compute the maximal intervals of the $quote$ fluent, as defined by rules (3.14) and (3.15), i.e., a specification of $quote$ without future initiations. The comparison included s(CASP), jREC$^{\text{fi}}$, as well as systems without support for events with delayed effects, i.e., Ticker, Fusemate, Logica and jREC$^{\text{rbt}}$. Figure 4.7 (left) shows the total reasoning times over streams with size ranging from 10 to 80 events. The window size and the step of RTEC$^{\rightarrow}$ were set to 10 time-points, while the remaining systems do not employ windows. RTEC$^{\rightarrow}$ outperformed all frameworks, in most cases by several orders of magnitude. Ticker, Fusemate and Logica are not interval-based. jREC$^{\text{rbt}}$ scaled better than jREC$^{\text{fi}}$ thanks to FVP interval indexing. RTEC$^{\rightarrow}$ and jREC$^{\text{rbt}}$ were the most efficient systems, and thus we compared them further, using streams with thousands of events. Figure 4.7 (right) presents our results, demonstrating the benefits of RTEC$^{\rightarrow}$.

In the last set of experiments, we evaluated RTEC$^{\rightarrow}$ on maritime situational awareness. We used real streams containing millions of events that describe the activities of thousands

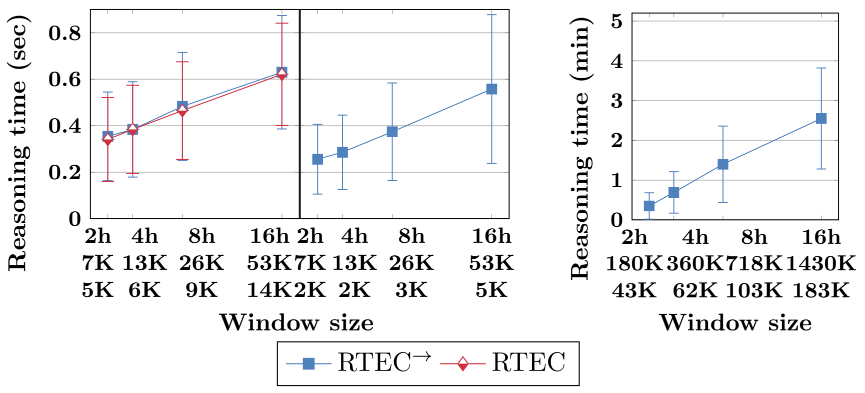


Figure 4.8: Maritime situational awareness on the datasets of Brest for FVPs without future initiations (left) and FVPs with future initiations (middle), and all European seas for FVPs with future initiations (right). The horizontal axes have the same format as in Figure 4.3.

of vessels. The volume of these streams did not allow for a comparison with the systems discussed earlier in this section. Figure 4.8 presents our results. The window size ranged from 2 to 16 hours and the step was set to 2 hours. Overlapping windows are necessary in the maritime domain due to the inherent lags in the arrival of vessel position signals, that may exceed 8 hours, especially concerning signals relayed through satellites [Pitsikalis et al., 2019]. First, we compared RTEC$^{\rightarrow}$ and RTEC (Section 2.3) on the dataset concerning the area of Brest, France, on all FVPs except those subject to future initiations, since RTEC cannot handle events with delayed effects. The results of Figure 4.8 (left) indicate that our extension of RTEC does not compromise efficiency in temporal specifications without future initiations. Second, we evaluated RTEC$^{\rightarrow}$ on the detection of maritime activities specified by means of FVPs with future initiations, i.e., trawling and search and rescue operations. Figures 4.8 (middle and right) display the results on the datasets of Brest and all European seas, respectively. Our results show that RTEC$^{\rightarrow}$ is very efficient in challenging, real applications including events with delayed effects and complex temporal specifications. In the case of Brest, e.g., RTEC$^{\rightarrow}$ required less than 0.6 seconds to reason over the events of a 16-hour window, while in the case of all European seas RTEC$^{\rightarrow}$ reasoned over 16-hour windows, including approx. 1.4M events, in less than 3 minutes.

4.2.3 Stream Reasoning with RTEC$_A$

We present the experimental results of our comparison of RTEC$_A$ against AEGLE and D^{2}IA. AEGLE does not support windowing or the computation of situations of interest. Thus, the aim of the first set of experiments was to compare RTEC$_A$, AEGLE and D^{2}IA in a batch setting, for Allen relation computation without deriving new situations. We instructed these systems to derive all interval pairs satisfying an Allen relation among lists of maximal intervals during which composite maritime activities were detected on the Brest dataset. We evaluated the efficiency of each framework as the number of input intervals increases, while making sure that all systems produced the same interval pairs (see Lemma 1 for the

Table 4.1: Average reasoning times for Allen relation computation ((a) and (b)) and CER (c) in milliseconds.

Batch size	Reasoning Time		
Input Intervals	RTEC$_A$	AEGLE	D^{2}IA
200	**1**	980	2K
2K	**14**	4K	6K
20K	**154**	71.5K	395K
200K	**1.8K**	MEM	$>$3.6M

(a) Batch setting.

Window size		Reasoning Time		Output Interval Pairs	
Days	Input Intervals	RTEC$_A$	D^{2}IA	RTEC$_A$	D^{2}IA
1	125	**1**	48	5K	5K
2	250	**2**	164	19K	18K
4	500	**4**	568	72K	71K
8	1K	**8**	1.7K	237K	236K
16	2K	**15**	7.8K	878K	874K

(b) Streaming setting.

Window size		Reasoning Time		Output Intervals	
Days	Input Intervals	RTEC$_A$	D^{2}IA	RTEC$_A$	D^{2}IA
1	19K	**40**	410	6K	6K
2	37K	**65**	592	9K	9K
4	74K	**99**	1.1K	16K	16K
8	148K	**156**	1.6K	32K	31K
16	297K	**285**	2.7K	77K	76K

(c) CER with Allen relations.

correctness of RTEC$_A$ in a batch setting). As expected, the most common Allen relation was before, while relations requiring endpoing equality, i.e., meets, starts, finishes and equal, were less frequently satisfied. Table 4.1a shows the average reasoning times of RTEC$_A$, AEGLE and D^{2}IA for computing all Allen relations among input lists containing 200–200K intervals. All results displayed in Table 4.1 are the average of 30 experiments. Since AEGLE and D^{2}IA do not assume that the input lists are temporally sorted, the interval lists of composite maritime activities were not sorted. The performance of AEGLE and D^{2}IA on sorted input lists is almost identical to that presented in Table 4.1a, and thus omitted here. For RTEC$_A$, we had to sort the input lists prior to Allen relation computation; the cost of sorting is included in the reported times of RTEC$_A$.

Table 4.1a shows that the reasoning time of RTEC$_A$ increases linearly with the input size, verifying our complexity analysis (see Proposition 10). Moreover, RTEC$_A$ outperforms AEGLE and D^{2}IA by 2–3 orders of magnitude. For example, in the experiments with 200K input intervals, RTEC$_A$ was able to compute all interval pairs satisfying an Allen relation in about 1.9 seconds. In contrast, AEGLE terminated with a memory error, while we killed the execution of D^{2}IA because it lasted for more than one hour. AEGLE sorted each input interval list by start or end endpoint, depending on the relation under evaluation. However, both sorting operations produce the same result on a list of maximal intervals, and thus only one of them is sufficient. RTEC$_A$ leverages the common assumption in stream reasoning that intervals are maximal, and avoids such unnecessary re-computations. D^{2}IA has higher reasoning times than RTEC$_A$ and AEGLE, because it is significantly slower when computing before, which is satisfied by most interval pairs in each experiment. RTEC$_A$ evaluates before very efficiently as it derives all target intervals satisfying before with some source interval in a single iteration. Furthermore, in contrast to D^{2}IA and AEGLE, RTEC$_A$ uses a compact representation for the computed interval pairs in order to avoid their explicit enumeration (see lines 6 and 8 of Algorithm 8).

In our next set of experiments, we compared RTEC$_A$ with D^{2}IA for Allen relation computation in a streaming setting, but without deriving the intervals of situations of interest. D^{2}IA does not support overlapping windows for Allen relations and does not cache intervals that may satisfy an Allen relation, such as before, in the future. Thus, to facilitate a fair comparison, we set the step of RTEC$_A$ to the window size and the threshold mem to zero.

Table 4.1b presents the average reasoning times of RTEC_A and D^2IA, and the average number of interval pairs computed by each system (see 'output interval pairs'). The input lists were provided to each system in windows, ranging from 1 day, including approx. 125 intervals, to 16 days, including 2K intervals. Our results show that RTEC_A remains orders of magnitude faster than D^2IA. Moreover, the cost of our caching mechanism is negligible (e.g., compare the second line of Table 4.1a with the last line of Table 4.1b), verifying our complexity analysis. Note that RTEC_A computed more interval pairs than D^2IA in most settings. This is due to the fact that D^2IA does not include a technique for transferring intervals to future windows (with the exception of open intervals), compromising correctness. See Lemma 2 for the correctness of RTEC_A in a streaming setting.

In the final set of experiments, we compared RTEC_A and D^2IA on stream reasoning, using fifteen patterns of composite maritime activities with Allen relations, such as those presented in Section 3.4.2. Given a pattern including $\mathsf{allen}(\mathsf{rel}, \mathcal{S}, \mathcal{T}, \mathsf{outMode}, I)$, RTEC_A computes all interval pairs of $\mathcal{S}$ and $\mathcal{T}$ satisfying rel, and applies $\mathsf{outMode}$ to the computed pairs, in order to produce the maximal intervals of the composite activity defined by the pattern. In contrast, D^2IA computes only the union of the interval pairs satisfying an Allen relation within a composite activity pattern, i.e., D^2IA does not allow the specification of another output mode. To facilitate a fair comparison, we set the $\mathsf{outMode}$ of RTEC_A to union in all maritime patterns.

Table 4.1c presents the average reasoning times of RTEC_A and D^2IA, and the average number of composite activity intervals (see 'output intervals'). The number of input intervals is significantly larger as compared to our previous experiments, because the input intervals correspond to activities performed by *all* vessels in the dataset. In stream reasoning, we are interested in the combinations of input items indicated by the composite activity patterns, and not on evaluating all possible interval combinations, as in the previous experiments. Consequently, the number of composite activity intervals is much smaller than the number of input intervals. Our results show that RTEC_A is significantly faster than D^2IA, without compromising correctness (in some cases D^2IA misses composite activities due to the absence of interval caching).

4.3 Discussion

We presented an experimental evaluation of $\text{RTEC}_\circ$, $\text{RTEC}^\rightarrow$ and RTEC_A on multi-agent systems, biological feedback processes and three composite event recognition applications, i.e., maritime situational awareness, human activity recognition and city transport management. $\text{RTEC}_\circ$ and $\text{RTEC}^\rightarrow$ employ incremental caching techniques in order to avoid redundant computations. We compared these frameworks with 'naive' versions of $\text{RTEC}_\circ$ and $\text{RTEC}^\rightarrow$ that do not use incremental caching techniques, demonstrating the benefits of such techniques. Moreover, we compared $\text{RTEC}_\circ$ and $\text{RTEC}^\rightarrow$ against RTEC on applications whose event descriptions do not include cyclic dependencies or events with delayed effects, demonstrating that our extensions of RTEC do not impose a computational overhead on reasoning in such applications. In order to assess whether $\text{RTEC}_\circ$, RTEC$^{\rightarrow}$ and RTEC$_{\mathrm{A}}$ are suitable for stream reasoning, we evaluated the efficiency of these systems on large, real data streams for maritime situational awareness, showing that our systems require a few seconds to reason over thousands of events, while deriving the maximal intervals of thousands of instances of composite maritime activities. In order to demonstrate the contributions of our proposed systems with respect to the literature, we compared RTEC$_{\circ}$, RTEC$^{\rightarrow}$ and RTEC$_{\mathrm{A}}$ with state-of-the-art frameworks that support temporal specifications with cyclic dependencies, events with delayed effects and situations defined in terms of Allen relations. Our experiments showed that RTEC$_{\circ}$, RTEC$^{\rightarrow}$ and RTEC$_{\mathrm{A}}$ outperform the state of the art, in some cases by several orders of magnitude. Moreover, in order to justify our choice of extending RTEC, as opposed to some other system found in the literature, we compared RTEC$^{\rightarrow}$ with state-of-the-art frameworks on an event description that does not include events with delayed effects, demonstrating the superior reasoning efficiency of RTEC$^{\rightarrow}$.

RTEC$_{\circ}$, RTEC$^{\rightarrow}$ and RTEC$_{\mathrm{A}}$ cannot handle noise and uncertainty, which is inherent in real applications [Alevizos et al., 2017]. The remainder of this thesis concerns the treatment of noisy data streams. We review the literature on (stream) reasoning under uncertainty, and then propose an extension of a probabilistic event recognition framework, in order to operate in a streaming setting.

5. BACKGROUND: REASONING UNDER UNCERTAINTY

We provide the necessary background material for the specification of the 'online Probabilistic Interval-based Event Calculus' (oPIEC), i.e., the framework that we propose for handling noisy data streams. oPIEC is based on logic programming and employs the Event Calculus for representing events and their effects over time (see Sections 2.1 and 2.2). First, we discuss the field of probabilistic logic programming, where the syntax of logic programming is extended with constructs for representing uncertainty. We focus our attention on the probabilistic logic programming language ProbLog, as it is the main building block of the oPIEC framework. Second, we describe Prob-EC, a probabilistic extension of the Event Calculus that is based on ProbLog. Prob-EC reasons over probabilistic events and derives the probabilities of situations of interest at each point in time. Prob-EC serves as the first component in the pipeline of oPIEC (see Figure 6.1). Third, we discuss PIEC, an interval-based extension to Prob-EC, which avoids the inconsistencies of point-based reasoning and improves upon its predictive accuracy. PIEC does not support stream reasoning; oPIEC is an extension of PIEC that is designed for reasoning over noisy data streams.

Prob-EC and PIEC have been presented in the context of composite event recognition and evaluated on a composite event recognition application, namely human activity recognition [Skarlatidis et al., 2015a; Artikis et al., 2021]. Moreover, our empirical evaluation of oPIEC and its bounded-memory variants, i.e., oPIEC^b and oPIEC^{bd}, concerns two composite event recognition tasks, i.e., human activity recognition and maritime situational awareness. Thus, in order to be consistent with the published material, we focus our attention on composite event recognition in the following chapters. Note, however, that these frameworks can be used in the broader class of stream reasoning tasks. For example, we could use oPIEC^{bd} to compute the probabilities that agents hold certain normative positions over some periods of time, based on streams of agent actions that are associated with probability values.

5.1 Probabilistic Logic Programming

Probabilistic logic programming is the combination of logic programming with probability theory. We summarise probabilistic logic programming, following [Riguzzi and Swift, 2018; Riguzzi, 2023], and ProbLog, following [Fierens et al., 2014; Raedt and Kimmig, 2015; Skarlatidis et al., 2015a].

The most prominent semantics that we find in the literature for probabilistic logic programs is the distribution semantics [Sato, 1995]. Under the distribution semantics, a probabilistic logic program without function symbols defines a probability distribution over normal logic programs, which are called *possible worlds*[1]. In this context, the *success probability of a*

[1]The term (possible) world has also been used to denote the well-founded models [van Gelder et al., 1991] of the logic programs that are induced by a probabilistic logic program [Fierens et al., 2014]. We restrict

query is derived by marginalisation, i.e., by summing up the probabilities of the possible worlds whose intended interpretation is a model of the query. The distribution over programs is defined by encoding random choices for clauses; these choices are independent from each other.

Numerous probabilistic logic programming frameworks have adopted the distribution semantics, including PRISM [Sato, 2008], Logic Programs with Annotated Disjunctions [Vennekens et al., 2004], P-log [Baral et al., 2009] and ProbLog [De Raedt et al., 2007; Fierens et al., 2014]. The differences in the languages of these frameworks are merely syntactical, as they lie only in the way the choices for clauses are encoded and the probabilities for these choices are stated. Thus, all the aforementioned frameworks have the same expressive power (see Section 2.4 of [Riguzzi, 2023]). Therefore, in the following, we focus on the semantics of the language of the probabilistic logic programming framework ProbLog, which is the main building block of oPIEC, without sacrificing the generality of our discussion.

A ProbLog program Π consists of a set of rules R and a set of *probabilistic facts* F, where probabilistic fact $p :: f$ indicates that fact f holds as true with probability p in each one of its groundings. The literals whose atoms appear in probabilistic facts, i.e., the *probabilistic literals*, are disjoint from the literals whose atoms appear in the heads of rules, i.e., the *derived literals*. Each ground probabilistic fact $p :: f$ expresses an *atomic choice*, i.e., we can choose to include f as a ground fact of the program (with probability p) or discard it (with probability $1-p$). A *total choice* is obtained by making an atomic choice for each ground probabilistic fact. Each total choice σ on the set of probabilistic facts F identifies a possible world w_σ of Π, i.e., w_σ is a normal logic program in the sample space of Π. The probability of possible world w_σ is equal to the product of the probabilities of the (independent) atomic choices that are included in total choice σ. The probability of a query q is computed as the sum of the probabilities of the possible worlds whose intended interpretation is a model of q.

Unfortunately, it is not guaranteed that the possible worlds of an arbitrary probabilistic logic program have a clear semantics. Recall, e.g., that the program containing the rules '$a \leftarrow$ not b' and '$b \leftarrow$ not a' has two models $M_1 = \{a\}$ and $M_2 = \{b\}$, and there is no way to distinguish which one of these models should be preferred (see Section 2.1). For this reason, ProbLog is restricted to probabilistic logic programs whose sample space contains only normal logic programs that have a two-valued well-founded model [Riguzzi and Swift, 2013]. We restrict attention to locally stratified logic programs (see Section 2.1), which have a two-valued well-founded model (see Theorem 6.1 in [van Gelder et al., 1991]), and are thus supported by ProbLog.

In activity recognition, input events may exhibit uncertainty, because, e.g., of the temporary malfunction of a camera or object occlusion. To address this issue, we may use ProbLog and model event occurrences as probabilistic facts. Consider the following example:

Example 23 (Human Activity Recognition in ProbLog): We define ProbLog program $\Pi = R \cup F$, where R contains the rules defining the fluent-value pair (FVP)

ourselves to probabilistic logic programs where each possible world has a unique well-founded model.

Table 5.1: Total choices over the ProbLog program of Example 23 and the probabilities of the corresponding possible worlds.

	Total choice	**Prob.**
σ_1	{happensAt($walking(mike), 1$), happensAt($walking(sarah), 1$), happensAt($walking(mike), 2$)}	0.264
σ_2	{happensAt($walking(mike), 1$), happensAt($walking(sarah), 1$)}	0.058
σ_3	{happensAt($walking(mike), 1$), happensAt($walking(mike), 2$)}	0.31
σ_4	{happensAt($walking(sarah), 1$), happensAt($walking(mike), 2$)}	0.113
σ_5	{happensAt($walking(mike), 1$)}	0.068
σ_6	{happensAt($walking(sarah), 1$)}	0.025
σ_7	{happensAt($walking(mike), 2$)}	0.133
σ_8	{ }	0.029

'$moving(P_1, P_2) =$ true' that we introduced in Example 1. We repeat these rules below:

$$\begin{aligned}
&\text{initiatedAt}(moving(P_1, P_2) = \text{true},\ T) \leftarrow \\
&\quad \text{happensAt}(walking(P_1),\ T), \\
&\quad \text{happensAt}(walking(P_2),\ T), \\
&\quad \text{holdsAt}(close(P_1, P_2) = \text{true},\ T), \\
&\quad \text{holdsAt}(similarOrientation(P_1, P_2) = \text{true},\ T).
\end{aligned} \tag{5.1}$$

$$\begin{aligned}
&\text{terminatedAt}(moving(P_1, P_2) = \text{true},\ T) \leftarrow \\
&\quad \text{happensAt}(walking(P_1),\ T), \\
&\quad \text{holdsAt}(close(P_1, P_2) = \text{false},\ T).
\end{aligned} \tag{5.2}$$

$$\begin{aligned}
&\text{terminatedAt}(moving(P_1, P_2) = \text{true},\ T) \leftarrow \\
&\quad \text{happensAt}(active(P_1),\ T), \\
&\quad \text{happensAt}(active(P_2),\ T).
\end{aligned} \tag{5.3}$$

We assume that there is uncertainty in the detection of simple human activities, such as walking and being 'active', whereas contextual information, such as coordinates and orientation, are derived with certainty. Based on the input stream, we may have the following contextual information:

$$\text{holdsAt}(close(mike, sarah) = \text{true},\ 1). \tag{5.4}$$

$$\text{holdsAt}(similarOrientation(mike, sarah) = \text{true},\ 1). \tag{5.5}$$

$$\text{holdsAt}(close(mike, sarah) = \text{false},\ 2). \tag{5.6}$$

Facts (5.4)–(5.5) express, respectively, that two people, Mike and Sarah, are close to each other and have a similar orientation at time-point 1. Fact (5.6) states that Mike and Sarah are not close to each other at time-point 2. Such non-probabilistic facts can be viewed as probabilistic facts with probability 1, and do not affect the distribution over the possible worlds of the program.

Suppose that F contains the following probabilistic facts, expressing uncertain simple activities:

$$0.7 :: \text{happensAt}(walking(mike), 1).$$
$$0.46 :: \text{happensAt}(walking(sarah), 1).$$
$$0.82 :: \text{happensAt}(walking(mike), 2).$$

F contains 3 independent probabilistic facts, and thus the numbers of total choices and possible worlds of Π are equal to $2^3 = 8$. Table 5.1 presents the total choices for the probabilistic facts in F, along with the probabilities of the corresponding possible worlds of Π. For example, total choice σ_4, according to which we include happensAt$(walking(sarah), 1)$ and happensAt$(walking(mike), 2)$ in the program, but omit happensAt$(walking(mike), 1)$, induces the possible world w_{σ_4}, i.e., a normal logic program that contains rules R and the following non-probabilistic facts:

$$\text{happensAt}(walking(sarah), 1).$$
$$\text{happensAt}(walking(mike), 2).$$

The probability of w_{σ_4} is equal to the product of the probabilities of the atomic choices that comprise σ_4, i.e., $P(w_{\sigma_4}) = (1 - 0.7) \times 0.46 \times 0.82 = 0.113$. The sum of the probabilities of all 8 possible worlds of Π is equal to 1.

Consider the following queries:

$$init_1 : \text{initiatedAt}(moving(mike, sarah) = \text{true},\ 1)$$
$$term_1 : \text{terminatedAt}(moving(mike, sarah) = \text{true},\ 1)$$
$$init_2 : \text{initiatedAt}(moving(mike, sarah) = \text{true},\ 2)$$
$$term_2 : \text{terminatedAt}(moving(mike, sarah) = \text{true},\ 2)$$

According to rule (5.1) and facts (5.4)–(5.5), $init_1$ is true in the possible worlds where Mike and Sarah are both walking at time-point 1. Thus, the probability of query $init_1$ is equal to sum of the probabilities of possible worlds σ_1 and σ_2, i.e., $P(init_1) = P(\sigma_1) + P(\sigma_2) = 0.264 + 0.058 = 0.322$. Similarly, based on rules (5.2)–(5.3) and fact (5.6), query $term_2$ is true in the possible worlds where Mike or Sarah is walking at time-point 2, and thus $P(term_2) = P(\sigma_1) + P(\sigma_3) + P(\sigma_4) + P(\sigma_7) = 0.82$. Moreover, based on rules (5.2)–(5.3), $P(term_1) = 0$, since Mike and Sarah are close to each other at time-point 1 (see fact (5.4)), and there is no possible world where Mike or Sarah is 'active' at time-point 1. Finally, $P(init_2) = 0$, because there is no possible world where Sarah is walking at time-point 2, and thus it is not possible for rule (5.1) to fire at time-point 2. $\diamondsuit$

As illustrated in Example 23, we may compute the probability of a query by enumerating the possible worlds of the program and then summing up the probabilities of the possible worlds in which the query is true. This method, however, is intractable, as the number of possible worlds is exponential to the number of probabilistic facts in the program. In general, exact probabilistic inference is computationally expensive (see Chapter 7 of [Riguzzi, 2023]).

The probability of a query q can be derived based on the set of the possible explanations/proofs of q. In ProbLog, the probability of a query q, which matches the head of some rules, is equal to the probability of the disjunction of the bodies of these rules, where each body is a conjunction of (possibly probabilistic) literals. As a result, the success probability of q is computed as follows:

$$P(q) = P(\bigvee_{e \in Proofs(q)} \bigwedge_{l_i \in e} l_i) \tag{5.7}$$

According to equation (5.7), the probability of a query q is equal to the probability that the disjunctive normal form (DNF) formula on the right-hand side of the equation evaluates to true, based on the probabilistic facts of the program. However, it is not straightforward to transform the probability of a DNF to a sum of products. The required independence conditions do not generaly hold, as the conjunctions in the DNF, i.e., the conditions of the rules defining q, are not necessarily disjoint. In order to make the proofs disjoint, one would have to enhance every conjunction with negative literals, in order to exclude worlds whose probability has already been computed in previous conjunctions of the DNF. This problem is called the disjoint-sum problem and is known to be #P-hard [Valiant, 1979].

For this reason, ProbLog does not perform probabilistic inference using equation (5.7). We employ the second version of ProbLog[2], which performs probabilistic inference as follows. First, the program is grounded and the part of the ground program that is relevant to the query is converted to a weighted Boolean formula. The weight of a probabilistic literal is derived from the probability of the corresponding probabilistic fact of the program, while the weight of a derived literal is always 1. As a result, probabilistic inference is reduced to weighted model counting (WMC) on the resulting weighted formula. Second, ProbLog handles the WMC task using knowledge compilation, i.e., a translation of the weighted Boolean formula into a form on which WMC is tractable [Darwiche and Marquis, 2002]. This way, the high cost of probabilistic reasoning is pushed into an off-line compilation process. ProbLog first transforms the logical part of the Boolean formula into a deterministic, decomposable negation normal form (d-DNNF) [Darwiche and Marquis, 2002], and then turns the resulting d-DNNF into an arithmetic circuit by taking into account the weights of the literals in the formula [Chavira and Darwiche, 2008]. WMC amounts to evaluating the resulting arithmetic circuit, which is a linear operation to the number of nodes in the arithmetic circuit. In the worst case, the size of the arithmetic circuit is exponential to the size of the weighted Boolean formula we started with. Nevertheless, the second version of ProbLog has proven scalable in several probabilistic inference tasks [Fierens et al., 2014].

ProbLog integrates probability theory into logic programming, and comes with several advantages, such as an intuitive modelling of uncertainty using probabilistic facts, a formal and declarative semantics, as well as efficient algorithms for probabilistic inference. Unfortunately, ProbLog is not designed to reason over the dynamic domains that we find in stream reasoning applications. For instance, ProbLog does not have a built-in representation of time, and thus does not directly support time-varying properties. In Example 23,

[2] *https://dtai.cs.kuleuven.be/problog/*

e.g., we used a ProbLog program to deduce the probability that Mike and Sarah start moving together at time-point $\mathit{1}$. It is not clear from the program, however, how the probability of the '$moving(mike, sarah) = \text{true}$' evolves over time and how it may be affected by subsequent initiations/terminations of the activity. To address this issue, we employ Prob-EC, a framework that incorporates the Event Calculus (see Section 2.2) into ProbLog.

5.2 Probabilistic Event Calculus

We describe Prob-EC, i.e., a framework that integrates the Event Calculus into ProbLog. Prob-EC serves as the first component of oPIEC. We summarise Prob-EC, following [Skarlatidis et al., 2015a]. We discuss Prob-EC, as well as the systems that will follow, in the context of composite event recognition (CER). For this reason, we often use the term 'simple-derived event' (SDE) for an input event of the domain, and the term 'composite event' (CE) for a situation of interest of the domain.

The language of Prob-EC extends ProbLog with the predicates of the Event Calculus (see Table 2.1 and Section 2.2). $\text{happensAt}(E,\ T)$ denotes that an SDE E takes place at time-point T. Prob-EC processes uncertain indications of SDEs, i.e., probabilistic facts in the form of $p :: \text{happensAt}(E,\ T)$, and derives the probabilities of CEs over time by means of Event Calculus-based reasoning. To do this, Prob-EC is equipped with a set of domain-specific $\text{initiatedAt}/\text{terminatedAt}$ rules, which are used the derive the probabilities of CE initiations/terminations over input streams of probabilistic happensAt facts. For instance, Prob-EC may employ the probabilistic facts and the rules of Example 23, in order to compute the probabilities of the initiations/terminations of the 'moving' CE over time. This type of probabilistic inference is done using the probabilistic reasoning algorithms of ProbLog.

To derive CE occurrences, i.e., $\text{holdsAt}(CE = \text{true},\ T)$, Prob-EC integrates the law of inertia (see axioms (2.1)–(2.4)) to its probabilistic framework. The probability of $\text{holdsAt}(CE = \text{true},\ T)$ is equal to the probability of the disjunction of the initiation conditions of $CE = \text{true}$ before T, assuming that $CE = \text{true}$ has not been 'broken' in the meantime. This probability is calculated as follows:

$$P(\text{holdsAt}(CE = \text{true},\ t)) = P(\vee_{\forall t_s < t}(\text{initiatedAt}(CE = \text{true},\ t_s) \wedge (\neg\ \text{broken}(CE = \text{true},\ t_s,\ t)))) \tag{5.8}$$

Therefore, multiple initiations of $CE = \text{true}$ increase its probability. Moreover, if $CE = \text{true}$ is 'broken' with probability p_b, then the probability of $CE = \text{true}$ becomes equal to the product of the probability of the disjunction of the initiations and $\mathit{1} - p_b$ (see expression (5.8)). The higher the probability p_b, the more significant the decrease in the probability of $CE = \text{true}$. Consecutive terminations decrease further the probability of $CE = \text{true}$.

Figure 5.1 illustrates the inference mechanism of Prob-EC when the task is to compute the probability of the $moving$ CE. In this example, $moving$ is initiated at time-point $\mathit{1}$, while no other initiations/terminations of $moving$ take place until time-point $\mathit{21}$. Then, $moving$ is being initiated repeatedly until time-point $\mathit{41}$. From then on, we have consecutive termi-

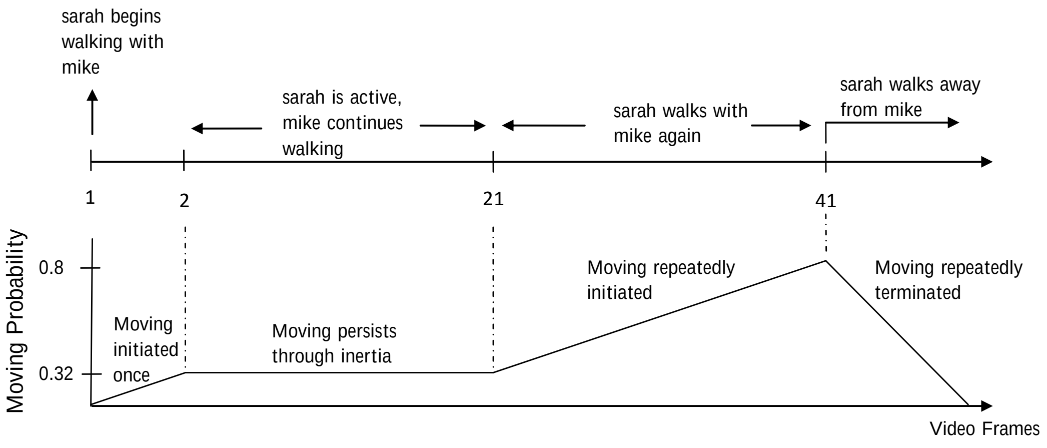


Figure 5.1: Probabilistic activity recognition with Prob-EC (after [Skarlatidis et al., 2015a]).

nations of $moving$ until the end of the video. The probability of the CE increases (resp. decreases) after each initiation (termination), while its probability does not change as long as no initiations/terminations occur. The increase/decrease of the probability is proportional to the probability values of the corresponding input events. For instance, the probability of $moving$ at time-point 22 is equal to the probability of the disjunction of the initiations of the CE at time-points 1 and 21, which is calculated as follows:

$$\begin{aligned} P(holds_{22}) = P(init_1 \vee init_{21}) &= P(init_1) + P(init_{21}) - P(init_1 \wedge init_{21}) \\ &= 0.32 + 0.1 - 0.32 \times 0.1 = 0.388 \end{aligned} \tag{5.9}$$

where $holds_t$ and $init_t$ are shorthands for holdsAt$(moving(mike,\ sarah) =$ true, $t)$ and initiatedAt$(moving(mike,\ sarah) =$ true, $t)$, respectively, and the probabilities that both people are walking close to each other, with a similar orientation at time-points 1 and 21 are 0.32 and 0.1, respectively.

Prob-EC incorporates the law of inertia into the probabilistic framework of ProbLog. This way, we can express the persistence of the probability of a CE, as well as the effects of further initiations/terminations of the CE on its probability of occurrence over time. Moreover, the use of Prob-EC allows for several efficiency optimisations. For instance, in order to monitor the probability of holdsAt$(CE =$ true, $T)$ as time progress, we can simply keep in memory the probability that the CE holds at the most recent time-point of the stream, and update this probability as new CE initiations/terminations appears in the stream, without sacrificing correctness. This method avoids the re-computation of the initiations/terminations of the CE at previous time-points, and thus is suitable for stream reasoning. However, Prob-EC performs point-based reasoning, i.e., it derives the probability that a CE holds at each point in time, which has proven to be less accurate than interval-based reasoning [Artikis et al., 2021]. Next, we describe an algorithm for probabilistic interval-based CE recognition.

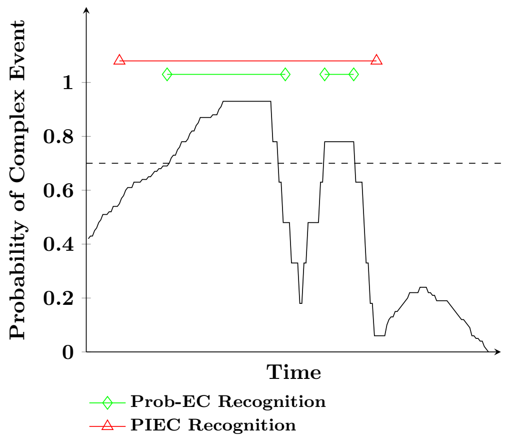


Figure 5.2: Probabilistic recognition over noisy data streams. The solid black line represents instantaneous CE probabilities, as computed by Prob-EC, while the dotted line designates the chosen threshold $\mathcal{T}$ for recognition. The outcome of Prob-EC, given this threshold value, is presented by the green straight lines. The red line expresses the recognition of PIEC.

5.3 Probabilistic Interval-based Event Calculus

The output of Prob-EC is a sequence of 'p :: holdsAt(CE = true, T)' indications which denote the evolution of the probability p of CE across time T. Some of these indications may be the result of a sensor's (temporary) malfunction. As an example, Figure 5.2 displays the probability of a CE as time progresses. Notice that there is an abrupt probability drop between the two peaks; in this example, the drop is the result of a sensor's malfunction. Prob-EC predicts that the target CE has a low probability of occurrence during this time period. In order to make sense of such indications, we employ a threshold that allows us to distinguish between positive and negative CE instances. In the example of Figure 5.2, the threshold is equal to 0.7, and the recognition of Prob-EC, i.e., the time-points at which the CE holds with a probability greater than the threshold, is depicted in the form of intervals with the green straight lines. Since, during the sensor's malfunction, Prob-EC computes probability values lower than the threshold for the CE, there is a gap in the recognition of Prob-EC. All time-points in this gap constitute false negatives, i.e., the CE took place during that gap but was not detected.

To address such issues, i.e., abrupt probability fluctuations which do not adhere to reality, the Probabilistic Interval-based Event Calculus (PIEC) [Artikis et al., 2021] consumes the output of such a point-based recognition, in order to compute the 'probabilistic maximal intervals' of a CE, i.e., the maximal intervals during which a CE is said to take place, with a probability above a given threshold. This way, PIEC is robust to short-term system failures. Below, we define the probability of an interval and probabilistic maximal intervals following [Artikis et al., 2021], and then present the way PIEC detects such intervals.

Definition 20: The **probability of interval** $I_{CE} = [s, e]$ of a CE, with $length(I_{CE}) = e - s + 1$

time-points, is

$$P(I_{CE}) = \frac{\sum_{t=s}^{e} P(\text{holdsAt}(CE = \text{true},\ t))}{length(I_{CE})} ,$$

where $P(\text{holdsAt}(CE = \text{true},\ t))$ is the probability of occurrence of the CE at time-point t.

In other words, the probability of an interval of a CE is equal to the average of the instantaneous CE probabilities at the time-points that the interval contains. Note that the instantaneous CE probabilities, i.e., $P(\text{holdsAt}(CE = \text{true},\ t))$, are not independent, since the computation of holdsAt is based on the common-sense law of inertia (see Section 5.2). Computing the average instantaneous probability allows us to smooth out abrupt probability drops caused by sensor malfunction, such as that presented in Figure 5.2. In contrast, interval probability definitions that employ instantaneous probability multiplication are sensitive to abrupt probability drops.

Definition 21: A **probabilistic maximal interval** $I_{CE} = [s, e]$ of a CE is an interval such that, given some threshold $\mathcal{T} \in [0, 1]$, $P(I_{CE}) \geq \mathcal{T}$, and there is no other interval I'_{CE} such that $P(I'_{CE}) \geq \mathcal{T}$ and I_{CE} is a sub-interval of I'_{CE}.

Probabilistic maximal intervals (PMIs) may be overlapping. To choose an interval among overlapping PMIs of the same CE, PIEC computes the *credibility* of each such PMI [Artikis et al., 2021].

By computing PMIs, PIEC addresses the problems of point-based recognition in the presence of noisy instantaneous CE probability fluctuations, and in the common case of non-abrupt probability change. See [Artikis et al., 2021] for an empirical analysis supporting these claims. As an example, Figure 5.2 shows PIEC operating on top of the probability values computed by Prob-EC. The PMI of PIEC spans over the region of noisy abrupt probability decay, leading to more robust recognition as compared to Prob-EC.

Given a dataset of n instantaneous CE probabilities $In[1..n]$ and a threshold $\mathcal{T}$, PIEC infers all PMIs of the CE in linear time, by translating the problem of PMI computation to the problem of 'maximal non-negative sum interval computation' [Allison, 2003]. To calculate the PMIs, PIEC constructs:

- The $L[1..n]$ list containing each element of In minus the given threshold $\mathcal{T}$, i.e.,

$$\forall\ i \in [1, n], L[i] = In[i] - \mathcal{T} \tag{5.10}$$

  It follows that:

$$\sum_{i=s}^{e} L[i] \geq 0 \xLeftrightarrow[Def.20]{Eq.(5.10)} P(I_{CE} = [s, e]) \geq \mathcal{T} \tag{5.11}$$

  Expression (5.11) states that the probability of an interval I_{CE} is above the given threshold $\mathcal{T}$ iff the sum of the L values of all the time-points i that I_{CE} contains is non-negative.

Table 5.2: PIEC with threshold $\mathcal{T} = 0.5$ (after [Artikis et al., 2021]).

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
In	0	0.5	0.7	0.9	0.4	0.1	0	0	0.5	1
L	-0.5	0	0.2	0.4	-0.1	-0.4	-0.5	-0.5	0	0.5
$prefix$	-0.5	-0.5	-0.3	0.1	0	-0.4	-0.9	-1.4	-1.4	-0.9
dp	0.1	0.1	0.1	0.1	0	-0.4	-0.9	-0.9	-0.9	-0.9

- The $prefix[1..n]$ list containing the cumulative or prefix sums of list L, i.e.,

$$\forall\ i \in [1, n], prefix[i] = \sum_{j=1}^{i} L[j] \tag{5.12}$$

- The $dp[1..n]$ list, where

$$\forall\ i \in [1, n], dp[i] = \max_{i \leq j \leq n} (prefix[j]) \tag{5.13}$$

  The elements of the dp list are calculated by traversing the $prefix$ list in reverse order.

Table 5.2 presents an example dataset $In[1..10]$ of instantaneous CE probabilities, along with the lists calculated by PIEC for a threshold value $\mathcal{T} = 0.5$. In this example, there are three PMIs: $[1, 5]$, $[2, 6]$ and $[8, 10]$.

PIEC additionally uses the following variable:

$$dprange[s, e] = \begin{cases} dp[e] - prefix[s-1] & \text{if } s > 1 \\ dp[e] & \text{if } s = 1 \end{cases} \tag{5.14}$$

Starting from Eq. (5.14), and substituting $prefix$ and dp with their respective definitions (Eq. (5.12) and (5.13)), we derive that $dprange[s, e]$ expresses the maximum sum that may be achieved by adding the elements of list L from s to some $e^* \geq e$, i.e.:

$$dprange[s, e] = \max_{e \leq e^* \leq n} (L[s] + \cdots + L[e^*]). \tag{5.15}$$

The following equivalence is a corollary of Eq. (5.15):

$$dprange[s, e] \geq 0 \Leftrightarrow \exists e^* : e^* \geq e, \sum_{s \leq i \leq e^*} L[i] \geq 0. \tag{5.16}$$

which, according to equivalence (5.11), entails that:

$$dprange[s, e] \geq 0 \Leftrightarrow \exists e^* : e^* \geq e, P([s, e^*]) \geq \mathcal{T}. \tag{5.17}$$

This means that $[s, e^*]$ is a potential PMI.

PIEC processes a dataset of instantaneous CE probabilities sequentially using two pointers directed towards the starting point s and the ending point e of a potential PMI. According to expression (5.17), if $dprange[s, e]$ is non-negative, then $[s, e^*]$ is a potential PMI, for some $e^* > e$. In that case, PIEC increments the e pointer until $dprange$ becomes negative. When $dprange$ becomes negative, PIEC produces the following PMI: $[s, e{-}1]$. Once a PMI is computed, PIEC increments the s pointer and re-calculates $dprange$. By repeating this process, PIEC computes all PMIs of a given dataset.

Example 24: Consider the dataset presented in Table 5.2 and a threshold $\mathcal{T} = 0.5$. Initially, $s = 1$, $e = 1$ and PIEC calculates that $dprange[1, 1] = 0.1 \geq 0$. Then, PIEC increments e as long as $dprange$ remains non-negative. This holds until $e = 6$ when $dprange[1, 6] = -0.4$. PIEC produces the PMI $[1, 5]$ and increments s. Afterwards, PIEC calculates that $dprange[2, 6] = 0.1$ and thus increments e, i.e., $e = 7$, while s remains equal to 2. PIEC proceeds by re-calculating $dprange$, i.e., $dprange[2, 7] = -0.4 < 0$, and then produces the PMI $[2, 6]$ and increments s, i.e., $s = 3$. The condition $dprange[s, 7] < 0$ holds $\forall s \in [3, 7]$. Hence, PIEC increments s until $s = 8$ when $dprange[8, 7] = 0$. Note that $\forall t,\ dprange[t + 1, t] = dp[t] - prefix[t] \geq 0$; see the definition of dp (Eq. (5.13)). Hence, PIEC avoids such erroneous pointer values, i.e., $s > e$, by incrementing e. Here, e increases as long as $dprange[8, e] \geq 0$. This holds for every subsequent time-point of the dataset. Finally, PIEC produces the PMI $[8, 10]$ as $P([8, 10]) \geq \mathcal{T}$ and there is no subsequent time-point to add. Summarising, PIEC computes all PMIs of the dataset $In[1..10]$. □

As already mentioned, the interval-based probabilistic CE recognition of PIEC has proven to be more accurate than the point-based recognition of Prob-EC. However, PIEC is not designed for applications where input data arrive to the system incrementally. For each new instantaneous probability in the input stream, PIEC would have reason over the entire dataset of instantaneous probabilities and compute the PMIs of the dataset from scratch, in order to ensure correct PMI computation. As a result, PIEC is not suitable for stream reasoning. To address this issue, we propose oPIEC, i.e., an extension of PIEC that operates online, as well as two bounded-memory extensions of oPIEC, in order to ensure that the size of the working memory remains constant as the stream progresses. Before moving to oPIEC and its extensions, we provide a literature review on probabilistic composite event recognition.

5.4 Literature Review

Composite event recognition (CER) involves the identification of high-level, composite events (CE)s of interest based on streams of simple, derived events (SDE)s [Giatrakos et al., 2020]. SDE streams may include various types of noise. For instance, in activity recognition, erroneous SDE indications are often the result of failures in the object tracker or object occlusion [Khan et al., 2019; Singh and Vishwakarma, 2019], while maritime data streams may include empty fields and erroneous field values [Bereta et al., 2021]. To address these issues, various frameworks have been proposed for probabilistic CER [Alevi-

Table 5.3: A comparison of probabilistic CER frameworks in terms of the underlying formalism, temporal model, support for background knowledge, uncertainty model and learning capabilities. B.K.: Background Knowledge, M.L.: Machine Learning.

Approach	Formalism	Temporal Model	B.K.	Uncertainty			M.L.
				Data	Pattern	Model	
Prob-EC	Logic programming	Points, Explicit, Event Calculus.	✓	✓		Probabilistic logic programming	
PIEC	Logic programming	Intervals, Explicit, Event Calculus.	✓	✓		Probabilistic logic programming	
SEC	Logic programming	Points, Explicit, Event Calculus.	✓	✓	✓	Probabilistic logic programming	
DeepProbCEP	Logic programming	Points, Explicit.	✓	✓		Neural probabilistic logic programming	✓
[Apriceno et al., 2021]	Logic programming	Intervals, Explicit.	✓	✓		Neural probabilistic logic programming	✓
CPT-L	Disjunctive logic programming	Points, Implicit.	✓		✓	Probabilistic logic programming	✓
Qualitative time CP-Logic	Disjunctive logic programming	Intervals, Explicit.	✓		✓	Dynamic Bayesian Networks	
WOLED	Answer set programming	Points, Explicit, Event Calculus.	✓		✓	Derivation of answer set that maximises the sum of weights of satisfied rules	✓
PEC	Action logic translated to answer set programming	Points, Implicit, Event Calculus.		✓	✓	Multiplication of independent probabilistic choices	
OnPad	First-order logic	Intervals, Explicit.		✓		Probabilities assigned to predicates for object equality	
MLN-EC	First-order logic	Points, Explicit, Event Calculus.	✓		✓	Markov Logic Networks	✓
MLN-Allen	First-order logic	Intervals, Explicit, Allen's Algebra.	✓	✓	✓	Markov Logic Networks	
OSLα	First-order logic	Points, Explicit, Event Calculus.	✓		✓	Markov Logic Networks	✓
[Apriceno et al., 2022]	First-order logic	Intervals, Explicit.	✓	✓		Mixed integer linear programming	✓
PEL	Event logic	Intervals, Implicit, Allen's Algebra.	✓		✓	Weighted event logic formulas	
CEP2U	TESLA	Points, Explicit.		✓	✓	Bayesian Networks	
Partial-State MTL	Metric temporal logic	Points, Implicit.		✓		Multiplication of state transitions probabilities	
ProbSTL	Signal temporal logic	Points, Implicit.		✓		Bayesian filtering	
Neuroplex	Finite state machines and logic programming	Points, Explicit.	✓		✓	Probabilistic logic programming approximated with a deep learning model	✓
SASE++	Non-deterministic finite automata	Points, Implicit.		✓		Probability distribution on time attribute	
[Sugiura and Ishikawa, 2020]	Deterministic finite automata	Intervals, Implicit.		✓		Multiplication of probabilistic transition matrices	
				Data	Pattern	Model	
Approach	Formalism	Temporal Model	B.K.	Uncertainty			M.L.

Table 5.4: A comparison of probabilistic CER frameworks in terms of the time complexity, empirical efficiency, predictive accuracy and code availability. C.A.: Code Availability.

Approach	Complexity	Efficiency	Accuracy	C.A.
Prob-EC	Linear to the number of time-points in the dataset and the number of rules and body literals in the event description.	Throughput of about 10 events/second in a human activity recognition task.	Has proven more accurate than its crisp variant when CEs are frequent and their definitions include few probabilistic conjuncts.	✓
PIEC	Linear to the number of time-points in the dataset.	Processes a stream of 8K probabilities for the 'rendez-vous' activity (more than 2 hours of data) in less than 3 minutes.	Often improves upon point-based recognition. Accuracy depends on threshold.	✓
SEC	Not reported.	Not reported.	Not reported.	✓
DeepProbCEP	Not reported.	Slower training and reasoning times than Neuroplex by more than an order of magnitude.	Outperforms state-of-the-art neural approaches when trained using sparse data.	✓
[Apriceno et al., 2021]	Not reported.	Not reported.	Improves upon fully neural benchmark.	
CPT-L	Program transformed into a BDD in polynomial time. Probabilistic inference is linear to the size of the BDD. The size of the BDD may be exponential to the size of the program.	Inference over 200K nodes in the ground network in approx. 20 seconds.	Approx. 84% in the most challenging domain of their experiments.	
Qualitative time CP-Logic	Not reported.	Not reported.	Not reported.	
WOLED	Reduction to weighted MaxSat, which is NP-complete.	MAP inference over batches including approx. 30K ground atoms in less than 1 second.	Induced patterns for 'moving' and 'meeting' yield f1-scores of 82% and 89%, respectively.	✓
PEC	Not reported.	Processes approx. 4K traces/sec.	Approximate solution computes probabilities within 95% confidence intervals w.r.t. exact probabilities by sampling 100 worlds.	✓
OnPad	$\mathcal{O}(max(\|O\|,\|l\|,\|pl\|)^{\|af\|})$, where is $\|O\|$ is the number of objects in a video, $\|l\|$ is the number of ground boolean atoms, $\|pl\|$ is the number of probabilistic ground atoms and $\|af\|$ is the number of atoms in the formula of the activity of interest.	Processes approx. 4 video frames per second.	Divergence from human reviewer annotation ranges from 9% to 20%.	
Approach	**Complexity**	**Efficiency**	**Accuracy**	**C.A.**

Table 5.4: Continued.

Approach	Complexity	Efficiency	Accuracy	C.A.
MLN-EC	Marginal inference with the approximate sampling algorithm MC-SAT. Approximate MAP inference through a transformation into Integer Linear Programming.	Marginal and MAP inference in less than 5 minutes.	Increased precision, slight decrease in recall w.r.t. deterministic solution.	✓
MLN-Allen	Not reported.	Not reported.	F1-score $>$ 65%, even for small window sizes.	
OSLα	Not reported.	Training time for 'meeting' CE approx. 2 hours. Uses MLN-EC for inference.	Accuracy of learned CE definitions is similar and often better than the accuracy of manually curated rules.	✓
[Apriceno et al., 2022]	Not reported.	Not reported.	Improves upon fully neural benchmark.	
PEL	Transforms the knowledge base in CNF in time linear to its size and applies an approximate stochastic local search approach for MAP inference.	Not reported.	On average, $>$75%.	
CEP2U	Not reported.	$<$10ms/event after introducing data and pattern uncertainty.	$>$80% in all of their experiements.	
Partial-State MTL	Not reported.	$<$3 minutes for formulas corresponding to automata with up to 15K states. Approx. 1.5 minutes with approximate reasoning.	Accuracy of formula probability approximation w.r.t. exact solution is controlled by a user-defined parameter.	✓
ProbSTL	Proportional to the length of the formula defining the target composite event and the complexity of the domain-specific functions mapping stochastic signals to real values.	Not reported.	Probabilistic approach improves upon the recall metric, which is crucial for safety formulas where no instances should be missed.	
Neuroplex	Not reported.	Not reported.	Outperforms corresponding neural solutions without human knowledge injected.	✓
SASE++	Exponential to window size if Kleene closure is included in the language.	Throughput 300K–7M events/sec or 13.62 events/second/node w.r.t. 6 queries.	Not reported.	
[Sugiura and Ishikawa, 2020]	$\mathcal{O}(\sqrt{w}n^2+n^3)$, where w is the window size and n is the number of states in the deterministic finite automaton representation of a pattern.	Throughput of $>$1K events/sec for a deterministic finite automaton with $<$10 states.	Removing states with probability $<$0.0001 to improve efficiency introduces an error of approx. 7%.	
Approach	**Complexity**	**Efficiency**	**Accuracy**	**C.A.**

zos et al., 2017; Artikis et al., 2021]. Tables 5.3 and 5.4 compare probabilistic CER frameworks, extending the discussion provided in [Alevizos et al., 2017]. Table 5.3 compares state-of-the-art frameworks in terms of the underlying formalism, temporal model, support for background knowledge, uncertainty model and learning capabilities. Table 5.4 reports the time complexity, empirical efficiency and predictive accuracy of these frameworks, as well as their code availability. In what follows, we review probabilistic CER systems along the dimensions-columns of Tables 5.3 and 5.4.

Expressive Power

Probabilistic CER frameworks are commonly automata-based or logic-based [Alevizos et al., 2017]. Automata-based methods, which often extend the CER engine SASE+ [Agrawal et al., 2008], have been designed to handle uncertainty in event symbols and their time-stamps [Zhang et al., 2013; Zhang et al., 2014]. The approach based on deterministic finite automata of Sugiura and Ishikawa [2019; 2020] employs adaptive sliding windows and optimisation techniques. Automata-based methods lack the explicit representation of time and the expressive power of logic-based frameworks, such as the Event Calculus, making the modelling of CE patterns with complex temporal constraints and background knowledge cumbersome, and often impossible.

Logic-based approaches are typically based on (subsets of) first-order logic. Several probabilistic Event Calculus dialects, such as Prob-EC and SEC [McAreavey et al., 2017], model CE definitions through logic programming rules. Other approaches employ extensions of linear temporal logic [de Leng and Heintz, 2019; Tiger and Heintz, 2020], action logics [D'Asaro et al., 2017] or the 'event logic' [Siskind, 2001; Selman et al., 2011]. These approaches model state machines where time is specified implicitly as the index of a state in an execution sequence. Metric Temporal Logic (MTL) augments the expressive power of state machines by supporting temporal operators whose scope can be constrained by user-defined temporal intervals. de Leng and Heintz [2019] presented Partial-State MTL, an incremental reasoning algorithm for computing formula satisfaction under uncertainty in MTL. ProbSTL [Tiger and Heintz, 2020] employs an extension of MTL with dense-time semantics where logical statements represent continuous time signals. Because of their point-based, implicit time model, these approaches are designed for path checking, i.e., finding whether a possible state evaluation is a model for a given query, and do not support durative CE computation. Probabilistic CER frameworks that are based on logic programming, such as Prob-EC, avoid the aforementioned representation issues.

There are additional point-based frameworks in the literature. Skarlatidis et al. [2015] proposed MLN-EC, i.e., an Event Calculus expressed in Markov Logic and implemented using Markov Logic Networks (MLNs). CEP2U [Cugola et al., 2015] extends the TESLA [Cugola and Margara, 2010] event specification language with probabilistic modelling. CPT-L [Thon et al., 2011] is a temporal extension of Causal-Probabilistic Logic (CP-Logic) [Vennekens et al., 2009] using Markov processes. SEC and PEC [D'Asaro et al., 2017] are probabilistic Event Calculus dialects based on probabilistic logic programming and an action logic, respectively. In all of these approaches, CE inference is performed at each individual time-point (e.g., video frame). It has been shown that point-based approaches are often insufficient for CER under uncertainty [Artikis et al., 2021]. Noisy instantaneous

CE probability fluctuations, and non-abrupt probability change affect the performance of point-based recognition. Therefore, it is preferable to employ interval-based techniques for probabilistic CER.

Towards this, Brendel et al. [2011] integrated the interval-based Probabilistic Event Logic (PEL) into an activity recognition framework for detecting CEs from a set of noisy, durative SDEs. MLN-Allen [Morariu and Davis, 2011] is an interval-based activity recognition framework that avoids the enumeration of all possible intervals of a CE. In [van der Heijden and Lucas, 2013], CP-logic was combined with Allen's interval relations [Allen, 1983] to perform interval-based CER under uncertainty. Contrary to PIEC, these frameworks cannot compute CE intervals with a single pass over the input stream.

Some frameworks support CE patterns that rely on background knowledge. For example, in the maritime domain, it is often necessary to retrieve the type of a vessel and its dimensions, or the regulations governing vessel behaviour in a designated area. Table 5.3 shows that the automata-based frameworks, i.e., SASE++ and the work of Sugiura and Ishikawa, do not support background knowledge, while most logic-based frameworks do. Prob-EC and PIEC are based on logic programming, and thus have inherent support for background knowledge.

A probabilistic CER framework may express data uncertainty in the input SDEs, and/or pattern uncertainty, i.e., uncertainty in the definitions of CEs. Moreover, a model is used to handle the uncertainty of SDE occurrences and/or CE definitions in reasoning. Table 5.3 compares the probabilistic CER frameworks along these dimensions. Prob-EC supports data uncertainty as SDEs are associated with probability values. Moreover, Prob-EC, like all probabilistic logic programming frameworks displayed in Table 5.3, is based on the distribution semantics [Sato, 1995]. PEL models uncertainty by defining CEs as weighted event logic formulas [Brendel et al., 2011]. OnPad supports probability annotations only on equality predicates and employs an ad-hoc algorithm to propagate their uncertainty to its output [Albanese et al., 2010]. Some state transition systems propagate uncertainty by multiplying the probabilities of all transitions in some sequence [Sugiura and Ishikawa, 2019; Tiger and Heintz, 2020], while CEP2U and Qualitative time CP-Logic [van der Heijden and Lucas, 2013] employ Bayesian Networks. PEC derives the instantaneous probability of a CE by computing all possible worlds in which the CE holds and summing up their probabilities. As discussed in Section 5.1, this approach is not tractable, and thus not suitable for probabilistic CER.

Learning

Several probabilistic CER frameworks support learning; see the last column of Table 5.3. WOLED combines inductive logic programming techniques with answer set programming (ASP) in order to learn the structure and the weights of probabilistic Event Calculus rules [Katzouris et al., 2023]. For example, WOLED was able to learn a more accurate definition for 'meeting' than the related learner ILASP [Law, 2018; Law, 2023], while being more efficient [Katzouris et al., 2023]. OSLα is a supervised framework for learning the structure of weighted Event Calculus rules in Markov Logic [Michelioudakis et al., 2016]. OSLα has been employed in a semi-supervised setting using an online supervision completion method adapted to first-order logic [Michelioudakis et al., 2019]. In this setting, OSLα was able to learn accurate definitions for the 'moving' and 'meeting' CEs, even in the presence of partially labelled data. Learning CE definitions is orthogonal to our work, as we focus on efficient CE recognition over probabilistic event streams. The CE definitions we employ may be learned or manually constructed.

Neuro-symbolic frameworks for CER typically employ a neural layer for deriving SDEs from multimedia data and a logic layer for defining patterns of CEs. Neuroplex [Xing et al., 2020] is a neuro-symbolic CER framework that translates logical rules expressing CE definitions into a 'neural reasoning layer', resulting in a fully differentiable neural architecture that may be trained end-to-end using CE annotations. DeepProbCEP [Vilamala et al., 2023] is a neuro-symbolic CER framework whose logic layer is based in DeepProbLog [Manhaeve et al., 2021], incorporating an Event Calculus dialect inspired by Prob-EC. The use of DeepProbLog enables training with CE labels without translating CE definitions into a neural representation. The probabilistic Event Calculus implementation of DeepProbCEP, however, is a bottleneck for the training and reasoning efficiency of the system, suggesting the need for further optimisations. Moreover, DeepProbCEP and Neuroplex are point-based, and thus exhibit the drawbacks of point-based recognition [Artikis et al., 2021].

Apriceno et al. [2021] proposed a neuro-symbolic framework based on DeepProbLog which leverages background knowledge about the SDEs and the visual objects in CE patterns. In a more recent work, Apriceno et al. [2022] employed a mixed integer linear programming formulation instead of DeepProbLog. SDE durations were used as soft constraints in order to find the most probable sequence of SDEs. The logic layers of [Apriceno et al., 2021; Apriceno et al., 2022] employ an Event Calculus dialect where SDEs and CEs are durative. Contrary to our Prob-EC, this dialect is very minimal as it contains only a durative incarnation of the happensAt predicate. Moreover, CE recognition is limited to one CE per video clip, while the SDEs generated by the neural network are mutually exclusive, i.e., concurrent or overlapping SDEs are not considered.

Performance

Table 5.4 reports the time complexity, empirical efficiency and predictive accuracy of each system. Note that some frameworks do not provide a complexity analysis and/or empirical results. Futhermore, we identify the systems with available code. Regarding time complexity, OnPad computes the probability that a durative CE takes place, given a possibly incomplete video sequence, with a cost exponential to the length of the formula defining the CE. PEL resorts to approximate inference techniques to mitigate its exponential worst-case cost for the task of durative CE computation. Sugiura and Ishikawa translate the definition of the target CE into a deterministic finite automaton and use an algorithm with cost $\mathcal{O}(\sqrt{w}n^2 + n^3)$, where w is the window size and n is the number of states in the automaton. Prob-EC computes CE probabilities incrementally, maintaining in memory only the probabilities of CEs at the previous time step. Subsequently, PIEC performs one pass over the derived instantaneous CE probabilities and computes PMIs with a cost that is linear to the number of input probabilities.

Table 5.4 describes the best reported empirical results of each system in terms of efficiency. OnPad processed approx. 4 video frames per second, indicating the need for further optimisations for supporting real-time processing. Sugiura and Ishikawa showed that their approach can process thousands of events per second, when the size of the automaton expressing the target CE pattern includes less than 10 states.

Table 5.4 also reports empirical results on the predictive accuracy of probabilistic CER frameworks. For OnPad, PEL and MLN-Allen, the accuracy of CE recognition was evaluated against the CE annotations provided by experts. For the approach of Sugiura and Ishikawa, which supports approximate reasoning, predictive accuracy was reported as a comparison against the CE recognition derived by the corresponding exact algorithm. In Chapter 7, we present an empirical evaluation of our framework, demonstrating that the predictive accuracy of reasoning in an online setting using our framework is comparable with the accuracy of batch reasoning over the entire dataset. Moreover, we demonstrate that our framework outperforms point-based recognition, using ground truth offered by experts.

The last column of Table 5.4 marks the systems with publicly available code. The systems we built upon to construct oPIEC, i.e., Prob-EC and PIEC, as well as the variants of oPIEC that we propose, are publicly available[3].

[3] *https://github.com/periklismant/opiec*

6. REASONING OVER NOISY DATA STREAMS

We propose oPIEC, an extension of the probabilistic interval-based composite event recognition framework PIEC for handling noisy data streams. Section 6.1 describes oPIEC, proves its correctness and outlines its complexity. Section 6.2 proposes two bounded-memory versions of oPIEC, paving the way for high predictive accuracy in scenarios where memory is scarce. In Section 6.3, we summarise our contributions and provide a discussion with respect to the state the of art.

6.1 Online Probabilistic Interval-based Event Calculus (oPIEC)

PIEC was designed to operate in a batch mode, as opposed to an online setting where data stream into the recognition system. In the online setting, reasoning has to be performed in windows/data batches and thus the predictive accuracy of PIEC may be compromised. To address this issue, we present oPIEC, i.e., *online Probabilistic Interval-based Event Calculus*, an extension of PIEC which operates on data batches $In[i..j]$, where i and j are time-points with $i \leq j$. oPIEC processes each incoming data batch $In[i..j]$ and then discards it. Moreover, it identifies the minimal set of time-points that need to be cached in memory in order to guarantee correct PMI computation. These time-points are cached in the *support set* and express the starting points of potential PMIs, i.e., PMIs that may end in the future.

Figure 3 presents the pipeline of online probabilistic CER. Prob-EC reasons over the domain-independent Event Calculus axioms, the domain-dependent rules (e.g., defining when two people are said to be moving together in activity recognition), and a probabilistic data stream, in order to compute instantaneous CE probabilities. With the use of a caching technique, Prob-EC may operate in an online mode [Skarlatidis et al., 2015a]. oPIEC consumes the instantaneous CE probabilities, as they stream out of Prob-EC, in order to calculate PMIs.

Upon the arrival of a data batch $In[i..j]$, oPIEC computes the values of the $L[i..j]$, $\mathit{prefix}[i..j]$, and $dp[i..j]$ lists. To allow for correct reasoning, the last prefix value of a batch is transferred to the next one. Consequently, the prefix value of the first time-point of a batch, $\mathit{prefix}[i]$, is set to $\mathit{prefix}[i{-}1]{+}L[i]$. (For the first batch, $\mathit{prefix}[1] = L[1]$.) This way, the computation of the values of $\mathit{prefix}[i..j]$ and $dp[i..j]$ is not affected by the absence of the data prior to i. Subsequently, oPIEC performs the following steps:

1. It computes all PMIs starting from some time-point in the support set or the data batch $In[i..j]$ and ending in $In[i..j]$.
2. It identifies the minimal set of time-points of the data batch $In[i..j]$ that should be cached in the support set.

In what follows, we first present Step 2 and then move to Step 1.

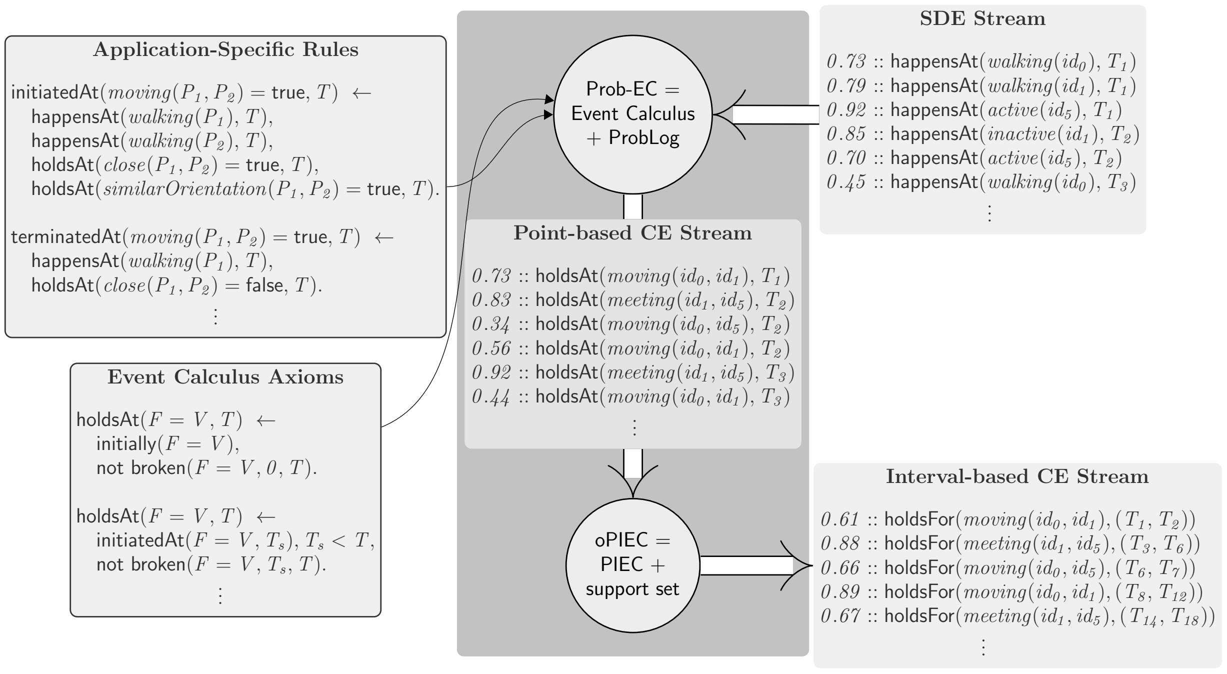

Application-Specific Rules
initiatedAt($moving(P_1, P_2)$ = true, T) ←
happensAt($walking(P_1)$, T),
happensAt($walking(P_2)$, T),
holdsAt($close(P_1, P_2)$ = true, T),
holdsAt($similarOrientation(P_1, P_2)$ = true, T).
terminatedAt($moving(P_1, P_2)$ = true, T) ←
happensAt($walking(P_1)$, T),
holdsAt($close(P_1, P_2)$ = false, T).
⋮
Event Calculus Axioms
holdsAt($F = V$, T) ←
initially($F = V$),
not broken($F = V$, 0, T).
holdsAt($F = V$, T) ←
initiatedAt($F = V$, T_s), $T_s < T$,
not broken($F = V$, T_s, T).
⋮
Prob-EC = Event Calculus + ProbLog
SDE Stream
0.73 :: happensAt($walking(id_0)$, T_1)
0.79 :: happensAt($walking(id_1)$, T_1)
0.92 :: happensAt($active(id_5)$, T_1)
0.85 :: happensAt($inactive(id_1)$, T_2)
0.70 :: happensAt($active(id_5)$, T_2)
0.45 :: happensAt($walking(id_0)$, T_3)
⋮
Point-based CE Stream
0.73 :: holdsAt($moving(id_0, id_1)$, T_1)
0.83 :: holdsAt($meeting(id_1, id_5)$, T_2)
0.34 :: holdsAt($moving(id_0, id_5)$, T_2)
0.56 :: holdsAt($moving(id_0, id_1)$, T_2)
0.92 :: holdsAt($meeting(id_1, id_5)$, T_3)
0.44 :: holdsAt($moving(id_0, id_1)$, T_3)
⋮
oPIEC = PIEC + support set
Interval-based CE Stream
0.61 :: holdsFor($moving(id_0, id_1)$, (T_1, T_2))
0.88 :: holdsFor($meeting(id_1, id_5)$, (T_3, T_6))
0.66 :: holdsFor($moving(id_0, id_5)$, (T_6, T_7))
0.89 :: holdsFor($moving(id_0, id_1)$, (T_8, T_{12}))
0.67 :: holdsFor($meeting(id_1, id_5)$, (T_{14}, T_{18}))
⋮

Figure 6.1: Online probabilistic CER. First, Prob-EC, equipped with the Event Calculus axioms (bottom left), and application-specific fluent initiation and termination rules (top left), reasons over a probabilistic data stream (top right) to derive the instantaneous probabilities of the target CEs ('Point-based CE stream' at the center of the picture). Second, oPIEC processes this stream of instantaneous CE probabilities to compute the probabilistic maximal intervals of the target CEs (bottom right).

6.1.1 Support Set

The support set comprises a set of tuples of the form $(t, \mathit{prev\_prefix}[t])$, where t is a time-point and $\mathit{prev\_prefix}[t]$ expresses t's previous prefix value, which is defined as follows:

$$\mathit{prev\_prefix}[t] = \begin{cases} \mathit{prefix}[t-1] & \text{if } t > 1 \\ 0 & \text{if } t = 1 \end{cases} \tag{6.1}$$

With the use of $\mathit{prev\_prefix}[t]$, oPIEC is able to compute $\mathit{dprange}[t, t']$ for any future time-point t', and thus determine whether t is the starting point of a PMI. For example, the arrival of a time-point $t' > t$ for which $dp[t'] \geq \mathit{prev\_prefix}[t]$ implies that $\mathit{dprange}[t, t'] \geq 0$, because

$$dp[t'] \geq \mathit{prev\_prefix}[t] \xRightarrow{Eq.(6.1)} dp[t'] \geq \mathit{prefix}[t-1] \xRightarrow{Eq.(5.14)} \mathit{dprange}[t, t'] \geq 0$$

Hence, t is the starting point of a PMI that may end either at t' or at a later time-point (see expression (5.17)).

Algorithm 10 identifies the time-points of a data batch $In[i..j]$ that should be cached in the support set. For each time-point $t \in [i, j]$, oPIEC checks whether $\mathit{prev\_prefix}[t]$

Algorithm 10 support_set_update$(min_prev_prefix,\ support_set)$

```
1: for $t \in [i, j]$ do
2:     if $prev\_prefix[t] < min\_prev\_prefix$ then
3:         $support\_set$.append($(t, prev\_prefix[t])$)
4:         $min\_prev\_prefix \leftarrow prev\_prefix[t]$
   return $(min\_prev\_prefix, support\_set)$
```

is less than the min_prev_prefix, i.e., the lowest $prev\_prefix$ value found so far. If $prev\_prefix[t]$ is less than min_prev_prefix, then $(t, prev\_prefix[t])$ is added to the support set and min_prev_prefix is set to $prev\_prefix[t]$. Finally, the updated support set and the min_prev_prefix value are passed to the next batch to ensure correct PMI computation. Example 25 illustrates this functionality of oPIEC.

Example 25: Consider the dataset in Table 5.2 arriving in three batches, $In[1..4]$, $In[5..8]$ and $In[9, 10]$. The values of the $prefix$ list are shown in Table 5.2—recall that oPIEC generates the same prefix list whether processing data batches or seeing all data in a single batch. oPIEC processes every time-point of each batch sequentially. For $t = 1$, $prev\_prefix[1] = 0$ is less than min_prev_prefix, since, initially, $min\_prev\_prefix = +\infty$. Thus, the tuple $(1, prev\_prefix[1] = 0)$ is added to the support set and min_prev_prefix is set to 0. Next, $t = 2$ and $prev\_prefix[2] = -0.5$, which is less than min_prev_prefix. Therefore, oPIEC caches the tuple $(2, -0.5)$ and updates min_prev_prefix. The remaining time-points of the first batch are not added to the support set as the condition $prev\_prefix < min\_prev\_prefix$ is not satisfied by their $prev\_prefix$ value. By processing the remaining batches in a similar way, the support set evolves as follows:

- $[(1, 0), (2, -0.5)]$; computed after processing $In[1..4]$.
- $[(1, 0), (2, -0.5), (8, -0.9)]$; computed after processing $In[5..8]$.
- $[(1, 0), (2, -0.5), (8, -0.9), (9, -1.4)]$; computed after processing $In[9..10]$. $\Diamond$

This example illustrates that oPIEC augments the support set with the time-points having the smallest $prev\_prefix$ value, regarding the data seen so far, and no other time-points.

Proposition 11: If $[t_s, t_e]$ is a PMI, then t_s satisfies the following condition:

$$\forall t_{prev} \in [1, t_s),\ prev\_prefix[t_{prev}] > prev\_prefix[t_s] \tag{6.2}$$

$\blacklozenge$

Proof Sketch. We assume that $\exists t'_s : t'_s < t_s$, $prev\_prefix[t'_s] \leq prev\_prefix[t_s]$ and follow a proof by contradiction. We show that the above expression entails that $dprange[t'_s, t_e] \geq 0$, which, according to corollary (5.16), implies that there is a PMI $[t'_s, t'_e]$, where $t'_e \geq t_e$. $[t_s, t_e]$ is a sub-interval of $[t'_s, t'_e]$. This is a contradiction because $[t_s, t_e]$ is a PMI. Thus, t_s must satisfy condition (6.2). $\square$

Proposition 12: oPIEC caches a time-point t_s in the support set iff t_s satisfies condition (6.2). $\square$

Proof. oPIEC adds a time-point t to the support set iff the condition of the *if* statement shown in line 2 of Algorithm 10 is satisfied. When this condition, $\mathit{prev\_prefix}[t] < \mathit{min\_prev\_prefix}$, is satisfied, the $\mathit{min\_prev\_prefix}$ is set to $\mathit{prev\_prefix}[t]$. Since Algorithm 10 handles every time-point in chronological order, in its i^{th} iteration, $\mathit{min\_prev\_prefix}$ will be equal to the minimum $\mathit{prev\_prefix}$ value of the first i time-points. So, a time-point's $\mathit{prev\_prefix}$ value is less than the $\mathit{min\_prev\_prefix}$ iff it satisfies condition (6.2). □

According to Propositions 11 and 12, the starting point of a PMI has the smallest $\mathit{prev\_prefix}$ value up to that point, and oPIEC caches exclusively the time-points which have this property. As a result, oPIEC caches in the support set the *minimal* set of time-points that guarantees correct PMI computation, irrespective of the data that may arrive in the future.

Note that a time-point t may satisfy condition (6.2) and not be the starting point of a PMI in a given dataset. For instance, see time-point 9 in Example 25. These time-points must also be cached in the support set, because they may become the starting point of a PMI in the future. Consider again Example 25 and assume that a fourth batch arrives with $In[11] = 0$. In this case, we have a new PMI: $[9, 11]$.

6.1.2 Interval Computation

We now describe how oPIEC computes PMIs using the support set. Algorithm 11 outlines this process. oPIEC uses a pointer pt_{ss} to traverse the support set; moreover, oPIEC uses two pointers to traverse the prefix and dp lists of the data batch $In[i..j]$, and indicate the starting point s and ending point e of a potential PMI. Note that the elements of all lists are temporally sorted. The task of Algorithm 11 is to compute all PMIs which end in data batch $In[i..j]$, starting from either a time-point in the support set (lines 3–8) or in the current batch $In[i..j]$ (lines 9–15). While processing time-points in the support set, oPIEC calculates $\mathit{dprange}$ for the elements which correspond to the pt_{ss} and e pointers. If the value of $\mathit{dprange}$ is non-negative, then there is a PMI starting at the examined element of the support set and e is incremented in order to find the ending point of the PMI. Otherwise, if the value of $\mathit{dprange}$ is negative, then there is no PMI starting from the given element of the support set. In this case, oPIEC checks whether the ending point of a PMI was identified in the previous iteration, which is indicated by a Boolean flag variable. If this is the case, the interval $[\mathit{support\_set}[pt_{ss}].\mathit{timepoint}, e{-}1]$ is added to the list of computed PMIs, where $\mathit{support\_set}[pt_{ss}].\mathit{timepoint}$ expresses the time-point of the tuple in the support set indicated by pt_{ss}. At this point, oPIEC increments pt_{ss} in order to check the next potential starting point in the support set.

Once the support set has been processed, oPIEC computes PMIs starting in $In[i..j]$. Again, $\mathit{dprange}$ is computed and, depending on its value, the starting and ending points of PMIs are selected, as explained above. Finally, if after processing $In[i..j]$ there is a pending interval, i.e., an interval whose starting point has been computed but its ending point may be found in future data, then, given the data seen so far, this interval is a PMI which ends at the last time-point of $In[i..j]$, and thus is added to the output of oPIEC.

Algorithm 11 intervalComputation(i, j, $prefix$, dp, $prev\_prefix[i]$, $support\_set$)

Input: Indices i and j indicating the starting and ending point of the current data batch, lists $prefix[i..j]$ and $dp[i..j]$, $prev\_prefix[i]$, i.e., the last value of the $prefix$ list of the previous data batch, and the support set after processing the previous data batch.

Output: A temporally sorted list of PMIs, given the data up to the current data batch $In[i..j]$.

```
1: $s \leftarrow i,\ e \leftarrow i,\ pt_{ss} \leftarrow 1,\ flag \leftarrow false$
2: while $s \le j$ and $e \le j$ do
3:   if $pt_{ss} \le length(support\_set)$ then        ▷ look for starting points in the support set
4:     $dprange \leftarrow dp[e] - support\_set[pt_{ss}].prev\_prefix$
5:     if $dprange \ge 0$ then $flag \leftarrow true,\ e{+}{=}1$
6:     else
7:       if $flag == true$ then $intervals.append((support\_set[pt_{ss}].timepoint, e-1))$
8:       $flag \leftarrow false,\ pt_{ss}{+}{=}1$
9:   else                                              ▷ look for starting points in the current data batch
10:    if $s == i$ then $dprange \leftarrow dp[e] - prev\_prefix[i]$
11:    else $dprange \leftarrow dp[e] - prefix[s-1]$
12:    if $dprange \ge 0$ then $flag \leftarrow true,\ e{+}{=}1$
13:    else
14:      if $s < e$ and $flag == true$ then $intervals.append((s, e-1))$
15:      $flag \leftarrow false,\ s{+}{=}1$
16: if $flag == true$ and $pt_{ss} \le length(support\_set)$ then        ▷ append pending interval
17:   $intervals.append((support\_set[pt_{ss}].timepoint, e-1))$
18: else if $flag == true$ then $intervals.append((s, e-1))$
19: return $intervals$
```

Example 26: We complete Example 25 by presenting the interval computation process for the same dataset arriving in batches $In[1..4]$, $In[5..8]$, and $In[9..10]$. Table 6.1 displays the contents of lists In, L, $prefix$, $prev\_prefix$ and dp for each data batch. The first three lists are the same as in Example 25. Please recall that the consistency of $prefix$ in the presence of data batches is achieved by transferring its last element to the next data batch. The last two lines of Table 6.1 present the elements added to the support set and the newly computed PMIs after processing each data batch. Upon the arrival of the first batch $In[1..4]$, the support set is empty. Therefore, Algorithm 11 only considers intervals starting in $In[1..4]$ and computes the interval $[1, 4]$, which is a PMI, given the data seen so far. When the second batch $In[5..8]$ arrives, the support set is $[(1, 0), (2, -0.5)]$ (see Example 25). Hence, Algorithm 11 initializes pointer pt_{ss} to 1 and pointer e to 5. Since $dp[5] \ge prev\_prefix[1]$, the $flag$ becomes $true$ and e is incremented (lines 4–5 of Algorithm 11). In the following iteration, $dp[6] < prev\_prefix[1]$ and thus Algorithm 11 produces the PMI $[1, 5]$, which replaces the interval $[1, 4]$. Next, pt_{ss} is set to 2. Because $dp[6] > prev\_prefix[2]$, Algorithm 11 decides that there is a PMI starting at $t = 2$. However, it fails to extend it in the following iteration. Therefore, Algorithm 11

Table 6.1: oPIEC operating on data batches.

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
In	0	0.5	0.7	0.9	0.4	0.1	0	0	0.5	1
L	-0.5	0	0.2	0.4	-0.1	-0.4	-0.5	-0.5	0	0.5
$prefix$	-0.5	-0.5	-0.3	0.1	0	-0.4	-0.9	-1.4	-1.4	-0.9
$prev\_prefix$	0	-0.5	-0.5	-0.3	0.1	0	-0.4	-0.9	-1.4	-1.4
dp	0.1	0.1	0.1	0.1	0	-0.4	-0.9	-1.4	-0.9	-0.9
$support\ set$ (new tuples)	$[(1,0),(2,-0.5)]$				$[(8,-0.9)]$				$[(9,-1.4)]$	
$PMIs$ (ending in batch)	$[[1,4]]$				$[[1,5],[2,6]]$				$[[8,10]]$	

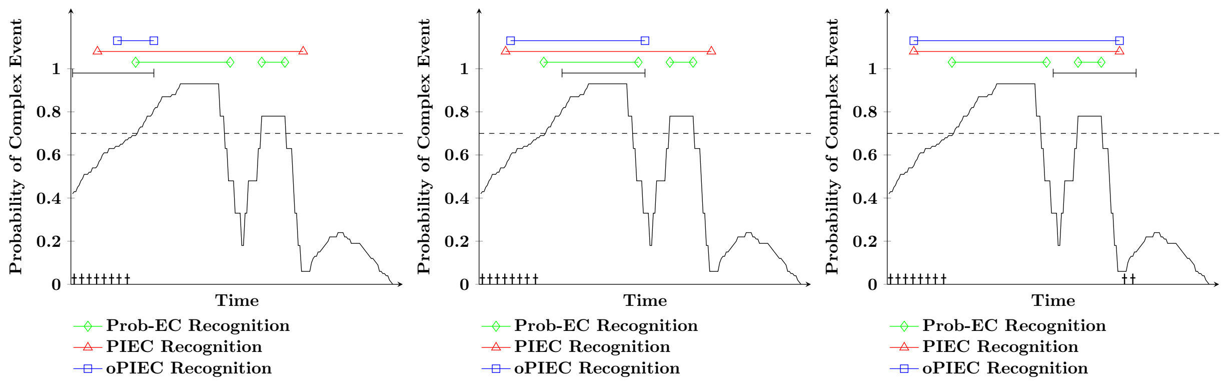


Figure 6.2: Probabilistic recognition over noisy data streams (continued from Figure 5.2). The solid black straight line denotes the current data batch. The black crosses on top of the horizontal axis denote the time-points stored in the support set, while the blue straight line denotes the recognition of oPIEC, given the data seen so far.

produces the PMI $[2,6]$ and terminates for this data batch, since $dprange$ is negative for all values of pointers s and e. When the third batch $In[9..10]$ arrives, the support set is $[(1,0),(2,-0.5),(8,-0.9)]$. Since $dp[9]=-0.9$ is less than any of the $prev\_prefix$ values of the first two elements of the support set, Algorithm 11 skips these elements, and sets $pt_{ss}=3$, for which $support\_set[3]=(8,-0.9)$. Finally, Algorithm 11 increments e as long as $dprange \geq 0$ and eventually computes the PMI $[8,10]$. $\Diamond$

In this example, oPIEC computes all PMIs of the dataset. In contrast, PIEC cannot compute any of the PMIs because, in the data partitioning of this example, the interval starting points have been discarded at the time when their ending points arrive at the system.

As another example, Figure 6.2 shows how oPIEC processes the instantaneous CE probabilities displayed in Figure 5.2 in three data batches. Figure 6.2 features three diagrams; each diagram shows the PMIs and the support set of oPIEC after processing each data batch. In the first batch (left diagram), the instantaneous probability of the CE is initially below the threshold $\mathcal{T}$, and it gradually increases, overcoming the threshold at some time-point t, i.e., $In[t] \geq \mathcal{T}$. oPIEC computes a PMI I_1 including all time-points $t' : t \leq t' \leq j$, where j indicates the end of the data batch, and as many of the previous time-points as possible, while maintaining that $P(I_1) \geq \mathcal{T}$. Next, oPIEC caches the potential starting points of future PMIs based on their $prev\_prefix$ values. All time-points in $[0,\ t]$ satisfy condition (6.2) and thus, according to Proposition 12, are cached in the support set.

In the second data batch (middle diagram), oPIEC extends I_1 to a longer PMI I_2 ending in this batch, because, given the newly arrived probability values, I_1 is not a PMI as it is a sub-interval of I_2 and $P(I_2) \geq \mathcal{T}$. Note that I_2 starts earlier than I_1 as the starting point of I_2 was cached in the support set. Moreover, the instantaneous probabilities in this data batch are either above the given threshold or temporally close to those that are above the threshold, and thus the support set does not have to be extended. In the third data batch (right diagram), oPIEC can extend I_2 further to PMI I_3, since $P(I_3) \geq \mathcal{T}$, and thus match the output of PIEC. This batch includes low instantaneous probabilities which decrease the values of the $prev\_prefix$ list. As a result, $prev\_prefix[t] < min\_prev\_prefix$ holds for some time-points t near the end of the data batch. All such time-points are cached in the support set because they may become starting points of PMIs if high probability values appear later in the stream.

6.1.3 Correctness and Complexity

We prove the correctness of oPIEC, i.e., we demonstrate that every interval computed by oPIEC is a PMI, and every PMI is computed by oPIEC, given the data seen so far. Moreover, we present the complexity analysis of the algorithms for updating the support set and computing PMIs. We provide proof sketches for the Lemmas and Propositions that follow. The full proofs are in Appendix C.

Algorithm 11 uses $dprange$ to identify PMIs. We associate $dprange$ with PMIs using the following lemma.

Lemma 3: If $I = [s, e]$ is a PMI, then $dprange[s, e]$ is non-negative. Moreover, for every interval $I' = [k, l]$, such that I is a sub-inteval of I', we have $dprange[k, l] < 0$. ▲

Proof Sketch. First, we prove that the fact that I has a probability above the chosen threshold implies that $dprange[s, e] \geq 0$. For the second part of the lemma, we assume that there is an interval $I' = [k, l]$, such that I is a sub-interval of I' and $dprange[k, l] \geq 0$, and prove that there must exist an interval I'' such that $I \subset I' \subseteq I''$ and the probability of I'' is greater or equal to the threshold. This is a contradiction because I is a PMI. Therefore, $dprange[k, l] < 0$. □

We proceed by presenting the proofs of soundness and completeness of oPIEC, focusing on PMIs starting with an element of the support set. The proofs for PMIs starting with an element of the current data batch are similar, and thus omitted.

In the analysis that follows, it is also useful to define the state of Algorithm 11 of oPIEC as a triple $\{pt_{ss}, e, flag\}$, in order to refer to the values of variables pt_{ss}, e and $flag$ at a specific iteration of the *while* loop of Algorithm 11. Recall that pt_{ss} and e are pointers traversing, respectively, the support set and the current data batch, indicating the starting and the ending point of a potential PMI, while $flag$ is a Boolean variable declaring whether $dprange$ was non-negative in the previous iteration. For every PMI in a dataset, oPIEC is guaranteed to reach the state corresponding to the starting and ending point of the PMI, as shown below in Lemma 4.

Lemma 4: Suppose that $In[i..j]$ is the current data batch, $[sst_1..sst_m]$ is the list of time-points in the support set, and the interval $I = [sst_k, l]$, where $1 \leq k \leq m$ and $i \leq l \leq j$, is a PMI. The execution of Algorithm 11 will eventually reach the state $\{k, l, true\}$ or the state $\{k, l, false\}$. ▲

Proof Sketch. Based on the condition of *while* loop of Algorithm 11, oPIEC reaches a state where either $pt_{ss} = k$ or $e = l$. We distinguish two possibilities for the first encounter of such a state. If oPIEC reaches the state $\{k', l, true/false\}$, where $k' < k$, then we prove that oPIEC increments pointer pt_{ss} in all subsequent iterations of the *while* loop, while keeping pointer e constant, until we reach the state $\{k, l, false\}$. Otherwise, if oPIEC reaches the state $\{k, l', true/false\}$, where $l' < l$, then, analogously, we prove that oPIEC increments pointer e in all subsequent iterations, while keeping pointer pt_{ss} constant, until we reach the state $\{k, l, true\}$. As a result, given that $I = [sst_k, l]$ is a PMI, the execution of Algorithm 11 reaches either the state $\{k, l, true\}$ or the state $\{k, l, false\}$. □

Proposition 13 (Soundness of Interval Computation): Every interval computed by oPIEC is a PMI of the data seen so far. ▲

Proof Sketch. We assume that $In[i..j]$ is the current data batch, $[sst_1..sst_m]$ is the list of time-points in the support set and that $I = [sst_k, l]$, where $1 \leq k \leq m, i \leq l \leq j$, is an interval computed by Algorithm 11. We prove that I is a PMI. We demonstrate that $dprange[sst_k, l] \geq 0$ and show that a non-negative $dprange$ value implies that the probability of I is greater or equal to the threshold. It remains to show that I is not a sub-interval of a PMI. We assume that I is a sub-interval of the PMI $I' = [sst_{k'}, l']$. Using Lemma 4, we show that oPIEC reaches the state $\{k', l'+1, true\}$. Since oPIEC computes I when its state is $\{k, l+1, true\}$, it suffices to show that the state $\{k, l+1, true\}$ cannot be reached starting from state $\{k', l'+1, true\}$. This result leads to a contradiction, and thus I is not a sub-interval of a PMI. Therefore, I is a PMI. □

Proposition 14 (Completeness of Interval Computation): oPIEC computes all PMIs of the data seen so far. ▲

Proof. Suppose that $In[i..j]$ is the current data batch, $[sst_1..sst_m]$ is the list of time-points in the support set, and $I = [sst_k, l]$ is a PMI, where $1 \leq k \leq m$ and $i \leq l \leq j$. Based on Lemma 4, oPIEC will eventually reach a state $\{k, l, true/false\}$ where $pt_{ss} = k$ and $e = l$. Also, since I is a PMI, from Lemma 3, we have that $dprange[sst_k, l] \geq 0$. Therefore, the *if* condition in line 5 of Algorithm 11 will be satisfied, e incremented and the $flag$ set to

$true$. In the following iteration, the state will become $\{k, l+1, true\}$. Since I is a PMI and a sub-interval of $[sst_k, l+1]$, based on Lemma 3, we have that $dprange[sst_k, l+1] < 0$. Therefore, the *if* condition will not be satisfied and, since the $flag$ will be set to $true$, I will be computed in line 7 of Algorithm 11. Hence, every PMI is computed by Algorithm 11. □

Propositions 13 and 14 show that, given the data seen so far, oPIEC computes every PMI and every interval computed by oPIEC is a PMI. Therefore, oPIEC is correct with respect to the data seen so far.

Before presenting the complexity of oPIEC, we revisit the complexity of PIEC [Artikis et al., 2021]. In PIEC, the computation of lists L, $prefix$ and dp takes linear time $\Theta(n)$, where n is the number of the input time-points. With respect to PMI computation, in the worst case, the two pointers looking for interval starting and ending points have to traverse the input list of size n. Therefore, this step requires $\mathcal{O}(2n)$. Consequently, the complexity of PIEC may be summarised by the following proposition:

Proposition 15 (PIEC complexity): The cost of computing the PMIs in a dataset of n time-points with PIEC is $\mathcal{O}(n)$. ▲

To update the support set, oPIEC iterates over the time-points of the current data batch $In[i..j]$. Therefore, this step is $\mathcal{O}(n_w)$, where n_w is the size of $In[i..j]$. The process of interval computation of oPIEC (Algorithm 11) iterates over the elements of the support set and time-points of $In[i..j]$. In the worst case, Algorithm 11 traverses the support set once and $In[i..j]$ twice. Hence, the complexity of interval computation is $\mathcal{O}(m + 2n_w)$, where m is the size of the support set. After simplifications, the complexity bound of Proposition 16 is reached.

Proposition 16 (oPIEC complexity): The cost of computing the PMIs of a data batch and updating the support set with oPIEC is $\mathcal{O}(m+n_w)$, where m is the size of the support set and n_w is the size of the data batch. ▲

The data batch size n_w can be constant, while the support set size m may increase over time. In rare cases, the support set may include a significant portion of the stream seen so far. Consider, e.g., the case where every probability value of the input stream is below the user-defined threshold. Then, the prefix list is strictly decreasing and oPIEC caches in the support set every time-point of the stream. In practice, $m + n_w$ is only a small fragment of the data seen so far, i.e., n. In other words, oPIEC is more efficient than PIEC, since the complexity of the latter increases much faster (with n) than the complexity of the former (which increases with m). In any case, streaming applications require constant complexity. To achieve this, the support set of oPIEC needs to be bounded. In the following section, we present ways in which this can be achieved.

6.2 oPIEC with Bounded Memory

In streaming applications, memory and performance requirements demand a bounded support set. We introduce oPIEC^b, that is, a version of oPIEC in which the support set has a bounded size. oPIEC^b is equipped with a support set maintenance algorithm which decides which elements, if any, should be deleted from the support set, in order to make room for new ones. Consequently, compared to oPIEC which computes all the PMIs of a dataset, oPIEC^b may detect, for instance, fewer or shorter intervals.

In practice, we observe that the support set of oPIEC^b often includes elements with time-stamps much prior to the current time. In most applications, each CE has a typical duration and it is not useful to anticipate an instance of a CE which exceeds that duration significantly. As an example, in human activity recognition, it is improbable that two people are fighting for more than five minutes, and thus support set elements with temporal distance greater than five minutes from the current time are very unlikely to be useful for computing PMIs. To take advantage of this property, we propose an extension of the support set maintenance algorithm of oPIEC^b in order to consider CE-specific statistics concerning the duration of PMIs. In the following, we first outline the support set maintenance algorithm of oPIEC^b, and then present oPIEC^{bd}, the variant of oPIEC^b which considers the expected PMI duration.

6.2.1 Support Set Maintenance with oPIEC^b

The support set maintenance algorithm of oPIEC^b works as follows. When a time-point t satisfying condition (6.2) arrives, i.e., t may be the starting point of a PMI, oPIEC^b considers caching it in the support set. If the designated support set limit is exceeded, then oPIEC^b decides whether to cache t, replacing some older time-point of the support set, by computing the ‘score range’, i.e., an interval of real numbers defined as follows:

$$score\_range[t] = [prev\_prefix[t], prev\_prefix[prev_s[t]]) \tag{6.3}$$

The $score\_range$ is computed for the time-points in set S, i.e., the time-points already in the support set and the time-points that are candidates for the support set. The elements in S are tuples of the form $(t, prev\_prefix[t])$, which are sorted in ascending time-point order and, since they satisfy condition (6.2), these tuples are also sorted in descending $prev\_prefix$ order. $prev_s[t]$ is the time-point before t in S.

With the use of $score\_range[t]$, oPIEC^b computes the likelihood that a time-point t in set S will indeed become the starting point of a PMI. The longer the $score\_range[t]$, i.e., the longer the distance between $prev\_prefix[t]$ and $prev\_prefix[prev_s[t]]$, the more likely it is that a future time-point t_e will arrive with $dp[t_e] \in score\_range[t]$, and thus t will be the starting point of a PMI. Therefore, oPIEC^b stores in the support set the elements with the longer score range.

Example 27: Consider the dataset $In[1..5]$ presented in Table 6.2. With a threshold value $\mathcal{T}$ of 0.5, this dataset has a single PMI: $[2, 5]$. Assume that the data arrive in two batches:

Table 6.2: Support set maintenance of oPIEC^b.

Time	**1**	**2**	**3**	**4**	**5**
In	0	0.3	0.3	0.6	0.9
L	-0.5	-0.2	-0.2	0.1	0.4
$prefix$	-0.5	-0.7	-0.9	-0.8	-0.4
$prev\_prefix$	0	-0.5	-0.7	-0.9	-0.8
dp	-0.5	-0.7	-0.8	-0.8	-0.4

$In[1..4]$ and $In[5]$. Given an unbounded support set, oPIEC would have cached time-points 1, 2, 3 and 4 into the support set. Assume now that the limit of the support set is set to two elements. oPIEC^b processes $In[1..4]$ to detect the time-points that may be used as starting points of PMIs. These are time-points 1, 2, 3 and 4. In order to respect the support set limit, oPIEC^b computes the score ranges:

- $score\_range[1]$ is set to $[0, +\infty)$ since $t = 1$ has no predecessor in the support set.
- $score\_range[2] = [prev\_prefix[2], prev\_prefix[1]) = [-0.5, 0)$.
- $score\_range[3] = [prev\_prefix[3], prev\_prefix[2]) = [-0.7, -0.5)$.
- $score\_range[4] = [prev\_prefix[4], prev\_prefix[3]) = [-0.9, -0.7)$.

Given these $score\_range$ values, oPIEC^b caches the tuples $(1, 0)$ and $(2, -0.5)$ in the support set, since these are the elements with the longest score ranges.

With such a support set, oPIEC^b is able to perform correct CER, i.e., compute PMI $[2, 5]$, upon the arrival of the second data batch $In[5]$. Note that we have $dprange[2, 5] = 0.1 \geq 0$ and $t = 5$ is the last time-point of the data stream so far. Also, for the other time-point of the support set, $t = 1$, we have a negative $dprange$, i.e., $dprange[1, 5] = -0.4 < 0$, and hence a PMI cannot start from $t = 1$. $\diamondsuit$

Figure 6.3 describes how oPIEC^b processes the instantaneous CE probabilities displayed in Figure 5.2 in three data batches, using a support set which may hold at most three elements. After processing the first data batch (left diagram), oPIEC^b computes the PMI I_1. Then, it identifies the same potential starting points of PMIs as oPIEC, but caches in the support set only three of them. These are the three starting points with the longest $score\_range$.

After processing the second data batch (middle diagram), oPIEC^b extends I_1. There are no potential starting points of PMIs in this data batch, and thus the support set of oPIEC^b is not updated. When oPIEC^b processes the third data batch (right diagram), it extends further I_1 to I_3^b. I_3^b, however, is not a PMI—compare it against the interval computed by PIEC and oPIEC. The starting point of the PMI was not cached in the support set after processing the first data batch, and oPIEC^b could only compute a slightly shorter interval. oPIEC^b updates the support set with a new element, because the low probability values near the end of this data batch have caused a large decrease in the values of $prefix$. As

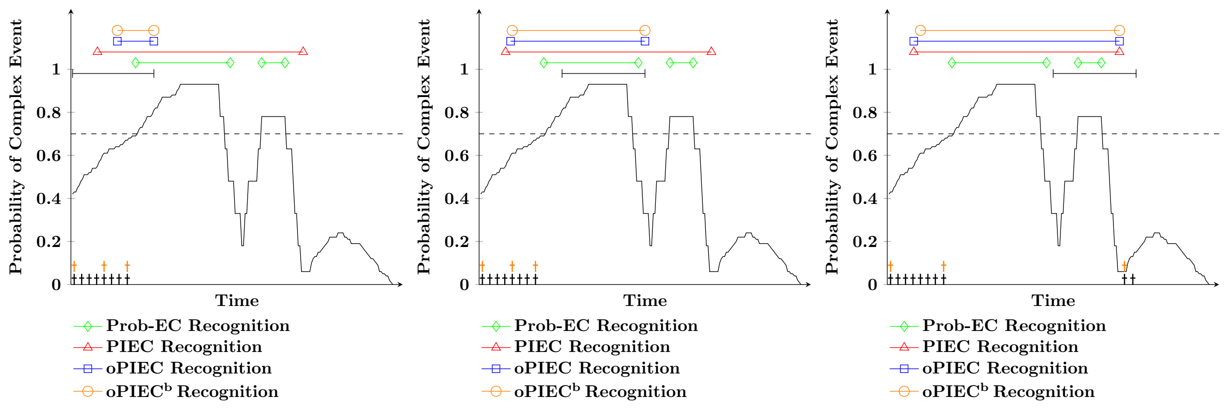


Figure 6.3: Probabilistic recognition over noisy data streams (continued from Figure 6.2). The orange straight line expresses the recognition of oPIEC^b after processing the current data batch. Black crosses denote the potential starting points of future PMIs computed by oPIEC/oPIEC^b and cached by oPIEC. Orange crosses depict the ones cached in the bounded support set of oPIEC^b after processing the current data batch.

a result, the first candidate element of this batch has a longer $score\_range$ than one of the time-points previously cached in the support set, and thus takes its place.

6.2.2 Support Set Maintenance with oPIEC^{bd}

It is often the case that oPIEC^b maintains in the support set elements whose time-points are much earlier than the current data batch, while the duration of a CE is usually within some bounds. See, for example, the first cached time-point, depicted as an orange cross, in the right diagram of Figure 6.3—it is unlikely that this time-point will become the starting point of a PMI in a future data batch. To address this issue, we introduce oPIEC^{bd}, a variant of oPIEC^b with a support set maintenance algorithm taking advantage of CE-specific knowledge regarding PMI duration, which may be obtained by observing historical data. We assume that the duration of PMIs is normally distributed and oPIEC^{bd} is given the mean μ and the standard deviation σ of the observed values. During online execution, when support set conflicts arise, oPIEC^{bd} utilises this information for support set maintenance.

The support set maintenance algorithm of oPIEC^{bd} works as follows. First, oPIEC^{bd} constructs the set S which contains every element of the support set and the candidate elements. When the size of S is greater than the support set limit m_b, oPIEC^{bd} computes, for each element t_i in S, its least duration $ld_i = t_{now} - t_i + 1$, where t_{now} expresses the last time-point of the current batch. ld_i denotes the minimum duration of a PMI starting from t_i and ending in some future data batch. (Upon each incoming data batch, support set maintenance takes place after PMI computation. At this stage, therefore, oPIEC^{bd} has computed the PMIs ending in the current data batch.) Then, oPIEC^{bd} calculates the dele-

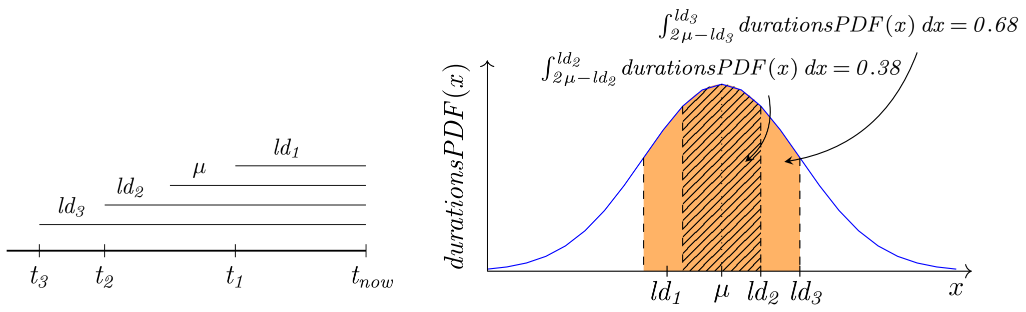

ld_1
μ
ld_2
ld_3
t_3
t_2
t_1
t_{now}
$\int_{2\mu-ld_3}^{ld_3} durationsPDF(x)\,dx = 0.68$
$\int_{2\mu-ld_2}^{ld_2} durationsPDF(x)\,dx = 0.38$
$durationsPDF(x)$
ld_1
μ
ld_2
ld_3
x

Figure 6.4: The deletion probabilities of the elements of set S whose time-points are t_1, t_2 and t_3. The left diagram shows the temporal positions of these time-points in relation to the last time-point of the current batch t_{now} and their least durations ld_1, ld_2 and ld_3 in relation to the mean duration μ of the observed values. The right diagram visualises the process of calculating the deletion probabilities of these time-points. For time-point t_1, $delPr(t_1) = 0$ because $ld_1 < \mu$. For time-points t_2 and t_3, whose least durations are greater than μ, $delPr$ is equal to the area under $durationsPDF$ defined by the integral of equation (6.4). As an example, $delPr(t_2)$ is equal to the area under $durationsPDF$ between $2\mu - ld_2$ and ld_2, i.e., $delPr(t_2) = \int_{2\mu-ld_2}^{ld_2} durationsPDF(x)\,dx$.

tion probability of t_i, defined as follows:

$$delPr(t_i) = \begin{cases} 0 & ld_i \leq \mu \\ \int_{2\mu-ld_i}^{ld_i} durationsPDF(x)\,dx & ld_i > \mu \end{cases} \tag{6.4}$$

$durationsPDF$ is the probability density function of the normal distribution $\mathcal{N}(\mu, \sigma^2)$ with mean μ and standard deviation σ of the observed duration values. Thus, $\int_{-\infty}^{y} durationsPDF(x)\,dx$ is equal to the probability that a sample generated from $\mathcal{N}(\mu, \sigma^2)$ is smaller than y. As an example, $\int_{-\infty}^{\mu} durationsPDF(x)\,dx = 0.5$ denotes that a sample generated from $\mathcal{N}(\mu, \sigma^2)$ is shorter than the observed mean value μ with a probability equal to 50%. According to expression (6.4), the deletion probability $delPr(ld_i)$ of a time-point t_i is 0 when its least duration ld_i is shorter or equal to μ. In such cases, it is possible for a future PMI starting from t_i to have a duration shorter or equal to the mean μ of the observed duration values, and therefore it is too early to delete t_i. Otherwise, when all possible future PMIs starting from t_i may have a duration greater than μ, i.e., $ld_i > \mu$, $delPr(ld_i)$ is positive and increases with the probability that a sample generated from the given probability distribution is smaller than ld_i, i.e., the area under the probability density function up to ld_i. The integral in expression (6.4), however, starts from $2\mu - ld_i$ instead of $-\infty$ in order to have $delPr(\mu) = 0$ and $delPr(\infty) = 1$, i.e., $delPr$ is a cumulative distribution function.

As an example, Figure 6.4 presents the deletion probabilities of three elements with time-points t_1, t_2 and t_3, given that the observed duration values follow the normal distribution $\mathcal{N}(\mu, \sigma^2)$. The left diagram of Figure 6.4 shows the temporal positions of these time-points in relation to t_{now}, as well as their least durations. The least durations of t_1, t_2 and t_3 are

$ld_1 < \mu$, $ld_2 = \mu + \frac{1}{2}\sigma$ and $ld_3 = \mu + \sigma$, respectively. The right diagram of Figure 6.4 shows that the deletion probability of t_1 is 0, because $ld_1 < 0$ (see expression (6.4)). Moreover, the deletion probabilities of t_2 and t_3, which are equal to the line-highlighted and the orange-coloured areas, respectively, are derived by computing the integral of expression (6.4). As an example, for time-point t_3, whose least duration is $ld_3 = \mu + \sigma$, we have:

$$delPr(t_3) = \int_{2\mu - ld_3}^{ld_3} durationsPDF(x)\, dx = \int_{\mu-\sigma}^{\mu+\sigma} durationsPDF(x)\, dx = 0.68$$

After each element is assigned a deletion probability, it is removed from set S if this probability is greater than a threshold generated randomly from the uniform distribution $\mathcal{U}(0, 1)$. Since this is a stochastic process, it is not guaranteed that the length of S will be equal to m_b after iterating over all of its elements. If the length of S remains greater than m_b, oPIEC^{bd} invokes the support set maintenance algorithm of oPIEC^{b} to resolve the remaining conflicts.

Algorithm 12 supportSetFiltering(S, m_b, μ, σ, t_{now})

Input: Set S containing the support set elements and the candidate elements of the current data batch, the maximum size of the bounded support set m_b, the mean and the standard deviation of the given duration values μ and σ, and the last time-point of the current data batch t_{now}.

Output: Set S containing m_b tuples which comprise the new support set.

```
1: $durationsPDF \leftarrow \mathcal{N}(\mu, \sigma^2)$
2: for each $(t_i, prev\_prefix[t_i]) \in S$ do
3:     $ld_i \leftarrow t_{now} - t_i + 1$
4:     if $length(S) > m_b$ and $ld_i > \mu$ then
5:         $delPr(ld_i) \leftarrow \int_{2\mu - ld_i}^{ld_i} durationsPDF(x)\, dx$
6:         $threshold \leftarrow random(\mathcal{U}(0, 1))$
7:         if $delPr(ld_i) > threshold$ then $S.delete((t_i, prev\_prefix[t_i]))$
8: if $length(S) > m_b$ then $S \leftarrow$ supportSetMaintenance($S$, $length(S) - m_b$)
9: return $S$
```

Algorithm 12 describes the support set maintenance algorithm of oPIEC^{bd}. First, oPIEC^{bd} generates $durationsPDF$ (line 1). Then, for each time-point t_i in set S, oPIEC^{bd} computes its least duration ld_i (line 3). If the support set limit is exceeded, Algorithm 12 checks whether ld_i is greater than μ (line 4). If it is, we compute the deletion probability of time-point t_i (line 5). Subsequently, Algorithm 12 generates a threshold based on the uniform distribution $\mathcal{U}(0, 1)$ (line 6) and deletes the element of S whose time-point is t_i if its deletion probability exceeds that threshold (line 7). Finally, if the support set limit is still violated after iterating over all elements of S, Algorithm 12 invokes the support set maintenance algorithm of oPIEC^{b} (line 8). After this step, the length of S is guaranteed to be equal to the maximum support set size, and thus Algorithm 12 returns S as the new support set (line 9).

The following example illustrates oPIEC^{bd} and compares it against oPIEC^{b}.

Example 28: Suppose that oPIEC^b and oPIEC^{bd} operate on the data batches $In[1..4]$, $In[5..8]$ and $In[9..10]$ of Example 26. The maximum size of the support set for both systems is two elements. Recall that in Examples 25 and 26, given the same data stream, oPIEC computed the following support set and PMIs:

$$support\_set = [(1,0),(2,-0.5),(8,-0.9),(9,-1.4)]$$
$$PMIs = [[1,5],[2,6],[8,10]]$$

After processing $In[1..4]$, the support set of both oPIEC^b and oPIEC^{bd} is $[(1,0),(2,-0.5)]$. Then, when $In[5..8]$ arrives, both systems compute the first two PMIs: $[1,5]$ and $[2,6]$. Subsequently, both systems attempt to add the tuple $(8,-0.9)$ to the support set, i.e., $S = [(1,0),(2,-0.5),(8,-0.9)]$, and thus the respective maintenance algorithm is invoked, since the support set size is limited to two elements.

oPIEC^b computes the $score\_range$ of each element of S:

- $score\_range[1] = [0,+\infty)$.
- $score\_range[2] = [-0.5,0)$.
- $score\_range[8] = [-0.9,-0.5)$.

Then, oPIEC^b removes element $(8,-0.9)$ because it has the shortest $score\_range$. Therefore, when data batch $In[9..10]$ arrives, oPIEC^b cannot compute the interval $[8,10]$.

Suppose that the normal distribution fitting the duration data seen so far has $\mu = 4.3$ and $\sigma = 2.4$. Given $t_{now} = 8$, i.e., the last time-point of the current data batch $In[5..8]$, for the first element of S, where $t_i = 1$, we have $ld_i = t_{now} - t_i + 1 = 8$. This means that any future PMI starting from the first element of S will have a duration of at least 8 time-points. However, the PMI duration statistics indicate that PMIs very rarely exceed 8 time-points. When $ld_i > \mu$, oPIEC^{bd} calculates the deletion probability of each element of the set, which for $t_i = 1$ is $delPr(1) = 0.88$. Therefore, element $(1,0)$ will very likely be deleted from S. If this happens, the support set of oPIEC^{bd} when $In[9..10]$ arrives will be $[(2,-0.5),(8,-0.9)]$ and the final PMI, i.e., $[8,10]$, will be computed. If $t = 1$ is not deleted, $t = 2$ will be deleted with probability $delPr(2) = 0.74$, leading again to the computation of PMI $[8,10]$. $\Diamond$

Figure 6.5 demonstrates how oPIEC^{bd} processes the instantaneous CE probabilities displayed in Figure 5.2 in three data batches, using a support set which may hold, at most, three elements. Similarly to oPIEC^b, after processing the first data batch (left diagram), oPIEC^{bd} computes all the potential starting points of future PMIs, but caches only three of them. Algorithm 12, using duration statistics regarding the target CE, derives that some of the earlier time-points of the data batch are not likely to be starting points of future PMIs, because these statistics favour intervals with a shorter duration. As a result, the support set maintenance algorithm of oPIEC^{bd} has cached more recent time-points, compared to oPIEC^b. Finally, after processing the third data batch (right diagram), oPIEC^{bd} computes the correct interval, i.e., the PMI computed by PIEC and oPIEC, because the starting point of the PMI is in the support set.

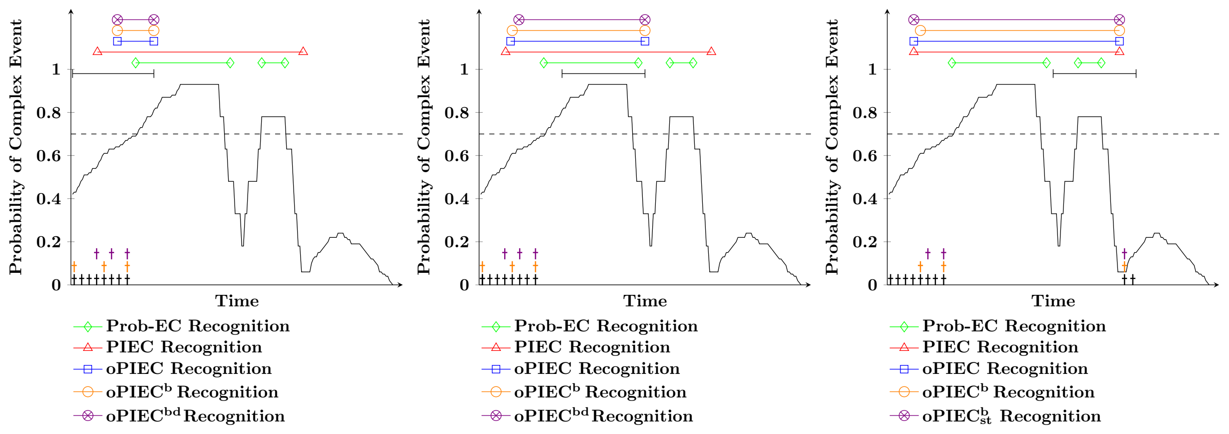


Figure 6.5: Probabilistic recognition over noisy data streams (continued from Figure 6.3). The violet straight line expresses the recognition of oPIEC^{bd} after processing the current data batch. Violet crosses depict the potential starting points of future PMIs cached by oPIEC^{bd} after processing the current data batch.

6.2.3 Complexity of Support Set Maintenance

In oPIEC, the cost of PMI computation given a data batch with size n_w and a support set with size m is $\mathcal{O}(m+n_w)$ (see Section 6.1.3). In oPIEC^b and oPIEC^{bd}, we have a bounded support set with, at most, m_b elements. The choice of the value of m_b constitutes a trade-off between accuracy and efficiency. For small values of m_b, it is more likely that a PMI is missed because its starting point is not present in the support set. Choosing a larger value for m_b, however, may hinder the efficiency of oPIEC.

Suppose that the support set is full (m_b elements) and there are n_w new candidate elements—in the worst case, there is one candidate element for each time-point in the current data batch $In[i..j]$. Both oPIEC^b and oPIEC^{bd} utilise set S, containing the elements of the support set and the candidate elements, i.e., m_b+n_w elements in total. The support set maintenance algorithm of oPIEC^b removes the n_w elements of S with the shortest score ranges. This is achieved with an iterative process which compares the score range of each element in S, with the largest score range among the n_w elements of S with the shortest score ranges found so far. This operation has a cost of $\mathcal{O}(m_b n_w)$. In total, the cost of oPIEC^b amounts to that of interval computation ($\mathcal{O}(m_b + n_w)$), support set candidate element identification ($\mathcal{O}(n_w)$, because oPIEC checks the $prev\_prefix$ value of each time-point in the data batch) and support set maintenance ($\mathcal{O}(m_b n_w)$). After simplifications, the complexity of oPIEC^b is presented in Proposition 17.

Proposition 17 (oPIEC^b complexity): The cost of computing the PMIs of a data batch and updating the support set with oPIEC^b is $\mathcal{O}(m_b n_w)$, where m_b is the size of the bounded support set and n_w is the size of the data batch. ▲

In the case of oPIEC^{bd}, support set maintenance is performed by Algorithm 12. The filtering step of Algorithm 12 (lines 1–7) iterates over each element of S once, and removes it from S if its deletion probability exceeds a threshold. Therefore, the cost of this step is $\mathcal{O}(m_b{+}n_w)$. Afterwards, if the support set limit is still exceeded, oPIEC^{bd} invokes the support set maintenance algorithm of oPIEC^{b}, whose complexity is $\mathcal{O}(m_b n_w)$. Therefore, the cost of support set maintenance in oPIEC^{bd} is $\mathcal{O}(m_b{+}n_w{+}m_b n_w)$. After simplifications, the complexity of oPIEC^{bd} is presented in Proposition 18.

Proposition 18 (oPIEC^{bd} complexity): The cost of computing the PMIs of a data batch and updating the support set with oPIEC^{bd} is $\mathcal{O}(m_b n_w)$, where m_b is the size of the bounded support set and n_w is the size of the data batch. ▲

In other words, the support set maintenance algorithm of oPIEC^{bd} (Proposition 17) is of the same complexity order as the corresponding algorithm of oPIEC^{b} (Proposition 18). This is verified by the empirical comparison presented in the following section.

In contrast to oPIEC^{bd} and oPIEC^{b}, the cost of oPIEC with an unbounded support set is $\mathcal{O}(m{+}n_w)$, where m is the size of the unbounded support set (see Proposition 16). m increases as the stream progresses, whereas m_b remains constant, and thus, in practice, $m_b \ll m$. Therefore, oPIEC with a bounded support set is the preferred choice for streaming applications.

6.3 Discussion

We presented oPIEC, a formal computational framework for online, interval-based CER over noisy events. oPIEC consumes the output of a point-based CER system to compute the most likely maximal intervals during which a composite activity is said to take place. oPIEC employs a ‘support set’, a memory structure with the minimal set of time-points that guarantee correct interval computation. We provided a theoretical analysis of oPIEC, proving its correctness and presenting its complexity, which demonstrates that the cost of oPIEC is linear to the size of the data batch and the size of the support set. Moreover, we proposed two bounded-memory versions of oPIEC, i.e., oPIEC^{b} and oPIEC^{bd}, in order to support stream reasoning with very limited memory. When the size of the support set exceeds the memory threshold, oPIEC^{b} and oPIEC^{bd} employ an algorithm that decides which elements of the support set will be deleted in order to make room for new ones. oPIEC^{b} keeps in memory the elements that are more likely to become starting points of CE intervals in the future, provided that we are agnostic about the expected duration of the CE. In contrast, oPIEC^{bd} leverages interval duration statistics to resolve memory conflicts. We assessed both oPIEC^{b} and oPIEC^{bd} by means of complexity analyses, demonstrating that our memory maintenance algorithms do not introduce a significant overhead in reasoning efficiency.

The variants of oPIEC that we proposed have several benefits compared to the state of the art (see Tables 5.3 and 5.4). Our frameworks are based on logic programming, and thus seamlessly support an explicit representation of time, which has several advantages compared to an implicit representation (see Section 5.4), as well as background knowledge in temporal specifications. Moreover, oPIEC performs interval-based reasoning, which is, in many cases, more accurate than point-based reasoning [Artikis et al., 2021]. The literature includes several interval-based frameworks that handle uncertainty, such as PEL, MLN-Allen and CP-logic; none of these frameworks derives the intervals of CEs with a single pass over the input stream. In oPIEC^{b} and oPIEC^{bd}, the cost of reasoning depends only on the size of the data batch and the size of the working memory, which are constant and low. In Chapter 7, we present an extensive, reproducible empirical analysis of oPIEC^{b} and oPIEC^{bd}, demonstrating the benefits of using interval duration statistics and showing that our frameworks outperform state-of-the-art systems.

7. EXPERIMENTAL EVALUATION OF oPIEC

We present an experimental evaluation of oPIEC^b and oPIEC^{bd}, i.e., our interval-based frameworks for reasoning over noisy data streams. In Section 7.1, we describe the applications that we employed in our experiments. In Section 7.2, we present our experiments. Section 7.3 provides commentary on our results.

7.1 Experimental Setup

We describe the setup of the experiments concerning human activity recognition and maritime situational awareness. We present the input data, i.e., the simple, derived events (SDE)s and accompanying contextual data, and the output composite events (CE)s. In all experiments, the task of oPIEC^{bd} is to compute the PMIs of CEs.

Human Activity Recognition

Data. To evaluate our work, we used CAVIAR[1], a benchmark activity recognition dataset. CAVIAR includes 28 videos with 26,419 video frames in total. The videos are staged, i.e., actors walk around, sit down, meet one another, fight, etc. Each video has been manually annotated by the CAVIAR team in order to provide the ground truth for both SDEs, taking place on individual video frames, as well as CEs. Table 7.1 describes the SDEs and the CEs of CAVIAR. The input to the activity recognition system consists of SDEs such as 'inactive', i.e., standing still, 'active', i.e., non-abrupt body movement in the same position, 'walking' and 'running', together with their time-stamps, that is, the video frame in which the SDE took place. The dataset also includes the coordinates of the tracked people and objects as pixel positions at each time-point, as well as their orientation. Consider the following example:

$$\begin{aligned}&\text{happensAt}(walking(id_0),\ 680).\\&\text{holdsAt}(coord(id_0) = (262, 285),\ 680).\\&\text{holdsAt}(orientation(id_0) = 0,\ 680).\end{aligned}$$

According to this video frame annotation, id_0 is walking at video frame 680, located at $(262, 285)$ in the two dimensional projection of the video, and facing towards the horizontal axis of this projection.

CAVIAR includes inconsistencies, as the members of the CAVIAR team that provided the annotation for SDEs and CEs did not always agree with each other [List et al., 2005; Skarlatidis et al., 2015a]. Furthermore, to allow for a more demanding evaluation, Skarlatidis et al. [2015] injected noise into CAVIAR, producing the following datasets:

[1] *https://groups.inf.ed.ac.uk/vision/DATASETS/CAVIAR/CAVIARDATA1/*

Table 7.1: The input SDEs (above the dashed line) and contextual data (below the dashed line), and the output CEs of human activity recognition.

	Entity	Description
SDEs & contextual data	$walking(P)$	Person P is walking.
	$running(P)$	Person P is running.
	$active(P)$	Person P performs non-abrupt body movements without changing position.
	$inactive(P)$	Person P is standing still.
	$abrupt(P)$	Person P performs abrupt body movements.
	$appear(P)$	Person P starts being tracked.
	$disappear(P)$	Person P stops being tracked.
	$coord(P)=(X, Y)$	Person P is at position (X, Y).
	$orientation(P) = \Theta$	Person P is facing at an angle of Θ degrees w.r.t. the x-axis in the two-dimensional video projection.
CEs	$moving(P_1, P_2)$	Persons P_1 and P_2 are moving together.
	$meeting(P_1, P_2)$	Persons P_1 and P_2 are having a meeting.
	$fighting(P_1, P_2)$	Persons P_1 and P_2 are fighting.

- *Smooth noise*: SDEs have been attached with probability values, generated by a Gamma distribution with a varying mean, which signify the probability of their occurrence.
- *Strong noise*: Apart from SDEs, probabilities have also been attached to contextual information (coordinates and orientation) using the same Gamma distributions. Moreover, spurious SDEs that do not belong to the original dataset have been added using a uniform distribution.

Task. Given the SDEs and contextual data of each video frame, the task is to recognise the CEs ‘moving’, ‘meeting’ and ‘fighting’.

Evaluation. We evaluated the predictive accuracy of oPIEC^{bd} and oPIEC^{b} on the CAVIAR dataset for human activity recognition. The streams of instantaneous CE probabilities we used as input were generated either by Prob-EC or OSLα. Recall that Prob-EC is an implementation of the Event Calculus in ProbLog, designed to handle data uncertainty (see Section 5.2). OSLα translates the Event Calculus into Markov Logic [Skarlatidis et al., 2015b], and uses supervised learning to generate weighted CE definitions [Michelioudakis et al., 2016; Michelioudakis et al., 2019]. OSLα is not designed to handle probabilistic data, and thus was trained on the original CAVIAR dataset (see [Michelioudakis et al., 2016] for the setup of the training process). The CE definitions used by Prob-EC were constructed manually and are not weighted.

Table 7.2: The input SDEs (above the dashed line) and contextual data (below the dashed line), and the output CEs of maritime situational awareness.

	Entity	Description
SDEs & contextual data	$enterArea(V, A)$	Vessel V enters area A.
	$leavesArea(V, A)$	Vessel V exits area A.
	$gapStart(V)$	Vessel V stops sending position signals.
	$gapEnd(V)$	Vessel V resumes sending position signals.
	$stopStart(V)$	Vessel V starts being idle.
	$stopEnd(V)$	Vessel V stops being idle.
	$slowMotionStart(V)$	Vessel V starts moving at a low speed.
	$slowMotionEnd(V)$	Vessel V stops moving at a low speed.
	$proximityStart(V_1, V_2)$	Vessels V_1 and V_2 start being close to each other.
	$proximityEnd(V_1, V_2)$	Vessels V_1 and V_2 stop being close to each other.
	$coord(V) = (Lat, Lon)$	Vessel V is at position (Lat, Lon).
	$velocity(V) = S$	Vessel V sails at speed S.
CEs	$rendezVous(V_1, V_2)$	Vessels V_1 and V_2 are nearby in the open sea, stopped or sailing at a low speed.
	$tugging(V_1, V_2)$	Vessels V_1 or V_2 is a tugboat and is pulling or towing the other vessel.

The support set maintenance algorithm of oPIEC^{bd} requires CE duration statistics. For this reason, the presented experiments were performed using 5-fold cross-validation. We made sure that all folds were balanced with respect to the number of CE instances, and that no video was split between the training and test sets. In each fold, the durations of the PMIs identified by PIEC in the training set were used when testing oPIEC^{bd} in the corresponding test set. In other words, oPIEC^{bd} had to compute the PMIs in the test set given the mean and the standard deviation of the PMIs of the training set. The overall (micro) f1-score was derived by combining the results of all folds. A perfect f1-score implies that the intervals computed by oPIEC^{b}/oPIEC^{bd} match precisely the PMIs of PIEC. oPIEC^{bd} and oPIEC^{b} operated on data batches of one time-point, i.e., performing reasoning at every time-point and then discarding it, unless cached in the support set.

Maritime Situational Awareness

Data. To test further oPIEC^{bd}, we employed a publicly available dataset[2] which includes 18 million AIS position signals collected from 5 thousand ships sailing in the area of Brest,

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

France, for a period of six months, between October 2015 and March 2016. AIS messages comprise dynamic information (position, speed, etc.), as well as static information (name, type, etc.) about vessels [Bereta et al., 2021]. This dataset has been pre-processed by means of trajectory simplification [Patroumpas et al., 2017; Fikioris et al., 2023] and spatial processing [Santipantakis et al., 2018]. The former process annotated position signals of interest as 'critical', signifying major changes in a vessel's behaviour such as a stop, a turn, and a gap in signal transmission, while the latter process computed spatial relations between vessels, such as vessels being close to each other. Table 7.2 describes the SDEs and CEs of the Brest dataset. Consider the following example of input data:

$$\begin{aligned}&\text{happensAt}(\mathit{leavesArea}(\mathit{id_2}, \mathit{fishing}),\ \mathit{492}).\\&\text{holdsAt}(\mathit{coord}(\mathit{id_2}) = (\mathit{48.12}, -\mathit{4.35}),\ \mathit{492}).\\&\text{holdsAt}(\mathit{velocity}(\mathit{id_2}) = \mathit{5.13},\ \mathit{492}).\end{aligned}$$

According to these records, vessel $\mathit{id_2}$ leaves some fishing area at time-point $\mathit{492}$; moreover, $\mathit{id_2}$ is located at $(\mathit{48.12}, -\mathit{4.35})$ and sailing with a speed of $\mathit{5.13}$ knots.

We injected noise into the dataset of Brest by assigning a probability value to every item of the dataset. We followed [Zocholl et al., 2019] and assumed that the confidence of an AIS signal decreases as the distance of the corresponding vessel from the nearest coastline base station increases, and annotated each signal of the dataset with a probability using the following function:

$$P_c(x) = \mathit{1} - \frac{x}{x + c}$$

x is the distance of the ship from the nearest base station and c is a distance threshold denoting our confidence concerning the veracity of signals. According to the function above, the probability of a position signal decreases as the value of c decreases. We constructed a 'smooth noise' and a 'strong noise' version of the Brest dataset by applying the noise functions P_{10000} and P_{5000} on the original dataset.

Task. Given the input presented in Table 7.2, the task is to recognise the CEs 'rendez-vous', i.e., a potential ship-to-ship transfer of goods in the open sea, and 'tugging', i.e., the activity of pulling a ship into or out of a port. These CEs were defined following expert opinion [Pitsikalis et al., 2019].

Evaluation. For our experiments on maritime situational awareness, we employed Prob-EC to produce probability values for the instantaneous occurrences of the target CEs, i.e., 'rendez-vous' and 'tugging'. Subsequently, we evaluated oPIEC^{b} and oPIEC^{bd} on the derived probability streams. We compared the intervals computed by oPIEC^{b} and oPIEC^{bd} when processing online the CE probabilities of Prob-EC against the PMIs computed by PIEC when processing the same data as a single batch. The duration statistics required by oPIEC^{bd} were derived by a cross-validation setting similar to that of the experiments on human activity recognition. We made sure that the folds were balanced in terms of CE instances, and that the set of vessels in the training set is disjoint from the set of vessels in the test set. The batch size of oPIEC^{b} and oPIEC^{bd} was set to one, while the probabilistic threshold $\mathcal{T}$ was set to $\mathit{50}\%$.

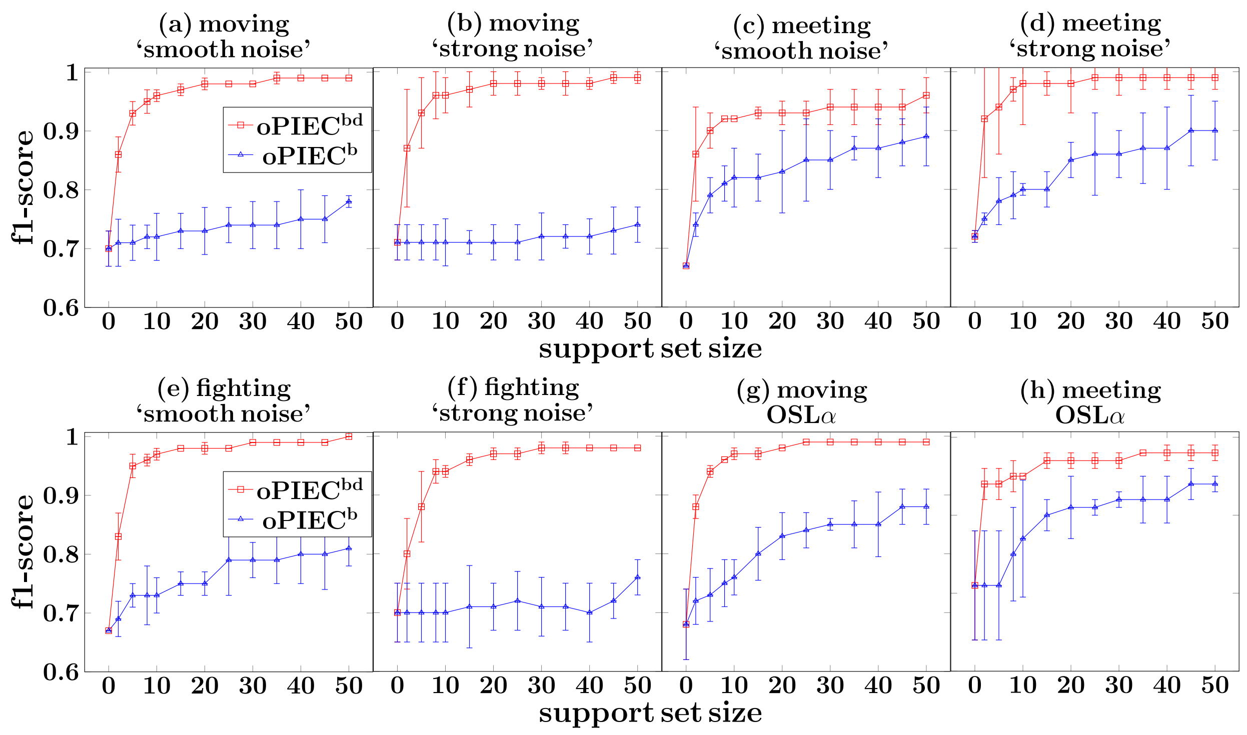


Figure 7.1: The predictive accuracy of oPIEC^{bd} and oPIEC^{b} on human activity recognition. The intervals computed by oPIEC^{bd} and oPIEC^{b} are compared to the PMIs computed by PIEC. The point-based systems providing the input probability streams are Prob-EC (diagrams (a)–(f)) and OSLα (diagrams (g)–(h)).

PIEC, oPIEC^{b} and oPIEC^{bd} are written in Python. For the remaining frameworks of our evaluation (see 'Comparisons with the State of the Art' in Section 7.2), we used the programming language (version) recommended by its developers. We used a single core of a desktop PC running Ubuntu 20.04, with Intel Core i7-4770 CPU @3.40GHz and 16GB RAM. The code of oPIEC^{b} and oPIEC^{bd}, as well as the complete event descriptions and the datasets we employed, are publicly available[3], allowing the *reproducibility* of our empirical analysis.

7.2 Experimental Results

Human Activity Recognition. Figure 7.1 presents the predictive accuracy of oPIEC^{bd} and oPIEC^{b}, operating with a probabilistic threshold $\mathcal{T} = 50\%$; using this value, PIEC outperforms point-based recognition. These results demonstrate that the use of oPIEC^{bd} leads to significant performance gains compared to oPIEC^{b}. In all cases, oPIEC^{bd} outperforms oPIEC^{b}, highlighting the importance of taking into consideration PMI duration statistics. Note that in the case of 'meeting' oPIEC^{bd} performs better under 'strong noise' than 'smooth noise', when Prob-EC provides the input stream (see Diagrams 7.1(c) and

[3]*https://github.com/periklismant/opiec*

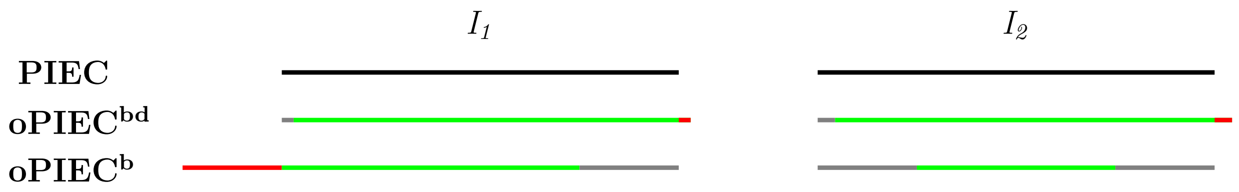


Figure 7.2: Indicative comparison of the intervals computed by PIEC (top) against those computed by oPIEC$^{\mathrm{bd}}$ (middle) and oPIEC$^{\mathrm{b}}$ (bottom). Green lines denote true positives, red lines denote false positives, while grey lines denote false negatives.

Table 7.3: Indicative interval computation comparison between PIEC, oPIEC$^{\mathrm{bd}}$ and oPIEC$^{\mathrm{b}}$. The first column presents the PMIs computed by PIEC. The second and third columns present the support sets (lists of tuples in the form of $(t,\ prev\_prefix[t])$) of oPIEC$^{\mathrm{bd}}$ and oPIEC$^{\mathrm{b}}$, respectively, at the time denoted by the ending point of the corresponding PMI.

Target PMI	oPIEC$^{\mathrm{bd}}$ support set	oPIEC$^{\mathrm{b}}$ support set
$I_1 = [17492, 18881]$	$[(17312, -7075), (17390, -7114), (17495, -7167), (17556, -7192), (17673, -7211)]$	$[(0, 0), (4858, -1096), (10391, -3628), (13025, -4936), (17133, -6986)]$
$I_2 = [21602, 22960]$	$[(21503, -8475), (21559, -8503), (21616, -8532), (21680, -8555), (21793, -8572)]$	$[(0, 0), (10391, -3628), (13025, -4936), (17133, -6986), (20715, -8081)]$

7.1(d)). Some intervals which were PMIs under ‘smooth noise’ are not PMIs under ‘strong noise’ because their probabilities have dropped below the threshold. This exclusion of lower probability PMIs produces more reliable duration statistics for oPIEC$^{\mathrm{bd}}$.

In order to illustrate the performance of oPIEC$^{\mathrm{bd}}$, we analyse some typical snapshots of our experiments. Figure 7.2 displays two PMIs, I_1 and I_2, computed by PIEC for ‘meeting’ under ‘strong noise’, and the corresponding intervals computed by oPIEC$^{\mathrm{bd}}$ and oPIEC$^{\mathrm{b}}$. Table 7.3 displays the endpoints of PMIs I_1 and I_2, along with the support sets of oPIEC$^{\mathrm{b}}$ and oPIEC$^{\mathrm{bd}}$ at the time denoted by the ending point of the corresponding PMI. The support sets of oPIEC$^{\mathrm{b}}$ and oPIEC$^{\mathrm{bd}}$ may hold at most five elements, while the mean and the standard deviation of the duration values provided to oPIEC$^{\mathrm{bd}}$ are 411 and 22 time-points, respectively.

Consider oPIEC$^{\mathrm{b}}$ and oPIEC$^{\mathrm{bd}}$ attempting to compute PMI $I_1 = [17492, 18881]$ with the corresponding support sets displayed in Table 7.3. oPIEC$^{\mathrm{b}}$ computed the interval $[17133, 18522]$ starting from the time-point of the last support set element, i.e., $(17133, -6986)$. The time-point $t = 17492$, with $prev\_prefix[t] = -7164$, was not stored in the support set because its $score\_range$ was shorter than the $score\_range$ of all support set elements. In contrast, oPIEC$^{\mathrm{bd}}$ maintained in the support set the time-point 17495, and thus computes the interval $[17495, 18883]$, having only three false negatives and two false positives. In oPIEC$^{\mathrm{bd}}$, support set elements which are ‘outdated’ with respect to duration statistics are eventually removed. Note that the intervals computed by oPIEC$^{\mathrm{b}}$ and

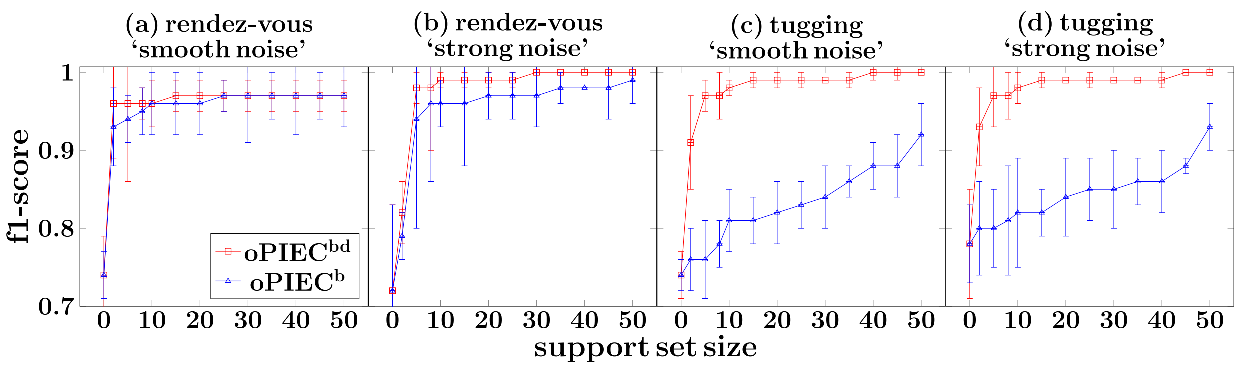


Figure 7.3: The predictive accuracy of oPIEC^{bd} and oPIEC^{b} on maritime situational awareness. The intervals computed by oPIEC^{bd} and oPIEC^{b} are compared to the PMIs computed by PIEC. The point-based system providing the input probability streams is Prob-EC.

oPIEC^{bd} are PMIs. PIEC chose I_1 over these intervals because I_1 has the highest 'credibility', i.e., among overlapping PMIs, PIEC chooses the one with the highest probability.

In the case of PMI $I_2 = [21602, 22960]$, all intervals starting from a time-point in the support set of oPIEC^{b} have a probability below the threshold. In such cases, the recognition of oPIEC^{b} follows point-based recognition, i.e., it returns the time periods during which instantaneous CE probability is greater or equal to the threshold, which resulted in the construction of a sub-interval of I_2, i.e., oPIEC^{b} could not compute a PMI. In contrast, oPIEC^{bd} maintained in its support set time-point 21616, which is the starting point of a PMI overlapping I_2.

Maritime Situational Awareness. Figure 7.3 presents the f1-scores of oPIEC^{bd} and oPIEC^{b}. oPIEC^{bd} performs at least as well as oPIEC^{b} in the case of 'rendez-vous' and clearly outperforms oPIEC^{b} in the case of 'tugging'. In the recognition of 'tugging', it is often the case that all intervals starting from a time-point in the support set of oPIEC^{b} and ending in the current data batch have a probability below the threshold. It such cases, the intervals computed by oPIEC^{b} are confined only to time-points with probability above the threshold, and thus oPIEC^{b} detects sub-intervals of PMIs, resulting in false negatives. In contrast, oPIEC^{bd} is more accurate than oPIEC^{b} because it promptly discards obsolete time-points from the support set. In the case of 'rendez-vous', the elements in the support set of oPIEC^{b} often coincide with those maintained after consulting the provided duration statistics. Thus, the performance of oPIEC^{b} is closer to that of oPIEC^{bd}.

The aim of the next set of experiments was to compare the run-times of oPIEC^{bd}, oPIEC^{b} and PIEC. We show the results of this comparison on 'rendez-vous' under 'strong noise'; the results for the other CEs in maritime situational awareness and human activity recognition are similar and thus not shown here. The support set size limit was set to 50 elements for both oPIEC^{bd} and oPIEC^{b}, as this value leads to near-perfect PMI computation (see Figure 7.3(b)). The data batch size was set to one time-point. Upon each new batch arrival, PIEC had to process every input probability value from the start of the stream until the latest data batch, in order to ensure correct PMI computation. Table 7.4(a) shows the

Table 7.4: Run-times of oPIEC^{bd}, oPIEC^{b} and PIEC when computing the PMIs of 'rendezvous' under 'strong noise'.

(a) Total run-times of oPIEC^{bd}, oPIEC^{b} and PIEC in seconds when processing data streams of increasing size.

total stream size (number of time-points)	**1K**	**2K**	**4K**	**8K**
PIEC	1.92 ± 0.32	7.53 ± 1.24	29.76 ± 4.9	134.63 ± 22
oPIEC^{b}	$\boldsymbol{0.09 \pm 0.02}$	$\boldsymbol{0.19 \pm 0.05}$	$\boldsymbol{0.38 \pm 0.1}$	$\boldsymbol{0.7 \pm 0.2}$
oPIEC^{bd}	$\boldsymbol{0.09 \pm 0.02}$	$\boldsymbol{0.19 \pm 0.05}$	0.39 ± 0.1	0.72 ± 0.23

(b) Total run-times of oPIEC^{bd} and oPIEC^{b} in seconds as the support set size increases.

support set size (number of elements)	**50**	**100**	**200**	**400**
oPIEC^{b}	$\boldsymbol{0.7 \pm 0.21}$	$\boldsymbol{1.27 \pm 0.5}$	$\boldsymbol{2.4 \pm 1.1}$	$\boldsymbol{4.79 \pm 2.41}$
oPIEC^{bd}	$\boldsymbol{0.7 \pm 0.23}$	$\boldsymbol{1.27 \pm 0.53}$	$\boldsymbol{2.4 \pm 1.1}$	$\boldsymbol{4.79 \pm 2.41}$

performance of PIEC, oPIEC^{b} and oPIEC^{bd} for input data streams with size ranging from 1,000 to 8,000 time-points. oPIEC^{bd} and oPIEC^{b} have effectively the same performance; their run-times increase linearly with the input stream size, as the cost of processing an incoming data batch depends only on the size of the data batch and the size of the support set, which remain constant, and not on the entire history of the stream. Not surprisingly, the use of PIEC becomes prohibitively expensive as the stream progresses. These results are compatible with the complexity analyses of Sections 6.1.3 and 6.2.3.

In order to stress test oPIEC^{bd} and oPIEC^{b} further, we performed experiments for support set sizes ranging from 50 to 400 elements, while the stream size remains constant and equal to 8,000 time-points. Table 7.4(b) shows the results. Again, the performance of both systems is nearly identical. In other words, the additional operations of oPIEC^{bd} impose practically no overhead to the system (see Section 6.2.3).

Comparisons with the State of the Art. We compared oPIEC^{bd} with three state-of-the-art systems: the Simplified Event Calculus implemented in ProbLog (SEC), the Probabilistic Event Calculus (PEC) and OSLα. SEC is a probabilistic Event Calculus framework that supports uncertainty in both SDEs and CE definitions [McAreavey et al., 2017]. Similar to Prob-EC, SEC performs point-based recognition. We compared the predictive accuracy of oPIEC^{bd} and SEC using the CE annotation provided by the CAVIAR team as the ground truth. We investigated the detection of the 'meeting' CE under 'strong noise' and the detection of the 'fighting' CE under 'smooth noise' and 'strong noise'. We used the threshold values leading to the best performance for each system. When recognising 'meeting' and 'fighting' under 'strong noise', we set $\mathcal{T} = 50\%$ for all systems. In the case of 'fighting' under 'smooth noise', the best performance for SEC is achieved with $\mathcal{T} = 90\%$, while oPIEC^{bd} reached its best performance with $\mathcal{T} = 50\%$. oPIEC^{bd} was evaluated using 5-fold

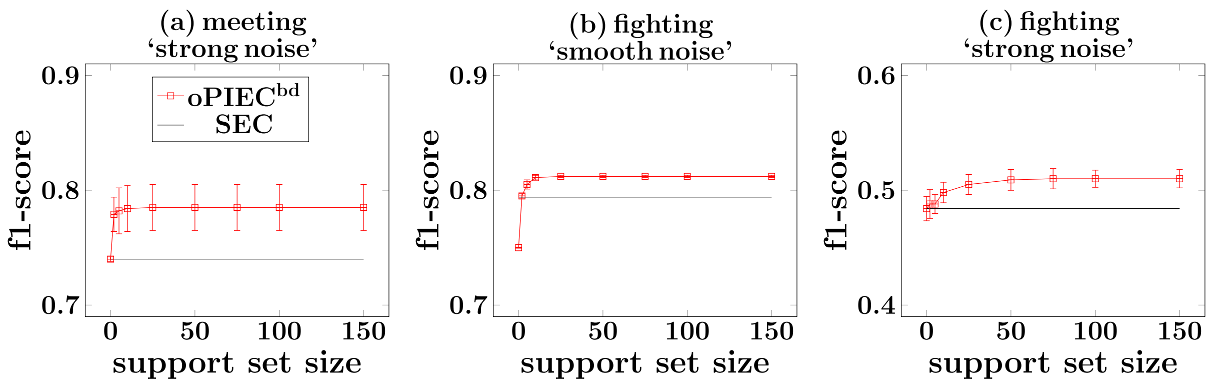


Figure 7.4: The predictive accuracy of oPIEC^{bd} and SEC against the ground truth provided by the CAVIAR team. The standard deviations in diagram (b) are small, and thus not visible. oPIEC^{bd} consumed the instantaneous CE probabilities computed by Prob-EC (which are the same as the instantaneous CE probabilities computed by SEC).

cross-validation, while SEC does not require training. Figure 7.4 shows the f1-scores of oPIEC^{bd} and SEC, as the size of the support set used by oPIEC^{bd} increases from 0 to 150 elements. Our results show that the use of oPIEC^{bd} improves upon the point-based recognition of SEC, while requiring only a small subset of the data.

Figure 7.5 shows the reasoning times of oPIEC^{bd} and SEC when processing SDE streams of increasing size. The reasoning times presented for oPIEC^{bd} include the time that Prob-EC required to derive the instantaneous CE probabilities which are necessary for PMI computation. As a matter of fact, PMI computation takes up approx. 0.2% of the times shown in Figure 7.5. Our results show that oPIEC^{bd} scales much better than SEC. SEC computes the probability of a CE at each time-point from scratch, without taking into consideration the CE probabilities derived at previous time-points. In contrast, oPIEC^{bd} uses Prob-EC, which derives CE probabilities incrementally as time progress, avoiding redundant computations and maintaining in memory only the probabilities of CEs at the previous time-point.

A closely related system is PEC, i.e., a probabilistic Event Calculus framework for point-based recognition, which is implemented in answer set programming (ASP) for evaluation with a state-of-the-art ASP solver [D'Asaro et al., 2017; D'Asaro, 2019]. The ASP implementation of PEC has been used in a decision-making system monitoring the attention levels of children engaging in rehabilitation exercises [D'Asaro et al., 2023]. We tried to compare the predictive accuracy of oPIEC^{bd} against that of PEC, but could not conduct a meaningful comparison due to the high reasoning times of PEC. Figure 7.5 shows that the reasoning time of PEC increased exponentially as we doubled the number of SDEs in the input stream. Moreover, given a stream with at least 256 SDEs, we had to terminate the execution of PEC because it ran for more than 5 hours.

OSLα is a supervised learning system optimising the structure and weights of CE definitions in an Event Calculus expressed in Markov Logic [Michelioudakis et al., 2016; Michelioudakis et al., 2019]. OSLα has proven effective in the task of learning human activity

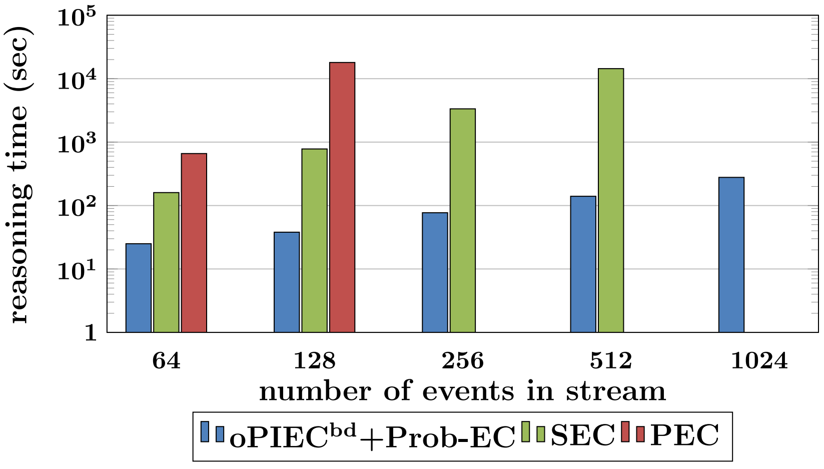


Figure 7.5: The reasoning times of oPIEC^{bd} and Prob-EC (oPIEC^{bd}+Prob-EC), SEC and PEC on the task of detecting the 'moving' CE under 'smooth noise' in logarithmic scale. PEC and SEC required more than 5 hours to process streams including at least 256 events and 1024 events, respectively, and thus we terminated their execution in these cases.

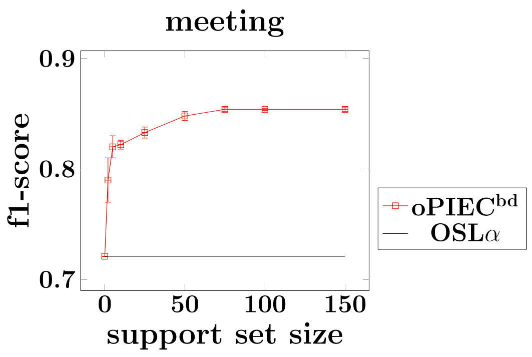


Figure 7.6: The predictive accuracy of oPIEC^{bd} and OSLα on the task of detecting the 'meeting' CE against the ground truth provided by the CAVIAR team.

definitions, often outperforming manually constructed rules. We compared the predictive accuracy of oPIEC^{bd} and OSLα with respect to detecting the 'meeting' CE against the ground truth provided by the CAVIAR team. OSLα is not designed to handle probabilistic data, and thus operated on the original CAVIAR dataset. OSLα included a training phase during which it learned a definition of 'meeting' in the form of weighted Event Calculus rules, given a training set containing a subset of the annotation provided by the CAVIAR team (see [Michelioudakis et al., 2016] for the setup of the training process). Afterwards, OSLα was evaluated in the remaining instances of 'meeting', using the learned pattern to compute the probability of 'meeting' at each time-point. oPIEC^{bd} processed the instantaneous probabilities derived by OSLα in order to compute the intervals of 'meeting' and was trained using 5-fold cross-validation. The threshold value for all systems was set to 70% because this choice maximises the predictive accuracy of each system.

Figure 7.6 presents our results, according to which the interval-based recognition of oPIEC^{bd} improves upon the recognition of OSLα. The instantaneous probabilities computed by OSLα include frequent fluctuations (similar to the one presented in Figure 5.2), which lead to false negatives. oPIEC^{bd} addressed this issue by computing PMIs and, as a result, outperformed the point-based recognition of OSLα.

7.3 Discussion

We outlined our experimental evaluation of oPIEC^{b} and oPIEC^{bd} on two composite event recognition applications, i.e., human activity recognition and maritime situational awareness. For both applications, we constructed noisy data streams by injecting varying levels of artificial noise into the original, crisp data streams. First, we compared the predictive accuracy of oPIEC^{b} and oPIEC^{bd} against the batch processing of PIEC on both applications, using support sets of increasing size. In all of our experiments, oPIEC^{bd} performed at least as well as oPIEC^{b}, while nearly reaching the accuracy of batch processing using support sets whose sizes were bounded by a very small portion of the stream. Second, we compared oPIEC^{b}, oPIEC^{bd} and PIEC in terms of reasoning efficiency, demonstrating the benefits of our extensions of PIEC for stream reasoning. Third, we compared oPIEC^{bd}, which proved to be more accurate than oPIEC^{b} in the previous experiments, with three state-of-the-art frameworks that perform CER over noisy events. Our results demonstrated that the interval-based recognition of oPIEC^{bd} is more accurate than the point-based recognition of these frameworks, when compared against the ground truth annotation of our human activity recognition dataset. Moreover, we showed that oPIEC^{bd} scales much better than the state-of-the-art frameworks (see Figure 7.5).

8. SUMMARY AND FUTURE WORK

8.1 Summary

The goal of this thesis was to develop novel stream reasoning techniques, with respect to the requirements of contemporary applications. In Chapter 1, we outlined several requirements for effective stream reasoning, such as a formal and declarative temporal specification language with a hierarchical pattern organisation, incorporating relational situations and background knowledge, an interval-based semantics for durative situations, as well as efficient reasoning algorithms with optimised windowing techniques that avoid re-computations, paving the way for reasoning with minimal latency. We identified the Event Calculus, i.e., a logic programming formalism for representing and reasoning about events and their effects, as a suitable language for reasoning over event streams. In Section 2.5, we reviewed the systems found in the literature, focusing on logic-based frameworks and frameworks that are based on the Event Calculus. Our analysis concluded that, based on the aforementioned requirements, RTEC is the most effective Event Calculus-based framework for reasoning over streams. RTEC, however, does not support three types of temporal specifications—cyclic dependencies, events with delayed effects and Allen relations—that are commonly required for modelling the domains of contemporary applications. Moreover, none of the systems that we find in literature supports the aforementioned specifications, while being suitable for reasoning over large data streams (see Table 2.3). To address this issue, we proposed three orthogonal extensions of RTEC, in order to capture each one of these three types of specifications.

In Section 3.2, we proposed RTEC$_{\circ}$, an extension of RTEC supporting temporal patterns, written in the Event Calculus, with cyclic dependencies. While there are other Event Calculus dialects that support cyclic dependencies, RTEC$_{\circ}$ is the only Event Calculus-based system that handles such dependencies efficiently, by employing an incremental caching algorithm that avoids redundant computations. We demonstrated that temporal specifications in RTEC$_{\circ}$ are locally stratified logic programs. We proved the correctness of the incremental caching algorithm of RTEC$_{\circ}$ and outlined its worst-case time complexity, demonstrating that our algorithm is suitable for reasoning over large data streams. Moreover, we compared RTEC$_{\circ}$ experimentally with a ‘naive’ version of RTEC$_{\circ}$ that does not feature incremental caching, as well as with jREC, an Event Calculus-based system that supports cyclic dependencies, demonstrating the superior reasoning efficiency of RTEC$_{\circ}$.

In Section 3.3, we presented RTEC$^{\rightarrow}$, i.e., an extension of RTEC for reasoning over streams of events with delayed effects. RTEC$^{\rightarrow}$ extends the language of RTEC with constructs expressing that a situation of interest may be initiated as a future effect of a set of events, as well as that such a future initiation may be postponed. We showed that temporal specifications in RTEC$^{\rightarrow}$ are locally stratified logic programs and outlined an optimal processing order, which can be determined at compile-time. At run-time, RTEC$^{\rightarrow}$ uses incremental caching to compute the maximal intervals of situations of interest that are defined based on events with delayed effects without any redundant computations.

Moreover, $\text{RTEC}^{\rightarrow}$ identifies the minimal information that needs to be transferred between windows, in order to compute these intervals correctly in a streaming setting. Our empirical evaluation of $\text{RTEC}^{\rightarrow}$ highlighted the positive effects of caching in reasoning efficiency and demonstrated that $\text{RTEC}^{\rightarrow}$ outperforms the state of the art, often by several orders of magnitude, when reasoning with patterns that include events with delayed effects, as well as when reasoning with patterns that do not include such events.

In Section 3.4, we proposed RTEC_{A}, an extension of RTEC that supports temporal patterns that involve the relations of Allen's interval algebra. RTEC_{A} extends the syntax of pattern specifications in RTEC, in order to allow the use of Allen relations as interval manipulation constructs. We showed that the semantics of the language is not affected by this extension, i.e., temporal specifications in RTEC_{A} are locally stratified logic programs. Moreover, we provided a linear time algorithm for computing all interval pairs that satisfy an Allen relation, among two sorted lists of maximal intervals. Subsequently, we proposed an extension of this algorithm, in order to support windowing, and proved the correctness of the algorithm in a batch and in a streaming setting. In our experimental evaluation of RTEC_{A}, we compared our approach with two state-of-the-art systems that support Allen relation. Our experiments assessed the efficiency of Allen relation computation in both batch and windowing mode, as well as composite event recognition with Allen relations on real maritime data. Our results showed that RTEC_{A} is more efficient than the state of the art, outperforming the systems we selected in all of our experiments, by at least one order of magnitude.

Real applications often exhibit uncertainty; thus, a stream reasoning framework needs to be robust in the presence of noisy data streams. In Section 5.4, we presented a literature review on computational frameworks that handle uncertainty, concluding that there is no framework in the literature that supports noisy data streams, while fulfilling the requirements we outlined in Chapter 1. Our analysis showed that the probabilistic event recognition framework PIEC exhibits several desired characteristics compared to the state of the art, as it is based on the Event Calculus, provides a language primitive for representing event probabilities and includes a linear time algorithm for deriving (the probabilities of) the maximal intervals of situations of interest. However, PIEC is not designed to operate over data streams, yielding several redundant computations when reasoning in a streaming setting. To address this issue, we proposed an extension of PIEC for reasoning over noisy data streams, along with two bounded-memory variants, paving the way for high predictive accuracy in scenarios where memory is scarce.

In Chapter 6, we proposed oPIEC, an extension of PIEC for handling noisy data streams. oPIEC processes the input stream in data batches and stores in the 'support set' the minimal set of time-points that may be starting points of intervals including future time-points. This way, oPIEC discards the remaining time-points, while guaranteeing correct maximal interval computation throughout the stream. In oPIEC, the cost of maximal interval computation is linear to the size of the data batch and the size of the support set. We proved the correctness of the support set updating and maximal interval computation algorithms of oPIEC. In order to further support contemporary applications, we proposed two bounded-memory versions of oPIEC, i.e., oPIEC^{b} and oPIEC^{bd}. Using a bounded-memory, the size of the support set, and thus the cost of reasoning, remains constant as the stream progresses. Our empirical analysis of oPIEC^{b} and oPIEC^{bd} showed that oPIEC^{bd} is, in many cases, much more accurate than oPIEC^{b}, demonstrating the benefits of the support set maintenance algorithm of oPIEC^{bd}, which leverages composite event duration statistics. Subsequently, we compared oPIEC^{bd} with three state-of-the-art computational frameworks handling noisy data streams. Our results demonstrate that the interval-based recognition of oPIEC^{bd} is often more accurate than the point-based recognition of state-of-the-art frameworks, while oPIEC^{bd} reasons much more efficiently than these systems, outperforming them by several orders of magnitude.

8.2 Future Work

We outline some directions for further research.

Explanations for Derived Situations. In order to facilitate transparency and fairness, the derivations of a stream reasoning framework need to be explainable, i.e., have explicit justifications that can be easily understood by humans [Arrieta et al., 2020]. A good explanation should have a causal structure, i.e., include a chain of inferences that leads to the explained result, be minimal, i.e., omit irrelevant information, and be understandable, i.e., easy to follow by non-experts [Vidal, 2022]. The systems we proposed do not provide justifications for their derivations, e.g., derived situations of interest are not accompanied by explanations. In a different line of work, we developed a visual analytics framework that facilitates the exploration of the composite maritime activities detected by RTEC using interactive visualisation, allowing for the identification of the features capable of distinguishing the detected activities from other types of behaviour by domain experts [Andrienko et al., 2024]. This type of explanation, however, may be regarded as too high-level and prone to human error, while lacking a causal structure. To avoid such issues, explanations should have a formal foundation.

RTEC$_{\circ}$, RTEC$^{\rightarrow}$ and RTEC$_{\mathsf{A}}$ are based in logic programming, where query answers can be justified via the rules and the substitutions used in their SLDNF proofs. However, SLDNF proof trees are typically large and difficult to understand by non-experts. Cabalar et al. [2014] proposed a semantics for logic programs where each atom of a model is associated with a set of causal graphs, reflecting the ordered rule applications that can be used to derive the atom. Causal graphs are compact and minimal, paving the way for explainability. Explainability has also been studied in the context of ASP, where the answer sets that satisfy a query cannot be used as its explanations, because they do not have a causal structure [Pontelli et al., 2009; Arias et al., 2020; Cabalar et al., 2020]. We would like to investigate explanation techniques for the derivations of RTEC$_{\circ}$, RTEC$^{\rightarrow}$ and RTEC$_{\mathsf{A}}$.

The problem of query explainability becomes more complex in probabilistic logic programming, where the explanations of a query need also include information about the total choices leading to possible worlds that model the query. The total choice with the highest probability is termed as the most probable explanation of the query, while its Viterbi proof is equivalent to the minimal set of atomic choices that justify the query [Shterionov et al., 2014]. Total choices—and sets of atomic choices in general—contain no causal structure, while some of the included atomic choices may be irrelevant to the considered query [Vidal, 2024]. Vidal has proposed two methodologies for explaining the derivations of a probabilistic logic programming framework. In [Vidal, 2024], explanations are generated by a top-down proof procedure, while the choices made at each step of the proof are expressed in a compact manner, aiding intelligibility. In [Vidal, 2022], explanations are represented as programs, generated from the given query using unfolding transformations. We aim to develop similar query explanation methods in the context of oPIEC.

The frameworks we proposed support hierarchical temporal specifications, where it would be useful to provide explanations with varying levels of detail with respect to the hierarchy. Zieliński [2023] proposed a pattern specification language with a denotational semantics, containing information about the sub-pattern instances that make up a detected situation in the form of a tree. The structure of the tree reflects the hierarchy of the given specifications, thus providing justifications over all possible levels of detail, facilitating the detection of errors in pattern specifications with respect to their intended meaning. In the future, we aim to propose similar denotational semantics for our proposed frameworks, paving the way for explainable derivations in the presence of complicated hierarchies.

Comparison with automata-based formalisms. The languages of the frameworks we proposed have a formal, declarative semantics that is based on logic programming, while most frameworks for event recognition over data streams are automata-based [Giatrakos et al., 2020]. For this reason, we would like to compare our proposed frameworks against automata-based formalisms in terms of expressive power. Note, however, that the languages of several automata-based frameworks do not come with a clear semantics, which makes them hard to understand and generalise [Grez et al., 2019].

Some recent works have proposed formal, denotational semantics for pattern specification languages that can be executed in an automata-based framework [Grez et al., 2020; Zielinski, 2023]. This type of semantics provides a clear meaning for each one of the temporal operators of the language, paving the way for pattern compositionality, while facilitating a comparison with other formalisms. Instead, we base our frameworks on a model-theoretic semantics, where the intended meaning of the temporal specifications of a domain is the perfect model of the corresponding locally stratified logic program (see Section 2.1). The model of the program does not inform us about the expressive power of the language regarding temporal operators, and thus it is not sufficient for a direct comparison against automata-based formalisms. For example, RTEC does not support event sequencing, which is a commonly supported in automata-based systems [Artikis et al., 2015; Giatrakos et al., 2020]. It would be interesting to compare the expressive power of our extensions of RTEC against the expressivity of event sequencing operators. Towards this goal, we aim to propose denotational semantics for our frameworks, and compare their expressive power against automata-based formalisms, such as [Bucchi et al., 2022; Zielinski, 2023].

Distribution and parallelisation techniques. Modern stream processing engines commonly feature distribution and parallelisation capabilities [Fragkoulis et al., 2024]. Giatrakos et al. [2020] argue that, in contemporary applications where large volumes of data are being generated continuously and with high velocity, centralised, serial event processing often becomes a bottleneck. To address this issue, the reasoning engine may be distributed over multiple processing units that can work in parallel, improving the processing rate of input events and reducing reasoning latency. In this setting, the computations required to derive a situation of interest are broken up into parts, and each part is assigned to a different processing unit. Subsequently, the outputs of these units are merged together, composing, e.g., the recognised instances of some situation.

dRTEC is a distributed version of RTEC where input items are allocated to processing units based on the domain entities they refer to [Mavrommatis et al., 2016]. We would like to develop similar entity-based parallelisation techniques for our proposed frameworks, as well as investigate further parallelisation possibilities. For instance, the evaluations of initiatedAt/terminatedAt rules at each time-point of the window are often independent, and thus can be parallelised (see, e.g., rule schema (3.1)). Furthermore, in $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A}, the evaluation of each one of the initiatedAt/terminatedAt rules of an FVP at some time-point can be assigned to a different processing unit, since the outcomes of these evaluations are independent from each other. The latter type of a parallelisation can only be employed in oPIEC when the initiatedAt/terminatedAt rules of the FVP have disjoint bodies. Otherwise, deriving the probability of an initiation/termination of the FVP by summing up the probabilities of the bodies of the corresponding initiatedAt/terminatedAt is incorrect, i.e., we encounter the disjoint-sum problem (see Section 5.1).

Logical Inference in Tensor Spaces. As another future work direction, we aim to reduce logical inference in our proposed frameworks into a set of algebraic operations on tensors, providing an one-to-one mapping between the models of a logical theory and the solutions of a system of tensor equations. Several approaches have been proposed for reducing logical inference into algebraic operations, focusing on Datalog [Sato, 2017b], logic programming [Sakama et al., 2021], and first-order logic [Sato, 2017a]. These approaches argue for the benefits of such transformations. First, the resulting tensor operations may be equivalent to a set of algebraic operations that can be computed in (low) polynomial time. Sato et al. [2017b] showed that transforming a Datalog program into a set of linear matrix equations and solving these equations is significantly more efficient than the traditional fixpoint evaluation of Datalog programs, as well as two Prolog engines using tabling and two state-of-the-art ASP systems. Second, Sato et al. [2017a] argue that the solutions of tensor operations may be approximated using tensor decomposition techniques, further increasing the scalability of this method. Third, some problems, such as scientific computations and neuro-symbolic reasoning, require both symbolic and algebraic computations. Integrating these two types of computation, by mapping symbolic expressions into vector spaces and solving the resulting tensor equations, is a promising approach for these areas [Sakama et al., 2021]. Fourth, algebraic operations can often be executed in parallel using GPUs, leading to further efficiency gains.

Tsilionis et al. [2024] proposed a formalisation of a point-based Event Calculus dialect in tensor spaces. $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A} employ an interval-based Event Calculus dialect, combined with stream reasoning techniques, such as windowing and caching.

oPIEC uses an implementation of the Event Calculus in ProbLog, featuring probabilistic facts as language primitives. Towards supporting the range of specifications of $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A}, we would like to extend the work presented in [Tsilionis et al., 2024], in order to map occurrences of durative events/situations into tensors. Moreover, towards translating reasoning in oPIEC into tensor operations, we would like to express event/situation probabilities as tensor elements and adjust the algebraic operators of [Tsilionis et al., 2024] in order to guarantee correctness in the probabilistic setting. We believe that transforming the symbolic computations of our systems into tensors operations could lead to significant reasoning efficiency gains [Sato, 2017a; Sato, 2017b; Sakama et al., 2021; Tsilionis et al., 2024].

Neuro-symbolic reasoning. Although neural networks typically exhibit high performance in modern applications, they have several disadvantages, such as brittleness, the lack of a formal semantics and result explainability, and the need for a large amount of training data and time. On the other hand, symbolic representation and reasoning techniques are commonly formal, interpretable and data-efficient. Neuro-symbolic artificial intelligence involves the integration of symbolic techniques with neural networks, aiming towards highly efficient and accurate frameworks that are semantically sound, explainable and trustworthy [Yu et al., 2023; Marra et al., 2024]. Neuro-symbolic frameworks typically include a neural layer, mapping sub-symbolic data into symbols, and a logic layer, modelling the relationships and constraints among symbolic data. These frameworks are trained end-to-end, using supervision on the output of the logic layer and adjusting the parameters of both layers simultaneously through backpropagation. For example, DeepProbLog extends ProbLog with neural probabilistic facts, the probabilities of which are determined by the outputs of neural networks. These neural networks are trained via backpropagation, using supervision on the output of the ProbLog-based logic layer [Manhaeve et al., 2021].

The literature includes some neuro-symbolic frameworks for temporal reasoning. For instance, DeepProbCEP employs an implementation of Prob-EC in DeepProbLog as its logic layer [Vilamala et al., 2023]. DeepProbCEP lacks a build-in representation for durative and ongoing situations, as well as optimised reasoning algorithms for event hierarchies. Apriceno et al. have proposed two neuro-symbolic frameworks that are based on DeepProbLog and incorporate an Event Calculus dialect in their logic layer [Apriceno et al., 2021; Apriceno et al., 2022]. These approaches are very limited in terms of expressive power; e.g., they do not allow concurrent events.

The state-of-the-art neuro-symbolic frameworks do not support stream reasoning, while the formulation of a semantics for neuro-symbolic approaches is still an open challenge [Marra et al., 2024]. To address these issues, we aim to integrate our proposed stream reasoning techniques into neuro-symbolic frameworks. For instance, oPIEC is based on probabilistic logic programming, which can be extended with neural probabilistic facts [Manhaeve et al., 2021]. We would like to extend oPIEC with such facts, leading to a neuro-symbolic framework with a formal, interval-based semantics, and optimisation techniques, such as windowing and hierarchical event processing.

ABBREVIATIONS - ACRONYMS

AIS	automatic identification system
ASP	answer set programming
CE	composite event
CER	composite event recognition
CQL	continuous query language
CWA	closed world assumption
d-DNNF	deterministic, decomposable negation normal form
DNF	disjunctive normal form
ESL	expressive stream language
FVP	fluent-value pair
GKL-EC	Generalised Kinetic Logic in the Event Calculus
oPIEC	online Probabilistic Interval-based Event Calculus
oPIEC^{b}	oPIEC with bounded memory and the $score\_range$ tactic
oPIEC^{bd}	oPIEC with bounded memory and the interval duration statistics tactic
PIEC	Probabilistic Interval-based Event Calculus
PMI	probabilistic maximal interval
Prob-EC	Probabilistic Event Calculus
RTEC	Run-Time Event Calculus
RTEC$_{\circ}$	RTEC with cycles
RTEC$^{\rightarrow}$	RTEC with events with delayed effects
RTEC$_{A}$	RTEC with Allen relations
SCC	strongly connected components
SDE	simple-derived event
SQL	structured query language
WCC	weakly connected components
WMC	weighted model counting

APPENDIX A. PROOFS FOR PROPOSITIONS OF CHAPTER 3: SEMANTICS

We prove that the event descriptions of $\text{RTEC}_{\circ}$, $\text{RTEC}^{\rightarrow}$ and RTEC_{A} are locally stratified logic programs. First, we present temporal logic programs and describe the cycle-sum test; if a temporal logic program passes the test, then it is locally stratified [Rondogiannis, 2001]. We outline a correspondence between event descriptions and temporal programs, justifying the use of the cycle-sum test to prove local stratification for event descriptions. Second, we demonstrate that $\text{RTEC}_{\circ}$ and $\text{RTEC}^{\rightarrow}$ event descriptions are locally stratified by applying the cycle-sum test. Third, we show that RTEC_{A} event descriptions are locally stratified, provided that there are no cyclic dependencies among the FVPs of statically determined fluents.

A.1 The Cycle-Sum Test

We outline temporal logic programs and the cycle-sum test, following [Rondogiannis, 2001; Nomikos et al., 2005]. We adopt the definition of the 'extended' cycle-sum test, described in [Nomikos et al., 2005], because it accepts a wider range of temporal programs that the original test presented in [Rondogiannis, 2001].

The syntax of temporal logic programs extends normal logic programs with temporal operators, i.e., first and next. A temporal reference is a sequence of temporal operators. next^k denotes a sequence of k next operators. A canonical temporal reference is a reference of the form first next^k, while an open temporal reference has the form next^k. More general forms of temporal references (e.g., next^{2} first next first next^{3}) are not interesting because the operators appearing before the rightmost first operator are superfluous, and thus such temporal references are prohibited. A temporal atom is an atom preceded by either a canonical or an open temporal reference. Given a temporal atom a, the temporal reference of a is denoted by $time(a)$. A temporal clause has the form '$h \leftarrow a_{\mathit{1}}, \dots, a_k, \text{not } b_{\mathit{1}}, \dots, \text{not } b_m$', where $h, a_{\mathit{1}}, \dots, a_k, b_{\mathit{1}}, \dots, b_m$ are temporal atoms. A temporal program is a finite set of temporal clauses.

We illustrate the meaning of first and next with the following temporal program, simulating the operation of traffic lights:

$$\text{first light(green)}. \tag{A.1}$$

$$\text{next light(amber)} \leftarrow \text{light(green)}. \tag{A.2}$$

$$\text{next light(red)} \leftarrow \text{light(amber)}. \tag{A.3}$$

$$\text{next light(green)} \leftarrow \text{light(red)}. \tag{A.4}$$

Rule (A.1) states that the traffic light is green at time-point $\mathit{0}$. Rules (A.2)–(A.4) describe how the color of the traffic light changes from one time-point to the next one. For instance, rule (A.4) expresses that, if the color of the traffic light is red at time-point T, then its color becomes green at time-point $T{+}\mathit{1}$.

Suppose that P is a temporal logic program. The skeleton S of P is the propositional program generated after removing all the arguments of the predicates in P. For example, the skeleton of the temporal logic program described in rules (A.1)–(A.4) is the following propositional program:

$$\text{first light.} \tag{A.5}$$

$$\text{next light} \leftarrow \text{light.} \tag{A.6}$$

The cycle-sum test is a procedure for deciding whether a temporal logic program is locally stratified; if a temporal logic program P passes the cycle-sum test, then P is a locally stratified logic program. First, we define the *temporal difference* between two atoms in the skeleton S of P and the *cycle-sum graph* of S. Then, we outline the *cycle-sum test*.

Definition 22 (Temporal Difference): Consider a temporal logic program P with a skeleton S. Suppose that h is the head of a clause in S that contains a body atom a. The temporal difference $dif(h, a)$ between h and a is defined as follows:

$$dif(h,a) = \begin{cases} k-m & \text{if } time(h) = \text{first next}^k \text{ and } time(a) = \text{first next}^m \\ k-m & \text{if } time(h) = \text{next}^k \text{ and } time(a) = \text{next}^m \\ k-m & \text{if } time(h) = \text{next}^k \text{ and } time(a) = \text{first next}^m \\ -\infty & \text{if } time(h) = \text{first next}^k \text{ and } time(a) = \text{next}^m \end{cases} \tag{A.7}$$

■

Intuitively, $dif(h, a)$ denotes the lower bound on the number of time-points by which atom a precedes atom h. In the case of $-\infty$, we cannot establish such a lower bound because, while atom h occurs k time-points after the first time-point, atom a occurs m time-points after the current time-point, which may have an arbitrary temporal distance from time-point k.

We describe the 'extended' version of the cycle-sum graph, as described in [Nomikos et al., 2005].

Definition 23 (Cycle-Sum Graph): Consider a temporal logic program P with a skeleton S. The cycle-sum graph of S is a directed, labeled multi-graph with self-loops $CG_S = (V, E)$. The set V of vertices of CG_S is the set of predicate symbols appearing in S. The set E of edges consists of triples (p, q, l), where p, $q \in V$ and $l \in (Z \cup -\infty) \times \{+, -\}$. An edge $(p, q, (w, s))$ belongs to E if there exists a clause in S with an atom h as its head and an atom a occurring in its body such that the predicate symbol of h is p, the predicate symbol of a is q, $w = dif(h, a)$, and $s = -$ if a occurs negatively in the clause body, and $s = +$, otherwise. ■

Definition 24 (Cycle-Sum Test): Consider a temporal logic program P with a skeleton S. P passes the cycle-sum test if, in every strongly connected component of CG_S that contains a negatively signed edge, the following conditions hold:

1. The sum of weights across every cycle is non-negative.
2. Every cycle which has a zero sum of weights does not contain a negatively signed edge. ■

A temporal program that passes the cycle-sum test is locally stratified [Rondogiannis, 2001]. For instance, the cycle-sum graph of the skeleton program described in rules (A.5)–(A.6) is $CG_S = (\{\mathsf{light}\}, \{(\mathsf{light}, \mathsf{light}, (\mathit{1}, +))\})$. This graph does not contain a negatively signed edge, and thus the corresponding temporal logic program passes the cycle-sum test.

Rondogiannis [2001] proposed the notion of the *classical counterpart* P^* of a temporal logic program P, and described a procedure for constructing P^* based on P, by removing the temporal operators first and next from clauses of P and adding to the predicates of P an additional argument, expressing explicitly the concept of time. There is a one-to-one correspondence between the models of P and P^* (see Theorem 3.1 in [Rondogiannis, 2001]).

Event descriptions in RTEC$_\circ$, RTEC$^\rightarrow$ and RTEC$_\mathsf{A}$ do not feature operators first and next, while they include a time argument for all their temporal predicates, such as happensAt and initiatedAt. Therefore, such event descriptions can be viewed as the classical counterpart of a temporal logic program. In order to construct the skeleton of an event description, we remove all arguments from the predicates of the program, except from predicate time-stamps. Next, we transform the representation of temporal predicates with a time argument into a representation that is based on the temporal operators first and next as follows. First, we identify the earliest time-stamp in the rule, remove the time argument of the corresponding predicate and consider that this time-stamp represents the current time. Then, we remove the remaining predicate time-stamps and add next^k operators before the corresponding atoms, where k is the temporal distance between the time-stamp of the atom and the current time. For example, consider the following formulation of the law of inertia in the Event Calculus[1]:

$$\begin{aligned}&\mathsf{holdsAt}(F = V,\ T) \leftarrow \\ &\quad \mathsf{initiatedAt}(F = V,\ T - \mathit{1}).\end{aligned} \tag{A.8}$$

$$\begin{aligned}&\mathsf{holdsAt}(F = V,\ T) \leftarrow \\ &\quad \mathsf{holdsAt}(F = V,\ T - \mathit{1}), \\ &\quad \mathsf{not\ terminatedAt}(F = V,\ T - \mathit{1}).\end{aligned} \tag{A.9}$$

$$\begin{aligned}&\mathsf{holdsAt}(F = V,\ T) \leftarrow \\ &\quad \mathsf{holdsAt}(F = V,\ T - \mathit{1}), \\ &\quad \mathsf{not\ initiatedAt}(F = V',\ T - \mathit{1}),\ V \neq V'.\end{aligned} \tag{A.10}$$

The skeleton S of the above program is:

$$\mathsf{next\ holdsAt} \leftarrow \mathsf{initiatedAt}. \tag{A.11}$$

$$\mathsf{next\ holdsAt} \leftarrow \mathsf{holdsAt}, \mathsf{not\ terminatedAt}. \tag{A.12}$$

$$\mathsf{next\ holdsAt} \leftarrow \mathsf{holdsAt}, \mathsf{not\ initiatedAt}. \tag{A.13}$$

All the edges of the cycle-sum graph CG_S induced by S have weight $\mathit{1}$, as, for each clause C in S, every body atom of C precedes the head of C by $\mathit{1}$ time-point. As a result, the

[1]Note that rule-set (A.8)–(A.10) is equivalent to rule-set (2.1)–(2.4). This equivalence can be verified by repeatedly expanding the recursive call to holdsAt in the first body condition of rules (A.9)–(A.10), until reaching time-point $\mathit{0}$.

sum of weights across every walk in CG_S is positive. Therefore, every program whose skeleton comprises rules (A.11)–(A.13) passes the cycle-sum test, and thus is a locally stratified logic program.

Given an event description P_{ED} in RTEC$_\circ$ (resp. RTEC$^\rightarrow$), we prove that RTEC$_\circ$ (RTEC$^\rightarrow$) programs are locally stratified using the cycle-sum test as follows. First, we identify a set of rules P_R that is equivalent with the reasoning algorithms of RTEC$_\circ$ (RTEC$^\rightarrow$) in terms of derivations. Second, we construct the skeleton of program $P_{ED} \cup P_R$ and prove that it passes the cycle-sum test. As a result, RTEC$_\circ$ and RTEC$^\rightarrow$ programs are locally stratified.

A.2 Proof of Proposition 2

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

Proof. We focus on the definition of simple fluents (see Definition 5); RTEC$_\circ$ does not extend statically determined fluent definitions. We repeat the rule schema for initiatedAt definitions in RTEC$_\circ$ (we omit the schema of terminatedAt rules because it is similar):

$$\begin{aligned}
&\text{initiatedAt}(F = V,\ T',\ T,\ T'') \leftarrow \\
&\quad \text{happensAt}(E_1,\ T), T' \leq T < T''[[, \\
&\quad [\text{not}]\ \text{happensAt}(E_2,\ T),\ \ldots, \\
&\quad [\text{not}]\ \text{happensAt}(E_i,\ T), \\
&\quad [\text{not}]\ \text{holdsAt}(F_1 = V_1,\ T),\ \ldots, \\
&\quad [\text{not}]\ \text{holdsAt}(F_k = V_k,\ T), \\
&\quad \textit{atemporal\_constraints}]].
\end{aligned} \tag{A.14}$$

After simplifications, the skeleton of the rules defining simple fluents in RTEC$_\circ$ is:

$$\begin{aligned}
&\text{initiatedAt} \leftarrow \text{happensAt}[[, [\text{not}]\ \text{happensAt}, [\text{not}]\ \text{holdsAt}]]. \\
&\text{terminatedAt} \leftarrow \text{happensAt}[[, [\text{not}]\ \text{happensAt}, [\text{not}]\ \text{holdsAt}]].
\end{aligned} \tag{A.15}$$

Since all atoms in simple fluent definitions contain the same time argument, we did not need to add any next operators in their skeleton. Moreover, the atemporal predicates in the body of the rule depend only on background knowledge, and thus may not be part of a cycle in the cycle-sum graph. Thus, we can omit these conditions without any effect on our result.

RTEC$_\circ$ employs Algorithms 1 and 2 to perform Event Calculus-style reasoning in the presence of cyclic dependencies. These algorithms constitute an optimised implementation of the law of inertia in the Event Calculus that employs incremental caching (see the proof of Proposition 3 in Appendix B). As a result, Algorithms 1–2 and rules (A.8)–(A.10) are semantically equivalent, i.e., they compute the same holdsAt atoms. Thus, we may analyse the semantics of RTEC$_\circ$ in terms of rules (A.8)–(A.10).

An event description P_{ED} in RTEC$_\circ$ is a set of initiatedAt and terminatedAt rules following schema (A.14), while P_R contains rules (A.8)–(A.10). Rules (A.11)–(A.13) and rule-set

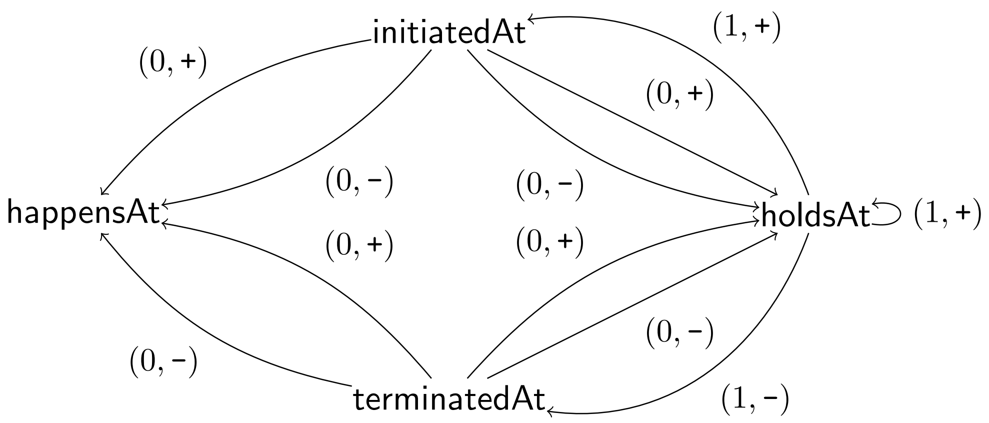


Figure A.1: The cycle-sum graph of an RTEC$_\circ$ program.

(A.15) comprise the skeleton S of $P_{ED} \cup P_R$. Figure A.1 presents the cycle-sum graph CG_S of S. The sum of weights across every cycle in CG_S is positive. Therefore, $P_{ED} \cup P_R$ passes the cycle-sum test, and thus an event description in RTEC$_\circ$ is a locally stratified logic program. □

A.3 Proof of Proposition 5

Proposition (Semantics of RTEC$^\rightarrow$): An event description in RTEC$^\rightarrow$ is a locally stratified logic program. ▲

Proof. RTEC$^\rightarrow$ extends the syntax of simple fluent definitions, by allowing future initiations of FVPs of simple fluents (see Definition 11). In Section 3.3.4, we outlined a set of rules that can be used for evaluating future initiations of FVPs. We repeat these rules below:

$$\begin{aligned}&\textsf{initiatedAt}(F = V',\, T{+}R) \leftarrow\\ &\quad \textsf{fi}(F = V, F = V', R),\\ &\quad \textsf{initiatedAt}(F = V,\, T),\\ &\quad \textsf{not cancelled}(F = V,\, T,\, T{+}R).\end{aligned} \tag{A.16}$$

$$\begin{aligned}&\textsf{cancelled}(F = V,\, T,\, T{+}R) \leftarrow\\ &\quad \textsf{brokenBetween}(F = V,\, T,\, T{+}R).\end{aligned} \tag{A.17}$$

$$\begin{aligned}&\textsf{cancelled}(F = V,\, T,\, T{+}R) \leftarrow\\ &\quad \textsf{p}(F = V),\\ &\quad \textsf{initiatedBetween}(F = V,\, T,\, T{+}R).\end{aligned} \tag{A.18}$$

RTEC$^\rightarrow$ reasons over events with delayed effects using Algorithms 3–6. These algorithms are an optimised implementation of rules (A.16)–(A.18) with windowing and incremental caching. Since Algorithms 3–6 and rules (A.16)–(A.18) lead to the same derivations (see the proof of Proposition 6 in Appendix B), we focus on rules (A.16)–(A.18) for this proof.

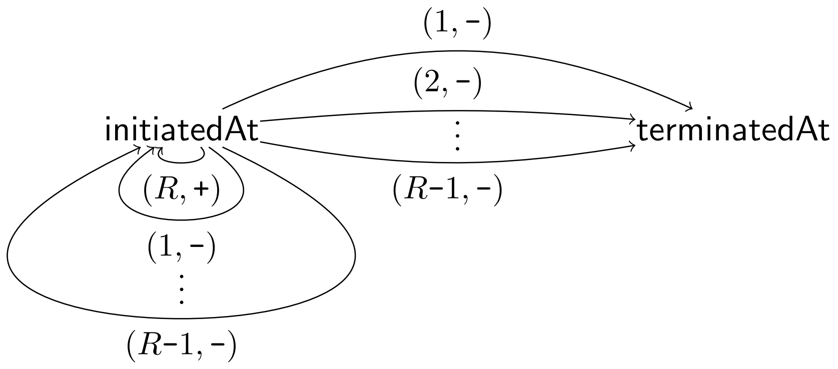


Figure A.2: The cycle-sum graph of an RTEC$^{\rightarrow}$ program.

An event description P_{ED} in RTEC$^{\rightarrow}$ contains simple and statically determined fluent definitions, following the syntax of RTEC, as well as fi and p facts. The reasoning procedure of RTEC$^{\rightarrow}$ can be modelled using (A.16)–(A.18), in conjunction with the law of inertia in the Event Calculus, as expressed in rules (A.11)–(A.13). It suffices to complement the proof we provided in Section A.2 with a proof outlining that rules (A.16)–(A.18) pass the cycle-sum test. In order to generate the skeleton of rules (A.16)–(A.18), we need to substitute the body conditions with predicates brokenBetween and initiatedBetween, which do not have a time-stamp argument, based on the following definitions:

$$\begin{aligned}
&\text{initiatedBetween}(F=V,\, T,\, T{+}R) \leftarrow \text{initiatedAt}(F=V,\, T{+}\mathit{1}).\\
&\qquad\vdots\\
&\text{initiatedBetween}(F=V,\, T,\, T{+}R) \leftarrow \text{initiatedAt}(F=V,\, T{+}R{-}\mathit{1}).\\
&\text{brokenBetween}(F=V,\, T,\, T{+}R) \leftarrow \text{terminatedAt}(F=V,\, T{+}\mathit{1}).\\
&\qquad\vdots\\
&\text{brokenBetween}(F=V,\, T,\, T{+}R) \leftarrow \text{terminatedAt}(F=V,\, T{+}R{-}\mathit{1}).\\
&\text{brokenBetween}(F=V,\, T,\, T{+}R) \leftarrow \text{initiatedAt}(F=V',\, T{+}\mathit{1}),\, V \neq V'.\\
&\qquad\vdots\\
&\text{brokenBetween}(F=V,\, T,\, T{+}R) \leftarrow \text{initiatedAt}(F=V',\, T{+}R{-}\mathit{1}),\, V \neq V'.
\end{aligned} \tag{A.19}$$

We combine rules (A.16)–(A.18) with rule-set (A.19), and generate the following skeleton program S:

$$\begin{aligned}
&\mathsf{next}^{R}\ \mathsf{initiatedAt} \leftarrow \\
&\quad \mathsf{initiatedAt}, \\
&\quad \mathsf{not\ next}^{\mathit{1}}\ \mathsf{initiatedAt}, \\
&\quad\quad \vdots \\
&\quad \mathsf{not\ next}^{R-\mathit{1}}\ \mathsf{initiatedAt}, \\
&\quad \mathsf{not\ next}^{\mathit{1}}\ \mathsf{terminatedAt}, \\
&\quad\quad \vdots \\
&\quad \mathsf{not\ next}^{R-\mathit{1}}\ \mathsf{terminatedAt}.
\end{aligned} \tag{A.20}$$

Figure A.2 presents the cycle-sum graph CG_S induced by S. The weights of all edges in CG_S are positive, and thus all programs with skeleton S pass the cycle-sum test. Moreover, the cycle-sum graph generated as the combination of the graphs in Figures A.1 and A.2 also passes the cycle-sum test, indicating that the addition of future initiations in event descriptions preserves local stratification. As a result, an event description in RTEC$^{\rightarrow}$ is a locally stratified logic program. □

A.4 Proof of Proposition 9

Proposition (Semantics of RTEC$_{\mathsf{A}}$): An event description in RTEC$_{\mathsf{A}}$ is a locally stratified logic program. ▲

Proof. RTEC$_{\mathsf{A}}$ extends the syntax of RTEC by allowing instances of the allen predicate in the bodies of statically determined fluent definitions. The introduction of the allen predicate in the body of a rule does not lead to a new dependency among the ground atoms of happensAt, initiatedAt, terminatedAt, holdsAt and holdsFor predicates. Moreover, RTEC$_{\mathsf{A}}$, following RTEC, does not allow cyclic dependencies among the rules defining statically determined fluents. As a result, a local stratification of an event description in RTEC$_{\mathsf{A}}$ can be constructed by following bottom-up the dependency graph of RTEC$_{\mathsf{A}}$ (see Proposition 1 and [Artikis et al., 2015]). □

APPENDIX B. PROOFS FOR PROPOSITIONS OF CHAPTER 3: CORRECTNESS AND COMPLEXITY

In the proofs concerning RTEC$_\circ$, RTEC$^\rightarrow$ and RTEC$_\mathsf{A}$, we assume that all operations inherited from RTEC, such as the evaluation of initiatedAt/terminatedAt rules and the pairing of initiation and termination points, are correct.

B.1 Proof of Proposition 3

Proposition (Correctness of RTEC$_\circ$): RTEC$_\circ$ computes all maximal intervals of the FVPs of an event description with cyclic dependencies, and no other interval. ▲

Proof. First, we demonstrate that the problem of maximal interval computation reduces to the problem of showing the correctness of holdsAt$(F = V, T)$ query computation for FVPs $F = V$ that are in a cycle C. Second, we prove that holdsAt$(F = V, T)$ is evaluated correctly at every time-point T of the first window $(q_1 - \omega, q_1]$ of the stream. To do this, we use an inductive proof on T. Third, we generalise the proof for all subsequent windows. We use an inductive proof which states that if holdsAt$(F = V, T)$ is evaluated correctly at every time-point in the first k windows of the stream, then holdsAt$(F = V, T)$ is also evaluated correctly at every time-point in the $k+1$-th window of the stream. Thus, we have proven that holdsAt$(F = V, T)$, where $F = V$ is part of a cycle, is evaluated correctly at every time-point of the stream. Based on our reduction, it follows that RTEC$_\circ$ computes all maximal intervals of the FVPs of an event description with cyclic dependencies, and no other interval.

Reduction to holdsAt$(F = V, T)$. In order to prove that RTEC$_\circ$ computes the maximal intervals of $F_i = V_i$, where $1 \leq i \leq n$, correctly, we need to show that RTEC$_\circ$ computes initiatedAt$(F_i = V_i, T)$ iff $T \in w$ is an initiation point of $F_i = V_i$, and terminatedAt$(F_i = V_i, T)$ iff $T \in w$ is a termination point of $F_i = V_i$. Afterwards, correct maximal interval computations follows, since the pairing of initiation and termination points is inherited from RTEC, and thus we assume its correctness. initiatedAt$(F_i = V_i, T)$ and terminatedAt$(F_i = V_i, T)$ are evaluated using domain-specific rules, following a 4-arity variant of schema (3.1). If such a rule contains a holdsAt$(F_j = V_j, T)$ condition, where $1 \leq j \leq n$ and $j \neq i$, i.e., $F_j = V_j$ is an FVP in cycle C, then RTEC$_\circ$ evaluates this condition using Algorithms 1 and 2. For the remaining conditions, RTEC$_\circ$ follows RTEC, and thus we assume that these conditions are evaluated correctly. Thus, to prove the correctness of evaluating rules with head initiatedAt$(F_i = V_i, T)$ or terminatedAt$(F_i = V_i, T)$ in RTEC$_\circ$, it suffices to show that holdsAt$(F_j = V_j, T)$ conditions, where $F_j = V_j$ is different from and in the same cycle as $F_i = V_i$, are evaluated correctly.

Proof for the first window of the stream. Suppose that the first window is $(q_1 - \omega, q_1]$ and that we have a cycle C in the dependency graph. The base of our induction relies on the common assumption in stream reasoning that the values of fluents at the start of the stream, if any, are known. Thus, the truth values of holdsAt$(F = V, q_1{-}\omega{+}1)$ are given for all FVPs $F = V \in C$. For the inductive step, we suppose that the truth values of holdsAt$(F' = V', T')$, for all $F' = V' \in C$ and $q_1{-}\omega < T' < T \leq q_1{-}\omega$, are evaluated correctly, and prove that the query holdsAt$(F = V, T)$, where $F = V \in C$, is evaluated correctly. RTEC$_\circ$ employs Algorithm 1, and moves to lines 8–11. This is because the truth value of holdsAt$(F = V, T)$ is not cached at this point, and, since we have the initial values of all FVPs in C, there is certainly some time-point T_{lCP} before T where the truth value of holdsAt$(F = V, T_{lCP})$ has been evaluated and cached. Since $T_{lCP} < T$, based on our inductive assumption, it holds that the cached tuple $(T_{lCP}, Tval_{lCP})$ has been evaluated correctly, i.e., $Tval_{lCP} = +$ if $F = V$ holds at T_{lCP}, and $Tval_{lCP} = -$ if $F = V$ does not hold at T_{lCP}. Then, RTEC$_\circ$ resolves holdsAtEC$(F = V, T_{lCP}, T, Tval_{lCP})$ using Algorithm 2.

We need to show that, given a correct input tuple $(T_{lCP}, Tval_{lCP})$, Algorithm 2 returns true if $F = V$ holds at T, and false if $F = V$ does not hold at T. To do this, we prove that Algorithm 2 follows the law of inertia of the Event Calculus, according to which:

1. If there is no initiation/termination of $F = V$ in $[T_{lCP}, T)$, then $F = V$ holds at T iff $F = V$ holds at T_{lCP}, i.e., the truth value $Tval_{lCP}$ persists up to time-point T.
2. If there is at least one initiation of $F = V$ in $[T_{lCP}, T)$ and no terminations, then $F = V$ holds at T.
3. If there is at least one termination of $F = V$ in $[T_{lCP}, T)$ and no initiations, then $F = V$ does not hold at T.
4. If there is at least one initiation and at least one termination of $F = V$ in $[T_{lCP}, T)$, then $F = V$ holds at T if the last initiation of $F = V$ in $[T_{lCP}, T)$ is after the last termination of $F = V$ in $[T_{lCP}, T)$, and $F = V$ does not hold at T in the opposite case.

Suppose that there is no initiation/termination of $F = V$ in $[T_{lCP}, T)$ (case 1). Then, Algorithm 2 returns true if $Tval_{lCP} = +$ (see lines 2 and 5), and false if $Tval_{lCP} = -$ (see lines 6 and 9). In case 2 (resp. 3), Algorithm 2 returns true (false) regardless of the value of $Tval_{lCP}$. In case 2, e.g., where we have initiations of $F = V$ and no terminations, Algorithm 2 returns true if $Tval_{lCP} = +$ (see lines 2 and 5), and calls recursively holdsAt$(F = V, T_i{+}1, T, +)$, where T_i is an initiation point of $F = V$, if $Tval_{lCP} = -$. In the latter case, the recursive call returns true, as we have $Tval_{lCP} = +$ and no terminations of $F = V$ in $[T_i{+}1, T)$ (lines 2–5). In case 4, where we have both initiations and terminations of $F = V$ in $[T_{lCP}, T)$, suppose that T_i and T_b are, respectively, the last initiation and the last termination of $F = V$ in $[T_{lCP}, T)$. If $Tval_{lCP} = +$ and $T_i > T_b$, then Algorithm 2 invokes holdsAtEC$(F = V, T_b{+}1, T, -)$ recursively (lines 2–5). Now we have $Tval_{lCP} = -$, and, since $T_i > T_b$, we have an additional recursive call to holdsAtEC$(F = V, T_i{+}1, T, +)$ (lines 6–9). This last recursive call returns true, because there are no terminations of $F = V$ in $[T_i{+}1, T)$. If $Tval_{lCP} = +$ and $T_b > T_i$, then Algorithm 2 calls holdsAtEC$(F = V, T_b{+}1, T, -)$, which returns false, because there are no initiations of $F = V$ in $[T_b{+}1, T)$. In both scenarios, the truth value of holdsAt$(F = V, T)$ that is computed by Algorithm 2 follows the last initiation/termination of $F = V$. The proofs for $Tval_{lCP} = -$ are symmetrical. Therefore, in all cases, Algorithm 2 is consistent with the law of inertia.

It remains to show that the initiatedAt and brokenAt queries invoked in Algorithm 2 are evaluated correctly. A holdsAtEC($F = V$, T_{lCP}, T, $Tval_{lCP}$) query may only lead to the evaluation of an initiatedAt/brokenAt query if $T_{lCP} < T$ (see line 1). The bodies of the rules defining brokenAt($F = V$, T_{lCP}, T_b, T) and initiatedAt($F = V$, T_{lCP}, T_i, T) may only contain holdsAt predicates with time-stamp T_b and T_i (see rule schema (3.1)). Thus, since we have $T_{lCP} \leq T_b < T$ and $T_{lCP} \leq T_i < T$, it follows from our inductive assumption that these holdsAt predicates are evaluated correctly. Thus, the brokenAt and initiatedAt queries that are invoked in Algorithm 2 are evaluated correctly.

We have proven that, when evaluating holdsAt($F = V$, T), the last cached tuple (T_{lCP}, $Tval_{lCP}$) before T is correct. Moreover, RTEC$_\circ$ follows the law of inertia when computing holdsAt($F = V$, T) based on (T_{lCP}, $Tval_{lCP}$), and all initiatedAt and brokenAt queries that are required to apply the law of inertia in this computation are evaluated correctly. It follows that RTEC$_\circ$ evaluates holdsAt($F = V$, T) correctly, according to the law of inertia in the Event Calculus.

Generalisation to the i-th window of the stream. We generalise our proof for the i-th window $(q_i{-}\omega, q_i]$. In order to prove that holdsAt($F = V$, T) is evaluated correctly at every time-point in $(q_i{-}\omega, q_i]$, we use the same induction on T as we did for the first window of the stream. Although the inductive step can be proven in exactly the same way, in this case we cannot assume that the values of all FVPs in cycle C at the first time-point of the window are known. In order to prove that holdsAt($F = V$, $q_i{-}\omega{+}1$) is evaluated correctly for $F = V \in C$, we use an inductive proof on the sequence of the windows of the stream.

We have proven that holdsAt($F = V$, T) is evaluated correctly for all time-points in the first window (inductive base). For the inductive step, we assume that holdsAt($F = V$, T) is evaluated correctly for all time-points in the first $i{-}1$ windows of the stream, and prove that holdsAt($F = V$, T) is evaluated correctly at the first time-point of the i-th window $(q_i{-}\omega, q_i]$ of the stream. According to the law of inertia, holdsAt($F = V$, $q_i{-}\omega{+}1$) is true iff initiatedAt($F = V$, $q_i{-}\omega$) holds, or holdsAt($F = V$, $q_i{-}\omega$) and it is not the case that terminatedAt($F = V$, $q_i{-}\omega$). The time-point $q_i{-}\omega$ is in the $i - 1$-th window, and thus the truth value of holdsAt($F = V$, $q_i{-}\omega$) is known. Moreover, based on rule schema (3.1), the initiation and termination points of $F = V$ in the $i - 1$-th window are also known. Thus, the truth value of holdsAt($F = V$, $q_i{-}\omega{+}1$) is evaluated correctly for all $F = V \in C$.

Therefore, we have a valid inductive base for proving that holdsAt($F = V$, T) is evaluated correctly at every time-point in $(q_i{-}\omega, q_i]$, and thus proceed with the same inductive proof as in the same first window of the stream. We have proven that RTEC$_\circ$ evaluates holdsAt($F = V$, T) correctly at every time-point of the stream, and that this fact implies that RTEC$_\circ$ computes the maximal intervals of the FVPs in an event description with cyclic dependencies, and no other interval. $\square$

B.2 Proof of Proposition 4

Proposition (Complexity of RTEC$_\circ$): The cost of evaluating the maximal intervals of an FVP whose definition includes cyclic dependencies is $\mathcal{O}(\omega k)$, where ω is the size of the window and k is the cost of evaluating all the initiatedAt/terminatedAt rules of an FVP at a given time-point without cyclic dependencies (see [Artikis et al., 2015] for an estimation of k). □

Proof. Suppose that the window is $(q_i{-}\omega, q_i]$ and that FVP $F = V$ is in a cycle C of the dependency graph. In order to compute the maximal intervals of $F = V$, RTEC$_\circ$ computes the initiation and termination points of $F = V$ by querying initiatedAt and terminatedAt for each FVP in cycle C and at each time-point in $(q_i{-}\omega, q_i]$, following an ascending time-point order. The bodies of the rules defining initiatedAt$(F = V, T)$ and terminatedAt$(F = V, T)$ contain happensAt predicates, holdsAt predicates for FVPs with level lower than the level of $F = V$, and holdsAt predicates for FVPs that are in cycle C. The cost of evaluating happensAt predicates and holdsAt predicates for FVPs with level lower than the level of $F = V$ is bounded by $\mathcal{O}(k)$. holdsAt predicates for FVPs that are in cycle C are evaluated using Algorithm 1 of RTEC$_\circ$.

We prove that the cost of evaluating initiatedAt$(F = V, T)$ is $\mathcal{O}(k)$ using an inductive proof on T. For the inductive base, we show that the cost of evaluating initiatedAt$(F = V, q_i{-}\omega{+}1)$ is $\mathcal{O}(k)$. Since the truth values of all holdsAt$(F' = V', q_i{-}\omega{+}1)$ predicates, where $F' = V' \in C$, are known and cached (see the proof of Proposition 3), each such body condition of a rule that matches initiatedAt$(F = V, q_i{-}\omega{+}1)$ can simply be fetched from memory in constant time. For each holdsAt$(F' = V', q_i{-}\omega{+}1)$ query, RTEC$_\circ$ retrieves the cached tuple $(q_i{-}\omega{+}1, +/-)$ from memory (see lines 4–7 of Algorithm 1). Such a rule may include at most $r{-}1$ holdsAt conditions about FVPs in cycle C, where r is the length of C. As a result, the cost of evaluating initiatedAt$(F = V, q_i{-}\omega{+}1)$ is $\mathcal{O}(k + r)$, where r can be absorbed by k since the number of fluents f of the event description is a factor of k and $f \leq k$. Therefore, the cost of evaluating initiatedAt$(F = V, q_i{-}\omega{+}1)$ is $\mathcal{O}(k)$.

For the inductive step, we assume that the conditions of initiatedAt$(F = V, T')$, at each time-point $T' < T$, are known and cached, leading to a cost of evaluating initiatedAt$(F = V, T')$ that is $\mathcal{O}(k)$. We prove that the cost of evaluating initiatedAt$(F = V, T)$ is also $\mathcal{O}(k)$. To compute initiatedAt$(F = V, T)$, we may need to evaluate holdsAt$(F' = V', T)$ queries, where $F' = V' \in C$. Based on the ascending time processing order of RTEC$_\circ$, the cache contains the truth values of holdsAt$(F' = V', T')$ at each time-point T', where $T' < T$. RTEC$_\circ$ retrieves the cached tuple that is closest to T, i.e., $(T{-}1, \mathit{PrevVal})$, and invokes holdsAtEC$(F' = V', T{-}1, T, \mathit{PrevVal})$ (see lines 4–11 of Algorithm 1). According to Algorithm 2, the evaluation of holdsAtEC$(F' = V', T{-}1, T, \mathit{PrevVal})$ involves the computation of brokenAt$(F = V, T{-}1)$ if $\mathit{PrevVal} = +$, or the computation of initiatedAt$(F = V, T{-}1)$ if $\mathit{PrevVal} = -$. In both cases, the truth values of all the required holdsAt conditions that involve an FVP in cycle C are present in the cache. As a result, the cost of evaluating initiatedAt$(F = V, T)$ is $\mathcal{O}(k)$.

We have proven that the cost of evaluating initiatedAt$(F = V, T)$ is $\mathcal{O}(k)$ at each time-point T in the window. Analogously, we may prove that the evaluation cost of terminatedAt$(F = V, T)$ is also $\mathcal{O}(k)$. Thus, the cost of computing all the initiation and termination points of $F = V$ in window $(q_i{-}\omega, q_i]$ is $\mathcal{O}(\omega k)$. The cost of maximal interval computation is bounded by $\mathcal{O}(\omega)$, and thus the overall cost of computing the maximal intervals of an FVP whose definition includes cyclic dependencies is $\mathcal{O}(\omega k)$. □

B.3 Proof of Proposition 6

We show that Proposition 6 holds for an FVP $F=V$, such that fi$(F=V, F=V', R)$. First, we prove that RTEC$^{\rightarrow}$ transfers the attempt events between windows that are required for correct FVP interval computation, and no other attempt event (see Lemma B.3). Second, we show that RTEC$^{\rightarrow}$ computes all future initiations of $F=V'$, and no other future initiation (see Lemma 5). Third, we demonstrate that, at the time of constructing the maximal intervals of $F=V$, lists $IP[F=V]$ and $TP[F=V]$ contain the initiations and terminations of $F=V$, and no other point, completing the proof for Proposition 6.

Lemma (Correctness of transferring attempt events): RTEC$^{\rightarrow}$ transfers the attempt events between windows that are required for correct FVP interval computation, and no other attempt event. ▲

Proof. We prove that, at query time q_i, RTEC$^{\rightarrow}$ selects the attempt event that may mark a future initiation of $F=V'$, if any, as specified in the definition of fi (see Definition 11). According to this definition, we have a future initiation of $F=V'$ at T_{att} iff there is an initiation of $F=V$ at $T_{att}-R$ and there is no cancellation point of the future initiation of $F=V'$ between $T_{att}-R$ and T_{att}. Given window size ω, we will prove that RTEC$^{\rightarrow}$ derives an attempt event att iff its time-stamp T_{att} satisfies the following conditions:

1. $T_{att} > q_i-\omega$.
2. $T_{att}-R \leq q_i-\omega$.
3. $F=V$ is initiated at $T_{att}-R$.
4. The future initiation of $F=V'$ is not cancelled between $T_{att}-R$ and the start of the window $q_i-\omega+1$.
5. The future initiation of $F=V'$ is not cancelled between $q_i-\omega+1$ and T_{att} by a future initiation of $F=V'$ that was generated before $q_i-\omega+1$.

Conditions 1 and 2 express that the attempt att falls inside the current window, while the initiation of $F=V$ that generated att is before the window. If this initiation of $F=V$ were also inside the window, then we would be able to perform the computation at q_i without transferring att. Conditions 3 and 4 express that the future initiation of $F=V'$ at T_{att} was staged and not cancelled up to $q_i-\omega+1$. The information before $q_i-\omega+1$ does not change at the q_i, and thus, if conditions 3 and 4 were violated at q_{i-1}, they are still violated at q_i. Condition 5 expresses that we should not select attempt event att if there is another attempt that satisfies conditions 1–4 and cancels the future initiation of $F=V'$ at T_{att}. This condition addresses the cases where we have several initiations of $F=V$ at $T_1, \dots T_n$, where $T_n-T_1 < R$, and the future initiations of $F=V'$ may not be postponed. In such cases, only the attempt generated by the initiation of $F=V$ at T_1 is selected, because the remaining ones will be cancelled, at the latest, by the future initiation of $F=V'$ at T_1+R.

RTEC$^{\rightarrow}$ derives the attempt event that should be kept using Algorithm 5. Suppose that the output T_{att} of Algorithm 5 is not $null$, i.e., T_{att} is the time-stamp of an attempt event. If the future initiations of $F = V'$ may be postponed, then we have $T_{att} > q_i{-}\omega$ and $T_{att}{-}R \leq q_i{-}\omega$ (see line 4 of Algorithm 5). If the future initiations of $F = V'$ may not be postponed, then we have $T_{att} > q_i{-}\omega$, while $T_{att}{-}R$ coincides with the starting point of an interval that contains the start of the window (see lines 8 and 10), and thus $T_{att}{-}R \leq q_i{-}\omega$. In both cases, conditions 1 and 2 hold for T_{att}.

Next, we prove that $F = V$ is initiated at $T_{att}{-}R$. T_{att} is one of the time-stamps that were cached at previous query time by Algorithm 6 of RTEC$^{\rightarrow}$. If the future initiations of $F = V'$ may be postponed, then Algorithm 6 caches all time-points $T{+}R$, such that T is an initiation point of $F = V$. Otherwise, if the future initiations of $F = V'$ may not be postponed, then Algorithm 6 caches all time-points $T_s{+}R$, such that T_s is the starting point of an interval of $F = V$. Since all starting points of intervals of $F = V$ are initiation points of $F = V$, it holds that, if Algorithm 6 caches time-point $T_s{+}R$, then $F = V$ is initiated at T_s. As a result, all time-stamps in list $Attempts[F = V]$ of Algorithm 5 are R time-points after an initiation of $F = V$. Since the time-stamp T_{att} computed by Algorithm 5 is one of the items of list $Attempts[F = V]$, $T_{att}{-}R$ is an initiation point of $F = V$, and thus condition 3 holds.

T_{att} satisfies conditions 1–3. We prove that RTEC$^{\rightarrow}$ derives an attempt att with time-stamp T_{att} iff the future initiation of $F = V'$ is not cancelled between $T_{att}{-}R$ and the start of the window $q_i{-}\omega{+}1$ (condition 4), and it is not cancelled between $q_i{-}\omega{+}1$ and T_{att} by a future initiation of $F = V'$ that was generated before $q_i{-}\omega{+}1$ (condition 5). Assume that RTEC$^{\rightarrow}$ derives attempt att. Then, T_{att} is computed by Algorithm 5, meaning that there is an interval $[T_s, T_f)$ of $F = V$ that contains $q_i{-}\omega{+}1$ (see line 2 of Algorithm 5). The existence of interval $[T_s, T_f)$ implies that $F = V$ is not 'broken' between T_s and $q_i{-}\omega{+}1$, as, otherwise, $[T_s, T_f)$ would have been segmented. If the future initiations of $F = V'$ may not be postponed, then $T_{att}{-}R = T_s$ (see line 10). Therefore, the future initiation of $F = V'$ is not cancelled between $T_{att}{-}R$ and $q_i{-}\omega{+}1$, i.e., condition 4 holds, and it is not cancelled between $q_i{-}\omega{+}1$ and T_{att} by a future initiation of $F = V'$ that was induced earlier, because such a future initiation would have been cancelled before interval $[T_s, T_f)$, i.e., condition 5 holds. Otherwise, if the future initiations of $F = V'$ may be postponed, then $T_{att}{-}R$ coincides with the last initiation of $F = V$ in $[T_s, q_i{-}\omega{+}1)$ (see lines 5–6). Thus, $F = V$ is not 'broken' between $T_{att}{-}R$ and $q_i{-}\omega{+}1$, i.e., condition 4 holds, and earlier initiations of $F = V$ do not cancel the future initiation of $F = V'$ at T_{att}, as such earlier initiations would be postponed by the initiation of $F = V$ at $T_{att}{-}R$, i.e., condition 5 holds. Therefore, if RTEC$^{\rightarrow}$ computes an attempt event att, then its time-stampt T_{att} satisfies conditions 4 and 5.

Assume that T_{att} satisfies conditions 1–5. Based on condition 4, $F = V$ is not 'broken' between $T_{att}{-}R$ and $q_i{-}\omega{+}1$, and thus there is a maximal interval $[T_s, T_f)$ of $F = V$ that contains the start of the window $q_i{-}\omega{+}1$. If the future initiations of $F = V'$ may not be postponed, then RTEC$^{\rightarrow}$ computes $T_s{+}R$. Since $F = V$ holds continuously in $[T_s, T_f)$ and in $[T_{att}{-}R, T_f)$, and T_s is the starting point of a maximal interval, it holds that $T_{att}{-}R \geq T_s$. If $T_{att}{-}R > T_s$, then, since the future initiations of $F = V'$ may not be postponed, the future initiation of $F = V'$ that was induced at T_s will cancel the future initiation of $F = V'$ at T_{att}, meaning that T_{att} does not satisfy condition 5, which is a contraction. Therefore, we have $T_{att}-R=T_s$, and thus RTEC$^{\rightarrow}$ derives time-point T_{att}. If the future initiations of $F=V'$ may be postponed, then RTEC$^{\rightarrow}$ computes the time-stamp T of the last attempt of initiating $F=V'$ that was induced before $q_i-\omega+1$. Earlier attempts of initiating $F=V'$ are cancelled, because the corresponding future initiations of $F=V'$ are postponed by the initiation of $F=V$ that generated the attempt at T. Thus, these attempts do not satisfy condition 4, and the attempt event at T is only attempt that satisfies conditions 4 and 5. As a result, T coincides with T_{att}. Therefore, if T_{att} satisfies conditions 1–5, then RTEC$^{\rightarrow}$ derives the attempt event att that takes place at T_{att}.

We proved that RTEC$^{\rightarrow}$ derives an attempt event att iff its time-stamp T_{att} satisfies conditions 1–5. □

Lemma 5 (Correctness of future initiation computation): Given fi$(F=V, F=V', R)$, RTEC$^{\rightarrow}$ computes all future initiations of $F=V'$, and no other future initiation. ▲

Proof. We start by presenting the proof for an acyclic cd-component containing the FVPs with fluent F; moreover, we assume that the vertex of FVP $F=V$ has no incoming f-edges, i.e., there are no future initiations of $F=V$. We will relax these assumptions in subsequent steps of the proof.

First, we prove that, at the time of evaluating the future initiations of $F=V'$ based on fi$(F=V, F=V', R)$ (see line 7 of Algorithm 3), the list $IP[F=V]$ of cached initiations of $F=V$ contains all initiations of $F=V$, and no other points, and the list $TP[F=V]$ of cached terminations of $F=V$ contains all terminations of $F=V$, except from those stemming from future initiations of $F=V'$, and no other points.

Correctness of $IP[F=V]$. $IP[F=V]$ contains the correct immediate initiations of $F=V$, because RTEC$^{\rightarrow}$ evaluated these initiations earlier, in line 4 of Algorithm 3, using an operation inherited from RTEC, which we have assumed to be correct. Since there are no future initiations of $F=V$, we conclude that $IP[F=V]$ contains all initiations of $F=V$, and no other points.

Correctness of $TP[F=V]$. $TP[F=V]$ contains the correct immediate terminations of $F=V$, because these terminations were computed at an earlier step by RTEC (see line 5 of Algorithm 3). $TP[F=V]$ does not contain future terminations of $F=V$ that stem from future initiations of an FVP $F=V_j$, where $V_j \neq V'$, as such future initiations do not terminate $F=V$. Suppose that we have fi$(F=V_i, F=V_j, R')$, where $V_i \neq V_j \neq V$ and $V_j \neq V'$. (We assume that $V_i \neq V$ without loss of generality, because there may be, at most, one fi with $F=V$ as its first argument. If there were a fact fi$(F=V, F=V'', R'')$, where $R'' > R$, then such a fact would never result in a future initiation of $F=V''$.) In order for a future initiation of $F=V_j$ to not get cancelled, $F=V_i$ must hold up to the time-point of the future initiation. Since fluents have at most one value at a time, it is not possible for $F=V$ to hold at the time of the future initiation of $F=V_j$. Therefore, a future initiation of $F=V_j$ cannot terminate $F=V$. As a result, the future initiations of $F=V'$ are the only future initiations of F that may terminate $F=V$. Therefore, the only terminations of $F=V$ that are missing from $TP[F=V]$, if any, are the ones stemming from future initiations of $F=V'$.

Second, we prove that, given lists $IP[F=V]$ and $TP[F=V]$, RTEC$^{\rightarrow}$ derives all the future initiations of $F=V'$, and no other time-points. According to Definition 5, there is a future initiation of $F=V'$ based on fi$(F=V, F=V', R)$ at $T{+}R$ iff $F=V$ is initiated at T and there is no cancellation point of the future initiation of $F=V'$ between T and $T{+}R$.

Correctness of future initiation evaluation at T_{att}. Suppose that there is a cached attempt event att with time-stamp T_{att}. RTEC$^{\rightarrow}$ computes a future initiation of $F=V'$ at T_{att} iff there is no cached cancellation point of the future initiation of $F=V'$ between the start of the window $q_i{-}\omega{+}1$ and T_{att} (see lines 3–4 of Algorithm 4). Since this is the earliest potential future initiation of $F=V'$ at q_i, $TP[F=V]$ contains all the termination points of $F=V$ that are between $q_i{-}\omega{+}1$ and T_{att}, and no other points. Therefore, the cache contains all cancellation points of future initiations of $F=V'$ between $q_i{-}\omega{+}1$ and T_{att}, and no other points. Moreover, there is no cancellation point between $T_{att}{-}R$ and T_{att} that was generated before $q_i{-}\omega{+}1$ (see Lemma B.3). Therefore, RTEC$^{\rightarrow}$ computes a future initiation at T_{att} iff there is no cancellation point between $T_{att}{-}R$ and T_{att}.

Correctness of future initiation evaluation after T_{att}. Suppose that $T_1, \dots, T_k$ are the temporally sorted initiations of $F=V$. We show that RTEC$^{\rightarrow}$ computes a future initiation of $F=V'$ at $T_i{+}R$, where $1 \leq i \leq k$, iff the future initiation of $F=V'$ is not cancelled between T_i and $T_i{+}R$. For the base case, according to lines 5–7 of Algorithm 4, RTEC$^{\rightarrow}$ computes a future initiation of $F=V'$ at $T_1{+}R$ iff there is no cancellation point between T_1 and $T_1{+}R$. Note that there are no initiations of $F=V$ before T_1, and thus no future initiations of $F=V'$ before $T_1{+}R$. In other words, the cache has all cancellation points of the future initiation of $F=V'$ between T_1 and $T_1{+}R$. Next, assume that RTEC$^{\rightarrow}$ has evaluated correctly the future initiations of $F=V'$ up to $T_{n-1}{+}R$, where $1 < n \leq k$. Since RTEC$^{\rightarrow}$ caches the future initiations that it derives (line 7 of Algorithm 4), the cache contains all cancellation points of the future initiations of $F=V'$ up to $T_{n-1}{+}R$. As a result, RTEC$^{\rightarrow}$ computes a future initiation of $F=V'$ at $T_n{+}R$ iff there is no cancellation point between T_n and $T_n{+}R$.

Third, we generalise the proof for the case where the vertex of $F=V$ has incoming f-edges, i.e., we may have future initiations of $F=V$. In this case, we may have additional initiations of $F=V$, and thus we have to re-establish the correctness of $IP[F=V]$.

Correctness of $IP[F=V]$ *with future initiations of* $F=V$. We consider the case where the vertex of $F=V$ has incoming f-edges, i.e., we may have future initiations of $F=V$, and prove that $IP[F=V]$ contains all initiations of $F=V$, and no other points. Suppose that we have n facts fi$(F=V_i, F=V, R_i)$, where $1 \leq i \leq n$, and all values V_i are pairwise different. Since there are no cycles in the dependency graph, RTEC$^{\rightarrow}$ processes FVPs $F=V_1, \dots, F=V_n$ before $F=V$ (see Definition 18). For each FVP $F=V_i$, if we may not have future initiations of $F=V_i$, then RTEC$^{\rightarrow}$ evaluates the future initiations of $F=V$ based on fi$(F=V_i, F=V, R_i)$ correctly, as proven above, and stores them in list $IP[F=V]$ (see lines 7 and 9 of Algorithm 3). Otherwise, if there are some fi facts defining $F=V_i$, then, based on the acyclicity of the cd-component, we may use an inductive proof on the vertices of the cd-component, in order to show that the future initiations of $F=V$ based on fi$(F=V_i, F=V, R_i)$ are derived correctly. As a result, when processing FVP $F=V$, $IP[F=V]$ contains all the future initiations of $F=V$.

Algorithm 13 initiatedAtCyclic($F\!=\!V, T$)

```
Input: $q_i, \omega, AttemptOfFVP$.
Output: true/false.
 1: if $T \leq q_i{-}\omega$ or $T > q_i$ then return false
 2: if cachedInit($F\!=\!V, T, +$) then return true
 3: if cachedInit($F\!=\!V, T, -$) then return false
 4: for each $body$ in bodiesOf(initiatedAt($F\!=\!V, T$)) do
 5:    if sat($body$) then
 6:       updateCache(cachedInit($F\!=\!V, T, +$))
 7:       return true
 8: for each fi($F\!=\!V^d, F\!=\!V, R$) do
 9:    if $T{=}{=}AttemptOfFVP[F\!=\!V^d]$ then
10:       if not cancelledCyclic($F\!=\!V^d, q_i{-}\omega, T$) then
11:          updateCache(cachedInit($F\!=\!V, T, +$))
12:          return true
13:    if $T > q_i{-}\omega{+}R$ and initiatedAtCyclic($F\!=\!V^d, T{-}R$) then
14:       if not cancelledCyclic($F\!=\!V^d, T{-}R, T$) then
15:          updateCache(cachedInit($F\!=\!V, T, +$))
16:          return true
17: updateCache(cachedInit($F\!=\!V, T, -$))
18: return false
```

Fourth, we generalise the proof for the case where we have a cd-component with cycles. In this case, we show that future initiation evaluation is reduced to initiatedAt/terminatedAt rule evaluations, which are inherited from RTEC, in conjunction with caching intermediate derivations.

Correctness of future initiation evaluation with cycles. Suppose that there is a cycle of f-edges that includes f-edge $(v_{F=V}, v_{F=V'})$. In this case, RTEC$^{\rightarrow}$ does not employ Algorithm 4; instead, it uses Algorithm 13 to evaluate the initiations of all FVPs in the cycle. When evaluating the future initiations of $F\!=\!V'$, Algorithm 13 does not assume that all initiations of $F\!=\!V$ and all cancellation points of future initiations of $F\!=\!V'$ have been evaluated and cached. To address this issue, Algorithm 13 follows a direct implementation of the definition of fi (see Definition 11), according to which any initiation/termination point that is required to evaluate a future initiation of $F\!=\!V'$ and has not been evaluated in a previous step, based on the cache, is determined using initiatedAt/terminatedAt rules. As a result, the correctness of Algorithm 13 follows from the correctness of rule evaluation, which we have assumed to be correct. Therefore, RTEC$^{\rightarrow}$ evaluates correctly future initiations with cycles. □

Proposition (Correctness of RTEC$^{\rightarrow}$): RTEC$^{\rightarrow}$ computes all maximal intervals of the FVPs of an event description, and no other interval. ▲

Proof. In the proof of Lemma 5, we demonstrated that, at the time of evaluating the future initiations of $F\!=\!V'$, the cache contains all initiations of $F\!=\!V$, and no other initiation points, and all terminations of $F = V$, except from those stemming from future initiations of $F = V'$, and no other termination points. Moreover, we proved that RTEC$^{\rightarrow}$ computes the future initiations of $F = V'$, and no other points. According to line 8 of Algorithm 3, the derived future initiations of $F = V'$ are added in the list of cached terminations of $F = V$. As a result, at the time of constructing the maximal intervals of $F = V$, the cache contains the initiations and terminations of $F = V$, and no other points, leading to correct maximal interval construction. Therefore, RTEC$^{\rightarrow}$ computes the maximal intervals of all FVPs of an event description, and no other interval. □

B.4 Proof of Proposition 7

We provide a proof for Proposition 7, describing the complexity of RTEC$^{\rightarrow}$. First, we demonstrate the complexity of transferring attempt events between windows (see Lemma 6). Second, we outline the cost of evaluating the future initiations of all FVPs in an acyclic cd-component (see Lemma 7). Third, we discuss the case of a cd-component with cycles (see Lemma 8). Combining Lemmas 6–8 leads directly to a proof for Proposition 7.

Lemma 6 (Complexity of transferring attempt events): The cost of transferring attempt events between windows is $\mathcal{O}(\omega)$. ▲

Proof. Consider fi$(F = V, F = V', R)$. RTEC$^{\rightarrow}$ employs Algorithm 6 to determine which attempt events of initiating $F = V'$ will be cached. In the worst case, Algorithm 6 iterates over the list of cached initiation points of $F = V$ once. The size of this list is bounded by the window size ω, and thus the worst case complexity of Algorithm 6 is $\mathcal{O}(\omega)$. RTEC$^{\rightarrow}$ employs Algorithm 5 to determine which one the cached attempt events, if any, may mark a future initiation of $F = V'$. In the worst case, Algorithm 5 iterates over the list of cached attempt events once, and the number of these events is bounded by the window size. As a result, the cost of Algorithm 5 is $\mathcal{O}(\omega)$. Therefore, the cost transferring attempt events between windows is $\mathcal{O}(\omega)$. □

Lemma 7 (Complexity of processing an acyclic cd-component): The cost of evaluating the future initiations of all FVPs in an acyclic cd-component is $\mathcal{O}(n_v(\omega{-}R)log(\omega))$, where n_v is the number of possible values of an FVP, ω is the window size and R is the delay of a future initiation. ▲

Proof. Consider fi$(F = V, F = V', R)$ and an acyclic cd-component that contains the vertex of FVP $F = V$. We compute the cost of evaluating the future initiations of $F = V'$. For each initiation point T of $F = V$, where $T \leq q_i{-}R$, we may have a future initiation of $F = V'$ that falls inside the window. The number of these initiations is bounded by $\omega{-}R$. Moreover, there may be a future initiation of $F = V'$ at the time of the cached attempt event, if any. Therefore, RTEC$^{\rightarrow}$ may evaluate a future initiation of $F = V'$ at most $\omega{-}R{+}1$ times. To evaluate each future initiation of $F = V'$, RTEC$^{\rightarrow}$ performs a cache retrieval operation from the sorted lists of initiation and termination points of $F = V$, in order to check whether one of them cancels the future initiation of $F = V'$. The size of each list is bounded by

ω, and thus the cost of determining whether a future initiation is cancelled is $\mathcal{O}(log(\omega))$. Therefore, after simplifications, the cost of evaluating the future initiations of $F=V'$ is $\mathcal{O}((\omega{-}R)\log(\omega))$. In an acyclic cd-component, all vertices are connected with f-edges, and thus correspond to FVPs with the same fluent. As a result, an acyclic cd-component may have at most n_v FVPs. Thus, the cost of evaluating the future initiations of all FVPs in a cd-component is $\mathcal{O}(n_v(\omega{-}R)log(\omega))$. □

Lemma 8 (Complexity of processing a cd-component with cycles): The cost of evaluating the future initiations of all FVPs in a cd-component with cycles is $\mathcal{O}(n_v(\omega{-}R)log(\omega))$, where n_v is the number of possible values of an FVP, ω is the window size and R is the delay of a future initiation. ▲

Proof. Consider fi$(F=V, F=V', R)$ and that the cd-component that includes the vertices of $F=V$ and $F=V'$ contains cycles with f-edges. In this case, RTEC$^{\rightarrow}$ does not assume that the cancellation points of future initiations of $F=V'$ are available in the cache. To address this issue, RTEC$^{\rightarrow}$ employs Algorithm 13, which may, at most, evaluate the future initiation of each FVP in the cd-component once, at each of the $\omega{-}R{+}1$ time-points of the window where we may have such an initiation. This is because Algorithm 13 uses the cache of RTEC$^{\rightarrow}$ in order to store previous evaluation of FVP initiations, and thus avoids re-computations. As a result, the cost of evaluating the future initiations of all FVPs in a cd-component including cycles with f-edges remains $\mathcal{O}((\omega{-}R)n_v log(\omega))$. □

Proposition (Complexity of RTEC$^{\rightarrow}$): The cost of evaluating the future initiations of the FVPs in a cd-component is $\mathcal{O}(n_v(\omega{-}R)log(\omega))$, where n_v is the number of possible values of the FVPs, ω is the window size and R is the delay of a future initiation. ▲

Proof. Based on Lemmas 6, 7 and 8, the worst-case cost of evaluating the future initiations of all FVPs in a cd-component, including windowing, is $\mathcal{O}(n_v(\omega{-}R)log(\omega))$. □

B.5 Proof of Lemma 1

Lemma: Algorithm 8 computes all interval pairs (i^s, i^t), where $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, satisfying an Allen relation, and no other interval pair. ▲

Proof. First, we demonstrate that Algorithm 8 is sound, i.e., if Algorithm 8 computes an interval pair (i^s, i^t), such that $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, then (i^s, i^t) satisfies the selected relation rel, i.e., rel(i^s, i^t) holds. When rel is not before, Algorithm 8 may only compute (i^s, i^t) if rel(i^s, i^t) holds (see line 9). When rel is before, (i^s, i^t) may only be computed in line 6 or line 8. If (i^s, i^t) is computed in line 6, p_s points to i^s and p_t points to i^t_b, where i^t_b is i^t or some interval before i^t in list $\mathcal{T}$. Moreover, according to line 5, before(i^s, i^t_b) holds, and thus $f(i^s) < s(i^t_b)$. If i^t_b is before i^t, then we have that $f(i^s) < s(i^t)$, because $\mathcal{T}$ is a sorted list of maximal intervals. Therefore, (i^s, i^t) also satisfies before. If (i^s, i^t) is computed in line 8, then the interval pair of the current iteration is (i^s, i^t_b), i^t_b is before i^t in $\mathcal{T}$ and

$f(i^s) \leq f(i^t_b)$ (see line 7). As a result, we have that $f(i^s) < s(i^t)$, and thus before(i^s, i^t) holds. Therefore, Algorithm 8 is sound.

We demonstrate that Algorithm 8 is complete, i.e., if, for the selected relation rel, rel(i^s, i^t) holds, where $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, then Algorithm 8 computes (i^s, i^t). First, we demonstrate completeness when rel is not before. Suppose that rel(i^s, i^t) holds and Algorithm 8 does not compute (i^s, i^t). In this case, according to line 9, there is no iteration of the *while* loop of Algorithm 8 such that pointer p_s points to i^s and p_t points to i^t. The condition of the *while* loop states that Algorithm 8 iterates over all items in at least one of the input lists (see line 2). Suppose that, in the current iteration, p_s points to i^s when p_t points to an interval i^t_b that is before i^t in list $\mathcal{T}$. Given that rel(i^s, i^t) holds, we demonstrate the necessary relative positions between the endpoints of i^s and i^t_b for each relation rel other than before below:

- meets : meets$(i^s, i^t) \iff f(i^s) = s(i^t) \Rightarrow f(i^s) > f(i^t_b)$
- starts : starts$(i^s, i^t) \iff s(i^s) = s(i^t) \wedge f(i^s) < f(i^t) \Rightarrow s(i^s) > f(i^t_b)$
- finishes : finishes$(i^s, i^t) \iff s(i^s) > s(i^t) \wedge f(i^s) = f(i^t) \Rightarrow s(i^s) > f(i^t_b)$
- during : during$(i^s, i^t) \iff s(i^s) > s(i^t) \wedge f(i^s) < f(i^t) \Rightarrow s(i^s) > f(i^t_b)$
- overlaps : overlaps$(i^s, i^t) \iff s(i^s) < s(i^t) \wedge f(i^s) > s(i^t) \wedge f(i^s) < f(i^t)$
  $\Rightarrow f(i^s) > f(i^t_b)$
- equal : equal$(i^s, i^t) \iff s(i^s) = s(i^t) \wedge f(i^s) = f(i^t) \Rightarrow s(i^s) > f(i^t_b)$

According to these results and the conditions in lines 10 and 12, Algorithm 8 increments pointer p_t, and not p_s. Algorithm 8 continues to increment only pointer p_t in the following iterations until p_t points to i^t.

Suppose now that, in the current iteration, p_t points to i^t when p_s points to an interval i^s_b that is before i^s in list $\mathcal{S}$. Given that rel(i^s, i^t) holds, for each relation rel other than before, the following conditions must hold for the endpoints of i^s_b and i^t:

- meets : meets$(i^s, i^t) \iff f(i^s) = s(i^t) \Rightarrow f(i^s_b) < s(i^t)$
- starts : starts$(i^s, i^t) \iff s(i^s) = s(i^t) \wedge f(i^s) < f(i^t) \Rightarrow f(i^s_b) < s(i^t)$
- finishes : finishes$(i^s, i^t) \iff s(i^s) > s(i^t) \wedge f(i^s) = f(i^t) \Rightarrow f(i^s_b) < f(i^t)$
- during : during$(i^s, i^t) \iff s(i^s) > s(i^t) \wedge f(i^s) < f(i^t) \Rightarrow f(i^s_b) < f(i^t)$
- overlaps : overlaps$(i^s, i^t) \iff s(i^s) < s(i^t) \wedge f(i^s) > s(i^t) \wedge f(i^s) < f(i^t)$
  $\Rightarrow f(i^s_b) < s(i^t)$
- equal : equal$(i^s, i^t) \iff s(i^s) = s(i^t) \wedge f(i^s) = f(i^t) \Rightarrow f(i^s_b) < s(i^t)$

By combining these results with the conditions in lines 10 and 12 of Algorithm 8, we deduce that Algorithm 8 increments only pointer p_s, and keeps incrementing only this pointer until it points to i^s.

We have proven that in, both cases, Algorithm 8 reaches an iteration such that p_s points to i^s and p_t points to i^t, which is a contradiction. Therefore, Algorithm 8 computes (i^s, i^t), and thus we have proven that Algorithm 8 is complete for all relations other than before.

Suppose that rel is before, before(i^s, i^t) holds and Algorithm 8 does not compute (i^s, i^t). According to line 5 of Algorithm 8, there is no iteration where p_s points to i^s and p_t points to i^t. Moreover, there is no iteration of Algorithm 8 where p_s points to i^s, p_t point to i^t_b, where is i^t_b is before i^t in $\mathcal{T}$ and $f(i^s) \leq f(i^t_b)$ holds (see line 7). Suppose that i^t_r is the earliest interval in $\mathcal{T}$ such that $f(i^s) \leq f(i^t_r)$. (Note that, since before$(i^s, i^t) \Leftrightarrow f(i^s) < s(i^t) < f(i^t)$ holds, i^t_r is guaranteed to exist.) Since Algorithm 8 processes all intervals in at least one of the input lists (see line 2), we distinguish three possibilities:

- p_s points to i^s when p_t points to i^t_b, where i^t_b is before i^t_r in $\mathcal{T}$. Since i^t_r is the first interval of $\mathcal{T}$ satisfying $f(i^s) \leq f(i^t_r)$, it holds that $f(i^s) > f(i^t_b)$. In this case, according to the conditions in lines 10 and 12, Algorithm 8 increments only pointer p_t, and thus p_t eventually points to i^t_r, reaching an iteration where p_s points to i^s and p_t points to a target interval i^t_r, such that $f(i^s) \leq f(i^t_r)$.

- p_s points to i^s when p_t points to i^t_c, where i^t_c is i^t_r or some interval between i^t_r and i^t in $\mathcal{T}$. By the definition of i^t_r, we have that $f(i^s) \leq f(i^t_c)$.

- p_t points to i^t when p_s points to i^s_b, where i^s_b is before i^s in $\mathcal{S}$. Since it holds that before(i^s, i^t), we have $f(i^s_b) < s(i^t)$. Therefore, according to the conditions in lines 10 and 12, Algorithm 8 increments p_s, and not p_t. Algorithm 8 continues to increment p_s until it points to i^s.

We have demonstrated that, in all cases, Algorithm 8 reaches an iteration such that p_s points to i^s and p_t points either to i^t or to a target interval i^t_c, such that $f(i^s) \leq f(i^t_c)$. This is a contradiction. Therefore, Algorithm 8 computes (i^s, i^t), and thus is complete in the case of before.

We have proven that Algorithm 8 is sound and complete. Therefore, Algorithm 8 computes all interval pairs (i^s, i^t), where $i^s \in \mathcal{S}$ and $i^t \in \mathcal{T}$, satisfying an Allen relation, and no other interval pair. □

B.6 Proof of Lemma 2

Lemma: Algorithm 9 caches all intervals that may satisfy an Allen relation other than before with an interval arriving in the future, and no other interval. In the case of before, Algorithm 9 caches the source intervals ending at most mem time-points before the start of the next window. ▲

Proof. Suppose that $s(w_{j+1})$ is the start of the next window. A target interval i^t ending before $s(w_{j+1})$ cannot satisfy an Allen relation with a source interval i^s ending after or at $s(w_{j+1})$, because all Allen relations require that $f(i^s) \leq f(i^t)$. Moreover, (a possibly revised incarnation of) all intervals ending after $s(w_{j+1})$ will be available in the next window. Therefore, an interval may need to be cached iff it is a source interval containing $s(w_{j+1})$, a target interval containing $s(w_{j+1})$ or a source interval ending before $s(w_{j+1})$.

- Caching a source interval i^s_* that contains $s(w_{j+1})$. We will prove that Algorithm 9 caches $[s(i^s_*), s(w_{j+1})]$ iff i^s_* may satisfy an Allen relation with a target interval i^t arriving in the future. meets/overlaps/before: if i^t occurs in the next window w_{j+1}, it holds that $s(i^t) > s(i^s_*)$, and thus i^s_* may satisfy meets, overlaps or before with i^t. Therefore, Algorithm 9 caches $[s(i^s_*), s(w_{j+1})]$ (line 6). starts/equal: starts(i^s_*, i^t) and equal(i^s_*, i^t) may hold only if $s(i^s_*) = s(i^t)$ and $f(i^s_*) \leq f(i^t)$, in which case i^t also contains $s(w_{j+1})$. Thus, we cache $[s(i^s_*), s(w_{j+1})]$ iff there is a target interval starting at $s(i^s_*)$ and containing $s(w_{j+1})$ (see the conditions for starts and equal in line 6). finishes/during: finishes(i^s_*, i^t) and during(i^s_*, i^t) may hold only if $s(i^s_*) > s(i^t)$ and $f(i^s_*) \leq f(i^t)$. Therefore, we cache $[s(i^s_*), s(w_{j+1})]$ iff there is a target interval starting before $s(i^s_*)$ and containing $s(w_{j+1})$ (see the conditions for finishes and during in line 6).

- Caching a source interval i^s that ends before $s(w_{j+1})$. In the case of before, Algorithm 9 caches i^s iff it ends in $[s(w_{j+1}) - mem, s(w_{j+1})]$ (see lines 17–18). In that case, Algorithm 9 may also cache the segment of a target interval i^t_* containing $s(w_{j+1})$, if any, that is before $s(w_{j+1})$ in order to compute before(i^s, i^t_*) in the next window (see lines 11–12). We will prove that Algorithm 9 caches i^s iff i^s may satisfy an Allen relation other than before with a target interval i^t arriving in the future. meets/starts/overlaps/during: meets(i^s, i^t), starts(i^s, i^t), overlaps(i^s, i^t) and during(i^s, i^t) may hold only for a target interval i^t arriving in the future if it contains $s(w_{j+1})$, because these relations require that $s(i^t) \leq f(i^s)$. Since the segment of i^t before $s(w_{j+1})$ will not change in the future, i^s may satisfy one of these relations with i^t in a subsequent window only if the relation is satified given the current prefix of i^t. Therefore, we cache i^s iff (i^s, i^t) satisfies the selected relation meets, starts, overlaps or during. Note that in these cases Algorithm 9 also caches the segment of i^t that is before $s(w_{j+1})$ in order to compute (i^s, i^t) correctly in the next window (see lines 9–10 and the condition for during in line 13). finishes/equal: finishes(i^s, i^t) and equal(i^s, i^t) require that $f(i^s) = f(i^t)$. Therefore, i^s may not satisfy these relations with a target interval ending after $s(w_{j+1})$, and thus we do not cache i^s when evaluating finishes or equal.

- Caching a target interval i^t_* that contains $s(w_{j+1})$. In the previous paragraph, we described that the segment $[s(i^t_*), s(w_{j+1})]$ of i^t_* may be cached in order to facilitate the computation of an interval pair (i^s, i^t_*), where i^s is a cached source interval arriving before $s(w_{j+1})$, in the next window. Apart from these cases, we will prove that Algorithm 9 caches $[s(i^t_*), s(w_{j+1})]$ iff i^t_* may satisfy an Allen relation with a source interval i^s arriving in the future. finishes/during: if i^s occurs in the next window w_{j+1}, it holds that $s(i^s) > s(i^t_*)$, and thus i^s_* may satisfy finishes or during with i^t. Therefore, Algorithm 9 caches $[s(i^t_*), s(w_{j+1})]$ (line 13). starts/equal: starts(i^s, i^t_*) and equal(i^s, i^t_*) may hold only if $s(i^s) = s(i^s_*)$ and $f(i^s) \leq f(i^t_*)$, in which case i^s also contains $s(w_{j+1})$. Thus, Algorithm 9 caches $[s(i^t_*), s(w_{j+1})]$ iff there is a source interval starting at $s(i^t_*)$ and containing $s(w_{j+1})$ (see the conditions for starts and equal in line 13). overlaps: may hold only if $s(i^s) < s(i^t_*)$. Therefore, Algorithm 9 caches $[s(i^t_*), s(w_{j+1})]$ iff there is a source interval starting before $s(i^t_*)$ and containing $s(w_{j+1})$ (see the conditions for overlaps in line 13). meets/before: meets(i^s, i^t_*) and before(i^s, i^t_*) require that $f(i^s) \leq s(i^t_*)$. Therefore, i^t_* cannot satisfy these relations with a future source interval, and thus Algorithm 9 does not cache $[s(i^t_*), s(w_{j+1})]$.

We demonstrated that, for all relations other than before, Algorithm 9 caches all intervals that may satisfy an Allen relation with an interval arriving in the future, and no other intervals. In the case of before, we showed that Algorithm 9 caches the source intervals ending at most mem time-points before the start of the next window. □

B.7 Proof of Proposition 10

Proposition (Complexity of RTEC$_\mathbf{A}$): The cost of computing the maximal intervals of a statically determined fluent defined in terms of an Allen relation is $\mathcal{O}(n)$, where n is the number of input intervals. ▲

Proof. The maximal intervals of statically determined fluents defined in terms of an Allen relation are derived by the allen construct, which is implemented in Algorithm 7. We will show that the cost of all execution steps of Algorithm 7 is $\mathcal{O}(n)$. To do this, we demonstrate the complexity of Algorithms 8 and 9, which are invoked by Algorithm 7.

- In our logic programming implementation of Algorithm 8, the suffix of the target list $\mathcal{T}$ starting from p_t is available in each iteration of the loop. Therefore, the tuples in lines 6 and 8 are derived without iterating over the corresponding elements of $\mathcal{T}$, and thus the cost of their addition in the $pairs$ list is $\mathcal{O}(1)$. In each iteration of Algorithm 8, we increment at least one of the pointers p_s and p_t, because the conditions in lines 10 and 12 are exhaustive, and the *while* loop terminates at the latest when both input lists have been traversed (see line 2). Therefore, Algorithm 8 performs at most one iteration for each input interval, and thus its cost is $\mathcal{O}(n)$.
- Algorithm 9 identifies the source intervals ending before the start of the next window $s(w_{j+1})$, and the source and the target interval, if any, containing $s(w_{j+1})$ with cost $\mathcal{O}(n)$. In lines 9, 11, 16 and 18, Algorithm 9 identifies the source interval(s) satisfying rel with cost $\mathcal{O}(n)$. The remaining operations of Algorithm 9 are variable comparisons and caching operations that are performed in constant time. Therefore, the cost of Algorithm 9 is $\mathcal{O}(n)$.

The remaining execution steps of Algorithm 7 comprise input list traversals and, possibly, the evaluation of an interval manipulation construct, i.e., union_all, intersect_all or relative_complement_all. The cost of these operations is $\mathcal{O}(n)$ [Artikis et al., 2015]. Therefore, the cost of computing the maximal intervals of a statically determined fluent defined in terms of an Allen relation is $\mathcal{O}(n)$. $\square$

APPENDIX C. PROOFS FOR PROPOSITIONS OF CHAPTER 6

C.1 Proof of Proposition 11

Proposition: If $[t_s, t_e]$ is a PMI, then t_s satisfies the following condition:

$$\forall t_{prev} \in [1, t_s),\ prev\_prefix[t_{prev}] > prev\_prefix[t_s] \tag{C.1}$$

♦

Proof. Suppose that t_s does not satisfy condition (C.1). Then,

$$\exists t'_s : t'_s < t_s \text{ and } prev\_prefix[t'_s] \leq prev\_prefix[t_s] \tag{C.2}$$

We have that:

$$\begin{aligned} dprange[t'_s, t_e] &= dp[t_e] - prefix[t'_s - 1] \\ &= dp[t_e] - prev\_prefix[t'_s] \\ &\overset{\text{from ineq. (C.2)}}{\geq} dp[t_e] - prev\_prefix[t_s] \\ &= dprange[t_s, t_e] \geq 0 \end{aligned}$$

Note that $dprange[t_s, t_e] \geq 0$ because $[t_s, t_e]$ is a PMI.

The fact that $dprange[t'_s, t_e] \geq 0$, as shown above, indicates that $\exists t'_e : t'_e \geq t_e$ and $[t'_s, t'_e]$ is a PMI — see corollary (5.16). Additionally, $[t_s, t_e]$ is a sub-interval of $[t'_s, t'_e]$ since $t'_s < t_s$. Therefore, by Definition 21, $[t_s, t_e]$ is not a PMI. By contradiction, t_s must satisfy condition (C.1). □

C.2 Proof of Lemma 3

Lemma: If $I = [s, e]$ is a PMI, then $dprange[s, e]$ is non-negative. Moreover, for every interval $I' = [k, l]$, such that I is a sub-inteval of I', we have $dprange[k, l] < 0$. ▲

Proof. Since I is a PMI, then $P(I) \geq \mathcal{T}$, i.e., the probability of I is greater or equal to the threshold $\mathcal{T}$. By taking advantage of the definitions of the lists of oPIEC (see Section 5.3), we have the following:

$$P(I) \geq \mathcal{T} \xLeftrightarrow{Eq.(5.11)} \sum_{s \leq m \leq e} L[m] \geq 0 \xLeftrightarrow{Eq.(5.12)} prefix[e] - prefix[s-1] \geq 0.$$

We proceed as follows. According to Eq. (5.13), we have that $dp[e] = \max_{e \leq m \leq n} prefix[m]$, where n is the last time-point seen by oPIEC, and therefore $dp[e] \geq prefix[e]$. Consequently, it holds that:

$$prefix[e] - prefix[s-1] \geq 0 \Longrightarrow \max_{e \leq m \leq n} prefix[m] - prefix[s-1] \geq 0 \xLeftrightarrow{Eq.(5.13)}$$

$$dp[e] - \mathit{prefix}[s-1] \geq 0 \overset{Eq.(5.14)}{\Longleftrightarrow} \mathit{dprange}[s,e] \geq 0$$

Therefore, we conclude that if $I = [s,e]$ is a PMI, then $\mathit{dprange}[s,e] \geq 0$.

We will now prove the second part of this lemma. Let $k \leq s, l \geq e, I' = [k,l], I' \neq I$, i.e., I is a sub-interval of I', and $\mathit{dprange}[k,l] \geq 0$. Then, we have:

$$\mathit{dprange}[k,l] \geq 0 \overset{Eq.(5.14)}{\Longleftrightarrow} dp[l] \geq \mathit{prefix}[k-1] \overset{Eq.(5.13)}{\Longleftrightarrow}$$

$$\max_{l \leq m \leq n} \mathit{prefix}[m] \geq \mathit{prefix}[k-1] \Longleftrightarrow \exists m : l \leq m \leq n, \mathit{prefix}[m] \geq \mathit{prefix}[k-1] \overset{Eq.(5.12)}{\Longleftrightarrow}$$

$$\exists m : l \leq m \leq n, \sum_{k \leq m' \leq m} L[m'] \geq 0 \overset{Eq.(5.11)}{\Longleftrightarrow} \exists m : l \leq m \leq n, P([k,m]) \geq \mathcal{T}$$

The probability of $I'' = [k,m]$, where $m \geq l$, is greater or equal to the threshold and $I \subset I' \subseteq I''$. Therefore, I is not a PMI. We reached a contradiction and thus $\mathit{dprange}[k,l] < 0$. $\square$

C.3 Proof of Lemma 4

Lemma: Suppose that $In[i..j]$ is the current data batch, $[sst_1..sst_m]$ is the list of time-points in the support set, and the interval $I = [sst_k, l]$, where $1 \leq k \leq m$ and $i \leq l \leq j$, is a PMI. The execution of Algorithm 11 will eventually reach the state $\{k, l, \mathit{true}\}$ or the state $\{k, l, \mathit{false}\}$.

Proof. The *while* loop of Algorithm 11 stops searching for PMIs starting with a time-point in the support set when either pt_{ss} has iterated over all elements in the support set, including sst_k, the starting point of I, or e has iterated over all elements of the current data batch, including l, the ending point of I. As a result, Algorithm 11 reaches a state for which either $pt_{ss} = k$ or $e = l$. There are two possibilities for the first encounter of such a state:

1. $\{k', l, \mathit{true/false}\}$, where $k' < k$. Since $I = [sst_k, l]$ is a PMI and a sub-interval of $[sst_{k'}, l]$, we have $\mathit{dprange}[sst_{k'}, l] < 0$ (see Lemma 3). Consequently, the condition of the *if* statement in line 5 of Algorithm 11 fails when its state is $\{k', l, \mathit{true/false}\}$, and pt_{ss} is subsequently incremented. The same holds for all the following states $\{k'', l, \mathit{true/false}\}$, where $k' < k'' < k$, because I is a sub-interval of $[sst_{k''}, l]$. Therefore, pt_{ss} will continue to increase until the algorithm reaches the state $\{k, l, \mathit{false}\}$; the flag is set to false because pt_{ss} is increased in the last iteration before state $\{k, l, \mathit{false}\}$ (see line 8 of Algorithm 11).

2. $\{k, l', \mathit{true/false}\}$, where $l' < l$. Since $I = [sst_k, l]$ is a PMI, we have $\mathit{dprange}[sst_k, l] \geq 0$ (Lemma 3). Equivalently, we have $dp[l] \geq \mathit{prefix}[sst_k - 1]$ (Eq. (5.14)). As a corollary of the definition of dp (Eq. (5.13)), for $t_1 < t_2$, we have:

$$\max_{t_1 \leq t \leq n}(\mathit{prefix}[t]) \geq \max_{t_2 \leq t \leq n}(\mathit{prefix}[t]) \overset{Eq.(5.13)}{\Longrightarrow} dp[t_1] \geq dp[t_2].$$

In other words, dp is a decreasing sequence and thus $dp[l''] \geq \mathit{prefix}[sst_k - 1]$, when $l'' < l$. So, for every state $\{k, l'', \mathit{true}/\mathit{false}\}$, where $l' \leq l'' < l$, we have that $\mathit{dprange}[sst_k, l''] \geq 0$. The *if* condition in line 5 of Algorithm 11 holds for all these states and e increases continuously until Algorithm 11 reaches the state $\{k, l, \mathit{true}\}$. The flag is set to true, because e is increased in the last iteration before state $\{k, l, \mathit{true}\}$ (see line 5 of Algorithm 11).

Hence, given that $I = [sst_k, l]$ is a PMI, the execution of Algorithm 11 reaches either the state $\{k, l, \mathit{true}\}$ or the state $\{k, l, \mathit{false}\}$. □

C.4 Proof of Proposition 13

Proposition (Soundness of Interval Computation): Every interval computed by oPIEC is a PMI of the data seen so far. □

Proof. Suppose that $In[i..j]$ is the current data batch and $[sst_1..sst_m]$ is the list of time-points in the support set. Moreover, $I = [sst_k, l]$, where $1 \leq k \leq m, i \leq l \leq j$, is an interval computed by Algorithm 11. We will prove that I is a PMI.

In order for Algorithm 11 to reach line 7 and add the interval $I = [sst_k, l]$ to its results, the state must be $\{k, l+1, \mathit{true}\}$. Recall that the value of flag is true iff the *if* condition in line 5 was satisfied in the previous iteration, i.e., $\mathit{dprange}[sst_k, l] \geq 0$. However, since the execution reaches line 7, the *if* condition in line 5 fails in the current iteration, which means that $\mathit{dprange}[sst_k, l+1] < 0$. Hence, we have the following:

$$\mathit{dprange}[sst_k, l] \geq 0 \xLeftrightarrow{Eq.(5.14)} dp[l] \geq \mathit{prefix}[sst_k - 1] \xLeftrightarrow{Eq.(5.13)}$$

$$\max_{l \leq l' \leq j}(\mathit{prefix}[l']) \geq \mathit{prefix}[sst_k - 1] \xRightarrow{(a)} \mathit{prefix}[l] \geq \mathit{prefix}[sst_k - 1] \xLeftrightarrow{Eq.(5.12)}$$

$$\sum_{sst_k \leq t \leq l} L[t] \geq 0 \xLeftrightarrow{Eq.(5.11)} P(I = [sst_k, l]) \geq \mathcal{T}$$

Implication (a) holds because if there existed a $l' > l$ for which $\mathit{prefix}[l'] \geq \mathit{prefix}[l]$, then we would have $\mathit{dprange}[sst_k, l+1] \geq 0$, contradicting the fact that line 7 has been reached. Thus, we conclude that the probability of interval I is greater or equal to the threshold. Next, we will show that I is not a sub-interval of a PMI.

Suppose that I is a sub-interval of the PMI $I' = [sst_{k'}, l']$, i.e., $k' \leq k, l' \geq l, I \neq I'$ and $P(I') \geq \mathcal{T}$. Based on Lemma 4, Algorithm 11 will reach the state $\{k', l', \mathit{true}/\mathit{false}\}$. Also, since I' is a PMI, we have $\mathit{dprange}[sst_{k'}, l'] \geq 0$ (Lemma 3). So, the *if* condition in line 7 of Algorithm 11 is satisfied, e is incremented and the flag is set to true, thus reaching the state $\{k', l'+1, \mathit{true}\}$.

Since $I = [sst_k, l]$ is computed by oPIEC, the state $\{k, l+1, \mathit{true}\}$ is also reached. Moreover, since $k' \leq k$, the following transition between states has to take place, possibly in multiple steps:

$$\{k', l'+1, true\} \rightarrow \ldots \rightarrow \{k, l+1, true\}$$

We distinguish between two cases depending on the value of l'. If $l' > l$, the above state transition is infeasible since e does not decrease in Algorithm 11. If $l' = l$, the state transition can only be achieved by increasing pt_{ss} up to the value k while e remains constant. However, incrementing pt_{ss} is accompanied by setting the $flag$ to $false$ (see line 8 of Algorithm 11). The $flag$ can only be reset when incrementing e (in line 5). Therefore, eventually either $e = l + 1$ and $flag = false$, or $e > l + 1$ and $flag = true$. In either case, the aforementioned transition cannot take place.

Consequently, oPIEC cannot reach the state $\{k, l+1, true\}$ and, as a result, cannot compute I. By contradiction, there is no PMI I' such that I is a sub-interval of I'.

Therefore, since $P(I) \geq \mathcal{T}$ and I is not the sub-interval of a PMI, I is a PMI. Thus, every interval computed by oPIEC is a PMI. □

REFERENCES

Aghasadeghi, A., den Bussche, J. V., & Stoyanovich, J. (2024). Temporal graph patterns by timed automata. *VLDB J.*, *33*(1), 25–47.

Agrawal, J., Diao, Y., Gyllstrom, D., & Immerman, N. (2008). Efficient pattern matching over event streams. *SIGMOD Conference*, 147–160.

Albanese, M., Chellappa, R., Cuntoor, N., Moscato, V., Picariello, A., Subrahmanian, V. S., & Udrea, O. (2010). PADS: A Probabilistic Activity Detection Framework for Video Data. *IEEE Trans. Pattern Anal. Mach. Intell.*, *32*(12), 2246–2261.

Alevizos, E., Artikis, A., & Paliouras, G. (2022). Complex event forecasting with prediction suffix trees. *VLDB J.*, *31*.

Alevizos, E., Skarlatidis, A., Artikis, A., & Paliouras, G. (2017). Probabilistic complex event recognition: A survey. *Commun. ACM*, *50*(5), 71:1–71:31.

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Commun. ACM*, *26*(11), 832–843.

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

Allison, L. (2003). Longest biased interval and longest non-negative sum interval. *Bioinform.*, *19*(10), 1294–1295.

Alrayes, B., Kafali, Ö., & Stathis, K. (2018). Concurrent bilateral negotiation for open e-markets: The conan strategy. *Knowl. Inf. Syst.*, *56*(2), 463–501.

Andrienko, N., Andrienko, G., Artikis, A., Mantenoglou, P., & Rinzivillo, S. (2024). Human-in-the-loop: Visual analytics for building models recognising behavioural patterns in time series. *IEEE Computer Graphics and Applications*, 1–15. https://doi.org/10.1109/MCG.2024.3379851

Anicic, D., Rudolph, S., Fodor, P., & Stojanovic, N. (2012). Stream reasoning and complex event processing in ETALIS. *Semantic Web*, *3*(4), 397–407.

Apriceno, G., Passerini, A., & Serafini, L. (2021). A neuro-symbolic approach to structured event recognition. *TIME*, *206*, 11:1–11:14.

Apriceno, G., Passerini, A., & Serafini, L. (2022). A neuro-symbolic approach for real-world event recognition from weak supervision. *TIME*, *247*, 12:1–12:19.

Apt, K. R., & Bol, R. N. (1994). Logic programming and negation: A survey. *J. Log. Program.*, *19/20*, 9–71.

Arasu, A., Babu, S., & Widom, J. (2006). The CQL continuous query language: Semantic foundations and query execution. *VLDB J.*, *15*(2), 121–142.

Arias, J., Carro, M., Chen, Z., & Gupta, G. (2020). Justifications for goal-directed constraint answer set programming. *ICLP Technical Communications*, *325*, 59–72.

Arias, J., Carro, M., Chen, Z., & Gupta, G. (2022). Modeling and reasoning in event calculus using goal-directed constraint answer set programming. *Theory Pract. Log. Program.*, *22*(1), 51–80.

Arias, J., Carro, M., Salazar, E., Marple, K., & Gupta, G. (2018). Constraint answer set programming without grounding. *Theory Pract. Log. Program.*, *18*(3-4), 337–354.

Arrieta, A. B., Rodríguez, N. D., Ser, J. D., Bennetot, A., Tabik, S., Barbado, A., García, S., Gil-Lopez, S., Molina, D., Benjamins, R., Chatila, R., & Herrera, F. (2020). Explainable artificial intelligence (XAI): concepts, taxonomies, opportunities and challenges toward responsible AI. *Inf. Fusion*, *58*, 82–115.

Artikis, A., Makris, E., & Paliouras, G. (2021). A probabilistic interval-based event calculus for activity recognition. *Ann. Math. Artif. Intell.*, *89*(1-2), 29–52.

Artikis, A., & Pitt, J. V. (2008). Specifying open agent systems: A survey. *ESAW Workshop*, *5485*, 29–45.

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

Artikis, A., Sergot, M. J., & Paliouras, G. (2010). A logic programming approach to activity recognition. *EIMM Workshop in MM*, 3–8.

Artikis, A., Sergot, M. J., & Paliouras, G. (2012a). Run-time composite event recognition. *DEBS*, 69–80.

Artikis, A., Sergot, M. J., & Paliouras, G. (2015). An event calculus for event recognition. *IEEE Trans. Knowl. Data Eng.*, *27*(4), 895–908.

Artikis, A., Skarlatidis, A., Portet, F., & Paliouras, G. (2012b). Logic-based event recognition. *Knowl. Eng. Rev.*, *27*(4), 469–506.

Artikis, A., & Zissis, D. (Eds.). (2021). *Guide to maritime informatics*. Springer.

Awad, A., Tommasini, R., Langhi, S., Kamel, M., Valle, E. D., & Sakr, S. (2022). D^{2}ia: User-defined interval analytics on distributed streams. *Inf. Syst.*, *104*, 101679.

Bai, Y., Thakkar, H., Wang, H., Luo, C., & Zaniolo, C. (2006). A data stream language and system designed for power and extensibility. *CIKM*, 337–346.

Baral, C., Gelfond, M., & Rushton, J. N. (2009). Probabilistic reasoning with answer sets. *Theory Pract. Log. Program.*, *9*(1), 57–144.

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

Baumgartner, P. (2021a). Combining event calculus and description logic reasoning via logic programming. *FroCoS*, 98–117.

Baumgartner, P. (2021b). The fusemate logic programming system. *CADE*, *12699*, 589–601.

Bazoobandi, H. R., Beck, H., & Urbani, J. (2017). Expressive stream reasoning with laser. *ISWC*, *10587*, 87–103.

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

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

Bellodi, E., Alberti, M., Riguzzi, F., & Zese, R. (2020). Map inference for probabilistic logic programming. *Theory Pract. Log. Program.*, *20*(5), 641–655.

Bereta, K., Chatzikokolakis, K., & Zissis, D. (2021). Maritime reporting systems. In *Guide to maritime informatics* (pp. 3–30). Springer.

Berreby, F., Bourgne, G., & Ganascia, J. (2018). Event-based and scenario-based causality for computational ethics. *AAMAS*, 147–155.

Bragaglia, S., Chesani, F., Mello, P., Montali, M., & Torroni, P. (2012). Reactive event calculus for monitoring global computing applications. *Logic Programs, Norms and Action - Essays in Honor of Marek J. Sergot on the Occasion of His 60th Birthday*, *7360*, 123–146.

Brendel, W., Fern, A., & Todorovic, S. (2011). Probabilistic event logic for interval-based event recognition. *CVPR*, 3329–3336.

Bromuri, S., Urovi, V., & Stathis, K. (2010). Icampus: A connected campus in the ambient event calculus. *Int. J. Ambient Comput. Intell.*, *2*(1), 59–65.

Bucchi, M., Grez, A., Quintana, A., Riveros, C., & Vansummeren, S. (2022). CORE: a complex event recognition engine. *Proc. VLDB Endow.*, *15*(9), 1951–1964.

Cabalar, P., Fandinno, J., & Fink, M. (2014). Causal graph justifications of logic programs. *Theory Pract. Log. Program.*, *14*(4-5), 603–618.

Cabalar, P., Fandinno, J., & Muñiz, B. (2020). A system for explainable answer set programming. *ICLP Technical Communications*, *325*, 124–136.

Cervesato, I., Franceschet, M., & Montanari, A. (2000). A guided tour through some extensions of the event calculus. *Comput. Intell.*, *16*(2), 307–347.

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

Chaudet, H. (2006). Extending the event calculus for tracking epidemic spread. *Artif. Intell. Medicine*, *38*(2), 137–156.

Chavira, M., & Darwiche, A. (2008). On probabilistic inference by weighted model counting. *Artif. Intell.*, *172*(6-7), 772–799.

Chawda, B., Gupta, H., Negi, S., Faruquie, T. A., Subramaniam, L. V., & Mohania, M. K. (2014). Processing interval joins on map-reduce. *EDBT*, 463–474.

Chekol, M. W., Pirrò, G., & Stuckenschmidt, H. (2019). Fast interval joins for temporal SPARQL queries. *WWW*, 1148–1154.

Chesani, F., Mello, P., Montali, M., & Torroni, P. (2009). Commitment tracking via the reactive event calculus. *IJCAI*, 91–96.

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

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

Chittaro, L., & Montanari, A. (1996). Efficient temporal reasoning in the cached event calculus. *Comput. Intell.*, *12*(3), 359–382.

Chopra, A. K., Christie, S. H., & Singh, M. P. (2020). An evaluation of communication protocol languages for engineering multiagent systems. *J. Artif. Intell. Res.*, *69*, 1351–1393.

Clark, K. L. (1977). Negation as failure. *Logic and Data Bases*, 293–322.

Cugola, G., & Margara, A. (2010). TESLA: A formally defined event specification language. *DEBS*, 50–61.

Cugola, G., Margara, A., Matteucci, M., & Tamburrelli, G. (2015). Introducing uncertainty in complex event processing: Model, implementation, and validation. *Computing*, *97*(2), 103–144.

Darwiche, A., & Marquis, P. (2002). A knowledge compilation map. *J. Artif. Intell. Res.*, *17*, 229–264.

Das, A., Gandhi, S. M., & Zaniolo, C. (2018). ASTRO: A datalog system for advanced stream reasoning. *CIKM*, 1863–1866.

D'Asaro, F. A. (2019). *Probabilistic epistemic reasoning about actions* (Doctoral dissertation). University College London, UK.

D'Asaro, F. A., Bikakis, A., Dickens, L., & Miller, R. (2017). Foundations for a probabilistic event calculus. *LPNMR*, 57–63.

D'Asaro, F. A., Raggioli, L., Malek, S., Grazioso, M., & Rossi, S. (2023). An application of a runtime epistemic probabilistic event calculus to decision-making in e-health systems. *Theory Pract. Log. Program.*, *23*(5), 1070–1093.

De Raedt, L., Kimmig, A., & Toivonen, H. (2007). Problog: A probabilistic prolog and its application in link discovery. *IJCAI*, 2462–2467.

de Leng, D., & Heintz, F. (2019). Approximate stream reasoning with metric temporal logic under uncertainty. *AAAI*, 2760–2767.

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

Demers, A. J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., & White, W. M. (2007). Cayuga: A general purpose event monitoring system. *CIDR*, 412–422.

Demolombe, R. (2014). Obligations with deadlines: A formalization in dynamic deontic logic. *J. Log. Comput.*, *24*(1), 1–17.

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

Eiter, T., Ogris, P., & Schekotihin, K. (2019). A distributed approach to LARS stream reasoning (system paper). *Theory Pract. Log. Program.*, *19*(5-6), 974–989.

Falcionelli, N., Sernani, P., de la Torre, A. B., Mekuria, D. N., Calvaresi, D., Schumacher, M., Dragoni, A. F., & Bromuri, S. (2019). Indexing the event calculus: Towards practical human-readable personal health systems. *Artif. Intell. Medicine*, *96*, 154–166.

Fierens, D., Van Den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., & De Raedt, L. (2014). Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory Pract. Log. Program.*, *15*(3), 358–401.

Fikioris, G., Patroumpas, K., Artikis, A., Pitsikalis, M., & Paliouras, G. (2023). Optimizing vessel trajectory compression for maritime situational awareness. *GeoInformatica*, *27*(3), 565–591.

Fragkoulis, M., Carbone, P., Kalavri, V., & Katsifodimos, A. (2024). A survey on the evolution of stream processing systems. *VLDB J.*, *33*(2), 507–541.

Galton, A., & Augusto, J. C. (2002). Two approaches to event definition. *DEXA*, *2453*, 547–556.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2019). Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, *19*(1), 27–82.

Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming. *ICLP/SLP*, 1070–1080.

Georgala, K., Sherif, M. A., & Ngomo, A. N. (2016). An efficient approach for the generation of allen relations. *ECAI*, *285*, 948–956.

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

Grez, A., Riveros, C., & Ugarte, M. (2019). A formal framework for complex event processing. *ICDT*, *127*, 5:1–5:18.

Grez, A., Riveros, C., Ugarte, M., & Vansummeren, S. (2020). On the expressiveness of languages for complex event recognition. *ICDT*, *155*, 15:1–15:17.

Havelund, K., Omer, M., & Peled, D. (2021). Monitoring first-order interval logic. *SEFM*, *13085*, 66–83.

Hindriks, K., & Riemsdijk, M. (2013). A real-time semantics for norms with deadlines. *AAMAS*, 507–514.

Hopkins, J., Kafali, Ö., Alrayes, B., & Stathis, K. (2019). Pirasa: Strategic protocol selection for e-commerce agents. *Electron. Mark.*, *29*(2), 239–252.

Jones, A. J. I., & Sergot, M. J. (1996). A formal characterisation of institutionalised power. *Log. J. IGPL*, *4*(3), 427–443.

Kafali, Ö., Romero, A. E., & Stathis, K. (2017). Agent-oriented activity recognition in the event calculus: An application for diabetic patients. *Comput. Intell.*, *33*(4), 899–925.

Karlsson, L., Gustafsson, J., & Doherty, P. (1998). Delayed effects of actions. *ECAI*, 542–546.

Katzouris, N., & Artikis, A. (2020). WOLED: A tool for online learning weighted answer set rules for temporal reasoning under uncertainty. *KR*, 790–799.

Katzouris, N., Paliouras, G., & Artikis, A. (2023). Online learning probabilistic event calculus theories in answer set programming. *Theory Pract. Log. Program.*, *23*(2), 362–386.

Kauffman, S., Havelund, K., Joshi, R., & Fischmeister, S. (2018). Inferring event stream abstractions. *Formal Methods Syst. Des.*, *53*(1), 54–82.

Khan, A., Serafini, L., Bozzato, L., & Lazzerini, B. (2019). Event detection from video using answer set programing. *CILC*, *2396*, 48–58.

Kimmig, A., Demoen, B., Raedt, L. D., Costa, V. S., & Rocha, R. (2011). On the implementation of the probabilistic logic programming language ProbLog. *Theory Pract. Log. Program.*, *11*(2-3), 235–262.

Körber, M., Glombiewski, N., Morgen, A., & Seeger, B. (2021). Tpstream: Low-latency and high-throughput temporal pattern matching on event streams. *Distributed Parallel Databases*, *39*(2), 361–412.

Kowalski, R. A., & Sergot, M. J. (1985). Computer representation of the law. *IJCAI*, 1269–1270.

Kowalski, R. A., & Sergot, M. J. (1986). A logic-based calculus of events. *New Gener. Comput.*, *4*(1), 67–95.

Laptev, N., Mozafari, B., Mousavi, H., Thakkar, H., Wang, H., Zeng, K., & Zaniolo, C. (2016). Extending relational query languages for data streams. In *Data stream management* (pp. 361–386). Springer.

Law, M. (2018). *Inductive learning of answer set programs* (Doctoral dissertation). Imperial College London, UK.

Law, M. (2023). Conflict-driven inductive logic programming. *Theory Pract. Log. Program.*, *23*(2), 387–414.

Law, Y., Wang, H., & Zaniolo, C. (2004). Query languages and data models for database sequences and data streams. *VLDB*, 492–503.

Lee, J., & 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.

Li, M., Mani, M., Rundensteiner, E. A., & Lin, T. (2011). Complex event pattern detection over streams with interval-based temporal semantics. *DEBS*, 291–302.

List, T., Bins, J., Vazquez, J., & Fisher, R. B. (2005). Performance evaluating the evaluator. *VS-PETS*, 129–136.

Lloyd, J. W. (1987). *Foundations of logic programming, 2nd edition*. Springer.

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

Manhaeve, R., Dumančić, S., Kimmig, A., Demeester, T., & De Raedt, L. (2021). Neural probabilistic logic programming in deepproblog. *Artif. Intell.*, *298*, 103504.

Marín, R. H., & Sartor, G. (1999). Time and norms: A formalisation in the event-calculus. *ICAIL*, 90–99.

Marra, G., Dumancic, S., Manhaeve, R., & Raedt, L. D. (2024). From statistical relational to neurosymbolic artificial intelligence: A survey. *Artif. Intell.*, *328*, 104062.

Mavrommatis, A., Artikis, A., Skarlatidis, A., & Paliouras, G. (2016). A distributed event calculus for event recognition. *AI-IoT Workshop in ECAI*, *1724*, 31–37.

McAreavey, K., Bauters, K., Liu, W., & Hong, J. (2017). The event calculus in probabilistic logic programming with annotated disjunctions. *AAMAS*, 105–113.

Michelioudakis, E., Artikis, A., & Paliouras, G. (2019). Semi-supervised online structure learning for composite event recognition. *Mach. Learn.*, *108*(7), 1085–1110.

Michelioudakis, E., Artikis, A., & Paliouras, G. (2024). Online semi-supervised learning of composite event rules by combining structure and mass-based predicate similarity. *Mach. Learn.*, *113*(3), 1445–1481.

Michelioudakis, E., Skarlatidis, A., Paliouras, G., & Artikis, A. (2016). OSLα: Online structure learning using background knowledge axiomatization. *ECML-PKDD*, 232–247.

Miller, R., & Shanahan, M. (2002). Some alternative formulations of the event calculus. In *Computational logic: Logic programming and beyond, essays in honour of Robert A. Kowalski, part II* (pp. 452–490). Springer.

Montali, M., Maggi, F. M., Chesani, F., Mello, P., & 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.

Morariu, V. I., & Davis, L. S. (2011). Multi-agent event recognition in structured scenarios. *CVPR*, 3289–3296.

Moura, J., & Damásio, C. V. (2015). Allowing cyclic dependencies in modular logic programming. *EPIA*, *9273*, 363–375.

Mueller, E. T. (2008). Event Calculus. In *Handbook of knowledge representation* (pp. 671–708). Elsevier.

Mueller, E. T. (2015). *Commonsense reasoning: An event calculus based approach*. Morgan Kaufmann.

Ngomo, A. N., Sherif, M. A., Georgala, K., Hassan, M. M., Dreßler, K., Lyko, K., Obraczka, D., & Soru, T. (2021). LIMES: A framework for link discovery on the semantic web. *Künstliche Intell.*, *35*(3), 413–423.

Nomikos, C., Rondogiannis, P., & Gergatsoulis, M. (2005). Temporal stratification tests for linear and branching-time deductive databases. *Theor. Comput. Sci.*, *342*(2-3), 382–415.

Paschke, A. (2005). *ECA-RuleML: An approach combining ECA rules with temporal interval-based KR event/action logics and transactional update logics* (tech. rep. No. 11). TU München.

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

Paschke, A., & Kozlenkov, A. (2009). Rule-based event processing and reaction rules. *RuleML*, *5858*, 53–66.

Patroumpas, K., Alevizos, E., Artikis, A., Vodas, M., Pelekis, N., & Theodoridis, Y. (2017). Online event recognition from moving vessel trajectories. *GeoInformatica*, *21*(2), 389–427.

Patroumpas, K., Artikis, A., Katzouris, N., Vodas, M., Theodoridis, Y., & Pelekis, N. (2015). Event recognition for maritime surveillance. *EDBT*, 629–640.

Phuoc, D. L., Dao-Tran, M., Parreira, J. X., & Hauswirth, M. (2011). A native and adaptive approach for unified processing of linked streams and linked data. *ISWC*, *7031*, 370–388.

Piatov, D., Helmer, S., Dignös, A., & Persia, F. (2021). Cache-efficient sweeping-based interval joins for extended allen relation predicates. *VLDB J.*, *30*(3), 379–402.

Pilourdault, J., Leroy, V., & Amer-Yahia, S. (2016). Distributed evaluation of top-k temporal joins. *SIGMOD Conference*, 1027–1039.

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

Pitt, J., Kamara, L., Sergot, M., & Artikis, A. (2006). Voting in multi-agent systems. *Comput. J.*, *49*(2), 156–170.

Pontelli, E., Son, T. C., & El-Khatib, O. (2009). Justifications for logic programs under answer set semantics. *Theory Pract. Log. Program.*, *9*(1), 1–56.

Poppe, O., Lei, C., Rundensteiner, E. A., & Maier, D. (2019). Event trend aggregation under rich event matching semantics. *SIGMOD*, 555–572.

Przymusinski, T. C. (1988). On the declarative semantics of deductive databases and logic programs. In *Foundations of deductive databases and logic programming* (pp. 193–216). Morgan Kaufmann.

Raedt, L. D., & Kimmig, A. (2015). Probabilistic (logic) programming concepts. *Mach. Learn.*, *100*(1), 5–47.

Reiter, R. (1977). On closed world data bases. *Logic and Data Bases*, 55–76.

Riguzzi, F. (2023). *Foundations of probabilistic logic programming: Languages, semantics, inference and learning, 2nd edition*. River Publishers.

Riguzzi, F., & Swift, T. (2013). Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory Pract. Log. Program.*, *13*(2), 279–302.

Riguzzi, F., & Swift, T. (2018). A survey of probabilistic logic programming. In *Declarative logic programming: Theory, systems, and applications* (pp. 185–228). ACM an Morgan & Claypool.

Ronca, A., Kaminski, M., Grau, B. C., & Horrocks, I. (2022). The delay and window size problems in rule-based stream reasoning. *Artif. Intell.*, *306*, 103668.

Rondogiannis, P. (2001). Stratified negation in temporal logic programming and the cycle-sum test. *Theor. Comput. Sci.*, *254*(1-2), 663–676.

Ryzhyk, L., & Budiu, M. (2019). Differential datalog. *Datalog 2.0 Workshop in LPNMR*, *2368*, 56–67.

Sakama, C., Inoue, K., & Sato, T. (2021). Logic programming in tensor spaces. *Ann. Math. Artif. Intell.*, *89*(12), 1133–1153.

Santipantakis, G. M., Vlachou, A., Doulkeridis, C., Artikis, A., Kontopoulos, I., & Vouros, G. A. (2018). A stream reasoning system for maritime monitoring. *TIME*, *120*, 20:1–20:17.

Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. *ICLP*, 715–729.

Sato, T. (2008). A glimpse of symbolic-statistical modeling by prism. *J. Intell. Inf. Syst.*, *31*, 161–176.

Sato, T. (2017a). Embedding tarskian semantics in vector spaces. *AAAI Workshop*, *WS-17*.

Sato, T. (2017b). A linear algebraic approach to datalog evaluation. *Theory Pract. Log. Program.*, *17*(3), 244–265.

Selman, J., Amer, M. R., Fern, A., & Todorovic, S. (2011). PEL-CNF: Probabilistic event logic conjunctive normal form for video interpretation. *ICCV Workshops*, 680–687.

Sergot, M. J. (2001). A computational theory of normative positions. *ACM Trans. Comput. Log.*, *2*(4), 581–622.

Shahid, N. S., O'Keeffe, D., & Stathis, K. (2021). Game-theoretic simulations with cognitive agents. *ICTAI*, 1300–1305.

Shahid, N. S., O'Keeffe, D., & Stathis, K. (2023). A knowledge representation framework for evolutionary simulations with cognitive agents. *ICTAI*, 361–368.

Shanahan, M. (1999). The event calculus explained. In *Artificial intelligence today* (pp. 409–430). Springer.

Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., & Zaniolo, C. (2016). Big data analytics with datalog queries on spark. *SIGMOD Conference*, 1135–1149.

Shterionov, D. S., Renkens, J., Vlasselaer, J., Kimmig, A., Meert, W., & Janssens, G. (2014). The most probable explanation for probabilistic logic programs with annotated disjunctions. *ILP*, *9046*, 139–153.

Singh, T., & Vishwakarma, D. K. (2019). Video benchmarks of human action datasets: A review. *Artif. Intell. Rev.*, *52*(2), 1107–1154.

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

Siskind, J. M. (2001). Grounding the lexical semantics of verbs in visual perception using force dynamics and event logic. *J. Artif. Intell. Res.*, *15*, 31–90.

Skarlatidis, A., Artikis, A., Filipou, J., & Paliouras, G. (2015a). A probabilistic logic programming event calculus. *Theory Pract. Log. Program.*, *15*(2), 213–245.

Skarlatidis, A., Paliouras, G., Artikis, A., & Vouros, G. A. (2015b). Probabilistic event calculus for event recognition. *ACM Trans. Comput. Log.*, *16*(2).

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

Sugiura, K., & Ishikawa, Y. (2019). Regular expression pattern matching with sliding windows over probabilistic event streams. *IEEE BigComp*, 1–8.

Sugiura, K., & Ishikawa, Y. (2020). Multiple regular expression pattern monitoring over probabilistic event streams. *IEICE Trans. Inf. Syst.*, *103-D*(5), 982–991.

Terry, D. B., Goldberg, D., Nichols, D. A., & Oki, B. M. (1992). Continuous queries over append-only databases. *SIGMOD*, 321–330.

Thomas, R. (1991). Regulatory networks seen as asynchronous automata: A logical description. *Journal of Theoretical Biology*, *153*(1), 1–23.

Thomas, R., & d'Ari, R. (1990). *Biological feedback*. CRC Press.

Thon, I., Landwehr, N., & De Raedt, L. (2011). Stochastic relational processes: Efficient inference and applications. *Mach. Learn.*, *82*(2), 239–272.

Tiger, M., & Heintz, F. (2020). Incremental reasoning in probabilistic signal temporal logic. *Int. J. Approx. Reason.*, *119*, 325–352.

Tsilionis, E., Artikis, A., & Paliouras, G. (2022). Incremental event calculus for run-time reasoning. *J. Artif. Intell. Res.*, *73*, 967–1023.

Tsilionis, E., Artikis, A., & Paliouras, G. (2024). A tensor-based formalization of the event calculus. *IJCAI*. https://cer.iit.demokritos.gr/publications/papers/2024/tensor-EC.pdf

Valiant, L. G. (1979). The complexity of enumeration and reliability problems. *SIAM J. Comput.*, *8*(3), 410–421.

van der Heijden, M., & Lucas, P. J. F. (2013). Describing disease processes using a probabilistic logic of qualitative time. *Artif. Intell. Medicine*, *59*(3), 143–155.

van Gelder, A., Ross, K. A., & Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *J. ACM*, *38*(3), 620–650.

Vennekens, J., Denecker, M., & Bruynooghe, M. (2009). CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory Pract. Log. Program.*, *9*(3), 245–308.

Vennekens, J., Verbaeten, S., & Bruynooghe, M. (2004). Logic programs with annotated disjunctions. *ICLP*, *3132*, 431–445.

Verwiebe, J., Grulich, P. M., Traub, J., & Markl, V. (2023). Survey of window types for aggregation in stream processing systems. *VLDB J.*, *32*(5), 985–1011.

Vidal, G. (2022). Explanations as programs in probabilistic logic programming. *FLOPS*, *13215*, 205–223.

Vidal, G. (2024). Explaining explanations in probabilistic logic programming. *CoRR*, *abs/2401.17045*.

Vilamala, M. R., Xing, T., Taylor, H., Garcia, L., Srivastava, M., Kaplan, L. M., Preece, A. D., Kimmig, A., & Cerutti, F. (2023). Deepprobcep: A neuro-symbolic approach for complex event processing in adversarial settings. *Expert Syst. Appl.*, *215*, 119376.

Walega, P. A., Kaminski, M., & Grau, B. C. (2019). Reasoning over streaming data in metric temporal datalog. *AAAI*, 3092–3099.

Walega, P. A., Kaminski, M., Wang, D., & Grau, B. C. (2023). Stream reasoning with datalogmtl. *J. Web Semant.*, *76*, 100776.

Wan, H. (2009). Belief logic programming with cyclic dependencies. *RR*, *5837*, 150–165.

White, W. M., Riedewald, M., Gehrke, J., & Demers, A. J. (2007). What is ”next” in event processing? *PODS*, 263–272.

Wooldridge, M. J. (2009). *An introduction to multiagent systems, second edition*. Wiley.

Wu, E., Diao, Y., & Rizvi, S. (2006). High-performance complex event processing over streams. *SIGMOD*, 407–418.

Wu, J., Wang, J., & Zaniolo, C. (2022). Optimizing parallel recursive datalog evaluation on multicore machines. *SIGMOD Conference*, 1433–1446.

Xing, T., Garcia, L., Vilamala, M. R., Cerutti, F., Kaplan, L., Preece, A., & Srivastava, M. (2020). Neuroplex: Learning to detect complex events in sensor networks through knowledge injection. *SenSys*, 489–502.

Yolum, P., & Singh, M. P. (2002). Flexible protocol specification and execution: Applying event calculus planning using commitments. *AAMAS*, 527–534.

Yu, D., Yang, B., Liu, D., Wang, H., & Pan, S. (2023). A survey on neural-symbolic learning systems. *Neural Networks*, *166*, 105–126.

Zaniolo, C. (2012). Logical foundations of continuous query languages for data streams. *Datalog in Academia and Industry*.

Zervoudakis, P., Kondylakis, H., Spyratos, N., & Plexousakis, D. (2021). Query rewriting for incremental continuous query evaluation in HIFUN. *Algorithms*, *14*(5), 149.

Zhang, H., Diao, Y., & Immerman, N. (2013). Recognizing patterns in streams with imprecise timestamps. *Inf. Syst.*, *38*(8), 1187–1211.

Zhang, H., Diao, Y., & Immerman, N. (2014). On complexity and optimization of expensive queries in complex event processing. *SIGMOD Conference*, 217–228.

Zhao, B., van der Aa, H., Nguyen, T. T., Nguyen, Q. V. H., & Weidlich, M. (2021). EIRES: efficient integration of remote data in event stream processing. *SIGMOD*, 2128–2141.

Zielinski, B. (2023). Explanatory denotational semantics for complex event patterns. *Formal Aspects Comput.*, *35*(4), 23:1–23:37.

Zocholl, M., Iphar, C., Pitsikalis, M., Jousselme, A., Artikis, A., & Ray, C. (2019). Evaluation of maritime event detection against missing data. *QUATIC*, *1010*, 275–288.