A Distributed Logic Reactive Programming Model

and its Application to Monitoring Security

Thierry Renaux

Dissertation submitted in fulfillment of the
requirement for the degree of Doctor of Sciences

March 28, 2019

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 9789493079205
NUR 989

Abstract

Processes in the world are increasingly managed by software systems that are centered around the notion of *events*. For instance, banking software responds to the occurrence of financial transaction *events* and parcel services respond to the arrival *event* of a package at a depot. To, e.g., identify fraud in a series of financial transactions, multiple transaction events must be correlated.

How to efficiently extract patterns from a stream of events — in an online fashion — is an active research topic. The state of the art enables extracting complex patterns from large streams, or performing simple processing with predictable latency. There are, however, no techniques which combine expressive event correlation mechanisms with guaranteed upper bounds on resource usage. Lacking such guarantees, event monitoring systems risk falling behind on the live data, or even crashing due to memory exhaustion. Preventing this is of utmost importance for modern systems that are online 24/7.

In this dissertation, we propose a novel programming paradigm for large-scale online event correlation: Logic Reactive Programming. LRP overcomes the issues in the state of the art by imposing maximum *lifetimes* on stored event data. Through *temporal reasoning*, LRP then guarantees that a fixed upper limit exists on the number of events whose data needs to be retained. *Stale* data is automatically discarded, thus enabling LRP systems to operate in a fixed resource budget. We introduce PARTElang, the first Logic Reactive Programming language. Its formal foundations are defined by means of an *event algebra*. We define Featherweight PARTE, an operational semantics for PARTElang, which guarantees that every operation evaluates in constant time, using constant space. We prove correctness of Featherweight PARTE, i.e., that steps defined by the formal model ensure that a PARTElang program is translated to a number of concurrent units of computation which jointly arrive at the results prescribed by the event algebra. We further validate PARTElang by building a security monitoring application on top of a prototypical implementation of the Featherweight PARTE model.

Samenvatting

Steeds vaker worden activiteiten in de buitenwereld beheerd door softwaresystemen die werken op basis van *events*. Zo reageren banksystemen bijvoorbeeld op het plaatsvinden van een transactie-*event*. De systemen van pakjesdiensten voeren acties uit wanneer een aankomst-*event* van een pakje zich bij een depot voordoet. Dergelijke events bevatten op zich reeds nuttige informatie, maar er kan nog meer informatie onttrokken worden uit een correlatie van meerdere events, bv. door fraude te detecteren in een reeks financiële transacties.

Dit alles zorgt ervoor dat er nood is aan middelen om grote stromen van events te monitoren op het voorkomen van patronen van events. Omdat het vaak nuttig is om snel te reageren, vindt deze monitoring best live plaats — terwijl de events plaatsvinden. De ontwikkeling van technieken om grootschalige eventstromen live te analyseren, is de focus van meerdere onderzoeksdomeinen. Er ontbreekt echter een oplossing die kan garanderen dat de grootschalige live analyse uitgevoerd kan worden met een op voorhand bepaalde hoeveelheid middelen. Zonder die garantie riskeren eventmonitoringsystemen om achterop te geraken ten opzichte van de live data, of zelfs om hun taak te moeten staken bij gebrek aan werkgeheugen. Voor systemen die continu beschikbaar moeten blijven, is dit onaanvaardbaar.

In deze doctoraatsverhandeling stellen we een nieuw programmeerparadigma voor voor grootschalige live eventcorrelatie: Logisch-Reactief Programmeren. LRP overkomt de problemen in huidige oplossingen door een garandeerbare bovengrens te vereisen op de levensduur van eventdata. Aan de hand van *temporal reasoning* garandeert LRP dat een vaste bovengrens bestaat op het aantal events dat bijgehouden moet worden. Oude events worden automatisch verwijderd. Dit laat LRP systemen toe om binnen een eindig tijds- en geheugenbudget te werken. We introduceren PARTElang, de eerste LRP taal. De formele onderbouwing van LRP berust op een *event algebra*. We definiëren een operationele semantiek voor PARTElang. Deze garandeert dat iedere operatie in constante tijd uitgevoerd wordt, gebruikmakend van een constante hoeveelheid geheugen. We bewijzen dat het evaluatiemodel het correcte gedrag voorschrijft, en demonstreren dit aan de hand van een prototype.

Acknowledgments

First of all, I would like to thank the members of my jury (Viviane Jonckers, Katrien Beuls, Tias Guns, Coen De Roover, Sebastian Erdweg, and Robert Hirschfeld) for their insightful comments and feedback.

I would like to thank my promotors, Wolf and Joeri, for their guidance and feedback. Their advice, encouragement, and occasional constructive criticism were essential throughout my PhD.

Of course, the rest of the Software Languages Lab was also a great bonus. Thank you all, both for offering a good environment to do work in, and for being great colleagues. I won't list you all, but know that I am thankful for the work environment you helped create. So, thanks to the Rete-and-event-based-things people: Kennedy, Humberto, Sam, etc. Thanks to those of you who arrived at SOFT around the same time as me, and therefore were often going through the same thing at the same time as me: Janwillem, Nathalie, Simon, etc. Thanks to those who at some point were doing the day-to-day guidance for the part of SOFT I was in at that time: Stefan, Lode, Yves, etc.

Outside of SOFT, I'd also like to thank my family and friends. Thanks Ansje, for being there for me through all these years. Thanks, mom and Evelyne too, for all the support you provided. Thanks, Raf, for your friendship and for proofreading this entire dissertation, even though you had no obligation to do so at all.

Finally, thanks, reader, for reading this. Why *are* you reading this? Are you waiting for my public defense to start? (Hello! Wave!) Or are you reading it during the presentation? Aww, I spent all that effort making it interesting... Or are you looking for inspiration while writing your own Acknowledgments section? Did you pass your private defense yet? Congratulations! Or are you just that well-prepared that you're writing it proactively? Wow, keep up the good work! In any case, thanks, and I hope the work described in this dissertation is useful to some of you.

This work is partially funded by a PhD scholarship of the Agency for Innovation by Science and Technology in Flanders (IWT).

Contents

List Of Definitions

1
Introduction

Increasingly, processes in the world are managed by software systems. Financial transactions are executed by autonomous software systems. Access to buildings and rooms is managed by smart locks. Parcels passing through distributions centers are scanned and automatically routed. Evidently, electronic communication is handled by software systems too, as are web applications. Developments such as Smart Cities or the Internet of Things lead to widespread deployment of sensors throughout the real world.

All these systems necessarily create some representation of the elements they are managing, and of what happens to them. The latter are commonly referred to as *events*. *Events* are digital records of things that occurred in the world surrounding some computational system. For a system managing finances, the fact that, e.g., 50 € was transferred from some account to another at five past noon today, is an event. For a smart lock, the fact that, e.g., the front door was unlocked at eight o'clock last Sunday, is an event. For a web application, the fact that, e.g., a user stopped scrolling down while a certain image was displayed, is an event.

The information carried by events is often valuable: finances need auditing, smart locks benefit from storing access logs, parcel services benefit from having a trace of their parcels' whereabouts, etc. Even when events are not critical to the execution of the base tasks, storing events can be invaluable. For this reason, many web applications store click-streams, i.e., a log of all interaction events a user executed on their platform, for analysis purposes.

The process of gathering and storing events is only part of the solution. A complete solution includes the means of understanding the relation between events. While events carry meaning in isolation, a better understanding of the world can often be reached by correlating multiple data points in an event stream. For instance, detecting fraudulent diffusion of crime money requires linking multiple transactions. Detecting where a parcel has disappeared from the pipeline requires correlating the point at which it was last registered with the first point at which it was supposed to appear, but never did. Measuring user engagement in a web application requires multiple data points from multiple users. Such combinations of multiple events are commonly referred to as *complex events* [94]. Stakeholders' desire to monitor their data streams for occurrences of complex events, has led to the need for software technologies by which complex events can efficiently be detected.

At the same time, modern software systems increasingly need to provide immediate feedback. Users expect web applications to be interactive. Companies increasingly expect their event monitoring to be reactive too. Parcel services do not want to be limited to the ability to investigate issues after the facts, they want their systems to autonomously inform them of issues as soon as possible. Users deploying smart locks expect anomalies to be signaled as soon as possible. Financial services do not want to wait for the results of an overnight batch processing of yesterday's transactions. Every second gained increases the chance that malicious actions can successfully be reverted. In application domains like security monitoring or fraud detection, this feature is not just "nice to have", but a hard requirement.

There hence exists a need for software systems that reliably and autonomously monitor large, live streams of event data for complex events, and that autonomously enact reactions to those complex events. The act of detecting and reacting to complex events is referred to as *Complex Event Processing* (CEP).

Over the last decade, much effort has been put into building systems which can process vast amounts of data. These systems are commonly referred to as "Big Data" processing systems. A large portion of the Big Data processing systems is aimed at *batch processing*. In batch processing, groups of data points are gathered into a *batch*, which is then jointly processed. For instance, an entire day's worth of financial transactions can be gathered, to be executed overnight. As discussed, many use cases cannot be properly served by batch processing systems, as such systems inherently have a high latency. Instead of *analyzing* the data in batches, such use cases require *real-time monitoring* of event streams, i.e., processes events incrementally as they unfold. Such *stream processing* systems may offer lower latency, but typically do so at the cost of lower throughput. Furthermore, while batch processing systems trivially are able to use data from all data points in a batch, stream processing systems typically only offer limited means of referring to earlier data points while processing a new event.

In this dissertation, we argue there is one use case where the state of the art of Big Data processing falls short: **reliably correlating events across and within large streams of events, in a streaming fashion**. For instance, the use case of financial fraud monitoring

	Distributed Big Data & Stream Processing		Event Handling	
	Big Data Stream Processing	**Streaming Databases**	**Complex Event Processing**	**Reactive Programming**
Throughput	✓✓ very high	✓ high	~ average to low	~ average to low
Cost per update	✓ constant (depth of program)	– proportionate to history	– proportionate to history	✓ constant (depth of program)
Bounds on resource usage	✓ yes (with basic operators)	– proportionate to history	– proportionate to history	~ possible, assuming no loops etc. in program
History	– none	✓✓large history size	✓ some history	– none
Pattern matching	– unsupported	✓ supported (relational join)	✓✓ expressive pattern language	– unsupported
Reaction logic	– often integrated, but Callback Hell	– outside of the model	– outside of the model, or Callback Hell	✓✓ well-supported (core of the paradigm)

Table 1.1: Summary of the state of the art of event stream processing

needs support for *a.)* responding to occurrences of financial transactions; *b.)* filtering financial transactions based on their attributes, e.g., selecting only transactions above a certain valuation; *c.)* joining financial transactions based on their attributes, e.g., joining all transactions whose source account matches another set of transactions' destination account; *d.)* detecting the absence of transactions matching some pattern; *e.)* performing well enough to handle an entire financial institution's stream of transactions; and to *f.)* provide the guarantee that the system can keep up with data as it comes in, i.e., that it will not detect fraud too late, let alone miss fraudulent transactions because the system was overloaded or had crashed due to resource exhaustion.

Implementing software systems for this task is complicated, as existing software solutions do not offer the required guarantees, or fail to concisely and unambiguously express the patterns to detect.

We discuss in chapters 2 and 3 how existing batch processing systems fail to meet the stringent latency requirements, while most streaming systems either do not offer ways of correlating events, or do so without guarantees on the time and space it takes them. Such streaming systems can thus not guarantee that they will be able to keep up with the arrival of new events, nor that they will not run out of memory. These shortcomings make them inherently irreconcilable with use cases such as security monitoring: security systems can not afford to fall behind, or to cease monitoring.

On the other hand, existing stream processing systems which guarantee bounded resource usage are restricted to very constraining language models. For instance, some of those systems expose a complex event matching strategy based on regular expressions. This paradigm does not lend itself very well for use cases such as monitoring financial transactions: fraud analysts are rarely interested in comparing a transaction to specifically the very next transaction that happened to be submitted to their institution, or in uninterrupted sequences of transactions. We discuss in chapter 3 how none of the existing systems offer a clean way of expressing complex event patterns while guaranteeing constant time processing in constant space.

We summarize the strengths and weaknesses of state of the art in table 1.1.

1.1 Problem Statement

We explained the need for software systems that can detect complex patterns in continuous streams of events, live as the events are occurring. For some application domains — such as security monitoring — the complex event detection system must guarantee it is able to keep up with the input stream, and to autonomously enact reactions to occurrences of the complex events. Given a set of complex event patterns to detect, and an upper bound on the rate at which events can be delivered via the input stream, a monitoring system must be able to determine its maximum resource usage. Within the bounds of that resource usage, the system must then guarantee that reactions to the complex event patterns are enacted, and that they are enacted in a timely manner. We define this guaranteed ability to react in a timely fashion to an input stream with a bounded maximum rate as *strong reactivity* (page 65).

Given this context, we formulate the problem statement as follows:

Problem Statement

There is a need for platforms which autonomously process complex events in high-throughput streams of events, incrementally in a data-driven, always-listening, *strongly reactive* manner.

1.2 Research Goal

The goal of this dissertation is to develop a programming paradigm that captures the concerns of strongly reactive Complex Event Processing over high-throughput streams of events. Programming languages implementing this paradigm should

- facilitate Complex Event Processing by offering the abstractions to reason about combinations of events; and
- guarantee that programs written in them are strongly reactive, i.e., that finite upper bounds are known on the rate at which primitive events may occur, and that execution of the program in turn is guaranteed to never fall behind on the input streams, nor consume more than some predictable, finite amount of memory. In general, strong reactivity precludes the possibility of the program crashing due to resource exhaustion.

Many forms of event stream processing exist. It would needlessly broaden the scope of this dissertation to discuss them all in detail. Informed by a number of driver scenarios from the application domain of security monitoring, we investigate only *streaming pattern matching*. Pattern matching is the act of finding sets of events which jointly satisfy the conditions of some pattern or template.

To disambiguate what we mean by that we consider the defining properties of this approach to be that *1.)* the patterns to match are decided up front (in contrast to, e.g., *outlier detection*); that *2.)* the patterns to match are explicitly defined from expert knowledge (in contrast to, e.g., machine learning techniques); and that *3.)* the patterns may express a correlation (in contrast to, e.g., simply filtering events, or computing aggregates such as running averages). Systems which do not conform to these properties are out of scope for this dissertation. Due to their historic importance we discuss some systems of the third category in our discussion of the state of the art (chapters 2 and 3).

1.3 Approach

In this dissertation, we propose a novel programming paradigm which tackles the problem stated in section 1.1. We call this new paradigm Logic Reactive Programming (LRP). As the name implies, Logic Reactive Programming is a logic programming paradigm. In LRP, complex event patterns can be specified using a restricted logic programming language.

Consider the code snippet shown in listing 1.1.

```
rule FraudPassThrough where
  incoming : MoneyTransferred {
    amount >= 100.0,
    uncommon_route(originator, destination)
  },
  outgoing : MoneyTransferred {
    uncommon_route(originator, destination),
    originator = incoming.destination,
    amount = incoming.amount
  }
when
  outgoing in incoming [0 days, 14 days]
then
  emit SuspiciousSequenceDetected
    at incoming.timestamp
    with {
      mule_account          = incoming.destination,
      amount                = incoming.amount,
      incoming_transactions = list(incoming.id),
      outgoing_transactions = list(outgoing.id)
    }
```

Listing 1.1: A first look at LRP: a PARTElang code snippet expressing a pattern consisting of a suspicious incoming and a suspicious outgoing transaction.

Of course, the concrete semantics of this snippet will only be clear once the language is explained in chapter 4. Skimming over the details, though, one can see that this snippet defines a rule, called “FraudPassThrough”, which has a `where`-clause specifying constraints on an incoming and an outgoing “MoneyTransferred” event, a `when`-clause specifying how far apart in time both events may take place, and a `then`-clause specifying that a “SuspiciousSequenceDetected” event must be emitted.

To enforce strong reactivity, the language requires that the `when`-clause imposes closed temporal constraints between every pair of events in a pattern: events must occur some fixed, finite amount of time before or after each other. As such, an upper bound can statically be determined on the duration during which events remains relevant.

Closed temporal relations between events can specify, e.g., that a certain event must occur between half a second before, and one second after another event. Open temporal constraints, such as requiring that three events occur in sequence, but arbitrarily far apart in time, cannot be expressed in LRP languages, as the duration during which the events' data must be retained is then unbounded. This is compatible with our use case in security monitoring: while having the ability to match events occurring arbitrarily far apart from each other is nice to have, it is not a hard requirement.

The high-level, declarative nature of the LRP paradigm serves two purposes:

- First, it allows us to hide irrelevant implementation details from the LRP programmer. Details such as event storage, retrieval of stored events, or purging of outdated information are common to all LRP programs. In essence, an LRP program manages a database of recent (complex) events. There is hence some overlap with other languages used for querying and updating databases.

  Consider the case of SQL. In SQL, queries are expressed in terms of relational algebra. Relational algebra can express the notion of a table of records, of conditions on attributes of those records, and of joins between tables. SQL does not provide an API to some storage of record. Bugs such as out-of-bound reads are prevented by construction, since the relational algebra does not even expose the concept of an addressable storage. Automatic rewriting of relational expressions to equivalent relational expressions enables large-scale performance gains in relational database systems. These properties largely depend on the non-imperative, declarative character of SQL.

  Logic Reactive Programming aims to exploit similar advantages for event processing. By abstracting over the inner workings, it is easier for the LRP programmer to write correct programs. Additionally, the abstractions give the language runtime more options for transparent optimization. This choice is in line with the choices made by related work: most systems for event correlation follow a declarative approach (see chapters 2 and 3).

- The second reason we opted for a declarative approach, has to do with guaranteed reactivity. Compare again with SQL. The maximum size of the (intermediate and final) results of a query written in plain SQL is proportionate to the size of the database. The same holds for the time it may take to enumerate those results. Programs written in general-purpose languages offer a weaker guarantee, as Turing completeness entails that time and space requirements are unbounded.

  Logic Reactive Programming explores the opposite side of the spectrum: the resource usage of an LRP program may not be proportionate to the size of all data that was added to the monitored stream. To remain strongly reactive, LRP programs must execute in constant time. Since enumerating results takes time proportionate to the number of results, the number of (intermediate and final) results generated in response to each event must be constant.

By carefully selecting which operations are allowed in an LRP program, LRP programs by construction have a guaranteed, static upper bound on reaction time. One of the main restrictions imposed to accomplish this, is the absence of arbitrary data structures. The only compound data types which exist in LRP, are events. Events are treated as timestamped records. Events may not store other events as attributes. Individual events cannot even be named by an LRP program, nor can they be explicitly stored somewhere. LRP programs merely express patterns of events. The pattern is instantiated for each combination of events that matches the pattern. The storage location is transparent to the LRP programmer.

As such, the declarative nature of LRP ensures that the memory layout of any given LRP program can be statically determined up front. As a result, **given an LRP program and an upper bound on the *event arrival rate*, one can statically determine whether a certain machine is capable of detecting all complex events occurring in the input streams**. LRP precludes the possibility that a CEP program falls behind on the input stream or crashes due to resource exhaustion.

We develop PARTElang, the first Logic Reactive Programming language. The formal foundations of PARTElang are defined by means of an event algebra. We propose a two-phase model for evaluating programs written in the PARTE language.

- In the first phase, the dependencies outlined in the PARTE program are extracted and compiled into a Rete-like [59] graph. Using the temporal constraints defined in the PARTE program and the maximum event arrival rates for all input streams, a finite minimum viable size is computed for each location where state is stored. These minimum viable sizes are the minimum sizes at which PARTE can guarantee to find all matches to the patterns that occur in the event streams.
- In the second, online phase, events are matched to the constraints in the graph in constant time, using constant space: at most the minimum viable size is needed. The operational semantics of the evaluation model is defined in the form of rewrite-rules on the PARTE graph.

We evaluate our work in two ways:

- First, we develop a prototypical implementation, and show its ability to reliably process a live stream of events. We compare this with the abilities of the state of the art of event stream processing techniques.
- Second, we formally prove that the evaluation model defines *strongly reactive* matching of complex event patterns.

1.4 Contributions

The contributions of this dissertation are the following:

A taxonomy of existing event processing systems with a focus on scalable Big Data stream processing engines and expressive event handling languages.

Logic Reactive Programming, a novel programming paradigm aimed at *strongly reactive* correlation of complex events. Characteristic of Logic Reactive Programming Logic are *1.)* declarative specification of event patterns and reaction logic, *2.)* built-in temporal reasoning, *3.)* automatic management of event data, *4.)* scalable online processing, and *5.)* guaranteed availability.

PARTElang, a first Logic Reactive Programming language which satisfies the five requirements of LRP. We further define an *event algebra*, which serves as the formal foundation for PARTElang. We define how PARTElang programs map onto the event algebra.

Featherweight PARTE, a formal model for evaluation of PARTElang which defines an *operational semantics* of PARTElang. Featherweight PARTE complements the event algebra by specifying which steps must be taken to achieve the behavior specified in the event algebra. Featherweight PARTE fits the Big Data context by not requiring a shared memory space, nor centralized control, instead coordinating multiple nodes through asynchronous message passing. The model defines a global evaluation language which specifies how messages are exchanged by the nodes, and a local evaluation language which specifies how node state is transformed in response to specific messages.

A prototypical PARTElang runtime implementing the Featherweight PARTE model. We evaluate how the prototype compares to the state of the art when implementing our driver scenarios, which we introduce in section 2.1.

1.5 Supporting publications

Parts of this dissertation were published in a number of papers. We summarize them below:

- Thierry Renaux, Lode Hoste, Stefan Marr, and Wolfgang De Meuter. Parallel Gesture Recognition with Soft Real-Time Guarantees. In *Proceedings of the 2nd edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! 2012, pages 35–46. ACM, 10 2012. ISBN 978-1-4503-1630-9. doi: 10.1145/2414639.2414646

This paper introduced the first step towards PARTE: an implementation of the Rete algorithm [59], adapted for event processing, which can make use of the parallelism offered by modern multi-core processors by implementing the Rete graph as concurrent actors. The use case — live gesture recognition — necessitated *soft real-time* guarantees: late responses are suboptimal, but not a hard failure.

- Stefan Marr, Thierry Renaux, Lode Hoste, and Wolfgang De Meuter. Parallel Gesture Recognition with Soft Real-Time Guarantees. *Science of Computer Programming*, 98: 159–183, 2 2015. ISSN 0167-6423. doi: doi:10.1016/j.scico.2014.02.012

  This paper extends the one listed above. We showed that the time requirement of the version of PARTE described in that paper was bounded by the data set size. A set of mostly synthetic benchmarks demonstrated that PARTE could run with predictable time costs, a requirement for *soft real-time* systems.

- Janwillem Swalens, Thierry Renaux, Lode Hoste, Stefan Marr, and Wolfgang De Meuter. Cloud PARTE: Elastic Complex Event Processing based on Mobile Actors. In *Proceedings of the 3rd International Workshop on Programming based on Actors, Agents, and Decentralized Control*, AGERE! 2013, pages 1–10. ACM, 10 2013. ISBN 978-1-4503-2602-5

  This paper demonstrates how PARTE can make use of the compute power of multiple networked computers. We split up computation on the actor-level granularity introduced in PARTE, and schedule different nodes on different machines. We empirically validate the possibility of balancing a workload over multiple networked computers, based on the number of messages in the inboxes of the nodes.

A number of our papers touch upon topics related to this dissertation, notably the implementation of rule-based languages to provide complex event processing capabilities:

- Kennedy Kondo Kambona, Thierry Renaux, and Wolfgang De Meuter. Reentrancy and Scoping for Multitenant Rule Engines. In *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST 2017)*, volume 1, pages 59–70. Scitepress, 2017

  This paper describes a *multi-tenant* rule-based system, i.e., one which hosts the business logic of multiple users. The rule-based system can share some data and work among the users, while other data can be kept private.

- Kennedy Kambona, Thierry Renaux, and Wolfgang De Meuter. Efficient Matching in Heterogeneous Rule Engines. In *Proceedings of the 30th International Conference on Industrial, Engineering, Other Applications of Applied Intelligent Systems (IEA/AIE 2017: Advances in Artificial Intelligence: From Theory to Practice)*, volume 10350 of *Lecture Notes in Computer Science*, pages 394–406. Springer, 6 2017. ISBN 978-3-319-60041-3

  This paper shows how the performance of a multi-tenant system can be improved by asymptotically speeding up the process of joining data from different hierarchical *scopes*.

- Kennedy Kambona, Thierry Renaux, and Wolfgang De Meuter. Harnessing Community Knowledge in Heterogeneous Rule Engines. In Tim A. Majchrzak, Paolo Traverso, Karl-Heinz Krempels, and Valérie Monfort, editors, *Web Information Systems and Technologies*, volume 322, pages 132–160. Springer International Publishing, 2018. ISBN 978-3-319-93527-0. doi: 10.1007/978-3-319-93527-0_7

  This paper shows how the efficient matching in multi-tenant rule engines makes it possible to harness *community knowledge* by combining the views that individual users have.

- Thierry Renaux, Lode Hoste, Christophe Scholliers, and Wolfgang De Meuter. Software Engineering Principles in the Midas Gesture Specification Language. In *Proceedings the 2nd International Workshop on Programming for Mobile and Touch, Portland*, PRoMoTo 2014, pages 9–16, Portland, Oregon, USA, 10 2014. ACM. ISBN 978-1-4503-2295-9

  This paper explores applying software engineering principles to rule-based languages. Central are the ideas of improved means of abstraction and reuse.

1.6 Outline of the Dissertation

The next two chapters sketch the state of the art in distributed Big Data and stream processing (chapter 2), and event handling (chapter 3). Chapter 4 introduces the novel Logic Reactive Programming paradigm, and proposes a first LRP programming language, PARTElang, as well as an event algebra for PARTElang programs. In chapter 5 we define an *operational semantics* for PARTElang programs, in the form of the Featherweight PARTE model.

In chapter 6 we describe our prototypical implementations of the Featherweight PARTE model: PARTE$^{\text{Rust}}$ and PARTE$^{\text{Elixir}}$. We show how our research artifacts can be used to implement a real-world use case in chapter 7.

Finally, the validity of claims from chapter 5 is established by formal proofs in chapter 8.

2 State of the Art in Distributed Big Data and Stream Processing

We stated in section 1.1 that there is a need for software systems which autonomously process complex events in high-throughput streams of events. In this chapter we discuss the evolution of large-scale data processing systems. We argue that two concrete historical turning points lead us to the current state of the art. The first turning point is the inception of relational databases. The second turning point is the point at which distributed processing became mainstream, i.e., the "*Big Data*" trend. Both relational databases and Big Data processing systems have evolved to support incremental, streaming processing of event data. This chapter outlines how that evolution took place, and even became confluent across both domains.

The chapter is structured as follows: we first introduce three driver scenarios from the domain of fraud detection for financial transactions. We then sketch the relevant parts of the history of Big Data stream processing. We identify three trends in the state of the art: the **introduction of time windows**, the **introduction of distributed computation**, and the **shift in programming language paradigm** used.

Afterwards, we discuss how the state of the art in Big Data stream processing the art fails to express two key concerns of our driver scenarios: **correlating events** and guaranteeing **timely processing** of data.

2.1 Driver Scenarios

We stated in chapter 1 that the research in this dissertation targets software tools used to monitor streams of events for occurrences of event patterns. The event monitoring should additionally satisfy the following constraints:

Declarative: event patterns should be declarative specifications of *complex events*, i.e., events which potentially consist of multiple sub-events; and

Reliably real-time: the monitoring should take place in real-time, detecting matches to the patterns as soon as the pattern occurred.

Many existing systems ostensibly tackle a similar problem. We describe the most relevant systems in section 2.2 and explain in section 2.4 why those systems do not suffice for tackling our use cases. To make the issues concrete, we now introduce a set of driver scenarios. These scenarios will be used throughout this dissertation to compare different approaches, and to explain new concepts as they are introduced.

2.1.1 Setting

Consider the domain of fraud detection in financial transaction processing. To detect fraudulent transactions, financial institutions track all transfers of money which pass through their system. The data available to the financial institutions is limited to — on the one hand — static information about accounts held at their own institution, and — on the other hand — a live feed of the money transfers from, to, or between local accounts.

The occurrence of a money transfer constitutes an event. For instance, if Alice sends 50 € to Bob on January 1st, 2018 at 10:05, the system registers the occurrence of an event that "money was transferred" with a timestamp of "2018-01-01 10:05", and attributes identifying the originator of the event as Alice, the destination as Bob, and the amount as 50 €.

In this setting, we focus on suspicious behavior that indicates possible cases of two types of fraud:

Money muling: an account is used to forward crime money (scenario A) or to diffuse crime money (scenario B).

Refund scams: an account holder is tricked into reimbursing a supposedly erroneous transaction, but the erroneous transaction is additionally claimed as erroneous with the originating bank, causing the transaction to be reverted. The sender hence gets reimbursed twice (scenario C).

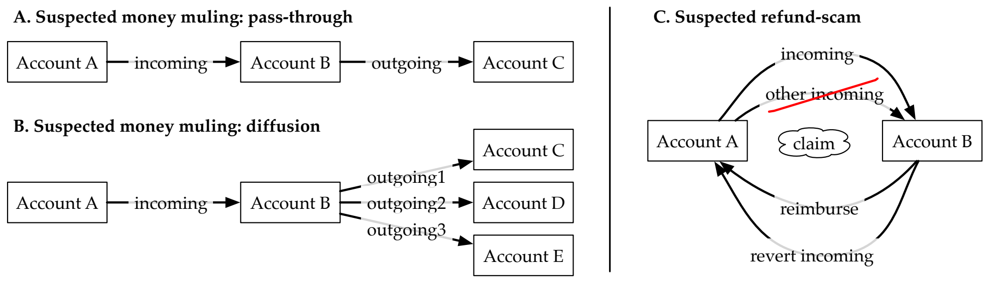


Figure 2.1: Schematic depiction of the driver scenarios — Rectangles represent accounts. Arrows represent money transfers between the accounts. A “claim” cloud represents the occurrence of a claim that a transaction was erroneous. Time progresses from left to right for scenarios A and B, and from top to bottom for scenario C.

2.1.2 Scenarios

A schematic depiction of the driver scenarios is provided in figure 2.1.

Scenario A: Suspected Money Muling: Detecting Pass-Through of Money

The first driver scenario is quite simple, and will be used to demonstrate basic concerns, such as basic syntax and concepts. Driver scenario A concerns the detection of an account possibly used as a money mule in the most simple way: the account receives a sum of money from an account it normally does not receive money from, and forwards it to a third account it normally does not send money to. In driver scenario A, the amount of money sent must be exactly the same as the amount of money received. Furthermore, the amount must be at least 100 €; individual transactions involving less money are ignored. To enable monitoring within fixed resource bounds, it is impossible to inspect the entire history of financial transactions. Therefore, a limited time span of 2 weeks is inspected. Scenario A only concerns cases of pass-through which conclude within 2 weeks.

Scenario B: Suspected Money Muling: Detecting Diffusion of Money

The second driver scenario extends the first one, by detecting a three-way split through a possible mule account. Again, the originator account is suspicious if it is an account that does not normally transact with the suspected mule account, as are the destination accounts. However, this time there are three destination accounts, and it is the sum of the value of all outgoing transactions that has to match the incoming transaction. Additionally, a discrepancy of up to 10% between incoming and outgoing amounts will be allowed to make the detection more resilient to obfuscation, and to account for possible included payment of the mule. The incoming transaction must again amount to at least 100 €. Again, a time window of up to 2 weeks is considered.

Scenario C: Detecting Refund Scam

The third driver scenario differs from the first, in that this case tries to detect a victim instead of an accomplice of fraud. Again, suspicious accounts are defined as accounts which do not normally wire money to each other. The time window for the sequence of events is again set to 2 weeks. The pattern of interactions is as follows: *1.)* an account receives a deposit from some originator, *2.)* the account sends the same amount, with a 10% margin, back to the originator, *3.)* the originator falsely claims the first transaction was in error, thereby recuperating the money, but keeps the money sent in step 2, and *4.)* there was no other transaction similar to the one in step 1, to which the refund in step 3 might apply.

Note that this scenario introduces the notion of **absence of events**: the last condition holds when no matching transaction is detected.

2.1.3 Conclusion

In all three scenarios, detection has to happen *live*, as the data comes in. The amount of data to handle can grow large, as both the rate at which new events can occur is large, as well as the time frame during which the events need to be retained. It is an operational requirement that the system is constantly online and keeping up with new events.

When these requirements present themselves, a reasonable first choice seems to be to turn to database management systems. Relational databases are a staple for warehousing and processing large amounts of structured data, providing high availability. Alternatively, distributed stream processing systems have shown to be able to handle "Big Data" with strong guarantees on availability. In the remainder of this chapter we sketch the state of the art in Big Data stream processing, and discuss why it falls short in properly implementing the driver scenarios. Similarly, chapter 3 sketches the state of the art of Event Handling, and discusses why existing event handling solutions fall short in properly implementing the driver scenarios.

2.2 History of Big Data Stream Processing

2.2.1 Origins

The need to store data has been part of computer science since the beginning. Multiple approaches have been devised, from low-level filesystems to domain-specific, tailored databases. For large amounts of structured data which can be made to fit a *schema*, relational database management systems have been in use for about half a century.

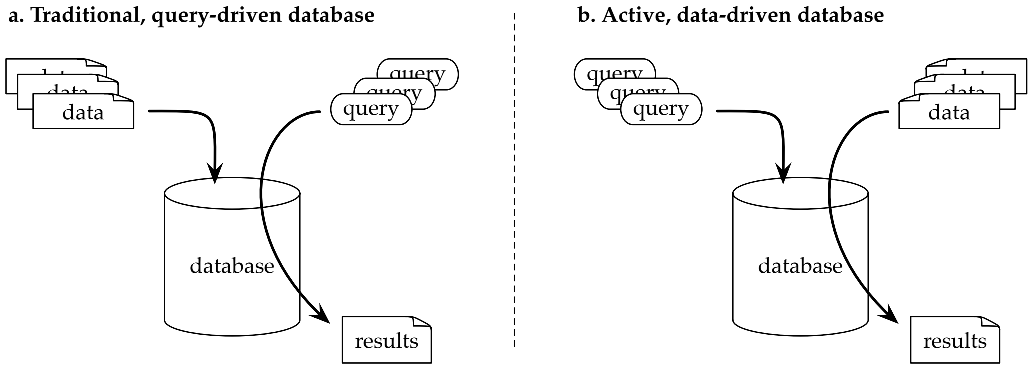


Figure 2.2: Traditional vs. active databases — In traditional databases, data is stored and the introduction of a query causes results to be computed. In an active database, queries are stored and the introduction of data causes results to be computed.

The need to handle data immediately, as new data arrives, gave rise to *active databases*. The seminal work by McCarthy and Dayal [99] summarize that an active database "is one which automatically executes specified actions when [specified] conditions arise". The difference is depicted schematically in figure 2.2. In essence, traditional relational databases store "passive" data as a bounded, unordered set of tuples, organized by abstractions based on relational algebra. Relational algebra can be used to optimize and plan queries over this data. Active databases move away from this notion of random access reads and updates, favoring instead *continuous queries* [134]. Typically, in active databases the order in which the tuples arrived matters. Furthermore, since the size of the database is unknown when the query is formulated, the utility of query planning is reduced. Some stream processing systems experimented with moving away from SQL — and even from relational operators in general — as the fit between the query language and the features required of — and offered by — the streaming active database system weakened. New constructs for structuring data were introduced instead, chiefly among them temporal *windowing*, i.e., selections of data points based on their timestamps. Simultaneously, constructs for splitting up storage and processing (such as database *sharding*) and fault-tolerance were modified to perform better in the novel context.

Data storage systems evolved to support incremental data processing. Concurrently, data processing systems evolved to support larger amounts of data. In symmetry with our discussion on the origins of active databases, we do not aim to provide an exhaustive survey of the origin and evolution of stream processing systems. We merely aim to sketch the context of the work described further in this thesis. As such, we skip the early history, including formalisms like Petri nets, Communicating Sequential Processes and Synchronous Dataflow languages, and applications in the domains of audio and video processing, or digital signal processing. We start instead at an early example of stream processing systems: Tribeca [131].

2.2.2 Early Stream Processing: Tribeca

Tribeca is a stream processing system tailored at network traffic stream analysis. Sullivan [131] describes that his system is a better fit for network traffic analysis than conventional relational database management systems because *a.)* "both [their] data and the storage medium are stream-oriented", offering fast sequential access to data; *b.)* "non-clustered indices will not work for traffic data"; *c.)* "a network traffic trace is a sequence of timestamped network protocol headers" whose processing requires "operators like those found in sequence and temporal DBMSs", yet they require "several dataflow operators and pattern matching operators that are not common in sequence databases"; *d.)* the format "means that even otherwise distinct queries [. . .] often share subqueries"; and *e.)* the need to work in a streaming fashion means that "users would rather reformulate an expensive query or drop an expensive query from the mix than overallocate" resource usage, whereas "relational systems run queries as fast as they can but typically do not provide this kind of capacity information".

While the paper by Sullivan focuses specifically on network traffic analysis, the concerns raised there can be generalized to what we consider to be the defining properties of *stream processing*:

Pre-registered processing Stream processing systems enable formulating queries or transformations that will be applied to the entire streams of data.

Temporal constraints Stream processing systems offer operators to express temporal constraints over the data.

Streaming operation Since they operate on data as the data is coming in, stream processing systems need to keep up with the data, which entails that the speed at which data can be processed is linked to the speed at which it is generated. If data arrives faster, the processing system has to be sped up; be it by augmenting its hardware, by improving its implementation, or by reducing its workload.

Since we use Tribeca as the baseline for our discussion of later systems, we describe Tribeca in some more detail.

The basic unit of computation in Tribeca is a network traffic query. Every query has exactly one source stream. Data on this stream can be transformed using three types of operators: qualification (i.e., filters), projection (i.e., transformations), and aggregation (e.g., averages, minima, or maxima). Results of a traffic analysis query are published in the form of one or more result streams. Both source streams and result streams can be fed from tape or file. In addition to considering entire streams at once, parts of streams can be selected into *temporal windows*. Tribeca offers two types of windows: a "fixed" window that gets reset after a fixed interval (commonly called a "tumbling window", see below) and a "moving" window that contains the last events for some fixed length of time (commonly called a "sliding window", see below).

The syntax for Tribeca is fairly straightforward: it consists of lines specifying one of the components listed above. For instance, consider the traffic query in listing 2.1.

```
source_stream a is {file sample1 someTrace}
result_stream b is {file someFile}
stream_pipe some_pipe
stream_window w on a defined by { a.time.interval 0.5 } is fixed
stream_proj {{a.some_field a.some_other_field}} some_pipe
stream_agg { some_pipe.some_field.min } b
```

Listing 2.1: The Tribeca syntax for the example from section 2.2.2.

This code specifies that the query sources its data from some stream `a` sampled of a file named `someTrace`. Results will be generated on some stream `b`, whose values will be stored into the file `someFile`. The attributes `some_field` and `some_other_field` will be read from all data on stream `a`, and pushed into an intermediate data pipe `some_pipe`. The values off `some_pipe` will be aggregated by taking the minimum of the `some_field` attribute, and pushed into stream `b`.

2.2.3 Later Stream Processing Systems

NiagaraCQ

The foundation laid by Tribeca was extended on by what became the field of stream processing. A first milestone after Tribeca was NiagaraCQ [32]. NiagaraCQ is arguably an *active database*, structuring its stream processing as *continuous queries*. Chen et al. [32] clarify the difference between traditional triggers and *continuous queries* by the purpose of both: the purpose of triggers is to maintain data integrity, whereas the purpose of continuous queries is to support repeated, continuous processing of queries. NiagaraCQ also supports queries involving multiple input sources, some of which need not even be present in a database. This distinguishes it from both triggers as well as Tribeca, which — as mentioned above — supports only a single source stream per query.

The NiagaraCQ Command Language offers the means to register named continuous queries, listing a query expressed in XML-QL [48], some reaction code, and optionally a start time and end time, and an interval. These timings specify how often the query is reevaluated. NiagaraCQ is not strictly data-driven; it is not necessarily the arrival of new data that causes reevaluation, but the firing of an internal timer.

The main contributions of NiagaraCQ are improved sharing of subqueries, which leads to higher processing performance on streams, and the use of a fully declarative, SQL-like query language in a streaming context.

Gigascope

Gigascope [39] merges the contributions of Tribeca and NiagaraCQ, by offering an SQL-like query language which reasons over data streams and produces new data streams. Cranor et al. [39] identify a number of issues with the sliding windows in existing continuous query languages. They claim composing windows is not possible in those languages, and result of continuous queries may be difficult to interpret when the inputs were filtered by means of time windows. For these reasons, they opt instead for selecting events using an SQL `GROUPBY` construct.

A translation into Gigascope syntax of the Tribeca query in listing 2.1 could look like what is listed in listing 2.2:

```
SELECT MIN(some_field), some_other_field
FROM SomeInput AS a
GROUP BY time/7200 -- Half-second buckets
```

Listing 2.2: The Gigascope syntax for a query similar to the one in listing 2.1.

In essence, it is an SQL query: Gigascope's GSQL is a subset of SQL. In contrast, the contemporaneous STREAM by Arasu et al. [13] uses a superset of SQL: CQL. CQL extends SQL with constructs for sliding windows, showing that no consensus existed in 2003 on the utility of sliding windows for stream processing.

Aurora and Borealis

We consider Aurora [27] to be the next big step in the evolution of stream processing systems. Aurora improved earlier work by offering more temporal reasoning capabilities. In addition to the simplistic time windows supported by Tribeca and NiagaraCQ, Aurora introduced the notion of a *latch* which retains a specific element for future processing steps, and the idea of *slack*: a means to handle events which arrived out of order. Aurora further featured *load shedding*, the principle of dropping tuples when compute time is insufficient to handle all input in a timely fashion. Evidently, this cannot in general be done without impacting the results. Carney et al. [27] therefore developed a strategy to select the data whose omission is least likely to impact results.

The Aurora project was combined with the Medusa project [17] to develop Borealis [1], a stream processing system that supports offloading its computation to a cluster of distributed computers. We consider this to be the next big milestone in the evolution of stream processing systems, as it enabled a significant increase in the load that can be handled by a stream processor. Not much later, Esper [56] combined the idea of SQL-based queries with distributed stream processing.

2.2.4 Early Big Data Processing: MapReduce

To enable very large data sets, the field of stream processing had to join forces with the field of *Big Data* processing. We temporarily shift the focus of our discussion away from streaming, and towards what could in retrospect be considered the point at which non-streaming *Big Data* processing became mature and gained wide use in industry: the introduction of MapReduce [44].

MapReduce is a programming model for applying computations to large amounts of data. As the name implies, MapReduce is based on the functional programming concepts of *mapping* a computation over a dataset, and *reducing* the result as if by an SQL `GROUPBY` where the values for each key are combined by an associative and commutative operation. The MapReduce programming model extends these concepts with a distribution model which is largely transparent to the user. Data is split up in subsets which get distributed across multiple physical computers. The same computation is then applied to each set, and the results are gathered to determine the overall result.

Advantages of the MapReduce Model

The main advantage of this model is that it allows for transparent distribution of the workload: each transformation can be done independently, and reduction can be done incrementally as the different compute nodes finish their transformation. Handling failures in the MapReduce model can always be achieved in one of two ways: either restart failed subparts, or ignore the failures and use only the results of the nodes that did not fail. Noteworthy is that neither option requires custom code to be written for a certain problem domain. MapReduce implementations come with these methods of failure handling built in, and the accidental complexity of handling failure, which typically burdens distributed programmers, is largely handled transparently.

Taking full advantage of the MapReduce model requires the transformation code to be purely functional, commutative, and associative, such that arbitrary retries do not influence the result, and such that merging in whichever order the system happens to select, leads to correct results. In practice, the MapReduce model is regularly used simply as a distributed task execution model with map-tasks that touch mutable state (which entails that repeated execution is not even strictly idempotent) and whose reduce-step is used only to synchronize on task completion. The simple to use nature of the MapReduce model led to widespread adoption. An open source implementation of the MapReduce model was made available in the form of the Hadoop [10] project. Hadoop's availability sped up the popularization of the notion of Big Data processing.

Drawbacks of the MapReduce Model

Despite its advantages, MapReduce suffers a number of drawbacks. Most importantly, the model does not deal well with pipelining. When the result of one computation serves as the input for another computation, both computations are scheduled one after the other. The results of one phase are returned to a central coordinator, only to be distributed again for the next phase. This inefficiency is targeted by data-parallel pipelines.

2.2.5 Data-Parallel Pipelines

Data-parallel pipelines offer ways of expressing a computation as a dataflow graph, where vertices are operations which can be executed in parallel to other vertices' operations, and edges are data-transfers. This architecture inherently supports distributing workloads over multiple compute nodes. The abstraction of data-parallel pipelines shares MapReduce's useful property that the way in which fault tolerance, failover, and recovery can be handled, are orthogonal to the application domain. As such, data-parallel pipelines can transparently handle these concerns.

A large number of novel systems operating by the data-parallel pipeline paradigm were developed to leverage MapReduce as their execution target. Examples include Pig [114], Dremel [102], JAQL [23], and Flume [73]. Simultaneously, a large number of existing data processing systems were made more scalable by developing a backend that schedules tasks as data-parallel pipelines on MapReduce clusters. For instance, SQL/MapReduce [60] offers a relational queriable data store backed by MapReduce, Yedalog [33] offers a Datalog system backed by MapReduce, and DryadLINQ [143] offers an implementation of .NET's Language INtegrated Query language that executes on a MapReduce backend.

Each of them differs somewhat in the way they map onto *data-parallel pipelines* scheduled on MapReduce, but they all operate roughly along one of the following avenues:

Parallel collections as used in systems such as Flume [73]. These systems offer the abstraction of parallel collections with a small set of operations, which can be efficiently implemented on top of the MapReduce model;

Relational datasets as used in systems such as Pig [114] and Dremel [102]. These systems feature an SQL-like language for querying the dataset, which is stored in such a way that it can be efficiently queried on top of the MapReduce model, e.g. by storing in a columnar fashion;

Knowledge base with declarative query language as used in systems such as Dyna [52], Jaql [23], or Yedalog [33]. These systems build on the concepts of Datalog to achieve near-Prolog expressivity with less risk for expensive or accidentally infinite regression in the deduction process.

The raw MapReduce model soon got superseded by those systems and their successors.

Drawbacks of the MapReduce Model in Data-Parallel Pipelines

As these data-parallel pipeline systems evolved, three significant weaknesses of the underlying MapReduce model became apparent. First, the high latency due to long startup time and slow communication made the model unfit for stream processing. Second, the model associates a high inherent cost to using previous results in a subsequent computation, as this requires a complete pass through the pipeline, including the long startup time and the slow communication. Third, the model does not deal well with shared global state, for what is essentially the same reason: by design, all synchronization in the MapReduce model happens by gathering results to the centralized master node at the end of a MapReduce cycle, and distributing that data again at the start of the next cycle, both of which are the slow parts of MapReduce.

Some solutions were devised which replace the MapReduce component with a custom-built distributed backend specifically targeting the pipelining of data. Frameworks such as Spark [144] offer shorter startup times, built-in mechanisms to efficiently reuse the results of previous pipeline stages, and ways of incorporating some sense of global state. At the same time, these frameworks still offered the main benefits of MapReduce: transparent distribution, load balancing, and fault tolerance through for instance checkpointing.

Shortcomings of Data-Parallel Pipelines

Data-parallel pipelines enabled efficient large-scale computations that required multiple steps. Some of them even partially tackled the problem of high processing latency. Still, none of these systems were really fit for streaming data processing: they offered neither the abstractions to reason about processing new data in the context of old data, nor the low latency necessary to respond in a streaming fashion. To tackle that problem, ideas from the field of stream processing had to be borrowed.

2.2.6 Towards Streaming Big Data Processing

Initially, solutions based on the existing systems were developed, leading to the Hadoop Online Prototype by Condie et al. [35] and Spark Streaming [146]. The former adapts pipelining within a job, and switches to a push-based data exchange model. The latter processes data on a Spark [145] cluster in many small jobs instead of a few large jobs, trading throughput for reduced latency. Despite the reduction in size per job, the underlying execution model remains batch-based, giving rise to the term "micro-batching". In both cases, the fault tolerance of the systems they were built on could be maintained. For instance, Spark Streaming guaranteed at-least-once semantics for processing, built on top of the at-least-once semantics offered by Spark for processing its Resilient Distributed Data Sets.

Industry-Grade Specialized Stream Processing Systems

In addition to those lower-latency versions of the existing tools, specialized stream processing systems were built, specifically aimed at Big Data processing. An early example of this is S4 by Neumeyer et al. [110]. S4 is built from the ground up to offer streaming support. It starts a new pipeline for each key/value-pair, offering low latency at the cost of peak throughput. S4 features a time-to-live based removal of data. Crucially, it coupled this with distributed processing support, and features some resilience to failure.

Work on S4 was halted when the industrial supporters of S4 switched to Storm [135]. Storm offers strong at-least-once semantic in its processing, coupled with a simple programming model. Sources of data (“spouts”) and transformations of data (“bolts”) can be defined in arbitrary imperative code. Coordination, transportation of data, and fault-tolerance is handled transparently by Trident [95], but imposes strict ordering on transactions. Storm in turn got superseded by a new system: Heron [89], which builds on the ideas underlying Storm. Storm and Heron both are supported by the Apache Software Foundation. A third distributed stream processing system is currently developed by Apache, which aims to more reliably store state. This system, Samza [112], is built on top of the Kafka distributed message broker [11], which enables it to guarantee strict exactly-once semantics — obviously under the constraint that it might take the system arbitrarily long to do this. Samza provides failure-handling not through checkpointing, but by keeping a changelog.

The tools described in the previous paragraph all target Java, Scala, or other languages running on the JVM. Other language communities have similar tools available. For instance, StreamPy [31] by Anomaly Systems Inc offers plain, but distributed stream processing for Python.

Specialized Event-Stream Processing Systems

Still, these tools merely provide a way of manually describing a topology, and implementing the components in imperative languages such as Java and Python. The need for tools which aid in processing event data, i.e., data whose temporal attributes are important, lead to a new wave of stream processing systems. We focus specifically on two of them: Naiad and MillWheel.

Google’s MillWheel [5] features *low watermarks* to track the remaining data belonging to a computation, essentially delimiting time windows. The ideas from MillWheel were transplanted into Google’s Cloude Dataflow platform [87].

In contrast, Microsoft Research’s Naiad [108] introduces the notion of *Timely Dataflow*. Timely Dataflow is a novel computational model centered around low-latency asynchronous message-passing, with minimal and lightweight coordination. Naiad distinguishes itself from earlier work by integrating incremental, streaming computation with iterative computation. Time windows in Naiad are formed by tracking not only time, but also by tracking an iteration count.

2.2.7 Summary

A summary of the history of Big Data stream processing can be found in figure 2.3. The figure depicts the systems discussed in this section, organized by the language paradigm (horizontally) and by the degree in which they support streaming (vertically). Dashed arrows depict the lineage of ideas, e.g., DryadLINQ builds on LINQ, by being a distributed version of the latter. Systems presented as green boxes define a domain specific language for stream processing. Systems presented as purple boxes offer only an API to an existing language. The borders of the boxes indicate the degree to which the systems support distribution: a solid border is a fully distributed system, a dashed border merely parallel. A dotted border indicates a system which is — in its original form — only suited for single-threaded execution.

2.3 Research Trends

The previous section sketched the history of Big Data stream processing in a chronological order. In this section we look at the evolution from active database management systems to distributed stream processing systems by focusing on three core trends: the introduction of **windowing** (section 2.3.1); the introduction of **distributed, streaming computation** (section 2.3.2); and the **shift in language paradigms** (section 2.3.3). We limit our discussion to the key languages and frameworks surveyed in previous section. Those languages and systems present a representative selection, based on prevalence in academia and industry.

The trends that will be described throughout this section are summarized in table 2.1. The last column presents a best-effort based indication of the year of introduction of the technology. For technologies introduced in academic papers, we use the year of publishing. Note that some of the systems only gained some distinctive features after that initial year.

2.3.1 Restricting Selections by Time: Windowing

Traditional Temporal Windows

When dealing with event data, the temporal component of the data is important. To provide a semantic fit for the temporal component, many streaming technologies discussed in the previous section introduced a form of *windowing*. Temporal windows represent a slice of time; a group of data elements whose membership to the window depends purely on temporal constraints. Windowing was an important innovation, and arrived in what Esmaili [55] calls the first generation of stream processing systems. Though Gigascope [39] lacks the means for specifying windows explicitly, systems like Tribeca [131] and Aurora [27] introduced windows as a built-in concept. A "latch" in Aurora is a window too albeit a single-slotted one. Similarly, Aurora's notion of *slack* is a related, somewhat

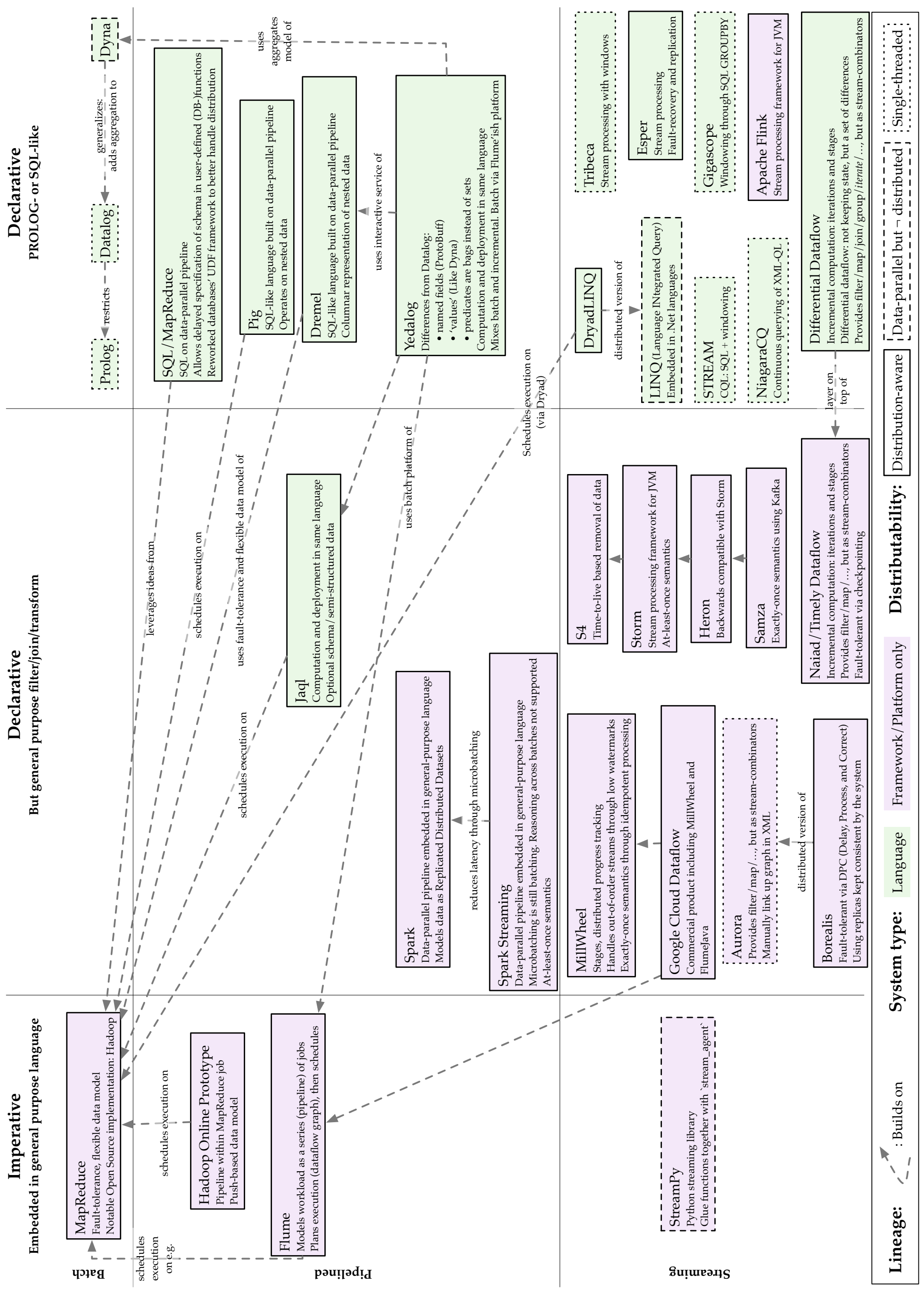

Imperative
Embedded in general purpose language
Declarative
But general purpose filter/join/transform
Declarative
PROLOG- or SQL-like
Batch
Pipelined
Streaming
MapReduce
Fault-tolerance, flexible data model
Notable Open Source implementation: Hadoop
Hadoop Online Prototype
Pipeline within MapReduce job
Push-based data model
schedules execution on
Flume
Models workload as a series (pipeline) of jobs
Plans execution (dataflow graph), then schedules
schedules execution on e.g.
StreamPy
Python streaming library
Glue functions together with `stream_agent`
Spark
Data-parallel pipeline embedded in general-purpose language
Models data as Replicated Distributed Datasets
reduces latency through microbatching
Spark Streaming
Data-parallel pipeline embedded in general-purpose language
Microbatching is still batching. Reasoning across batches not supported
At-least-once semantics
Jaql
Computation and deployment in same language
Optional schema / semi-structured data
leverages ideas from
schedules execution on
uses fault-tolerance and flexible data model of
schedules execution on
uses batch platform of
Schedules execution on (via Dryad)
MillWheel
Stages, distributed progress tracking
Handles out-of-order streams through low watermarks
Exactly-once semantics through idempotent processing
Google Cloud Dataflow
Commercial product including MillWheel and FlumeJava
Aurora
Provides filter/map/…, but as stream-combinators
Manually link up graph in XML
distributed version of
Borealis
Fault-tolerant via DPC (Delay, Process, and Correct)
Using replicas kept consistent by the system
S4
Time-to-live based removal of data
Storm
Stream processing framework for JVM
At-least-once semantics
Heron
Backwards compatible with Storm
Samza
Exactly-once semantics using Kafka
Naiad / Timely Dataflow
Incremental computation: iterations and stages
Provides filter/map/…, but as stream-combinators
Fault-tolerant via checkpointing
layer on top of
Prolog
restricts
Datalog
generalizes: adds aggregation to
Dyna
uses aggregates model of
SQL/MapReduce
SQL on data-parallel pipeline
Allows delayed specification of schema in user-defined (DB-)functions
Reworked databases' UDF framework to better handle distribution
Pig
SQL-like language built on data-parallel pipeline
Operates on nested data
Dremel
SQL-like language built on data-parallel pipeline
Columnar representation of nested data
uses interactive service of
Yedalog
Differences from Datalog:
• named fields (ProtoBuff)
• 'values' (Like Dyna)
• predicates are bags instead of sets
Computation and deployment in same language
Mixes batch and incremental. Batch via Flume'ish platform
DryadLINQ
distributed version of
LINQ (Language INtegrated Query)
Embedded in .Net languages
STREAM
CQL: SQL + windowing
NiagaraCQ
Continuous querying of XML-QL
Tribeca
Stream processing with windows
Esper
Stream processing
Fault-recovery and replication
Gigascope
Windowing through SQL GROUPBY
Apache Flink
Stream processing framework for JVM
Differential Dataflow
Incremental computation: iterations and stages
Differential dataflow: not keeping state, but a set of differences
Provides filter/map/join/group/iterate/…, but as stream-combinators
Lineage:
: Builds on
System type:
Language
Framework/Platform only
Distributability:
Distribution-aware
Data-parallel but ¬ distributed
Single-threaded

Figure 2.3: A summary of the state of the art in Big Data stream processing

System	Windowing						Distribution		Expressivity		Year
	Sliding	Landmark	Tumbling	Session	Semantic	Other	Supported	Fault-tolerance	Language	Paradigm	
Tribeca	✓	-	✓	-	-	-	-	-	custom	D	1996
NiagaraCQ	✓	-	-	-	-	-	-	-	XML-QL	S	2000
Aurora	✓	-	✓	-	-	latch	-	-	-	B	2002
Gigascope	-	-	✓	-	-	-	-	-	GSQL	S	2003
STREAM	✓	-	✓	-	-	-	-	-	CQL	S	2003
Borealis	✓	-	✓	-	-	latch	✓	R	-	B	2005
Esper	✓	✓	✓	∼	∼	-	✓	L	SQL	S	2006
DryadLINQ	-	-	-	-	-	element index	✓	C + R	LINQ	S	2008
HOP	∼	∼	∼	∼	∼	-	✓	C	-	I	2010
S4	∼	∼	∼	∼	∼	TTL	✓	R	-	I	2010
Flume	∼	∼	∼	∼	∼	-	✓	C + R	-	I	2010
Spark Streaming	-	-	✓	-	-	-	✓	C + R	-	S	2012
Naiad	∼	∼	∼	∼	✓	timely dataflow	✓	C	-	I	2013
MillWheel	∼	∼	∼	✓	✓	low watermarks	✓	C	-	I	2013
Storm	∼	∼	∼	∼	∼	-	✓	C + L	-	I	2014
Heron	∼	∼	∼	∼	∼	-	✓	C + L	-	I	2015
Flink	✓	✓	✓	✓	∼	-	✓	C	SSQL	S	2016
Samza	∼	∼	✓	✓	∼	-	✓	L	-	I	2017

Table 2.1: A taxonomy of streaming systems, ordered by time of introduction — For windowing: "∼" indicates that the column's windowing type is manually implementable in the system on top of the existing abstractions, but is not offered out of the box — For fault-tolerance: "R" indicates replication, "C" checkpointing, and "L" the use of a replayable changelog or eventlog — For language paradigm: "D" indicates declarative programming languages, "B" box-and-arrow, i.e., a manually construct graph, "S" SQL-like languages, and "I" imperative, manual coding.

dual concept to windowing: *slack* identifies a time period during which events' relative order is not yet known. Instead of identifying the "current" most relevant data elements, as a time window does, *slack* identifies the "current" not yet fully consolidated data elements. The notion of *slack* was generalized to a notion of *revision processing* in Aurora's successor: Borealis [1].

The types of windows offered by these early systems were rather straightforward. They were all one of three types:

sliding windows, i.e., a slice of either the last *n* events, or the last *n* seconds of events;

landmark windows, i.e., a slice delimited at the front by a "landmark" event, and either another "landmark" event, or the current time at the end; or

tumbling windows, a slice of at most *n* events, which fills up gradually, and once full gets reset.

A "latch" can be thought of as a sliding window of length 1, or — equivalently — as a tumbling window of length 1. The upper 3 timelines in figure 2.4 visualize how a stream of events can get mapped onto windows in all 3 traditional temporal window types.

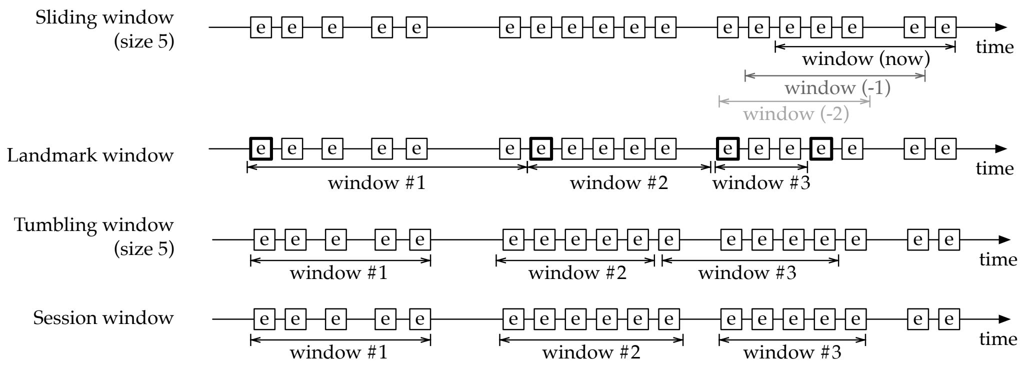


Figure 2.4: Visualization of four traditional window types — For each window type, the first three windows are indicated. The squares marked "e" are events, laid out on a timeline. The rectangles outlined in bold are "landmark events".

These window types offer good primitives for building stream processing systems. Moreover, they easily lend themselves for high performance implementations. A sliding window of length n, is just a ring buffer of length n. A landmark window is just a dynamically sized array which explicitly gets emptied when a "landmark" event occurs. Tumbling windows are even simpler, being merely a bounded vector in which events get placed until it is full, at which point the vector gets emptied. To this day, industry-strength stream processing systems largely leverage those window types. The "micro-batches" offered by Spark Streaming [146] are essentially tumbling windows, with the added limitation that processing is delayed until the window is full. As typical with the tradeoff between batch and streaming, Spark Streaming's batches offer high throughput, at the cost of latency.

Beyond Traditional Window Types

Still, those early types of windowing have their shortcomings. The type and size of the window must be determined statically, by the programmer. Furthermore, all runtime events are placed in the same window data structure. An abstraction that turned out to be useful for application domains such as click-stream monitoring — i.e., analyzing the interactions users have with e.g. a web application — is that of a *session window*. A session window is a best-effort approach to categorizing events that constitute one "session" of user interaction. It is used when a session cannot be established by discrete actions such as a login and logout, but instead has to be inferred. A session window is a window of events, delimited by periods of inactivity.[1] The fourth timeline in figure 2.4 visualizes session windows. Support for session windows is present in some of the later streaming

[1]We use the definition of session windows used by Flink, for instance at `https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/windows.html`. Other systems use definitions closer to our notion of "landmark window", where a certain start and end event delimits the session.

systems, or in recent updates to older system. Table 2.1 marks systems as supporting session windows if some version of the system supports them.

Semantic Windows

Many real-world use cases can not be expressed using these four window types. By design, reasoning is only possible within the window. Reasoning over large time windows quickly becomes resource intensive. A number of stream processing systems sought to tackle these problems. Jiang et al. [81] define the overarching solution as *semantic windows*.

With *semantic windows*, the time windows are no longer a static, one-dimensional length or duration. Instead, window membership is determined by the specific semantics of some pattern, in conjunction with the runtime values of previous events. Conceptually, a new window is created for each event, containing the events that are within the temporal constraints with respect to that first event. An alternative way of looking at this, is that events are only stored for the duration during which matching is actually possible with other concrete events which actually occurred. Among the surveyed systems, only Naiad [108] and MillWheel [5] (and by extension, Google Cloud Dataflow [87]) offer the means to tie the lifetime of data to some property that can only be detected at run time. Both use a system of logical timestamps that enables lightweight coordination: Naiad uses *Timely Dataflow* to keep track time and iteration count, whereas MillWheel uses *low watermarks* to track the remaining data belonging to a computation. Semantic windows do not lend themselves to a simple visualization, since they — by definition — depend on the concrete patterns being matched. We therefore did not include them in figure 2.4.

Many production-grade systems — even those introduced after Jiang et al. coined the term *semantic windows* — do not offer built-in windowing support at all. Such systems instead position themselves as Big Data stream processing, and do not necessarily concern themselves specifically with event data, or with correlating data points over large spans of time.

2.3.2 Distribution and Fault-Tolerance

Using the categorization of Esmaili [55], systems from both the second and third generation of stream processing innovated largely by providing and improving support for distributing the processing of data across multiple computers, and by adding the fault-tolerance features that go with it.

The Introduction of Distribution

Consider the case of Aurora: the Aurora stream processing system got subsumed by a new project, Borealis [1]. The stream processing constructs offered by Borealis are largely the same as those of Aurora. New in Borealis are support for distribution and fault tolerance.

From the stream processing programmer's perspective, little changes: the novel concerns of the distribution and fault tolerance are tackled by a set of new constructs — for instance the addition of constructs to store and replay streams — but the old concerns are mostly handled as they were in Aurora.

Even systems which do not offer full-fledged support for distribution at least sought to improve performance by parallelizing their workload across multiple devices. One instance of this is Gigascope [39], which brought computations closer to the data (achieving so-called *edge processing*) by moving parts of the computation away from the CPU and onto the Network Interface Card.

Overall, the move towards distributed systems in stream processing is clearly visible in the timeline shown in table 2.1. Support for distribution became standard for all stream processing systems in the early 2000's. The lack of centralized control in a distributed system led to the introduction of distributed *backpressure* in e.g. Heron [89]: distributed components can reject new data if their processing cannot keep up with what is being sent to them. Components earlier in the task dependency graph buffer their output, or in turn temporarily cease processing. This entails that the problem gets propagated backwards — the components "push back", as it were — to the data ingress point, where the problem is to be handled.

Fault-Tolerance

Distributing software across multiple machines may lead to higher throughput, and offers a novel opportunity: the system can deal with partial failures. Unlike in single-machine solutions, in distributed solutions, failure of one computer does not necessarily mean that the system as a whole fails. It may still complete its task, if the system features *fault-tolerance*.

On the flip side, the chance of some subcomponent of a system failing, increases with the number of components in the system. Hence, distributing computations across multiple machines does not only enable failure handling, it also increases the necessity thereof.

Borealis, and later Esper, demonstrate this well: they introduce distribution, and supplement that with the means to handle failures: replication and replayable logs respectively. The systems built on top of the MapReduce model inherit its checkpoint/restart-based fault-tolerance. The ZooKeeper [12] component got disentangled from Apache's Hadoop project, and used as a general-purpose coordination mechanism for many distributed data processing systems, offering replication-based fault-tolerance out of the box. Similarly, Spark Streaming [146] inherits the checkpointing and replication offered by Spark [144]. Finally, systems built on top of distributed message brokers such as Kafka [11] automatically receive the fault-tolerance the message broker offers. For Kafka, that means replaying the event log, with support for enforcing exactly-once semantics.

The stream processing systems which are not built on top of such preexisting technologies provide custom solutions playing to the strengths of the technology used. For instance, the fault-tolerance offered by MillWheel [5] uses MillWheel's low watermarks system to determine what data to retain.

2.3.3 Expressivity and Language Paradigm

We stated in section 2.2.1 that active databases are at the origin of stream processing systems. The origins of active databases are *database triggers* as supported by most relational database management systems. Such triggers can respond to the insertion of new data by executing SQL statements. Active databases started using the term *continuous queries* to refer to SQL queries which were registered to trigger whenever new matches to the query arrived in the database.

Early in the history of active databases, the use of plain SQL was found to be limiting.

Tribeca [131] introduced a custom stream selection and filtering language. In NiagaraCQ [32], continuous queries could be expressed in XML-QL. The XML-QL query language has a novel XML-based syntax, but returns to SQL's underlying relational algebra. While the streaming execution model prevents traditional query planning, common subexpressions in multiple continuous queries could be optimized by sharing the results of common join operations.

Later systems continued this return to SQL, even embracing an SQL-like syntax for their languages. For instance, Cranor et al. [39] states that "[u]nlike Tribeca the Gigascope uses an SQL-like language (GSQL) rather than a procedural language to express its queries, allowing query composition and query optimization". Around 2010, this trend reversed, when stream processing systems dropped the relational schemas from their systems, requiring instead procedural data selection and manipulation code to be written using *stream programming* constructs. The evolution can be observed in table 2.1. Note that before the start of the timeline, relational databases were based on traditional SQL.

A Taxonomy of Stream Processing Language Paradigms

From the viewpoint of interaction paradigm, the systems depicted in figure 2.3 and categorized in table 2.1 can be subdivided into the following categories:

Precursors to stream processing follow a non-streaming approach. Data is first inserted into the database. Queries are later submitted to the database, reasoning is performed, and results are returend. Language paradigms can include SQL in the case of traditional RDBMS'es, or general-purpose declarative, logic programming in the case of PROLOG-derived systems.

Trigger-enabled databases are a first step towards streaming data processing. The interaction model of the precursors to stream processing is typically still available, but triggers may be registered, which will activate when matching data arrives. Triggers may modify the incoming data, produce derived data, remove data depending on the inserted data, etc. The language paradigm is typically SQL.

Early stream processing systems like Tribeca, Aurora, and Borealis follow a different approach, offering full support for streaming, incremental processing. These stream processing systems are first programmed with some behavior. Next, whenever data is inserted, that data may be transformed, and notifications or transformed data may be produced. The language paradigm of these early systems was either declarative, or required manually linking up different processing steps, using the so-called “boxes-and-arrows model” [1].

Streaming databases like NiagaraCQ, Gigascope, and Flink (when using SSQL) [8] combine database concepts with proper streaming. First, a continuous query or rule is installed. Next, whenever data is inserted, that data may be transformed, and notifications or transformed data may be produced. In addition, the data itself is usually stored for some time. Such systems offer a streaming version of relational databases, and the language paradigm used, is hence derived from SQL.

Big Data stream processing systems like S4, Spark Streaming, Flink (when using the stream API) [8], and MillWheel do not mimic RDMBS'es. Nevertheless, the interaction model starts of similar to that of streaming databases: the systems first are programmed with some behavior. Next, however, whenever (a batch of) data is inserted, a processing pipeline is deployed across a number of compute devices. Input data is sent to those computers for processing. In the end, results are gathered and sent to a consumer. The language paradigm used by these systems is typically general-purpose imperative programming, somewhat constrained by an API. This API is typically centered around stream programming constructs like maps, filters, `GROUPBY`, and aggregation within a group. The API might impose restrictions on which data is accessible, and control over which processing step is executed when is left to the framework's runtime system.

Trend

A trend is visible from these data points: the domains of (active) databases and stream processing have merged, but have then split up again into streaming databases on the one hand, and Big Data stream processing frameworks on the other. The former has fully embraced an SQL-based language paradigm for processing structured data. The latter has fully embraced general-purpose languages, and may be used where only schemaless data is available.

While the early systems had built-in support for approximation, later systems assume exact results are achievable. Aurora, Borealis, and STREAM assumed their processing only got to see the stream once, and operated under the assumption that systems in practice would occasionally be forced to drop data. Newer systems assume that — between replication, replayable persisted logs, backpressure, and elastic scaling of available cloud resources — exactly-once semantics and transactional behavior are achievable. The interface offered to users of the systems suggests they transparently take care of the issues. Adoption-levels suggests that users prefer the apparent easy of use that is provided by the simplification. It should be noted, however, that the assumptions of exactly-once semantics and transactional behavior cannot formally be guaranteed due to theoretical limitations on distributed systems.

2.3.4 Summary

We identified three trends in the state of the art of distributed Big Data and stream processing.

- First, the need for temporal reasoning when handling event data led to the introduction of **multiple windowing techniques**. The traditional window types are widely supported by streaming systems. Semantic windows — which define window membership based on more involved reasoning than merely the timestamp of the event itself — only appear in specialized systems.
- Second, stream processing systems increasingly support spreading their workload across multiple machines. **Support for distribution** is by now ubiquitous. The mechanisms for handling failures were quickly added as well. In addition to the typical techniques of replication and checkpointing, many streaming systems offer a log-and-replay approach where the streamed input can be replayed after failure.
- Third, **two subcategories** of stream processing systems emerged: SQL-based streaming databases and Big Data stream processing frameworks enabling the processing of large amounts of data using general-purpose programming languages.

We summarize this in table 2.2, which reproduces the left-hand side of table 1.1 on page 3.

2.4 Discussion: Shortcomings in the State of the Art

At first sight, the streaming databases and Big Data stream processing systems described in this chapter seem able to encode the driver scenarios defined in section 2.1. The scalability of these systems implies that they can handle any arbitrary processing load. Their streaming computational model tackles the latency-problem that is inherent to batch processing. With their windowing support, it should be possible to express temporal constraints such as the 2-week period. Constraints such as the 100 € minimum should be expressible too: for stream programming approaches by means of filters, for streaming databases as `WHERE` clauses in the SQL. However, a more in depth analysis reveals some issues with both categories of streaming systems.

	Distributed Big Data & Stream Processing	
	Big Data Stream Processing	**Streaming Databases**
Throughput	✓✓ very high	✓ high
Cost per update	✓ constant (depth of program)	– proportionate to history
Bounds on resource usage	✓ yes (with basic operators)	– proportionate to history
History	– none	✓✓ large history size
Pattern matching	– unsupported	✓ supported (relational join)
Reaction logic	– often integrated, but Callback Hell	– outside of the model

Table 2.2: Summary of the state of the art of distributed Big Data and stream processing

2.4.1 Event Correlation in Big Data Stream Processing Frameworks

Data-parallel pipelines and their streaming successors do not aid a programmer in correlating pairs of events based on their attributes. Built-in support for windowing is limited in those systems, and the very concept of finding all pairs of events jointly satisfying some constraint is not at all covered by the APIs offered. Indeed, we defined the main distinction between Big Data stream processing frameworks and streaming databases to be the lack of support for a declarative programming paradigm in the former.

The stream processing frameworks offer constructs to filter elements or to compute running averages, or minima or maxima over a window. Thanks to improvements in the way shared global state is managed, they can even offer a high-performance incremental *fold* over data. Obviously, their use of Turing-complete languages makes them technically capable of computing anything Turning-computable during such a *fold*. However, that entails moving the computation and its accumulated state “out of the model”: the stream processing system can hence no longer manage the data, collect garbage, or transparently distribute data in an optimal way. The time and space constraints of programs executing arbitrary code in a *fold*, are in principle unrestricted, and complexity may depend on the size — or even values — of a number of data elements whose structure is not explicitly understood by the framework.

Some of the stream processing frameworks internalize this concern to some extent, by offering SQL-like *GROUPBY* and *JOIN* constructs as functions in their API. However, those too do not impose any bounds on the size of the (intermediate) results. While for, e.g., running averages or *zip*s between two streams produce predictable, fixed-cost pipelines, data processing pipelines involving *joins* start behaving like generic computations with arbitrary run times and resource requirements. When there is no statically guaranteed upper bound on the resources required to implement the processing pipeline, processing may be unable to keep up with the incoming data, or run out of storage and subsequently crash. This conflicts with the requirements of the problem statement outlined in section 1.1.

We conclude that Big Data stream processing systems which offer the means of (relationally) joining streams, effectively double as streaming databases, and therefore suffer from the shortcoming discussed in the next section. The shortcoming of other Big Data stream processing systems for implementing our problem case is as follows:

Shortcoming of Big Data Stream Processing Systems

Existing Big Data stream processing systems do not support matching event patterns.

In the absence of pattern matching, the driver scenarios cannot be implemented.

One could manually implement the scenarios from section 2.1 using e.g., Spark Streaming, Flink, or Samza. In fact, we do so as part of our evaluation in chapter 7. As we show there, though, the stream processing frameworks do not aid the programmer in tackling the hard parts of the scenarios. The scenarios are only implementable due to the ability to escape the domain-specific streaming API that Spark Streaming tries to enforce, and using the Turing-completeness of the underlying Java-based platform; the streaming API gets in the way instead of helping.

2.4.2 Data Processing Guarantees of Data in Streaming Databases

Streaming databases do offer the right programming paradigm to correlate events, but inevitably need to deal with the concern of being overloaded, too. As mentioned in the history of stream processing (specifically, in section 2.2.3), Aurora [27] and Borealis [1] deal with overloading through a process known as *load shedding*: when the stream processing system would fall too far behind on the data, Aurora and Borealis purposefully drop data. Though the systems attempt to minimize the impact this has, dropping any event data at all is irreconcilable with the requirements of the problem statement from section 1.1.

More recent systems use backpressure and elastic scaling to deal with overloading. Backpressure — by construction — can only handle short-term imbalances: if a stream processing system chronically cannot keep up with the rate at which new events arrive, slowing down processing will not help. Elastic scaling can deal with long-term inbalances, but only to a limited degree. The amount of compute power offered by a cluster does not scale linearly with the amount of resources available. Increasing the resources that are available, has diminishing returns. Hence, even with elastic scaling in a cloud setting, one cannot be certain that a given system will be able to handle a certain load without careful analysis of the system and the maximum expected load.

We conclude that shortcoming of streaming databases for implementing our problem case is as follows:

Shortcoming of Streaming Databases

Existing streaming databases do not offer the means to accurately model the load on a system given a certain set of patterns to detect, and a certain set of event streams.

In the absence of such a formal model, the hard requirements of our problem domain cannot be guaranteed, even in the absence of discrete hardware faults.

2.5 Conclusion

We surveyed the state of the art in Distributed Big Data and stream processing. We discussed how techniques from the domains of active databases, early stream processing systems, and Big Data processing were combined. Three trends emerged: the introduction of increasingly specialized temporal windowing, ubiquitous support for distribution and fault-tolerance, and a schism based on the programming language paradigm that is offered by the systems. This schism in programming paradigms, splits the domain in two: on the one hand we have streaming databases, on the other Big Data stream processing frameworks.

We discussed how stream processing frameworks failed to tackle the hard problems of our driver scenarios: searching for patterns in the streams. In turn, streaming databases were revealed to be unfit for the setting of our driver scenarios: while streaming databases do support searching for patterns in event streams, this features comes at a cost in predictability. Streaming databases cannot guarantee they will stay online and keep monitoring events under increased load. We conclude that no existing Big Data stream processing system or streaming database satisfies the requirements of live security monitoring.

3 State of the Art in Event Handling

The Big Data stream processing systems and streaming databases discussed in the previous chapter failed to satisfy the requirements exemplified by our driver scenarios on page 12. We therefore shift our focus to the remainder of the related work which deals with processing event streams: *event-based* frameworks and languages.

The languages and frameworks discussed in this chapter are not typically envisioned for dealing with huge streams of data, but instead consist of programming language paradigms which seek to improve programmers' ability to construct programs which detect and process combinations of events. We will discuss two such paradigms in sections 3.2 and 3.3. In section 3.4 we build a taxonomy of the different event handling languages, first based on their event-detection semantics, and then based on their operational event-detection model. In section 3.5 we discuss how the state of the art of event handling languages falls short in order to support our driver scenarios.

3.1 Traditional Approaches to Event Handling

Hinze and Voisard [72] describe event-based systems as systems which "trigger actions based on observed events. The most basic triggered action is the sending of a notification to an interested party". The set of driver scenarios sketched on page 12 hence calls for an event-based system: an implementation of those scenarios must respond to certain patterns of events.

Traditional programming paradigms are not event-based: they are designed first and foremost for describing a transformation from input to output. Of course, it is possible to deal with runtime events in traditional programming paradigms (cf. *streaming* as opposed to *batch* processing, as discussed in section 2.2), but a programmer must specifically structure their program to deal with events. There are two main ways in which programs are traditionally structured to deal with runtime events:

Event handling by callbacks requires programs to register pieces of code with sources of events, henceforth *event sources*. These pieces of code, named callbacks, can take the form of first-class procedures, function pointers, objects, etc. depending on the paradigm in use. Event sources can be other logical parts of the same application, network sockets, inter-process message channels, and so on.

Event handling by waiting structures programs in such a way that they read data from some input channel. Traditionally, the program would *block* when no more data is present on the channel. Whenever the occurrence of an event outside the system creates new data, the program would be *unblocked* and execution could resume. This synchronous approach severely hampers performance compared to what an asynchronous approach using callbacks can achieve. Language facilities have been developed which enable programmers to write asynchronous event handling code which syntactically takes the form of waiting event handling. The most well-known thereof are the async/await [24] constructs of C$^\sharp$, Java, or Scala.

Event handling by waiting offers a straightforward programming model when the program's main concern is processing events one by one. On the other hand, the model of event handling by waiting leads to rather rigid event handling. A callback-based approach handles this more flexibly, as event handling code can easily be associated with components. The drawback of callbacks is that they break the flow of code by requiring an anonymous procedure to be introduced. Depending on the language used, either those callback procedures capture relevant variables in their lexical scope, or explicit context-objects must be constructed. Regardless, application logic gets spread over multiple procedures, increasing cognitive load for the programmer. The problem gets exacerbated as the reaction logic becomes more complex: callbacks installed from within callbacks face the choice of either increasing the nesting level, or spreading the application logic even further apart. As Bierman et al. [24] point out, such an inadvertent "inversion of control-flow impedes the use of structured control constructs, the staple of sequential code". The problems of callback-based approaches are commonly refered to as the *Callback Hell* [51].

The need for handling events is commonplace in modern programs: if the program, e.g., exposes a user interface, accepts network connections, or listens to sensors, event handling is necessary. Understandably, the research domain of event-based systems is too large to warrant in-depth discussion in this dissertation. To limit the scope of our discussion, we omit an in-depth discussion of single-event systems such as Ptolemy [26] or EventJava [57].

Similarly, we do not discuss middleware which purely offers publish-subscribe.[1] We instead focus our discussion on paradigms for reacting to combinations of events: Complex Event Detection and Reactive Programming.

3.2 Averting the Callback Hell: Reactive Programming

In the previous section we introduced the notion of *Callback Hell*. Callback-based programs which depend on multiple event sources are written substantially different from how one would write a program which depends on multiple input variables. Most of the difference is boilerplate code.

3.2.1 The Functional Reactive Programming Paradigm

The *Reactive Programming* paradigm [16] seeks to tackle the Callback Hell. Reactive programming has its basis in purely functional programming languages. The seminal work is FRAN, a language for Functional Reactive ANimations by Elliott and Hudak [54]. Other reactive programming languages built on the purely functional programming paradigm followed, such as Frappé [38], Yampa [75], and Flapjax [103]. The breakthrough outside of academia was achieved by Czaplicki and Chong [42], whose Elm language introduced reactive programming to the industry. These purely functional reactive programming languages are known as *Functional Reactive Programming* languages, FRP languages for short.

The original FRP languages mostly concerned themselves with computing values which might change over time. Later FRP languages switched their focus to processing concrete occurrences of events. The unifying idea in FRP languages, though, is that the program flow merely models how values are assigned to the outputs based on the input values. It is the responsibility of the language and its runtime to reevaluate some portions of the code whenever one of the inputs changes.

Representing Values: Behaviors and Streams

In FRP, these inputs — but also the outputs and intermediate variables — take the form of either *behaviors*, or *event streams*. Behaviors are "time varying values" [77], i.e., they hold a single, current value. Streams represent the entire sequence of events that occurred up to the current time. It is well-established [16] that in a purely functional setting, behaviors and streams are complementary. Using the terminology from FrTime [37]: a `hold` operator can be defined which turns a stream into a behavior by exposing only its

[1]For the remainder of this dissertation, we assume that the reader is familiar with the idea of registering reaction logic — such as callbacks — to be executed in response to events. We introduce a more principled and structured way of specifying reaction logic in section 3.2, where we introduce *reactive programming*.

last value, and a `changes` operator can be defined to reveal the stream of all values a behavior has had.

Streams require the programmer to use stream-oriented operations to transform the data. On the other hand, they make it simpler to express e.g., filters over those streams. Evidently, remembering all previous values leads to unbounded memory growth, linear in the number of events that occurred. Practical implementations of long-running programs may hence need to be constrained to behaviors and bounded streams.

3.2.2 Beyond Functional Reactive Programming

Outside of purely functional programming, reactive programming has recently caught on too, for example in the form of FrTime by Cooper and Krishnamurthi [37]. However, the major breakthrough outside of the purely functional world came in the form of frameworks for existing languages. Reactive extensions were introduced [100] for a number of existing languages, for instance Java, Scala, C++, and JavaScript [119]. Many of the recent Web frameworks are based on the reactive programming approach, including Facebook's React [78].

In the absence of functional purity those reactive runtimes cannot determine which expressions may be safely reevaluated. Programmer effort is required to specify which side-effects may be triggered in response to updates. Usually these side-effects must be idempotent or otherwise cancel each other out for the system to be well-behaved.

3.2.3 Active Research in Reactive Programming

Since reactive programming is being used for real-world scenarios, theory and practice of reactive programming languages and frameworks are an active field of research. Real-world use cases are often built for the Web, an inherently distributed platform. While for instance Elm [42] already supported the notion of propagating updates through a distributed dependency graph, more recent work focuses on reducing the cost thereof [50, 109]. Similarly, since real-world use cases tend to run on top of object-oriented languages, and are often programmed by programmers who are used to object-orientation, recent work by Salvaneschi et al. [124] looks into how to optimally bridge between functional and object-oriented programming. Experiments by Salvaneschi et al. [123] show that use of the reactive programming paradigm effectively improves program comprehension.

3.3 Detecting Event Patterns: Complex Event Processing

Reactive programming solves the issues of the Callback Hell, but does not help programmers write software which detects *patterns* of events. Detecting series of events conforming to some pattern, in an on line stream of events, is known as Complex Event Detection (CED) [117] or Complex Event Processing (CEP) [40].[2]

CEP is in wide use in business processes: many companies employ CEP on the stream of business events generated by their systems to detect problems in the supply line, to trigger automatic reporting, or to rapidly discover and act on business opportunities [47, 46, 40, 72]. However, the techniques underlying CEP can also be used on a much lower level of processing. For instance, the software driving a touch screen might detect swipes on the screen by pattern matching periodic samples of where the screen is being touched [97].

3.3.1 Differentiating CEP from Streaming Databases

Evidently, CEP systems and the stream processing systems discussed in the previous chapter share some overlap. Especially streaming databases are often used for CEP workloads [61, 62, 49].

Differences in Use Cases

Looking back on the existing literature, we find that systems which are considered as CEP systems by their authors, differ from those considered streaming databases in at least one meaningful way: stream processing systems and active databases are invariably **optimized for processing large amounts of data**. In those systems it is assumed that the dataset is large proportionate to the processing power available at the time the system was developed.[3] Stream processing systems process some abstract "stream" of events with a high throughput. Streaming databases just have tables where data is put into. By default, streaming databases store all historic data.

In contrast, CEP systems focus on three aspects:

- First — by definition of *complex event* — CEP finds meaning in the **correlation of multiple events**. Typically, the means of correlation offered by stream processing systems and active databases takes the form of aggregation such as running averages, or relational joins, respectively. CEP systems instead offer constructs like regular expressions or pattern matching.
- Second, CEP focuses on the expressivity of the event correlation language. This entails that CEP systems are typically based around **declarative languages**, and that they include support for some form of **temporal reasoning** to correlate events.

[2]In this dissertation, we do not distinguish between Complex Event Detection and Complex Event Processing. The line between both is after all not very clear, and existing literature on both largely overlaps. For consistency, we stick to the terms "Complex Event Processing" and "CEP".

[3]Evidently, what was a "large dataset" a few decades ago may very well fit in main memory of a mobile device nowadays.

- Third, CEP systems typically are meant to support **large numbers of concurrent queries**. Contrast this to stream processing systems, which are typically meant to subject (a large number of) events to a relatively small set of processing paths.

Hinze and Voisard [72] additionally point out that many active databases "can rely on the transactional context for the composition of events. Trigger conditions can be defined based on the old and new state of the database, thus using the concept of states rather than describing the event itself".

In most cases, the difference between a CEP system and a stream processing system are clear. Nevertheless, some systems fit in both categories, or have extensions that bridge the gap. For instance, while Flink [8] is arguably a stream processing system, FlinkCEP [9] is obviously considered a CEP system by its authors.

Spatio-Temporal Reasoning in CEP

CEP systems are used in diverse application domains. In all of those domains, the temporal aspect of the data is important, hence the use of CEP. However, in many of those domains, location is important too. Consider for instance security monitoring, which includes, e.g., detecting malicious actions, or monitoring for possible malfunctions Demers et al. [47] in cyber-physical systems. The physical location at which the problem occurs, must be taken into account. Or consider the case of web analytics (e.g., click-stream analysis) [40] or multi-touch gesture detection [74], where the location of user-interactions on a screen must be taken into account. The processing back-end of environmental monitoring using sensor networks [25] evidently must take into account the physical location of the sensors. Similarly, application domains like traffic control [72], supply chain management [72], and inventory management Wang and Liu [139] contain the notion of the physical location of traffic and supplies. Even in credit card fraud Schultz-Møller et al. [127] detection, the country from which a transaction was executed is important.

This has given rise to the prevalence of support for spatio-temporal reasoning in CEP languages. With spatio-temporal reasoning, matching not only based on how two (complex) events relate in time, but also on whether the events occurred (physically) close to each other spatially.

Spatial reasoning for events has some commonalities with temporal reasoning for events. For instance, events can be ordered in time as well as in space. An event can be within some region in space as well as time. On the other hand, there are some clear differences between the temporal and spatial dimensions: time is typically treated as a one-dimensional concern, whereas spatial concerns are typically two- or three dimensional. This entails that, while time constraints are typically limited to an interval, spatial patterns can be defined in more elaborate ways, such as by means of polygons (for membership-testing) or polylines (for sequencing), This has led to the creation of specialized tools for authoring spatial complex event patterns for application domains such as gesture detection [74]. For instance, EventHurdles [86] enables CEP pattern authors to specify spatial constraints by drawing “hurdles” on a canvas. From this, a CEP pattern is derived that matches “gestures” which pass those hurdles. These gestures are generic movements in time and space, e.g., a swipe over a touch screen, or changes in the orientation of a mobile device.

A less domain-specific example of support for spatio-temporal reasoning for CEP can be found in SpaTeC [128]. SpaTeC was originally conceived for a mobile computing context. To process complex interaction events between multiple mobile devices, the proximity of the devices must be incorporated into the CEP. To this end, SpaTeC introduces a number of event composition operators. In SpaTec syntax, a pattern $E_1\{<>\}E_2$ requires that some complex event E_1 occurred at the same location as some complex event E_2. $E_1\{><\}E_2$ matches when both events occurred at a different location. SpaTeC allows combining these spatial event composition operators with temporal event composition operators: e.g., two events occurring concurrently at the same location are written as $E_1\{\substack{<>\\ ||}\}E_2$, while a pattern specifying that E_1 must occur before E_2, and both must happen at a different location, is written as $E_1\{\substack{><\\ ;}\}E_2$.

In general, many CEP systems come with built-in support for spatial reasoning. For our driver scenarios (see page 12), however, these concerns are not relevant. We will therefore not further discuss spatial reasoning in this dissertation. For our overview of the state of the art of CEP we focus on the most related work in the state of the art. Specifically, we will continue this chapter from the SASE CEP language by Wu et al. [141], and outline the main evolution of CEP in the last decade from that starting point.

3.3.2 A Baseline for Modern CEP: SASE

The SASE language [141] is a pattern specification language for CEP. A SASE program can be installed on a single stream of timestamped events. Each event has a type, corresponding to a named relation which defines which attributes the event has. The basic syntax of a SASE program looks as follows:

```
EVENT <event-pattern> [WHERE <filter>] [WITHIN <window-expr>]
```

The event pattern can name a sequence of events. For instance, `SEQ(A a, B b)` matches all pairs of events of the relation named A and of the relation named B, such that the event of relation A occurred before the event of relation B. For each instance, the match to relation A is bound to a logical variable `a`, and the match to relation B to a logical variable `b`. These logical variables can be used elsewhere in the pattern, and in the `WHERE`-clause and window-expression.

The `WHERE`-clause expresses join-constraints, e.g., `EVENT SEQ(A a, B b)WHERE a .id = b.id` expresses a relational join over A and B’s `id` attributes, in addition to requiring that `a` occurs before `b`. The window-expression in the `WITHIN`-clause expresses a time window (cf. section 2.3.1 on page 23). For instance, `EVENT SEQ(A a, B b) WHERE a.id = b.id WITHIN 1.5 seconds` restricts the previous pattern to only match pairs of `a` and `b` where `b` occurs at most 1.5 seconds after `a`. Finally, SASE supports *negation-as-failure*: `EVENT SEQ(A a, !B b)` matches all occurrences of events of relation A for which are not followed by an occurrence of an event of relation B.

The results of a SASE program is another stream of timestamped events, containing all matches.

Among the chief limitations of SASE is its inability to express patterns which match a variable amount of events: the multiplicity of a SASE sequence is determined by the number of named events, and is hence static. This shortcoming is resolved in SASE+ [3]. SASE+ has a similar syntax to SASE, but allows writing e.g. `SEQ(A+ a[], B b)` to signify a variation on the previous pattern where all matches to relation A are combined with the subsequent of events of relation B.

The basic syntax of a SASE+ program is depicted in listing 3.1.

```
FROM <stream>
   [PATTERN <event-pattern>]
   [WHERE <filter>]
   [WITHIN <window-expr>]
   [HAVING <pattern-condition>]
   RETURN <output-stream-name>
```

Listing 3.1: Basic syntax of SASE+

As shown, in SASE+ an output stream must explicitly be named. Furthermore, SASE+ adds a `HAVING`-clause, which supplements the `WHERE`-clause in a way analogous to the `HAVING`-clause in SQL: instead of making groups using `GROUP BY` as in SQL, groups in SASE+ are matches to the entire event pattern [66]. In other words: the `WHERE`-clause can reason about values bound to logical variables; but the `HAVING`-clause can reason about matches to Kleene closures of matches too.

3.3.3 Aggregation and Monitoring Multiple Streams: Cayuga

The Cayuga [47] CEP language is modeled even more like SQL, albeit a minimalistic version. In their survey paper, Cugola and Margara [40] state that Cayuga "trades expressiveness for performance, providing an extremely simple rule language, with a small number of operators". Despite the apparent simplicity of the language, it is an expressive CEP language.

Like SASE+, Cayuga supports aggregation. Furthermore, Cayuga does not assume all events occur in one single stream of events. Multiple, named streams can exist in Cayuga. Each of those streams has its own relational schema. Cayuga programs can select events from multiple streams by refering to the streams by name.

A formal underpinning for the matching behavior of Cayuga programs is provided in the form of a Cayuga algebra [46].

Cayuga's basic syntax looks as follows:

```
SELECT <attribs> FROM <stream-expr> PUBLISH <output-stream-name>
```

A tuple of all listed attributes is published on the named output stream, for each match returned by the stream expression. The stream expression in the `FROM`-clause consists of either a `FILTER`, a `NEXT`, or a `FOLD` expression:

Filter-expressions take the form `FILTER{pred-expr} (stream)`. A filter-expression discards all events for which the predicate does not hold from the stream to which it is applied.

Next-expressions take the form `stream-expr NEXT{some-expr} stream-expr`. Next-expressions take the "next" value of the right-hand stream for each element on the left-hand stream, i.e., for each event which occurs on the left-hand stream, the first event to occur on the right-hand stream is picked, and both are returned. Cayuga uses an explicit "followed by" relation: the "next" event selected by a `NEXT`-expression really means the very next element that satisfies the condition, not just some element some time later which satisfies the constraints. We come back to this in section 3.4.2.

Fold-expressions take the form `stream-expr FOLD{selection-predicate, stop -condition-expr, aggr-expr} stream-expr`. A fold-expression selects each element from the stream that satisfies the selection predicate, and aggregates it using the aggregation-expression, until an event is encountered which satisfies the stop-condition. Again: the very first occurrence of an event matching the stop-condition terminates the match. Compare this to our driver scenarios, where we do not want the occurrence of a non-fraudulent financial transaction to purge all data on potentially fraudulent transaction patterns which have not yet completed.

Cayuga models events as data points with a start timestamp and an end timestamp, i.e., an interval during which the event occurred, instead of a single timestamp. This is especially useful for complex events which abstract over an arbitrary number of events — those aggregated by a `FOLD`-expression — whose duration can vary greatly.

Consider a concrete example, inspired by our earlier work on multi-touch gesture recognition [97]. Consider a query which detects a rightwards swipe gesture on a multi-touch screen: it detects that the screen surface is touched at some location, and that in a time windows of two seconds at least three touches occur, in such a way that each successive touch is more to the right, i.e., has a larger x coordinate. The query selects the minimum (i.e., begin) and maximum (i.e., end) x coordinate, as well as the starting y coordinate.

This gesture is expressed in Cayuga syntax in listing 3.2:

```
1   SELECT min_x, x AS max_x, start_y
2   FROM
3     FILTER{count > 3, DUR > 2sec} (
4       -- subquery saving initial `x` and `y`, initializing `count` to one
5       (SELECT x, x AS min_x, y, y AS start_y, 1 AS count FROM Touches)
6         FOLD{TRUE, -- select any next event
7              $2.x <= $.x, -- stop when the x-value would decrease this step
8              $.count+1 AS count -- aggregate the count
9             } Touches)
10  PUBLISH Swipes
```

Listing 3.2: The Cayuga syntax for a “swipe right” multitouch gesture

The query first selects an event from the `Touches` stream (on line 5). It then accumulates (using `FOLD`) more `Touches` until the next touch has a smaller x coordinate than the previous touch. The resulting stream is then filtered (on line 3): only streams which accumulated a `count` above 3, and have a duration above 2 seconds, are retained. For each of those streams, a tuple is formed with attributes `min_x`, `max_x`, and `start_y` (by means of line 1). Those tuples are then published to a stream named `Swipes` (on line 10).

3.3.4 Expressive and Efficient CEP with Distributed Event Sources: TESLA

The main contribution of Cugola and Margara [41] is their push towards a generic, all-purpose CEP. They tackle this in two ways. First, they developed TESLA, an expressive CEP language. Second, they built an efficient CEP algorithm for TESLA into the T-Rex engine. We come back to the processing algorithm in section 3.4.4, when we describe the different detection models used in CEP languages.

The TESLA Language

Primitive events are captured by event sources. These sources may be physically distributed. Cugola and Margara introduce support for such event sources “[A]t the peripheral of the system”. This does not necessitate a distributed, parallel processing, and indeed their system follows a single-threaded approach to the processing itself.

For the TESLA language itself, consider the code snippet in listing 3.3:

```
define    Swipe(min_x: double, max_x: double, start_y: double)
from      TouchStart(min_x, start_y) and
          each TouchContinued(x=$x)
          within 2 sec. from TouchStart and
          first TouchStop(max_x)
          within 2 sec. from TouchStart
where     min_x<=$x and $x<=max_x and Increasing($x) and Count($x)>3
consuming TouchStart, TouchContinued, TouchStop
```

Listing 3.3: The TESLA syntax for a “swipe right” multitouch gesture

This snippet of TESLA syntax solves a similar, though slightly different use case than the Cayuga snippet in listing 3.2: a `Swipe` event is emitted, listing its `min_x`, `max_x`, and `start_y` attributes, whenever the pattern below is detected (line 1). The pattern itself consists of a primitive `TouchStart` (line 2), followed by some `TouchContinued` events (line 3). The values of those events' `x` attributes are bound to the logical variable `$x`. All `TouchContinued` are gathered for up to 2 seconds after the `TouchStart` (line 4). Hence, `$x` refers to a collection of all `x` attributes of all continued touches in the two-second time window. The first `TouchStop` event after the `TouchStart` concludes the pattern (line 5). The `where`-clause enforces that each value bound to `$x` are between `x` attribute of the `TouchStart` and the `TouchStop`, that the bound values form an increasing series, that that there are at least 3. Finally, the TESLA program specifies that all events used to match this pattern are consumed (line 8), i.e., that they are no longer available for matching by other patterns or other instances of this same pattern.

Semantical Differences between TESLA and Earlier CEP

The TESLA snippet expresses a different program than the Cayuga snippet. First, since Cayuga does not offer the means to specify consumption, the snippet in listing 3.2 does not consume its input. Second, the TESLA snippet uses explicit start and stop events, whereas the Cayuga snippet does not. This has the same cause as the last significant difference in the snippet: the TESLA snippet allows at most two seconds for the swipe, whereas the Cayuga snippet requires at least two seconds. We had to change the way we express a swipe because both systems offer different features: when a Cayuga pattern is installed on a stream, the pattern can only filter events, take the next event, or fold over events. Cayuga cannot respond to the progress of elapsed time without the occurrence of a new event, only to the occurrence of an event which is on the inspected stream, and which matches the end condition of a fold. Therefore, the Cayuga example starts by eagerly matching all sequences of touches which move rightwards, and then filters out those who are too short in time or multiplicity of events. TESLA, instead, is more time-aware, and can therefore immediately start gathering all events during a concrete time window of two seconds that satisfy the conditions. When the time window is finished, it needs not wait for a non-conforming event before recognizing the swipe gesture.

This difference is neither trivial nor unimportant. Consider a scenario akin to the one sketched on page 12, where a pattern detects a problem occurred when an event which satisfies some conditions fails to occur within five seconds after some other event. If such an event does not occur, but no other event occurs either in the next day, a Cayuga-style CEP system would detect the problem one day too late, whereas a TESLA-style CEP system would detect it immediately at the end of the five second time window.

To conclude: while Cayuga improved upon SASE by offering aggregation and better handling of multiple streams, Cayuga regressed by losing explicit windowing support. TESLA brought back windowing support, which has an important benefit for live processing.

3.3.5 A Formal Foundation for Modern CEP: EVA

EVA [72] is an *event algebra* rather than a CEP language. Like Cayuga's algebra [46], EVA aims to provide a solid formal foundation for CEP. EVA's aims go further than just specifying their own language, though: EVA attempts to capture the semantics of all CEP, enabling all CEP patterns to be expressed in EVA. Hinze and Voisard [72] identified four issues with the state of the art of CEP:

"unspecified temporal semantics" of composition operators;

"unclear semantics" of patterns;

"lack of collaboration" between different frameworks; and

"lack of adaptivity", both in the sense that linking up event sources to CEP systems typically requires manual programming of adapters, and in the sense demonstrated by listings 3.2 and 3.3: the semantics of CEP systems differ sufficiently that patterns need to be rewritten considerably when porting from one system to the next.

Their solution is a parameterized event algebra, called EVA. In EVA, patterns of interest are specified as "profiles" or "subscriptions". Events gathered at event sources are sent to the central CEP processor, where they are optionally consumed. The event algebra can express at least sliding windows, tumbling windows, and landmark windows. Events in a time window can be aggregated using some aggregator expression. The core features of EVA, though, are the windowing mechanisms and event sequencing.

While EVA explicitly aims to tackle the unclear semantics in existing CEP languages, some issues persist even in EVA itself. We discuss these in section 4.6.2 on page 90.

3.3.6 Complex Event Patterns and Reaction Logic as Declarative Rules

A trend is apparent in modern CEP languages: they focus on declarative specification of complex event pattern. Some modern CEP language — discussed before – come with their own, custom pattern matchers. Others use specialized descendents of the continuous query languages stemming from the streaming database world [30, 61, 62], which we described in the previous chapter. Another branch of systems use **production rules** [25, 115, 97]. Production rule — traditionally found in expert systems — encode *if-then* logic, consisting of a pattern of logical facts to match, and a number of new logical facts to create.

Forward-chaining rule engines are a good fit for event-based systems, as they enable incremental matching of declarative rules. In response to matching an event pattern, a production rule can emit one or more new events. Production rules are hence more flexible than the systems discussed above, which were limited to emitting one event onto a (named) stream.

Through the tight integration of event detection and enactment of responses, rule-based CEP systems are able to overcome the *impedance mismatch* [68, 136] which typically exists between detection code and reaction code. Responses — by construction — are executed whenever a match is found. Response logic — by construction — has all relevant data in scope, insofar that the response logic should only depend on attributes of the matched complex event detected by the pattern of the rule. In the context of CEP, production rules essentially replicate the advantages of the Reactive Programming paradigm we described from page 37 onwards.

3.4 A Taxonomy of Event Handling

The previous two sections briefly sketched the recent history of Complex Event Processing languages and Reactive Programming languages. Both paradigms improve event handling code, albeit in different ways. To enable a structured discussion on the limitations of the state of the art, we now define a taxonomy of event handling languages. For an in-depth, general-purpose overview of CEP languages, we refer to the papers by [40] and Hinze and Voisard [72]. We use their terminology where applicable. This section specifically aims to make the discussion in this dissertation self-contained. We therefore outline the aspects that are relevant to the thesis.

To this end, we categorize existing event handling techniques along two core axes: first, language support for expressing semantic concerns such as temporal constraints, negation, multiplicity, etc., and second the detection model of the CEP system, i.e., how the system is described to (operationally) match events. These two axes serve as the foundation from which we define the semantic (language) properties (in chapter 4) and the operational semantics (in chapter 5) of a programming paradigm fit for the domain of our driver scenarios. We split the discussion of the detection semantics in three parts: event consumption semantics, the semantics of "followed by", and support for temporal constraints.

3.4.1 Event Consumption Policies

One main property that differentiates different CEP languages' semantics, is the **event consumption policy** [147]. An event consumption policy dictates whether an event can match more than one pattern. In some languages, an event which is detected as part as one complex event, cannot be part of another complex event. In other languages, events can be part of many complex events. Cugola and Margara [40] calls the event consumption policy of the first category "zero consumption". For the former case, they surveyed only systems which make the consumption optional, and programmatically determinable. They hence call the event consumption policy for those other systems "programmable consumption".

All reactive programming languages we surveyed employ a zero consumption policy: every dependent of a *behavior* or *stream* is updated for every change of the event source.

Event Consumption in our Driver Scenarios

For the scenarios we envision, "zero consumption" is needed. Consider our driver scenarios introduced in section 2.1. A pair of financial transactions can be suspected of muling money between two accounts, using a third account. Yet, it is possible that another step of muling takes place, continuing from the second transaction. Even though this transaction was matched to a CEP pattern, it should still be available for future matches with other transactions.

Representation of Results

Related to the event consumption, is the representation of results in event handling systems. Results are represented in one of the following ways:

As a stream CEP systems built on top of stream processing systems typically represent results as a single, new stream. Feedback loops are explicitly impossible: results are generated downstream of the place where event occurrences are inspected. The same holds for some purpose-built CEP systems, e.g., SASE and SASE+ (see page 41). Other purpose-built CEP systems like the ones described from page 42 onwards represent CEP programs as compositions of stream-transformations. Hence, in addition to the final results, intermediary results are represented as streams, too.

As entries in a knowledge base In production-rule-based CEP, detected complex events are just part of a knowledge base. By design, results can be used by any matching pattern in the same CEP system again, as long as they are not consumed or explicitly removed.

As records in a table Streaming databases often materialize the generated results in whichever representation of streams or tables they use.

As behaviors or streams Reactive programming languages by design represent (intermediary and final) results as either *behavior*s or *stream*s.

3.4.2 Semantics of "followed by"

To express complex event patterns, CEP languages offer ways to express that one event follows another. For instance, SASE has the `SEQ` operator (see page 41), Cayuga has `NEXT` expressions (see page 42), and TESLA has conjunctions with time windows (see page 44). What exactly is meant by these, differs from language to language. In some cases, sequencing should be read as "is followed immediately by", whereas in others it should be read as "is followed eventually by".

Reactive programming languages lack support for "eventually followed by" semantics. Many reactive programming languages do offer a construct to select a sequence of events;

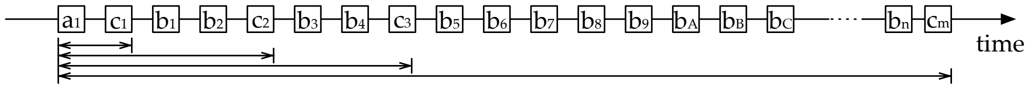


Figure 3.1: Matches to Kleene closure on an event stream — When the "follows by" relation allows skipping matches, selecting the shortest match is no longer viable, as that renders `b*` and `b?` equal.

specifically: to select an event and another event that occurred before on the same stream. Such constructs carry names like `foldp` (in Elm [42]) or `latch` (in the "Stream Processing with a Spreadsheet" paper by Vaziri et al. [137], as well as in the Aurora stream processing system as described on page 18). These constructs offer "immediately followed by" semantics: one can only select an event a fixed, finite number of occurrences ago.

Semantics of "followed by" in our Driver Scenarios

The "immediately followed by" semantics are inadequate for the scenarios we envision. Consider again our driver scenarios. When one wants to detect an incoming money transfer followed by an outgoing money transfer which satisfies some constraints, arbitrarily many other events may occur in the event stream in between those two transfers. It is even possible that other outgoing transfers which satisfy the constraints occurred in between the two.

Relation of "followed by" Semantics with Kleene Closure

The "eventually followed by" semantics however lead to unclear semantics for Kleene closures, such as those designated by the "star" operator in regular expressions. When a "followed by" relation may skip over arbitrarily many events, a Kleene closure may mean multiple things. Consider the timeline of events in figure 3.1, which depicts events as instances of either a, b, or c. Events have no other attribute than a timestamp. Consider a regular expression of the form `a b* c`. The extent of possible matches to that expression are depicted at the bottom of figure 3.1: all the pairs of a and c. Any selection of the b's between those two events may constitute a valid match. The matches are hence a_1c_1, a_1c_2, $a_1b_1c_2$, $a_1b_1b_2c_2$, $a_1b_2c_2$, a_1c_3, $a_1b_1c_3$, $a_1b_1b_2c_3$, $a_1b_2c_3$, $a_1b_1b_2b_3c_3$, etc.

Evidently, specifying the semantics of Kleene closure for events in such a way that all options are considered as a "match" leads to an impractical and inefficient matching behavior. Similarly, specifying that the longest match should be used, is impractical and inefficient: it is impossible to match `a b* c` in a streaming fashion using those semantics: at no point can the pattern matcher know that no other closing c will still arrive on the event stream. Finally, opting to specify the semantics of Kleene closure to select the shortest match is dissatisfying as there would then be no difference between `a b* c` and `a b? c`. Under these semantics, Kleene closures are not a means of aggregating multiple matches.

In summary, Kleene closures is not the clean abstraction it is in the case of strings when used on streams where a "followed by" relation allows skipping over elements, but that semantics is the most useful for CEP. If sequencing has "eventually followed by" semantics, Kleene closure is only a useful abstraction if it is coupled with a time window.

An analogous issue exists for the case of the positive closure (the "plus" operator in regular expressions).

3.4.3 Support for Temporal Constraints in Event Handling Languages

A final aspect of the semantics of event handling languages is their support for *temporal reasoning*. When multiple events need to be correlated, time often plays a crucial role. Some CEP languages — like SASE [141] or Cayuga [47] — allow patterns without temporal constraints, though many CEP languages require some form of time windows, as we introduced in section 2.3.1 on page 23. In contrast, reactive programming languages lack such features. As mentioned in previous section, reactive programming languages offer operators like `foldp` or `latch` to refer to specifically the previous event, but have no constructs for specifying, e.g., all events within 5 seconds of another event.

Timestamps vs. Time Intervals

CEP languages use one of two approaches for positioning events in time: either an event occurs at a timestamp (e.g., in SASE), or an event occurs during a time interval (e.g., in Cayuga). We refer to the paper by Hinze and Voisard [72] for an extensive comparison of temporal point semantics vs. temporal interval semantics.

The different representations of time have their most significant impact in the context of composition: when events are modeled as occurring at a timestamp, abstracting multiple events into a complex event necessarily loses information. However, even when events are modeled as occurring during a time interval, event composition can still discard useful information. We discuss this in detail in section 4.6.2 on page 90, and show that these problems can be alleviated by retaining the concrete event times of all involved events while the pattern is being matched, and only discarding this information when the complex event is explicitly abstracted into a complex event. This is not possible in systems which apply *implicit hierarchical composition*, such as Cayuga or EVA, where every sub-pattern implicitly abstracts its constituents into a new compound event.

Relation of Temporal Constraints with Negation

The ability to reason with temporal constraints — and with the notion of actual progression of time in the outside world — becomes even more important when *negation* is involved. Consider for instance a CEP program which deals with the occurrence of a financial transaction, and the absence of a claim that that transaction was fraudulent. Such a pattern cannot in general be implemented in a streaming fashion: if a transaction and a claim are found, it trivially holds that the pattern is not matched. If instead a transaction occurred, but no claim has yet been made, the system cannot conclude anything. Consider evaluation of this program on an event stream that contains the occurrence of a transaction, and the occurrence of a fraud claim for that transaction 4 seconds later. It would be wrong to initially conclude that the program's rule is matched. Conversely, consider a situation where the stream contains an occurrence of a transaction, but no claim for that transaction. Evaluation of the program would have to wait indefinitely.

In the context of negation, *time windows* are therefore necessary. A pattern matching a financial transaction, and no fraud claim e.g. within two weeks, can be processed in a streaming fashion. Regardless, evaluation of that pattern may require waiting: if no fraud claim occurs, this pattern can be successfully matched two weeks after the transaction. In general, a CEP language which supports negated patterns must restrict negated patterns within some time window, and matching events to those negated patterns necessarily requires that evaluation can wait for the entire duration of the time window.

3.4.4 Event Detection Models

We now shift our focus from what the different existing event languages do, to *how* they accomplish this.

Some aspects of the detection model are common across most event handling systems. For instance, they all have two distinct phases: first, the pattern specifications are interpreted, and data structures are built from the specifications. Second, the actual event handling phase is started, where live events are pushed to those pre-built data structures. These data structures can either be *a.)* (finite state) automata, *b.)* search-trees, or *c.)* dependency-graphs.

The remainder of this subsection discusses these three kinds of detection model.

a.) Automata-based Models and Methods

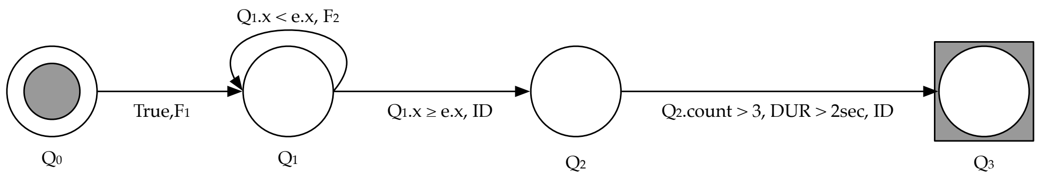
The first kind of detection model, are automata-based models. A huge body of work exists on matching strings using finite state automata or push-down automata. Efficient pattern matchers can be built using these techniques, as long as the patterns can be expressed in regular languages or as context free grammars. Finally, automata inherently match their inputs incrementally. Thus, matching (event) streams to patterns using these techniques is trivial. It warrants little additional explanation why the detection model of a large number of CEP systems builds on automata and regular expressions, including the work by Pietzuch et al. [117], SASE(+) [141, 3], Cayuga [47], and TESLA [41].

Mismatches between CEP and FSA Multiple mismatches were soon discovered between what traditional finite state automata offer and what CEP needs:

- First, obviously, anything not expressible in a regular language cannot be implemented using only finite state automata.
- Second, traditional finite state automata give rise to "immediately followed by" semantics, which have the problems described in section 3.4.2.
- Third, as a result of the second point, Kleene star and Kleene plus have unclear semantics, rendering the main point of using regular expressions moot.
- Fourth, automata typically also imply a *consuming* semantics, which — while fine for many CEP application domains — is not applicable to some CEP domains including ours, as discussed briefly in section 3.4.1.
- Fifth, traditional finite state automata operate on a reasonably small, finite input alphabet. In contrast, transitions in CEP depend not only on small finite sets such as the types of events. Consider a small pattern of two events, which are joined when they share the same value for some string-valued attribute. In general, this amounts to a pattern A^nB^n, which is not a regular pattern, and hence cannot be expressed by a finite state automaton. Limiting the length of the string to some value — say, 100 bytes — makes the pattern finite-sized, and therefore expressible by a finite state automaton. However, to support all possible values of the shared string-valued attribute, a number of states in the order of 2^{8*100} should be generated, which is clearly impractical.
- Sixth, traditional finite state automata do not support having arbitrarily many ongoing possible matches, in such a way that all matches to all patterns will be reported. For CEP it is often not sufficient to know that a complex event was matched; the reaction logic invoked by a CEP program typically needs access to the concrete attribute value of all constituent events.

Domains in which patterns occur which cannot be expressed in regular languages evidently cannot use models based on finite state automata. Approaches have been devised which bypass the other mismatches, while still retaining some of the advantages of automata.

Example of Automata-based Matching For a concrete example of how CEP with (non-deterministic) finite state automata can proceed, consider the state transition graph depicted in figure 3.2. This graph represents the finite state automaton created by Cayuga for the code snippet from listing 3.2 on page 44. This automaton receives events from the `Touches` stream. The arc between state Q_0 and Q_1 is labeled with the condition on the event from the `Touches` stream ("TRUE", i.e., no additional condition) and with a function which specifies which values have to be bound to which logical variables ("F_1", i.e., bind the event's attribute `x` to logical variables `x` and `min_x`, bind its attribute `y` to logical variables `y` and `start_y`, and bind the constant 1 to logical variable `count`). From Q_1, each new event on `Touches` whose `x` attribute is larger than the value bound to the logical variable `x` of Q_1 causes a self-transition, which increments the value bound to the logical variable `count` (using F_2). The first touch whose `x` attribute is too small causes



where
F1 = e.x ↦ x, e.x ↦ min_x, e.y ↦ y, e.y ↦ start_y, 1 ↦ count
F2 = Q1.count+1 ↦ count

Figure 3.2: Cayuga automaton for Cayuga Event Language example from listing 3.2 — States are represented as circles. Possible transitions are represented as arrows. Q_0 is the start state. Q_3 is an accepting state. Each transition is labeled by a condition and a function updating bindings. Functions F_1 and F_2 are depicted at the bottom. “ID” is the identity, i.e., changes no binding.

a transition to Q_2. If the accumulated `count` is larger than three, and the (implicitly measured) duration is larger than 2 seconds, accepting state Q_3 is reached. While this specific example is expressed in Cayuga, both SASE and T-REX (TESLA’s runtime engine) largely operate similarly.

Differences between CEP Automata and Traditional FSA To ensure that multiple events occurring at the same time are processed correctly, Cayuga uses *epoch-based* processing. This entails that events occurring at same time are processed jointly, but may not cause automata produced in response to another event from the same time to take a step.

To enable detecting multiple independent matches to patterns — where the events of multiple patterns optionally interleave — systems like Cayuga and T-REX deviate from traditional finite state automata behavior by duplicates automata instead of letting one single automaton take a step. In response to the occurrence of an event for which a state transition is defined for some automata, Cayuga and T-REX duplicate those automata, and have the newly created copy take a step. The occurrence of an event does not destructively mutate any automaton. In case of zero-consume semantics, the CEP system updates each instance of the automata it thusly created whenever a new event occurs. Part of the complexity of Cayuga’s and T-REX’s design comes from the creation of indexes to provide fast access to the automata that may be affected by a certain event.

For each instance of an automaton, the set of matched events is retained. As exemplified by figure 3.2, edges between states are labeled not only by the type of event that causes the transition, but also by predicates on the attributes of those events. Time windows can be enforced by having a different state transition for events outside of the current window compared to those that still fit in the window. Language runtimes can clean up automata which cannot make any more steps, e.g. because their last time window ended, or because a counterexample was encountered for a negated pattern.

b.) Tree-based Models and Methods

A second kind of detection model, are tree-based models. These are typically used for *logic programming*-based event handling, such as those based on production rules.

Supporting Order-Independent Matching The CEP languages mentioned in the previous section are a great fit whenever the CEP patterns one wishes to detect can be expressed in a regular language. One major limitation of automata-based solutions is that automata have a fixed recognition order, i.e., the pattern order and the recognition order are the same. Order-independent patterns have to be explicitly implemented as a disjunction of the different orders. This opens up the option of a combinatorial explosion of states, which in turn constitutes a performance problem.

The fact that matching happens in-order brings more subtle limitations, too. Consider the following sequence pattern in SASE syntax:

```
... SEQ(..., !A a, B b)... WHERE a.attr = b.attr ...
```

The pattern expresses the sequence of some events, followed by the absence of an event `a`, followed by an event `b`, where the `attr` attributes of `a` and `b` are the same. Informally, the semantics of this pattern make sense. For an automata-based system, though, this pattern is impossible to implement: it can only progress to the event `b` after not having found an `a` whose attribute matches that of an event it has not encountered yet. A different detection model must be used, which can implement patterns relating events to (absence of) events regardless of which event arrives first. Time windows should be enforced, but time windows should be able to extend forwards as well as backwards. This necessitates moving away from the notion of a current *state* of the pattern matching process.

Event Handling by Incremental Logic Programming Solutions take the form of logic-based query-planners (e.g., RTEC and its Prolog implementation [15]), or incremental algorithms such as the Rete algorithm [59]. The former category closely approximates the streaming databases discussed in chapter 2. As a result, they suffer the same issues in the context of our driver scenarios. Incremental pattern matchers like Rete are a better fit. We extensively explore their properties, how they work, and what they lack in detail in section 5.1. The gist of these systems, however, is that matches for all sub-patterns are tracked in parallel, and all correlation between events happens by incrementally joining sub-patterns pairwise.

c.) Dependency Graph-based Models and Methods

A final kind of event detection model, are dependency graph-based. This is the detection model used by reactive programming languages.

Like the other models, dependency graph-based models first build a detection data structure, and subsequently feed events into that structure. For Reactive Programming languages, this takes the form of propagating events to the root nodes representing the primitive stream for that event kind.

Push-based vs. Pull-based Reactive Programming Programs written in FRP languages interact with the outside world in one of two ways: they are either *push-based* or *pull-based*.

In a *pull-based* approach to FRP, whenever an entity in the outside world requests a new result from some output *behavior* or *stream*, the current value of the input *behaviors* or *streams* is determined, and evaluation chains backwards from the output, triggering reevaluation of every *behavior* or *stream* on which they depend.

Conversely, in *push-based* approaches, the occurrence of an event in the outside world causes the system to chain forward from the input at which the value arrived, updating the values associated with the intermediate *behaviors* or *streams*, until the wave of updates reaches the outputs.

Maintaining a Consistent State In both approaches, the order normally imposed by means of a program counter is imposed by means of a dependency graph. Care must be taken when evaluating an FRP program to prevent *glitches*. The customary example — reproduced from Cooper [36] — of an expression which requires glitch prevention in a FRP language is the expression

$$\mathit{seconds} < \mathit{seconds} + 1,$$

where *seconds* is a *behavior* representing the current time in seconds since some epoch.

Outputs computed by FRP programs should of course reflect the state of their inputs. In all FRP languages we discussed, it is required that a computed output can be traced back to a consistent set of input values; a set of values that were actually the current values of the inputs at some point in time. As such, evidently the above expression should always evaluate to the boolean value `true`: the set of inputs consists of a single rational number for the variable *seconds*, and any rational number is strictly smaller than its successor.

A too naive implementation of FRP logic might erroneously expose the order in which the dependency graph is traversed, and lead to a situation where the update of the *seconds* behavior on the left-hand side of the relational operator is already materialized, but the $\mathit{seconds} + 1$ behavior a the right-hand side of the operator is not yet. Such an evaluation scheme would cause the wrong output — `false` — to be momentarily produced. In a subsequent evaluation of the behavior $\mathit{seconds} + 1$, the semantically correct value `true` would be restored, making this situation an intermittent bug, i.e., a *glitch*.

In FRP languages these glitches can be prevented by updating the dependencies in a topologically sorted order of the dependency graph [36]. Evidently, this has the drawback of sequentializing evaluation, which leads to underutilization of compute resources on computers with multiple hardware threads, let alone in distributed settings.

In general, topological sorting no longer suffices as a glitch prevention strategy in a distributed setting [124]. A partial solution was designed in Elm [42], which defines a way of propagating changes through a potentially distributed graph. However, that "technique is inherently incompatible with dynamic dependencies" [50], such as those introduced by `if` conditions which depend on reactive values. A solution to that problem was developed by Drechsler et al. [50], in the form of the SID-UP propagation protocol. SID-UP offers guaranteed glitch-freedom in a distributed setting while allowing dynamic reconfiguration of dependencies. In SID-UP, the propagation of events through the reactive graph is performed in two phases: an admission phase, and a propagation phase. The latter phase can execute multiple propagations in parallel, and can proceed without contacting a centralized coordinator. This behavior significantly reduces the number of messages that need to be sent to execute a *turn*, and reduces the depth of the message-chain (i.e., the number of "steps" the algorithm takes), and with that: the time it takes to process an update.

3.4.5 Summary

We created a taxonomy of the state of the art of event handling, based on four characteristics:

- First, different **event consumption policies** exist. For our driver scenarios defined on page 12, a "zero consume" policy is needed.
- Second, different **semantics of "followed by"** exist in event handling languages. For our driver scenarios, "eventually followed by" semantics are needed.
- Third, existing event handling languages express **temporal constraints** either on even timestamps or on time intervals. Our driver scenarios can be expressed in either approach. However, to support negation we definitely need support for time windows.
- Fourth, multiple **detection models** exist in the state of the art. Matching multiple complex event patterns — as required by our driver scenarios — can be accomplished by either tree-based or automata-based detection models.

We summarize this in table 3.1, which reproduces the right-hand side of table 1.1 on page 3. The shortcomings of the state of the art are discussed in the next section.

3.5 Discussion: Shortcomings in the State of the Art

On page 12 we described three driver scenarios. In section 2.4 we described how the state of the art of stream processing frameworks fails to capture the whole scope of the

	Event Handling	
	Complex Event Processing	**Reactive Programming**
Throughput	~ average to low	~ average to low
Cost per update	– proportionate to history	✓ constant (depth of program)
Bounds on resource usage	– proportionate to history	~ possible, assuming no loops etc. in program
History	✓ some history	– none
Pattern matching	✓✓ expressive pattern language	– unsupported
Reaction logic	– outside of the model, or Callback Hell	✓✓ well-supported (core of the paradigm)

Table 3.1: Summary of the state of the art of event handling systems and languages

driver scenarios. Crucially, no technique provided event correlation mechanisms as well as guaranteed upper bounds on processing time and memory requirements. In this chapter we introduced two other research branches which can be leveraged to process events: Reactive Programming and Complex Event Processing. We now indicate why these techniques fall short at expressing the driver scenarios, too.

First, we note that most techniques discussed in this chapter do not offer the means of **distributed processing**. In their survey, Bainomugisha et al. [16] identified only three reactive programming languages with support for distribution, and noted that none of the three offered glitch-freedom in a distributed setting. Among the CEP languages, support for distribution — or even just multi-threading — is largely absent. Of course, distributed event based systems exist; e.g. those discussed by Mühl et al. [107], or the streaming databases discussed in chapter 2. We discussed their shortcomings in section 2.4.2. The expressive, formalized CEP languages described in this chapter are mostly single-threaded. SASE, Cayuga, TESLA/T-REX, and EVA all define single-machine algorithms.

The remainder of our discussion on the shortcomings in the state of the art of event handling, is split up into a discussion on the shortcomings of (functional) reactive programming languages, and a discussion on the shortcomings of CEP languages.

3.5.1 Shortcomings of Functional Reactive Programming

The shortcomings of reactive programming languages are twofold. We name the first shortcoming the lack of support for pattern matching, and we name the second shortcoming a mismatch between the pace inside the dependency graph and outside of the dependency graph:

No support for pattern matching Consider again the driver scenarios introduced on page 12. The plain-text description of the multiple parts that constitute a fraudulent set of transactions, specifies a *pattern* of transactions. Whether a transaction fits the pattern, depends on the amount of money transferred, but also on what other transactions took place in the two-week time window. For driver scenario A, any transaction of 100 € or more can — in isolation — match either the incoming or the outgoing transaction, or indeed even both. Yet, filtering events based on their attributes is the full extent to which reacting programming languages aid the programmer: FRP languages offer the means to exclude transactions below 100 € from the stream of transaction events, but do not aid in managing event windows — of only implicitly defined length — let alone searching within those windows.

Consider the following example — taken from one of our papers [122] — and the graphical depiction in figure 3.3: consider an event pattern consisting of the occurrence of an event of a type A, together with an occurrence of an event of a type B, and one of a type C. Let the pattern specify that

- occurrences of A and B may be up to t_{a-b} time units removed from each other;
- occurrences of A and C may be up to t_{a-c} removed; and
- occurrences of B and C may be up to t_{b-c} removed. Furthermore,
- no relative order is specified for A, B, and C; only the maximum time span.

This pattern gives rise to time windows as depicted in figure 3.3. Part 1 shows how any occurrence of an event of stream A gives rise to a time window on stream B, selecting $a - b$ time units before and after each occurrence of an event of type A. Multiple events can occur in a window, e.g., events b_1 through b_3 all occur in the time window around a_1. The multiplicity can go the other way, too, as exemplified by event b_8: since the semantic time windows can overlap, a single event can be in multiple time windows.

Reactive programming languages excel at combining one single event occurrence on a stream to another single event occurrence on another stream. Retaining events for use in different contexts, requires manual programmer intervention. This **manual pattern matcher state management** is a cumbersome and error-prone job when complex search patterns are involved.

Consider the remainder of the example: part 2 of figure 3.3 shows how each event of type A gives rise to a time window on stream C. Similarly, each event of type B gives rise to a time window on stream C. In reactive programming languages, the programmer has to manually create new data structures for each window, add events to them when the window is live, process the windows using stream operators, and discard them when they become obsolete. Whenever more than two events are involved in a pattern, the semantic time windows have to be combined, as depicted in part 3 of figure 3.3. A programmer using reactive programming languages is left to implement their own pattern matching algorithm, managing event state manually. The reactive programming paradigm fails to address the complicated parts of event correlation.

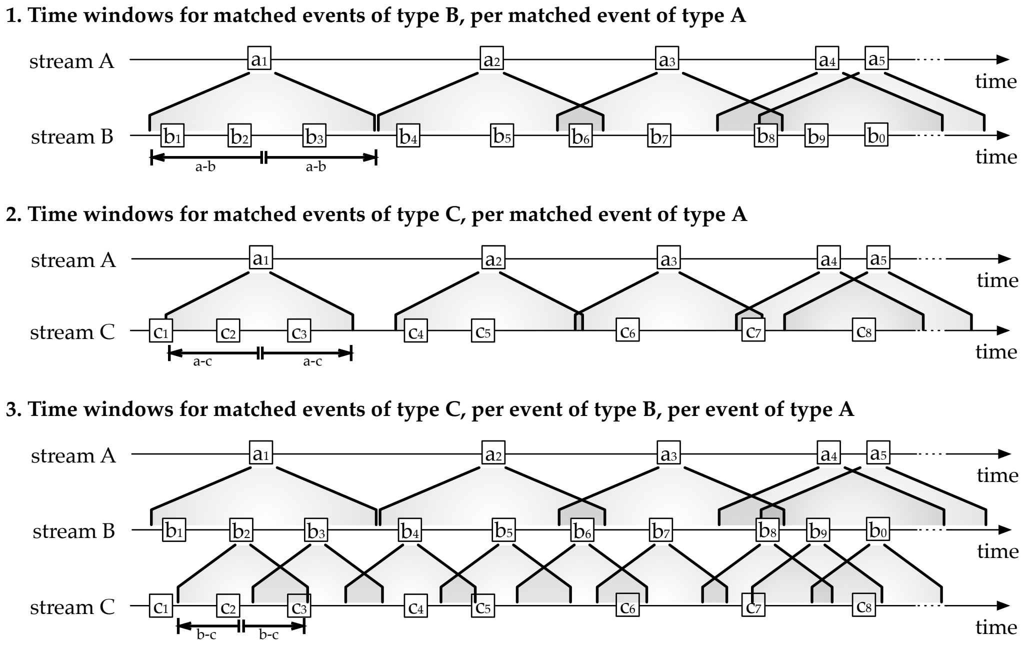

1. Time windows for matched events of type B, per matched event of type A
stream A
a1
a2
a3
a4
a5
time
stream B
b1
b2
b3
b4
b5
b6
b7
b8
b9
b0
time
a-b
a-b
2. Time windows for matched events of type C, per matched event of type A
stream A
a1
a2
a3
a4
a5
time
stream C
c1
c2
c3
c4
c5
c6
c7
c8
time
a-c
a-c
3. Time windows for matched events of type C, per event of type B, per event of type A
stream A
a1
a2
a3
a4
a5
time
stream B
b1
b2
b3
b4
b5
b6
b7
b8
b9
b0
time
stream C
c1
c2
c3
c4
c5
c6
c7
c8
time
b-c
b-c

Figure 3.3: Events matching part of a complex event pattern give rise to semantic windows on other streams — The white squares depict occurrences of events of the corresponding type, laid out on a time line. The shaded hexagons depict the extend of a semantic time window from the event on the top, mapped onto the stream below.

Pace mismatch inside and outside of the reactive dependency graph Conceptually, reactive programs are not driven by an internal clock, but by the need to react to external events. The notion that processing takes time, and with that the notion of a "processing delay", is not part of the abstraction. Mapping the concept of reactive programming onto hardware introduces a problem: on real hardware, reacting takes time. A reactive program may hence still be busy processing one event while the next event arrives. The next event will have to wait until the reactive program loops around to the part of its code that inspects the event streams or behaviors. This makes reactive programming a **leaky abstraction**: events cannot wait to occur; event occurrence is external to the system and happens without regard to the state of the reactive program.

While we think that guaranteed reactivity is the crux of reactive programming, most reactive programming languages sweep this issue under the rug. The languages assign a separate semantics to the time at which an event occurred, at which it is registered to the system, and at which it is processed. Events may occur, yet not be registered and processed. Reactive programming languages reach this point in one of two ways: they either drop some events, or introduce a buffer where they store events until the processing catches up.

Reactive programming languages which drop some events can maintain an illusion of reactivity, but only by sacrificing correctness. For instance, consider again the `seconds` behavior in FrTime [37]. If all code depending on the `seconds` behavior takes longer than a second to evaluate, FrTime's *signal manager* will skip some updates when the next second already occurred. While this enables FrTime to mitigate the issue of falling behind on the data, it does so at the cost of the guarantee that all data points are processed.

Reactive programming languages which introduce an event buffer instead sacrifice reactivity: when events have to wait to be processed by a program, that program is not reactive. Furthermore, buffering only solves transient issues; if the reaction code is consistently too slow, the buffer keeps growing and evaluation falls increasingly behind on the data. In the language model of existing reactive programming languages, the only way to resolve this issue would be to limit the size of the buffer. This introduces the problems outlined in the previous paragraph.

By ignoring the **difference in pace between the inside and the outside of the reactive dependency graph**, existing reactive programming languages risk either falling behind on the data, or having to drop events without even inspecting them. In a security context such as the one of our driver scenarios, such behavior is unacceptable. **Security monitoring systems may not drop events when they cannot keep up.** Instead, they have to be designed in such a way that they can keep up. This is obviously a hard problem, as it requires one to model the outside world as well as the reactive computation.

Shortcoming of (Functional) Reactive Programming Languages
Existing reactive programming languages do not support matching complex event patterns. Furthermore, they cannot statically guarantee they can keep up with incoming event streams.   In the absence of pattern matching, the driver scenarios cannot be implemented.

3.5.2 Shortcomings of Current Complex Event Processing

CEP languages evidently do not suffer from the first shortcoming of reactive programming languages: CEP languages offer extensive support for pattern matching. Still, they suffer from two shortcomings when used in application domains exemplified by our driver scenarios. We name the first shortcoming the semantic unclarity, and we name the second shortcoming a mismatch between the pace inside the matcher and outside of the matcher:

Unclear semantics For this shortcoming, we echo the concerns raised by Hinze and Voisard [72]. Existing CEP languages suffer from "unspecified temporal semantics", "unclear semantics" of patterns, "lack of collaboration", and "lack of adaptivity", as we explained in section 3.3.5. Even languages like Cayuga which have a formal algebraic specification [46] expose semantics which — while consistent — are counterintuitive. Evidently, Hinze and Voisard set out to solve these problems by creating EVA, an event algebra.

Future CEP languages can be specified in terms of a shared algebra such as EVA's, solving at least the lack of collaboration and adaptivity, and — by virtue of mapping onto EVA's semantics — have as clear and well-specified semantics as EVA itself. Still, some issues persist even in EVA itself. We discuss these in detail in section 4.6.2 on page 90.

Pace mismatch inside and outside of the pattern matcher Regardless of whether the pattern matcher is implemented as an automaton, search-tree, or dependency graph, the need to match the pace inside and outside of the pattern matcher remains critical. For traditional finite state automata, this is relatively simple to achieve: a state transition can be executed in a constant amount of time, and a constructed finite state automaton has a finite, constant number of states. As we explained in section 3.4.4, though, CEP languages which fit our driver scenarios' application domain do not use traditional finite state automata. Automata-based pattern matchers get rid of the requirement of having a constant number of states (or rather: automata), and keep track of the attributes of the matched events. As a result, existing expressive CEP languages do not guarantee that any event can be matched in constant time, and hence do not guarantee that the matcher can keep up with any input stream.

In conclusion, current CEP languages suffer from at least one of the shortcomings of reactive programming languages outlined above: current CEP languages either offer limited expressivity in their patterns, or fail to match the pace inside the pattern matcher to the pace of the outside world.

Overall, the core principles of CEP most closely match the requirements of our driver scenarios. Expressive CEP languages with a solid formal foundation, e.g., TESLA [41] and EVA [72], form our closest related work in many aspects.

Still, the *pace mismatch* in these languages makes them unable to guarantee that events which occur, will be matched to all relevant patterns. In part due to this distinction, more subtle differences exist in the semantics of these languages, both with respect to each other, and with respect to the solution we propose. We delay an in-depth discussion about the relevant differences to section 4.6, as we have to introduce more context before the nuances can be properly explained. For now, we summarize the shortcomings as follows:

Shortcoming of Complex Event Processing

Existing Complex Event Processing systems suffer from a pace mismatch. Because of this mismatch, they cannot statically guarantee an upper bound on their response time.

In the absence of such upper bounds, the hard requirements of our problem domain cannot be guaranteed by CEP systems.

3.6 Conclusion

In this second context chapter we surveyed the state of the art in event handling. We identified two categories of event handling languages which aim to enable programmers to write programs which depend on multiple events: (Functional) Reactive Programming, and Complex Event Processing.

We discussed how reactive programming languages do not aid the programmer in expressing patterns of complex events, and searching for matches to them in streams of events. Nor do reactive programming languages tackle the core problem of reactivity: ensuring that the pace inside the reactive dependency graph matches the pace of the outside world. CEP languages *do* aid the programmer in expressing complex event patterns and searching for them in event streams. Yet, they too do not ensure that the pattern matcher can keep up with the pace at which events occur in the outside world.

We hence conclude that — in addition to the Big Data stream processing system or streaming database discussed in chapter 2 — no event handling language exists which satisfies the requirements of live security monitoring.

4 Logic Reactive Programming

In chapters 2 and 3 we surveyed the state of the art in techniques by which large streams of events can be monitored for occurrences of complex event patterns. We defined three driver scenarios (page 12) where a financial institution monitored financial transactions for suspected cases of fraud. Using the scenarios we demonstrated that a class of event monitoring problems exist which are not catered to by the state of the art. This class of problems is characterized by the combination of *a.)* a large volume of events; *b.)* patterns involving multiple events spread out in time; *c.)* an unknown, *attacker-controlled* order of events; *d.)* the need to detect the events online, as quickly as possible after their occurrence; and *e.)* the need to guarantee that no transaction is inadvertently ignored.

Existing systems for detecting complex events in large, online event streams suffer a number of shortcomings, as discussed in the previous chapters. Stream processing systems and existing reactive programming languages fail to express the notion of searching for complex event patterns. Streaming databases and Complex Event Processing languages whose detection model is sufficiently expressive to support order-independent complex event patterns, invariably do so at a cost: these systems do not ensure that the pace inside of their pattern matcher logic can keep up with the pace at which events are generated.

In this chapter, we argue that a novel kind of reactive programming is needed. We call this novel variation *Logic Reactive Programming* (LRP). LRP aims to offer solutions to the class of event monitoring problems exemplified by the scenarios in section 2.1. LRP exists at the confluence of Big Data stream processing systems, complex event processing, and reactive programming. The next section positions LRP with respect to these precursor paradigms (section 4.1). We then introduce the PARTE language (PARTElang), a novel

Logic Reactive programming language (section 4.2). A formal foundation for PARTElang is laid in section 4.3, in which we define an event algebra for PARTElang. This event algebra formally defines what patterns are, and when a stream of events matches a pattern. We further discuss the limitations of PARTElang (section 4.5), describe future work (section 4.5), and compare PARTElang and its event algebra to related work (section 4.6).

4.1 A Programming Paradigm for Reactive CEP

We concluded last chapter with the notion that a class of event monitoring problems (see page 12) cannot be tackled by the state of the art in either streaming databases, Big Data stream processing, complex event processing, or reactive programming. In this section, we describe how concepts from these domains can be reused, and what needs to be added on top, to produce a paradigm fit for such security monitoring use cases.

4.1.1 LRP as the Combination of CEP and Behavior-based RP

Paradigm-wise, CEP comes closest to what LRP aims to provide: CEP offers ways to match complex event patterns to live streams of events. Many existing CEP languages offer either the expressivity, or the constant-time processing that is required for our driver scenarios. However, we argue that the *Callback Hell* [24] exists in the state of the art of CEP. CEP languages hence need to be augmented with reactive programming principles.

In turn, we argued in a position paper to the Workshop on Reactive and Event-based Languages and Systems workshop at SPLASH 2015 [122] that the traditional, *procedure oriented*, reactive programming languages fall short for expressing reactions to complex event patterns. We repeated this claim in section 3.5.1, where we demonstrated the complexity of matching semantic windows using functional reactive programs, in figure 3.3. A new paradigm is hence needed, which resolves the Callback Hell in CEP, or — equivalently — enables expressing reactions to complex events in a reactive programming language.

As a reactive programming paradigm, LRP could be expressed in terms of either *behaviors* or *streams*. Variables in a Logic Reactive program should hence either represent a single, current value, or a stream of values. As a CEP paradigm, LRP should use declarative rules mapping complex event patterns to reaction logic. In these complex event patterns each bound variable represents a single event. As such, the need for expressing searchable patterns dictates the choice of event representation: variables in LRP patterns bind a single event at a time, thus act as *behaviors*, not *streams*. Similarly, since LRP handles discrete events which occur in the outside world, the push-based approach to RP lends itself best.

One could thus imagine an LRP program as a behavior-based, push-based reactive program augmented with features for searching complex event patterns. Alternatively, one could imagine an LRP program as a rule-based CEP program whose reaction logic is freed from the Callback Hell by means of reactive programming principles.

4.1.2 Strong Reactivity and Event Arrival Rate

We discussed in section 3.5 how current event processing techniques fail to align the pace inside of the event processing system with the pace at which events occur in the outside world. They (sometimes implicitly) distinguish between the time at which an event occurs, the time at which the occurrence is reported to the system, and the time at which the system determines which patterns the event matches. Whenever the event processing system has finished processing an event by the time a new event arrives, this distinction is invisible. However, if the system has not finished processing an event by the time a new event arrives, processing of that new event has to wait until the event processor is ready.

The concept of time between occurrence and detection of occurrence is incompatible with the semantics of reactivity. Worse, if the event processor continuously takes too long to be ready for a next event, the processing increasingly falls behind on the event stream, and is forced to either drop events, or fail as it runs out of storage space.

This makes reactive programming a leaky abstraction: in the abstraction, events cannot wait to occur; the occurrence of an event is external to the system and happens without regard to the state of the reactive program. We use the term *strong reactivity* to refer to a type of reactivity where this leak in the abstraction cannot take place.

Definition 1: Strong reactivity

A reactive program is *strongly reactive* if the abstraction of reactivity is never broken, i.e., if the program always reacts, immediately, whenever an event occurs.

Strong reactivity can in general only be achieved when the computer on which the program runs is arbitrarily fast, or if the program can execute arbitrarily many tasks in parallel. However, if an upper bound is known on the rate at which new events arrive, a program can be *strongly reactive* on a real computer. To this end, the program must guarantee to fully process an event in a constant amount of time and space, independent of the number of events that occurred before. With this guarantee, the program is never not ready for the next event, as long as the constant upper bound on processing time is not larger than the minimal time between events arrivals.

We define *event arrival rate* as follows:

Definition 2: Event arrival rate

The rate at which events arrive at the inputs of a reactive program, i.e., the *event arrival rate* is expressed as a number of events per time unit.

The event arrival rate is an absolute upper bound, not an average upper bound. The event arrival rate is defined by the minimum time between subsequent events.

This definition imposes the requirement that events are *periodic*. This restriction is not excessive, in fact it is the norm in real-time systems. Mellin [101] writes in his PhD thesis that "[no] system can handle a completely random load. *Aperiodic* events, that is, events whose interarrival time is unspecified, are not tractable in a deterministic way in terms of resources needed to handle them".

4.1.3 LRP means Strongly Reactive Complex Event Processing

With definition 1 formulated, we can now define the next deviation from the state of the art that constitutes LRP: existing Big Data stream processing and event processing techniques do not enforce strong reactivity.

While filter conditions tend to operate in constant time, stream processing system often do not enforce this. The systems where constant time processing is enforced, do so by only supporting limited means to specify event patterns, e.g., in the form of regular expressions. Other models used for CEP are more expressive, but do so by exposing a pattern language which can express programs that require arbitrarily much time to detect, and may require arbitrarily large storage for intermediary results. The application logic invoked in response to a pattern, too, undermines strong reactivity. Often times [142, 118] the language exposed for writing reaction logic in, is even Turing-complete. In Complex Event Processing, where the detection of an event pattern can cause a new complex event to be emitted, this emission-reaction-cycle inherently constitutes an unbounded loop when patterns may emit events of a type they may react to.

All these concerns with the state of the art explain why we consider strong reactivity to be a core feature of the Logic Reactive Programming paradigm. At the surface level, a declarative pattern language as used in streaming databases or rule-based CEP languages is the best fit for expressing the scenarios described on page 12. The need for strong reactivity however necessitates that LRP imposes a number of restrictions on the patterns it can express. We achieve this by means of three restrictions. First, the pattern language must be restricted in such a way that only those patterns can be expressed which match a fixed, finite number of other events for each event. Second, the depth of the matching process must be restricted in such a way that only a fixed, finite number of pattern matching steps are required for each complex event. Third, no other side-effects can be created, except emitting new complex events from the reaction logic of a rule. We use the high-level nature of a declarative, rule-based language to enforce these restrictions: we make it statically impossible to express LRP programs which violate these restrictions. We discuss both cases in more detail:

Restricted pattern language In LRP, a program can only be expressed as a set of declarative rules. Each rule consists of a pattern — matching one or more events — and a set of expressions implementing the reaction logic to execute whenever a match to the pattern is found. Matching events to a pattern can be done in time proportionate to the number of events and to the number of events involved in a single *match* to a pattern. To achieve strong reactivity, LRP must hence enforce that only a fixed, finite number of events can be involved in a pattern for each event, and that only a fixed, finite number of events match a pattern at a time.

The latter is easily enforced syntactically: a pattern binds events to logical variables. Each logical variable binds a single event at a time (cf. a *behavior*, as discussed in section 4.1.1), and each logical variable has to appear in the source code of the LRP program. The source code of any LRP program is of a finite size, and is of a size which does not change during the run time of the program. By transitivity, a pattern hence binds only a fixed, finite number of events per match to a pattern.

To achieve the former, we make use of the fact that the *event arrival rate* (definition 2) is finite and constant. Since a fixed, finite upper limit exists on the number of events arriving in a certain time interval, any fixed, finite time window contains only a fixed, finite number of events. If LRP requires that all patterns define a (semantic) time window for each event, the patterns necessarily enforce that for any event, only a fixed, finite number of other events can be involved in a match. It suffices that for any two logical variables in a pattern a maximum distance in time between both can be determined.

Restricted matching depth If an LRP program contains a rule which emits a complex event in response to an event, and if the type of event that rule responds to is the same as the type of event that rule emits, the LRP program encodes an unbounded loop. Unbounded loops are trivially incompatible with strong reactivity, and must hence be made impossible to achieve in LRP. To remove the option of introducing cycles, while still retaining as much of the expressivity as possible, rules must be disallowed to emit events of types which they may react to, either directly or indirectly. We define the notion of *stratification* as a requirement for LRP programs:

Definition 3: Stratification

Let p be a Logic Reactive Program p.

The program p is *stratified* if there exists a numeric *stratum* for each type of event which appears in p, such that if the occurrence of an event e_a of type t_a can cause p to emit of a complex event e_b of type t_b, then the stratum $s_a \in \mathbb{N}$ of type t_a is strictly lower than the stratum $s_b \in \mathbb{N}$ of t_b.

Intuitively: an LRP program is stratified if no event can influence the later occurrence of an event of the same type. Or, from the perspective of declarative rules: an LRP program is stratified if no rule can emit events which cause complex events to occur which match the rule's pattern.

Restricted side-effects Any expressions in an LRP language, other than the *stratified* emission of new complex events, must be free of side-effects. Filter conditions in the patterns — e.g., filtering out financial transactions involving less than 100 € — as well as transformations — e.g., adding 10% to a value — can be restricted to purely functional predicates that operate in constant time and space.

> The application logic invoked whenever a match is found must be restricted to only emitting new complex events. This event emission must of course satisfy the stratification requirement.

Even with those three restrictions put in place, the Logic Reactive Programming paradigm would not yet be a strongly reactive Complex Event Processing paradigm. One key part is missing: to handle the driver scenarios sketched in section 2.1, LRP programs should be stateful. After all, if a program does not store previous events, it cannot correlate newly arrived events with previous events, as it no longer has access to those events. However, searching a history of previous events for matches to patterns involving new events takes time proportionate to the size of the event history. Existing solutions — either streaming databases or rule-based CEP languages — do not guarantee that a fixed, finite limit exists on this history.[1] In existing solutions, enumerating all matches may hence take arbitrarily long, preventing strongly reactive operation.

To alleviate this problem, an LRP program must ensure it stores only a fixed, finite amount of previous events. To do this without impacting the semantics of the program, i.e., without changing which complex events get detected, the LRP paradigm must ensure that no LRP programs expresses a pattern which requires arbitrarily many events' data to be retained. We wrote earlier that we restrict the pattern language allowed in LRP programs in such a way that the amount of relevant earlier events for any event is fixed and finite. Since time moves forwards, and events must be related by relative time windows, only a fixed, finite number of events can have any bearing on future matches.

LRP languages must leverage this property, and automatically remove data pertaining to such events which are too old to still have any bearing on future matches. This should not be a responsibility of the LRP programmer: the language runtimes of LRP languages must be able to analyze the relative temporal windows expressed in the LRP program, and extract all information necessary to manage the event storage. The system should determine which events are still relevant, and remove those which are not as soon as possible. A finite upper limit should exist on the number of irrelevant events which can accumulate before being removed: together with the fixed, finite upper limit on the number of potentially relevant events, this forms the fixed, finite upper limit on the state size of an LRP program's knowledge base.

A program for which no such upper limit can be determined, is an invalid LRP program. Such a program must be rejected by the LRP language runtime before monitoring would start.[2]

[1]Of course, whenever a streaming database or CEP language maintains only a fixed-length (tumbling or sliding) window, the event history is limited by the size of the window, which can be fixed and finite. In cases with more expressive patterns, though, data is retained arbitrarily long.

[2]While we do not explicitly present the requirements on the temporal constraints as a type system, it serves a similar purpose. Syntactically correct programs may still be rejected prior to runtime based on the static conformance check.

4.1.4 LRP means Distributed Processing

With the restrictions from last paragraphs put in place, Logic Reactive Programming enables a strongly reactive form of CEP. But to be really useful in a Big Data context, it should additionally be able to utilize the processing power of a cluster of computers. This entails that the algorithms underlying LRP evaluation may not assume a shared memory space, nor may they assume that any part of the program can inspect any part of the program's state. Synchronization between different distributed components must be introduced, and *glitches* (as introduced in section 3.4.4) must be prevented in a distributed context, where topological sorting no longer necessarily suffices [124].

LRP languages must define their own approach for guaranteeing correct and glitch-free exchange of data among multiple distributed computers. LRP has an advantage over other RP paradigms, though: LRP programs must specify temporal constraints, and data in LRP programs must be tagged with a timestamp. These timestamps can be used to order the interleaving of messages exchanged between the distributed computers. As we will show in chapter 5, the workload of an LRP language runtime can be divided across multiple distributed computers in such a way that the automatic removal of old events (described near the end of section 4.1.3) can be done using only local data and a small constant amount of meta-data added to the messages that are sent to propagate events through the LRP program. As such, LRP programs can keep their internal state consistent in a distributed setting.

Keeping the externally visible behavior consistent in a distributed setting, requires another set of restrictions. Non-determinism in scheduling across multiple computers should not impact which matches are detected. We therefore impose the requirement that matches in LRP must be atomic: a pattern is either fully matched, or fully unmatched. No interleaving of processing should ever allow a rule to find some events generated in response to the matching of a rule, but not find some other events generated by the same matching of that rule. Consider for instance an LRP program which contains two rules: the first rule emits a complex event of type A, and a complex event of type B when the rule is matched. The second rule matches on the occurrence of an event of type A and the absence of an event of type B. Consider the case where the evaluation of the first rule reached the point where it emitted an event of type A, but not yet the point where it emitted an event of type B,[3] and consider that at that point the evaluation of the second rule commences. The LRP program would momentarily find an A but not B, and hence erroneously trigger the second rule. LRP prevents such glitches by combining three features we have already introduced: stratification, correlation using temporal reasoning, and semantic windows which automatically remove old events' data, but retain them while they may have an impact on the result:

- first, thanks to stratification, an LRP program may only react by emitting new events of types to which it itself does not react. This prevents feedback loops, and makes it

[3]Or if only the emission of A had been propagated to the other computers in the cluster, and the emission of B was still ongoing.

possible to accurately determine lower bounds on which timestamps future events of a certain type may still have (we define this in detail later, for instance from section 5.3.3 onwards);

- second, since all correlation of events includes verifying how the events correlate in time — based on the timestamps they are explicitly tagged with — no combination of events which does not satisfy the pattern is detected; and
- third, since events are guaranteed to be retained while they are in a semantic window — based again on their explicit timestamps — all combinations of events which do match the pattern are detected.

4.1.5 Requirements for a Logic Reactive Programming Language

In this section we have discussed how a novel Logic Reactive Programming language would differ from the state of the art in distributed stream processing, streaming databases, (functional) reactive programming, and Complex Event Processing. We conclude this section by synthesizing the hard requirements for the Logic Reactive Programming paradigm.

R_{LRP}1 First, it is evident from the driver scenarios introduced in section 2.1 that LRP languages must enable **expressing patterns of events**. It must be possible to relate events to each other. The attributes of the events — e.g., the timestamp of a financial transaction, or the destination account — should be usable in filter conditions or as join-criteria. Beyond just using the attributes directly, a limited set of arithmetic operations should be exposed. In the driver scenarios, this ability is required to sum up the amount of money transferred, and to compare that sum to the amount carried by another transaction. It must also be possible to abstract over and consult additional domain knowledge, for instance historical information identifying trusted accounts. Negated patterns, such as the absence of a fraud claim, must be supported too.

LRP programs should of course be able to communicate successful matches to the patterns to some external system. To this end, the language should be able to express **reaction logic** as the consequent of matching a pattern. The reaction should be a natural extension of the conditions: the syntax should be similar and the conditions and reaction logic should share the same namespace and scoping rules. The reacting logic must execute in constant time, using a constant amount of space, to allow for strong reactivity throughout the system. Reaction logic is the only part of an LRP program which can cause side-effects in the outside world. It may only do so by emitting new compound events which abstract over the matching of a set of events to a pattern.

R_{LRP}2 Second, the language must provide the means to **relate events in time**. Events — by definition — have a temporal component. When the time of occurrence is modeled as a single *timestamp* attribute, this entails that each event has an attribute which is non-decreasing over successive emissions of events of the same type. LRP languages must provide "eventually followed by" semantics (as explained on page 48) for event sequences.

We refer to filters and join-criteria involving events' timestamps as *temporal constraints*. LRP languages must be able to express temporal constraints such as (semantic) time windows (as explained on pages 27 and 50). The temporal constraints must be validated in constant time per event, using constant space for the whole system, to allow for strong reactivity throughout the system.

R_{LRP}3 Third, our driver scenarios require a specific event retention scheme. It is important that events are not *consumed* when they match a pattern (i.e., a zero-consume policy must be used, as explained on page 47). Consider the driver scenarios: a transaction that seems to be part of a fraud combined with some other transaction, may also be suspicious when coupled with another transaction. Notifying the security team in only one of both cases, would lead to false-negatives.

At the opposite side of the same concern, events may not be retained indefinitely. Retaining events indefinitely implies that memory consumption grows without bounds, which precludes strong reactivity. Only part of the event history can be preserved. All the driver scenarios offer a way to steer clear of this issue, though: all patterns imply a duration for which events stays relevant, but events eventually becomes irrelevant. At that point, they can automatically be purged from the system. In LRP, the part of the event history that is still relevant must be automatically determined from the temporal constraints specified in the program source. The LRP language — not the programmer — is responsible for discarding irrelevant old events. We refer to this behavior as **managed event storage**. A program for which the storage cannot be managed automatically is an invalid LRP program, and should be rejected by an implementation of a LRP language.

R_{LRP}4 Fourth, to maximize the chance of responding to the attempted fraud in time, the **detection has to happen as soon as possible**. This precludes e.g. solutions based on running a processing batch at night, or even hourly, in favor of a streaming approach. Note that despite the online, streaming semantics, the detection must be able to correlate new events with events that took place some time earlier, necessitating some way of storing historical data, with the constraints imposed by the previous requirement. Since the number of transactions which must be cross-referenced in the driver scenarios may grow large, the system responsible for detecting the LRP patterns must be **high-performance and scalable**.

R_{LRP}5 Fifth, while the detection runs on trusted hardware, the data it operates on is generated by the outside world, and must be treated as potentially adversarial. It is in the adversary's interest to bring down the detection system, so the detection system must be hardened against this. The system should not have any inherent weaknesses (such as e.g., a resource consumption exponential in the number of non-fraudulent but suspicious transactions, which would allow a cheap DDoS with plausible deniability), and it must be possible for the entity deploying the logic reactive program (in the driver scenarios: the financial institution) to determine up front how high a load the detection system will be able to handle, such that a sufficient amount of resources may be allocated for it. Using the terminology introduced in definition 1, LRP should be **strongly reactive**.

It is self-evident that the requirement of bounded execution time is irreconcilable with Turing completeness. We consider this a strength of LRP: while there are programs which cannot be expressed in the LRP paradigm, all programs that can be expressed in LRP are guaranteed to be strongly reactive, and hence dependable: they will never fall behind on their inputs, nor run out of working memory.

We define the Logic Reactive Programming paradigm as follows:

Definition 4: Logic Reactive Programming

Logic Reactive Programming is a programming paradigm characterized by the following five requirements:

R_{LRP}1 LRP programs are expressed as declaratively specified patterns of events, together with reactions.

R_{LRP}2 LRP languages provide built-in support for temporal reasoning over those events.

R_{LRP}3 LRP language runtimes automatically manage the storage of events.

R_{LRP}4 Deployed LRP programs are scalable and online.

R_{LRP}5 Deployed LRP programs are guaranteed to stay online, and offer strong reactivity.

To enforce reactivity, all parts of the program — including the reaction logic — must take a worst-case constant execution time. If a part of the execution requires time proportional to the size of some data structure, that data structure must have a constant upper limit on its size. The central premise is that arrival of events should never have to wait because the system is still processing the previous events. Informally, if a certain cluster of physical computers cannot process a logic reactive program fast enough, it should be possible to fix the problem by buying a faster computer. There should exist a predictable, fixed, finite number of instructions-per-second such that a computer running at that speed can execute the LRP program without falling behind on the event stream.

4.1.6 Conclusion

A Logic Reactive Programming language compares to the state of the art of event stream processing as depicted in table 4.1:

	Distributed Big Data & Stream Processing (Chapter 2)		Event Handling (Chapter 3)		Our Contribution
	Big Data Stream Processing	**Streaming Databases**	**Complex Event Processing**	**Reactive Programming**	**Logic Reactive Programming**
Throughput	✓✓ very high	✓ high	~ average to low	~ average to low	✓ high
Cost per update	✓ constant (depth of program)	– proportionate to history	– proportionate to history	✓ constant (depth of program)	✓ constant (depth of program)
Bounds on resource usage	✓ yes (with basic operators)	– proportionate to history	– proportionate to history	~ possible, assuming no loops etc. in program	✓✓ yes (guaranteed)
History	– none	✓✓large history size	✓ some history	– none	✓ some history
Pattern matching	– unsupported	✓ supported (relational join)	✓✓ expressive pattern language	– unsupported	✓ restricted, expressive pattern language
Reaction logic	– often integrated, but Callback Hell	– outside of the model	– outside of the model, or Callback Hell	✓✓ well-supported (core of the paradigm)	✓ supported (with restrictions)

Table 4.1: Comparison between the state of the art of event stream processing and LRP

	Distributed Big Data & Stream Processing (Chapter 2)		Event Handling (Chapter 3)		Our Contribution
	Big Data Stream Processing	**Streaming Databases**	**Complex Event Processing**	**Reactive Programming**	**Logic Reactive Programming**
$R_{LRP}1$	– no patterns – Callback Hell	✓ relational join – outside of the model	✓✓ expressive patterns – Callback Hell	– no patterns ✓✓ well-supported	✓ restricted patterns ✓ supported (restricted)
$R_{LRP}2$	– unsupported	✓ supported	✓✓ well-supported	– unsupported	✓ supported
$R_{LRP}3$	~ no storage	✓✓ large managed storage	✓ managed storage	~ no storage	✓ managed storage
$R_{LRP}4$	✓✓ very scalable	✓ scalable	– not scalable	– not scalable	✓ scalable
$R_{LRP}5$	~ not guaranteed	– unsupported	– unsupported	~ not guaranteed	✓ guaranteed

Table 4.2: Requirements attained by event stream processing systems and by LRP

- Compared to Big Data stream processing systems, LRP languages preserve the history-independent cost per update. The complexity of pattern matching is greater than that of stream processing, which will have some negative impact on the throughput.

- Compared to streaming databases, LRP languages preserve the ability to keep historical data, albeit in a more restricted way. This restriction is not unwanted, though: it allows for imposing upper bounds on resource requirements.

- Compared to CEP languages, LRP languages preserve the ability to keep historical data. LRP additionally allows imposing upper bounds on resource requirements. This comes at a cost in language expressivity.

- Finally, compared to (F)RP languages, LRP languages preserve the property that the cost per update is independent of the history. LRP retains the software engineering benefit of preventing the *Callback Hell*, and adds support for pattern matching.

An alternative view is offered by table 4.2, which depicts which requirements of LRP are met by the state of the art of event stream processing.

The remainder of the dissertation defines and evaluates a language and evaluation model which enables these constraints.

4.2 A Logic Reactive Programming Language: PARTElang

In this section, we introduce PARTElang,[4] a novel Logic Reactive Programming language. PARTElang is aimed at the distributed Big Data context sketched in chapter 2, though its syntax and semantics can be applied more generally. We use PARTElang to demonstrate that the ideas of LRP can be put to practice.

After introducing the language itself in this section, section 4.3 defines an *event algebra* for the matching behavior of PARTElang. Chapter 5 defines an operational semantics of PARTElang programs. Chapter 6 describes a prototypical implementation of a PARTElang runtime. Chapter 8 proves correctness of the operational semantics. Finally, in chapter 7 we explore expressing the driver scenarios in PARTElang and executing them on our prototypical implementation.

4.2.1 Informal Semantics

This section gives a brief overview of the PARTElang semantics. The precise details are discussed further in this dissertation.

As an LRP language (see the requirements listed on page 72), PARTElang needs to be able to correlate large amounts of complex events, incrementally, as they occur ($\mathbf{R_{LRP}}\,\mathtt{4}$), with guarantees on availability ($\mathbf{R_{LRP}}\,\mathtt{5}$). The correlation to compute — and the reaction logic to enact — must be specified declaratively ($\mathbf{R_{LRP}}\,\mathtt{1}$ and $\mathbf{R_{LRP}}\,\mathtt{2}$). We opted for a language design centered around declarative rules, but with an automatically managed knowledge base ($\mathbf{R_{LRP}}\,\mathtt{3}$). At the top level, a program in PARTElang contains two kinds of items: definitions of the types of events the program can reason about, and named *rules* that reason about events of those types.

Events and Event Templates

Event types are defined by means of *event templates*.[5] Event templates are named, and identify a *relation* (see definition 5). Event templates specify a list of *attribute names* and their type. For instance, an event template could be named "MoneyTransferred", and could have attributes such as "amount" and "destination_account". The types of those attributes could be "Euros" and "AccountId".

Each event that is inserted into the system must be tagged to identify the template it conforms to, and the attributes of the event must match those of the template. For instance, a MoneyTransferred event could be inserted with the amount attribute set to the constant value 100.00 € and the destination account set to some account number.

[4]The acronym "PARTE" used to stand for "Parallel, Actor-based ReTe Engine" in an earlier iteration, but that description no longer applies. "PARTE" is now just a name.

[5]This terminology is borrowed from the logic programming language/expert system CLIPS [142].

Declarative Rules

Declarative rules give a name to a pattern and some reaction logic. The pattern of a rule is a conjunctive query: each sub-pattern expresses the constraints on an event that need to be present or the conditions on an event that may not be present for the rule to match. Events such as money transfers or reports of fraudulent transactions can be bound to *logical variables*. Only events themselves can be bound to variables. A variable cannot bind a Euro amount, an account identifier, a name, etc. Those are instead reached by taking the value bound to an *attribute* of an event bound to a variable.

Constraints can filter out events based on their types, or on the values of their attributes. Attributes of events bound to logical variables can be compared across events, e.g., a conjunction of patterns might match on the occurrence of a MoneyTransferred event bound to `t` and an AccountClosed event bound to `c`, where `t`'s originator account is equal to the account closed in `c`, and where `c` occurred within a day of `t`. We refer to constraints which compare across event boundaries as *join-criteria*, and to other event-local constraints as *filters*. Join-criteria enforcing only equality are functionally equivalent to *unification*.

Absence of an event meeting certain criteria is modeled by *negated patterns*. While *positive patterns* enforce the presence of events matching some conditions, *negated patterns* require the absence of events. A negated pattern implements a time-aware form of *negation-as-failure*: an event is considered not to have occurred if and only if time progressed beyond the latest point at which it could have occurred, and it is not yet known to have occurred. Critically, the point beyond which an event can no longer occur is based solely on its timestamp: an LRP program cannot determine that e.g. no more MoneyTransferred event will occur, but it can determine that no more MoneyTransferred event with a timestamp t_m can occur if all event sources declare they have sent all events up to t_s, where $t_m < t_s$.

PARTElang upholds the invariant that an event that was determined not to have occurred cannot later be found to have occurred. Universally quantifying that no event matching a certain pattern will ever take place, cannot possibly be done in constant time — evaluation would have to wait indefinitely for the occurrence of the event. Therefore, a negated pattern must always express a fixed-size time window relative to at least one positive event. The time window is delimited by a finite minimum and a maximum duration relative to the positive event. The minimum duration may be negative, thereby including events gathered before the occurrence of the positive event.

Disjunction of Rules

While enforcing more than one constraint at a time is expressed by having multiple patterns in a rule (a conjunctive query), choice is expressed by having multiple rules. Applying the approach from previous rule-based systems, all rules of a PARTElang program are regarded as being in an implicit disjunction. Arriving events are matched with every rule,[6] i.e., events satisfying multiple rules' pattern, match all those rules.

[6]An implementation can optimize this by tracking which rules have patterns for the which event template.

Reactive Knowledge Base

To ensure strong reactivity, both the pattern matching and the reaction logic of PARTElang program execute in constant time and space. Determining whether or not the conditions are met, takes a constant amount of time per new event and uses a constant amount space for the whole system. Expressions are restricted to a non-Turing complete set of arithmetic expressions. Data may be transformed, but no constructs exist in PARTElang to, e.g., construct lists.

In general, the data structures available to the PARTElang programmer are purposefully limited to guarantee bounded runtime. As a Logic Reactive Programming language runtime, the PARTElang runtime is responsible for *managed storage*. The rule programmer [7] needs not be able to indicate which events are to be preserved, and which are to be discarded. PARTE programs consequently need only limited support for data structures. The only available data structures in PARTE programs are event templates, which are non-recursive records. Output streams and intermediary results are handled behind the scenes, and have no syntactic representation. Input streams are represented by a *behavior*-like approach: for each interpretation of the pattern, variables in the positive patterns of the rule bind exactly one event.

Mandatory Temporal Constraints

Each rule which binds more than one variable must impose temporal constraints on the events it binds. Temporal constraints impose a minimum and maximum difference in time between events bound to two variables bound in the rule's patterns. These temporal constraints are obviously enforced on events — as the temporal reasoning component of the join-criteria — but they also serve as the basis of reachability analysis used to manage storage, i.e., the temporal constraints specify *semantic windows*. The timestamps of bound events are additionally used to determine the timestamp of newly emitted events.

Reaction Logic

Finally, reaction logic identifies the template of an event to emit, an expression to compute the new event's timestamp, and a set of assignments of values to attributes. PARTElang does not define any other form of reaction code, though practical systems can expose the occurrence of a *rule matching* (see section 5.3.5), or emit events to an external destination.

4.2.2 Structure of a PARTElang Program

The grammar of PARTElang is depicted in figure 4.1.

[7]By "rule programmer" we refer to the programmer writing rules in PARTElang, to be executed on some implementation of the PARTE model.

```
<prog>  ::= <templ>+ <rule>+

<templ>::= template <ident> { <slot>* }
<slot>  ::= <ident> : <btype>
<btype>::= Integer | String | …

<rule>  ::= rule <ident> where
               <ppat> <pat>*
            when
               <Δθ>*
            then
               <rexpr>*
<pat>   ::= <ppat> | <npat>
<npat>  ::= no <ppat>
<ppat>  ::= <ident> : <ident> { <cond>,* }

<cond>   ::= <expr> <relop> <expr>
           | <pred> "(" <expr>,* ")"
<rel_op> ::= "=" | "≠" | "<" | ">" | "≥" | "≤"
<expr>   ::= (<ident> .)? <ident>
           | <constant>
           | <expr> <op> <expr>
           | <func> "(" <expr>,* ")"
<op>     ::= <rel_op> | "+" | "-" | …
<func>   ::= min | max | avg | …
<pred>   ::= <func>returning a boolean
<Δθ>     ::= <pΔθ> | no <pΔθ>
<pΔθ>    ::= <ident> in <ident> [<θ>, <θ>]
<θ>      ::= <number> (ms | seconds | minutes | …)
<rexpr>  ::= emit <ident> at <expr> with { <assign>,* }
<assign> ::= <ident> = <expr>
```

Figure 4.1: The formal grammar of PARTElang — Nonterminals are typeset in bold and enclosed in pointy brackets.

At the top level, a program in PARTElang consists of a number of template definitions `<template>`, followed by rule definitions `<rule>`. Template definitions identify the kinds of events the rules can reason about, e.g., "MoneyTransferred" or "AccountClosed". They are named using an identifier `<ident>`, and list a set of slots. Each slot maps an attribute name `<ident>` — e.g., "account_balance" — to a base type `<btype>`, e.g., the set of whole numbers representable in 64 bits.

Rules are named (using an identifier `<ident>`), and list a `where`-clause, a `when`-clause, and a `then`-clause.

The `where` clause is an implicit conjunction of patterns `<pat>`. Patterns exist in two varieties: negated patterns `<npat>` and positive patterns `<ppat>`. Positive patterns specify that an event matching the pattern must be present for the rule to match. Negated patterns specify that no event matching the pattern may be present for the rule to match. Syntactically, negated patterns take the form of a positive pattern, with the constant "`no`" prefixed. All patterns identify a variable and a template, as well as a set of non-temporal constraints `<cond>`. For instance, a pattern could bind the identifier `incoming_transaction` to all events of template "MoneyTransferred", where the amount of money involved in `incoming_transaction` is greater than 100 €, and `incoming_transaction` occurred within two days of an event `account_closed` which was bound in another pattern.

The scope of variables starts at the pattern in which they are bound. For positive patterns, the scope extends to the end of the rule, i.e., for the remainder of the patterns and for the entire span of the reaction logic. Variables bound in negated patterns are only in scope in the pattern which binds the variable, and when expressing temporal constraints for that variable in the `when` clause. Accessing attributes of the matches to negated patterns outside of their scope is akin to accessing attributes of events which have been proven absent, and is hence an error. Because of this, inter-event constraints involving negated patterns must be expressed in the negated pattern, and the positive patterns must hence syntactically appear before the negated pattern. Note again that this does not mean that the negated patterns must refer to timestamps before those of events bound the positive patterns: the implicit conjunction of patterns does not imply causality or temporal order.

Non-temporal constraints relate two expressions `<expr>` by means of a relational operator `<relop>`. Expressions either

- identify an attribute of the event bound in that pattern using an `<ident>`, e.g., `amount` yields the value of the amount attribute of the event;
- identify a previously bound variable and an attribute of the event bound to that variable using the syntax `<ident>.<ident>`, e.g. `account_closed.account_id` yields the account number attribute of the event previously bound to the logical variable `account_closed`;
- name a constant `<constant>`, e.g., 0, 1, or 0.3 yield those numeric values;
- apply an arithmetic operator to two subexpressions using `<expr> <op> <expr>`), e.g., `amount * 0.9` yields the value of the amount attribute of the current event, scaled by 90%; or
- apply an arithmetic function to a comma-separated set of subexpressions using `<func> "(" <expr>,* ")"`, e.g. `avg(a.amount, b.amount)` yields the average of the amount attributes of the events bound to the logical variables `a` and `b`.

The `when` clause of a rule contains temporal constraints <$\Delta\theta$>. Temporal constraints identify two variables and an interval of minimum and maximum difference in time <θ> between the events bound to both variables. Any common unit of duration can be used here, PARTElang itself currently supports milliseconds ("ms"), seconds, minutes, hours, and days. To specify that, e.g., the event bound to logical variable `a` must occur within two days before or after the event bound to `b`, one could write `a in b [-2 days, 2 days]`. Syntactically, temporal constraints are prefixed by the constant "`no`" if the event on which they are specified, is bound in a negative pattern. For instance, if a rule specifies no event `a` may exist, temporal constraints relative to `a` will take the form `no a in ... [...]`.

Finally, the `then`-clause contains reaction code `<rexpr>`. Reaction code identifies the template of an event to emit, an expression to compute the new event's timestamp, and a comma-separated set of assignments `<assign>` from the results of expressions to attributes. PARTElang does not define any other form of reaction code, though practical systems can expose the occurrence of a *rule matching* (see section 5.3.5), or emit events to an external destination.

4.3 An Event Algebra for Logic Reactive Programming

This section introduces a formal foundation for the PARTE language in the form of an *event algebra*.[8] An event algebra is an algebraic notation for relational event patterns. Examples of event algebras are the works by Hinze and Voisard [71], or their later event algebra for EVA [72]. Event algebras encode relational operators over events and their attributes in a clean, algebraic form. This makes them less suited for direct use as a programming language, but a better fit for formal proofs.

We show how to map patterns expressed in PARTElang onto equivalent expressions in the event algebra in section 4.4. Proofs of correctness for PARTElang can then be expressed in terms of the event algebra defined in this section.

4.3.1 Definitions of Base Concepts

To start, we must define the base concepts used in our algebra. Building on the common definitions in database literature and CEP literature, we define relations, schemas, values, and streams. Let **A** be the countable infinite set of *attribute names*, let **K** be the countable infinite set of *constants* (i.e., all numeric constants, timestamps, etc. that can appear in a rule), and let **V** be the countable infinite set of *variables*, such that **A**, **K** and **V** are pairwise-disjoint.

Definition 5: Relation

A *relation name* is a set of attribute names.

Let **R** be a relation name. A *tuple in* **R** is a function $t : \mathbf{A} \to \mathbf{K}$ that maps each attribute in **R** to a constant.

The set of all tuples in **R** is denoted by $\mathrm{val}(\mathbf{R})$.

For instance, consider a relation named "MoneyTransferred". "MoneyTransferred" consists of the attribute names "amount", "destination_account", etc. The set of all tuples in "MoneyTransferred" is denoted val(MoneyTransferred). Each tuple in val(MoneyTransferred) maps the attribute "amount" to some value, e.g., 75.0 €, maps the attribute "destination_account" to some account number, etc.

[8]This event algebra is based on joint work with Martin Ugarte (Université Libre de Bruxelles). An earlier version of this event algebra was delivered as deliverable 1.4a of the "Scalable Processing and mIning of Complex Events for Security-analytics" (SPICES) project, an Innoviris Bridge project.

Definition 6: Schema

A relational *schema* $\mathcal{R}$ is a set of relation names.

The set of tuples in $\mathcal{R}$, i.e., $\bigcup_{\mathbf{R} \in \mathcal{R}} \text{val}(\mathbf{R})$, is denoted by $\text{val}(\mathcal{R})$.

A schema could for instance be { MoneyTransferred, AccountClosed, TransferFraudDetected }.

The set of *types* in a schema $\mathcal{R}$ is defined recursively as follows:

Definition 7: Type in a schema

Let $\mathcal{R}$ be a schema.

The set of types in $\mathcal{R}$ is denoted by $\text{type}(\mathcal{R})$, and consists of all of the following:

$$\begin{cases} \textbf{a base type, } \text{i.e, } \texttt{base} \text{ is a type in type}(\mathcal{R}) \\ \textbf{relation names as types, } \text{i.e., } \forall \mathbf{R} \in \mathcal{R} : \mathbf{R} \in \text{type}(\mathcal{R}) \\ \textbf{mapping types } \{x : \tau\}\textbf{, } \text{i.e., } \forall x \in \mathbf{V}, \forall \tau \in \text{type}(\mathcal{R}) : \{x : \tau\} \in \text{type}(\mathcal{R}) \\ \textbf{unions } \tau_1 \cup \tau_2 \textbf{ of two types mapping over disjoint sets of variables, } \text{i.e.,} \\ \quad \forall \text{ mapping types } \tau_1, \tau_2 \text{ in } \mathcal{R} : \text{dom}(\tau_1) \cap \text{dom}(\tau_2) = \emptyset \implies \tau_1 \cup \tau_2 \in \mathcal{R} \end{cases}$$

Unions of *mapping types* are hence themselves *mapping types*.

Using our previous example, the relation name “MoneyTransferred” is a type in a schema that included the relation name “MoneyTransferred”. Additionally, a mapping from some variable `incoming_transaction` to the relation name “MoneyTransferred” is a type in the schema, i.e., {`incoming_transaction` : MoneyTransferred} is a type in the schema. A mapping from some variable `accound_closed` to the relation name “AccountClosed” is a type in that schema too, and thus the union of both us a type in the schema. The base type, `base`, is always a type, in any schema. The base type abstracts over the types of constants, as shown by the following recursive definition of the set of *values of a type*:

Definition 8: Values of a type

Let τ be a type. The set of values of τ is denoted by $\text{val}(\tau)$, such that

$$\begin{cases} \text{val}(\tau) = \mathbf{K} & \text{if } \tau = \texttt{base} \\ \text{val}(\tau) = \text{val}(\mathbf{R}) & \text{if } \tau = \mathbf{R} \\ \text{val}(\tau) = \{\{x : v\} \mid v \in \text{val}(\tau')\} & \text{if } \tau = \{x : \tau'\} \\ \text{val}(\tau) = \{v_1 \cup v_2 \mid v_1 \in \text{val}(\tau_1), v_2 \in \text{val}(\tau_2)\} & \text{if } \tau = \tau_1 \cup \tau_2 \end{cases}$$

We use the common notation to denote access of data contained in values. For relation names — i.e., when $\tau = \mathbf{R}$ — access of the value bound to attribute $a \in \mathbf{R}$ of a tuple $t \in \text{val}(\tau)$ is denoted as $t.a$. For mapping types, $v[x]$ denotes the access of the value bound to variable $x \in \text{dom}(v)$ in mapping v. Finally, the event algebra supports built-in functions which take as arguments a number of values of the types introduced above, and return a new value. These built-in functions include simple arithmetic operators such as a sum or difference, as well as functions to determine the minimum, maximum, or average of the arguments.

Note that the event algebra does not support lists, sets, or other arbitrary-size container types. The functions always range over a fixed number of arguments of constant size. In the interest of reactivity, the built-in functions are required to operate in constant time, with constant space requirements.

We further define the notion of an *instance of a schema*.

Definition 9: Instance of a schema

Let $\mathcal{R}$ be a schema. An *instance of a schema* assigns a *relation* to each *relation name* in $\mathcal{R}$.

In other words, $\forall \mathbf{R} \in \mathcal{R}$, an instance of $\mathcal{R}$ constructs an assignment from $\mathbf{R}$ to a finite subset of $\text{val}(\mathbf{R})$. For instance, an *instance of the schema* of the driver scenarios would be a concrete dataset of financial transactions.

Finally, we introduce the notion of a *stream* in our event algebra.

Definition 10: Stream

A *stream* is an instance of a relational schema which prescribes that an attribute named `timestamp` occurs in every relation. This `timestamp` identifies the temporal component of the relation.

A stream has an *event arrival rate* (definition 2), which identifies how many events may occur within a certain range of timestamps.

Relations in a stream have an order: relations whose timestamp is smallest, rank lower, and are said to have "occurred first".

4.3.2 Patterns

The event algebra EA is the set of all patterns. The following are the five different forms of patterns and their types:

Definition 11: Pattern

A pattern and its type is recursively defined as one of the following:

Bindings of relations to variables, i.e., constructs of the form ($\mathbf{R}$ as v), where relation $\mathbf{R} \in \mathcal{R}$ and variable $v \in \mathbf{V}$.

The type of a binding pattern of the form ($\mathbf{R}$ as v) is the *mapping type* $(v : \mathbf{R})$.

Conjunctions of patterns binding disjoint sets of variables, i.e., constructs of the form $(P_1 \wedge P_2)$, where $P_1, P_2 \in \text{EA}$ and $\text{dom}(\text{type}(P_1)) \cap \text{dom}(\text{type}(P_2)) = \emptyset$.

The type of conjunctive patterns of the form $(P_1 \wedge P_2)$ is $\text{type}(P_1) \cup \text{type}(P_2)$, which is well defined since the sets of variables they map over are disjoint.

Disjunctions of patterns binding the same sets of variables, i.e., constructs of the form $(P_1 \vee P_2)$, where $P_1, P_2 \in \text{EA}$ and $\text{dom}(\text{type}(P_1)) = \text{dom}(\text{type}(P_2))$.

The type of disjunctive patterns of the form $(P_1 \vee P_2)$ is $\text{type}(P_1)$, which by definition is the same as $\text{type}(P_2)$.

Filters consisting of a pattern P and a filter condition F, denoted by P WHERE F.

The type of filter patterns of the form P WHERE F is $\text{type}(P)$.

Filtered negations of a pattern, called the *negated pattern*. A filtered negation consists of a *positive pattern* P_p, a *negated pattern* P_n, and a filter condition F, and take the form $P_p \setminus P_n$ WHERE F.

The type of a filtered negation $P_p \setminus P_n$ WHERE F is the type of the positive pattern, i.e., $\text{type}(P_p)$.

Intuitively: a binding pattern introduces a variable binding, which maps an instance of a relation name to a variable, e.g., (MoneyTransferred as `incoming_transaction`). Conjunctions and disjunctions operate as usual. Filters enforce constraints on an existing pattern, e.g. requiring that the `amount` attribute of `incoming_transaction` is larger than 100 €. A pattern that matches when two financial transactions occurred, jointly satisfying some constraints, can be modeled as a *conjunction* and a *filter* in the event algebra. A pattern that matches when either of two transactions occurred, can be modeled as a *disjunction* and a *filter*. The absence of contestation of a transaction can be modeled by *filtered negation* of a "contested_transaction" satisfying the constraints. Filtered negation thus encodes what PARTElang achieves using the syntax `no`.

4.3.3 Evaluation

Patterns can be evaluated on a stream of relations to extract the values matching the pattern.

Definition 12: Evaluation of a pattern

Let P be a pattern in an event algebra EA. Let S be a stream with schema $\mathcal{R}$. The evaluation of a pattern P over S is then denoted by $[\![P]\!]_S$. $[\![P]\!]_S$ is a set of values in $\mathrm{val}(\mathrm{type}(P))$, defined recursively as follows:

$$\begin{cases} [\![P]\!]_S = \{\{x : v\} \mid v \in \mathrm{val}(\mathbf{R})\} & \text{if } P = (\mathbf{R} \text{ as } x), \mathbf{R} \in S \\ [\![P]\!]_S = \{v_1 \cup v_2 \mid v_1 \in [\![P_1]\!]_S, v_2 \in [\![P_2]\!]_S\} & \text{if } P = P_1 \wedge P_2 \\ [\![P]\!]_S = [\![P_1]\!]_S \cup [\![P_2]\!]_S & \text{if } P = P_1 \vee P_2 \\ [\![P]\!]_S = \{v \mid v \in [\![P']\!]_S, v \models F\} & \text{if } P = P' \text{ WHERE } F \\ [\![P]\!]_S = \{v \mid v_p \in [\![P_p]\!]_S, \forall v_n \in [\![P_n]\!]_S : (v_p \cup v_n) \not\models F\} & \text{if } P = P_p \setminus P_n \text{ WHERE } F \end{cases}$$

To paraphrase, a binding pattern of the form $(\mathbf{R}$ as $x)$ yields a binding of all relations identified by $\mathbf{R}$ in the schema to a variable x; a conjunctive pattern yields the pairwise union of the results of both conjuncts (cf. a Cartesian product); a disjunctive pattern yields all results that occur in at least one of both disjuncts (cf. a logical *or*); a filter pattern yields results for the filtered pattern, as long as they satisfy the filter condition; and filtered negation patterns yield results for the positive pattern, as long as no results for the negated pattern satisfy the filter condition when paired with the positive pattern. Note that filtered negation implements negation-as-failure: absence of relations satisfying the negated patterns is concluded when no relations can be found that satisfy the negated pattern.

4.4 Mapping PARTElang onto the Event Algebra

Intuitively, it should be clear that the patterns in the `where` clause of a PARTElang program map onto patterns in the event algebra EA defined in section 4.3. Similarly, temporal constraints in the `when` clause of a PARTElang program map onto additional filters to add into the patterns in EA. The reaction logic in the `then` clause, however, does not map as straightforwardly, since EA does not specify how new events are made to occur. Still, apart from the actual `emit` itself, every part of the reaction logic can be expressed in terms of concepts defined in EA.

In this section, we specify in detail how declarative rules in PARTElang map onto expressions of EA.

4.4.1 Mapping a Single Pattern

A positive event pattern in the `where` clause of a PARTElang program, with no additional constraints specified for the pattern, takes the form

```
<var-name> :  <template-name> {}.
```

The template name identifies an *event template*, which corresponds to a *relation name* in EA. The binding of instances of the template to `<var-name>` corresponds to a binding of the relation to a fresh variable. For ease of reading, we will refer to a variable in PARTElang and its corresponding variable in EA by the same name. The type of a single, positive pattern is therefore a mapping type from the variable name to the relation corresponding to the template.

Non-temporal constraints may be specified in the `where` clause. A positive event pattern with non-temporal constraints takes the form

```
<var-name> :  <template-name> { <cond>* },
```

where `<cond>` is a sequence of conditions. For each condition specified in this way, a filter must be added in the corresponding pattern in EA. By definition 12, filters do not impact the type, so the type of a filtered pattern is still a mapping type from the variable name to the relation corresponding to the template.

4.4.2 Mapping Multiple Patterns

We wrote before that when multiple patterns occur in a PARTElang rule's `where` clause, they are implicitly understood to form a conjunction. This conjunction of patterns naturally maps onto the notion of a conjunction of patterns in EA.

Though timestamps of events are implicit in PARTElang, they must be made explicit when mapping onto EA. For the sake of this mapping, the timestamp attribute of an event in a PARTElang program is a hidden attribute, with a name different from any name given to attributes by an event's template. For each temporal interval `a in b[l, u]` given in the `when` clause of a PARTElang program, two new filter conditions must be added into the pattern in EA. One filter enforces the lower bound of the interval, by imposing a.timestamp $\leq b$.timestamp + l. The other filter enforces the upper bound of the interval, by imposing a.timestamp $> b$.timestamp + u. These filter conditions have to be added to the conjunction of pattern such that both a and b are in scope.

The number of filters that is required to encode the temporal constraints, as well as the rate of false-positives in the intermediary matches, can be reduced by applying the technique of Teodosiu and Pollak [133]. By first taking the transitive closure over the temporal constraints, some filters can be made more strict without excluding valid results, and some filters can be found to be redundant, and omitted.

When multiple patterns exist in a rule, some of them may be negated. Thanks to our scoping rules, all positive events mentioned in inter-event constraints involving negated patterns must syntactically appear before the negated pattern.[9] This makes the transformation to EA rather straightforward. Consider the following PARTElang pattern, which encodes a negated pattern enforcing the absence of an event `<neg>` satisfying the conditions in `<conds>`:

```
no <neg> { <conds> }.
```

Consider also an arbitrary conjunction of positive and negated PARTElang patterns, which we will call `<before>`. Let P_{before} be the result of mapping `<before>` onto a pattern in EA. Let P_{neg} be the result of mapping `<neg>` onto a pattern in EA. Let `F` be a direct translation of `<conds>` from the PARTElang condition syntax to the EA filter syntax. The pattern formed by the implicit conjunction

```
<before>
no <neg> { <conds> }
```

can then be mapped onto the filtered negation $P_{\text{before}} \setminus P_{\text{neg}}$ WHERE F.

4.4.3 Mapping Multiple Rules

Multiple rules in PARTElang form an implicit disjunction. Consequently, the patterns of multiple rules can be mapped onto EA by mapping the different rules' patterns onto EA, and then applying disjunction as defined in definition 12. In other words, the results of matching a disjunction of patterns, is the combination of all results for each of the patterns, i.e., a set union.

4.5 Limitations of PARTElang and Future Work

PARTElang and its event algebra EA are useful in their current form, as we show in chapter 7. Still, there are three major limitations that need to be solved in future work. First, when specifying time windows in PARTElang **only closed time intervals can be used**. Second, **PARTElang does not support aggregation constructs**. Third, **PARTElang does not include the software-engineering constructs** which facilitate building and maintaining LRP software.

[9]Remember that PARTElang requires that each negated pattern is correlated to at least one positive pattern. This requirement is similar to e.g. the *safety condition* in Datalog, which mandates that variables appearing in a negative literal in a rule's body must also appear in at least one positive literal.

Our event algebra enforces this at a lower level by making it syntactically impossible to not do so: the only syntax that introduces negation is *filtered negation* (see definition 11), and that syntax requires both a positive and a negated pattern.

4.5.1 Restriction to Closed Time Intervals

PARTElang and EA require specifying time windows between different event patterns. These time windows must be written as closed intervals. PARTElang programs cannot currently express patterns with semi-open temporal intervals. It is therefore not possible to, e.g., specify that an event should take place any time after another event, or that an event should take place at least three months before another event (but may take place arbitrarily far before).

In the general case, we argue that this limitation is required, since the alternative may require unlimited memory sizes to store old events. However, specific patterns can be useful while requiring only constant storage, or storage linear in some reasonable size:

Constant storage requirements Consider, e.g., a pattern which matches when any event matching certain conditions occurred ever. Such a pattern might be useful in real-world use cases, and an implementation could keep track of this information using only one single bit.

Linear storage requirements Consider a pattern which matches when an event occurred for the first time for a certain bank account. Such a pattern would require storage space linear in the number of bank accounts managed by the entity employing the detection. While this is not strictly constant, a reasonable upper bound could be defined on this number, and a "new account creation rate" could be defined in a similar vein to the event arrival rate defined on page 65.

Both these cases can be useful in real-world settings. Both cases could — in principle — be implemented in a strongly reactive way. Yet neither case can be expressed in PARTElang.

Another useful constraint which cannot yet be expressed in PARTElang, is the the notion of absolute timestamps, such as "every Wednesday" or "on March 28th, 2019". In general, while PARTElang supports reasoning about historical data in its managed storage, it currently lacks the ability to correlate events that it keeps in that managed storage with unmanaged data. Future work may explore how this limitation can safely be relaxed.

4.5.2 Lack of Aggregation Constructs

PARTElang programs cannot currently reason about *the set* of events satisfying a pattern. Every rule activation reasons about exactly one match per pattern. Support for aggregation is prevalent among existing databases, streaming systems, and even rule-based systems [2]. Many pattern matching systems based on regular expressions allow capturing the result of a Kleene star of a sub-pattern, using the set of matches to the sub-pattern as the match for the Kleene star. Its wide use suggests that it is worthwhile to explore how PARTElang and its underlying event algebra can be extended to support matching sets of events satisfying a pattern, and imposing intra-set conditions on the elements in the set.

Mellin [101] rejects the use of aggregation in real-time contexts. However, a restricted form of aggregation could be applicable. If the aggregated sets are of static, fixed size, operations on them need not preclude strong reactivity. Additionally, if the sets are defined by semantic windows, where the semantic windows are expressible in PARTElang, then this extension will not affect the strong reactive guarantees either. Future work may explore under which constraints matching multiple events to one pattern may be allowed.

4.5.3 Lack of Software-Engineering Constructs

In addition to being a theoretical model, LRP should be useful as a paradigm for practical programming languages. We therefore investigated how general software-engineering principles can be applied to Logic Reactive Programming. For instance, it should be possible for LRP programmers to write a pattern once, abstract over it, and compose it with other patterns. However, PARTElang as described in this dissertation is purposefully kept minimal, and does not feature all the software-engineering constructs we developed earlier, in context of Midas. Midas [126] is a declarative gesture specification language. Using declarative rules, a Midas programmer can specify a pattern of interaction events and the expected response. In his PhD thesis, Hoste [74] described the design of Midas, including some bits of language design aimed at improving the usability of the language. We studied ways in which the usability of Midas can be further improved [121].

Because of their co-development at the same lab around the same time, the designs of Midas and PARTElang influenced each other. Two features of Midas have not found their way into PARTElang yet. First, the Midas gesture specification language introduces ways of interleaving event pattern specifications with reaction code, while retaining the straightforward mental model of variable-scope found in procedural languages. Second, Midas provides a module system which allows reuse of patterns as *attempt*s, in a way reminiscent of *mixins* or trait composition.

PARTElang does not define a module system, and retains a rigid division between patterns and reaction code. However, the design from Hoste [74] is directly applicable. Future work may formally prove that Midas' software-engineering constructs can be ported to PARTElang while maintaining PARTElang's guarantees.

4.6 Related work

4.6.1 Guaranteed Constant Time Processing

The idea of a programming language with known bounds on execution time is not new. We compare PARTElang with three main categories of language research that tackles this problem.

Constant-Time Programming Languages

Constant-time programming languages like FaCT [28] or Jasmin [7] offer strong guarantees on processing time of programs written in those languages: the running time of those programs does not depend on secret data. As such, such languages guarantee resilience against timing-attacks. This makes them invaluable in cryptographic contexts, but does not help with LRP. *Constant-time programming languages* offer no specific support to the programmer to develop complex event processing systems. In contrast, PARTElang is a DSL which removes most of the boilerplate code for handling streams of complex events, for managing semantic windows, etc. PARTElang's guarantee on constant processing time specifically leverages domain-logic to define semantic windows, whereas this logic would have to be manually extracted and fit to the static analysis model of those constant-time programming languages. In contrast, the real value of constant-time programming languages, is their ability to operate under a more stringent threat model where not only the worst-case run time matters.

Similarly, synchronous data flow [20] programming languages like Esterel [21] and Lustre [67] provide excellent support for reasoning with time, and for formally reasoning about run time characteristics of the programs. This makes them a good fit for real-time domains such as safety-critical control software, e.g., in aerospace or industrial process management. However, like the constant-time programming languages discussed above, synchronous data flow languages do not aid programmers in expressing programs which match complex event patterns.

Both these categories of languages hence share a similar shortcoming to the one we described for (functional) reactive programming languages in section 3.5.1 on page 57: they offer no support for pattern matching; the job they leave for the programmer to fix, is cumbersome and error-prone when complex search patterns are involved.

Bounded-Size Reactive Programming

Some reactive programming languages and tools enforce bounded size. A representative example is the work by Krishnaswami [88]. Such work removes *spacetime leaks* from reactive programming languages. These leaks can cause the programs to accidentally have a non-constant update cost per new event, as state accumulates between updates. The language overcomes this issue by splitting an update into two categories: current expressions which are evaluated immediately (preventing the accidental build-up of state), and future expressions which are delayed as a constant-sized thunk. However, being aimed at functional reactive programming, these languages do not tackle the problem of maintaining a constant-size set of non-stale events which can *match patterns* at a future step. Events are either delayed, executed once, or deleted. Reusing events in multiple matches is not part of their model. Manually storing sets of events is allowed by their semantics, but no guarantees are offered by the language in that case.

Another interesting take on reactive programming within a bounded resource budget was explored by Vaziri et al. [137] in ActiveSheets. ActiveSheets enrich a popular spreadsheet program with streaming semantics, and linkage to external, streaming sources and sinks. Strongly reactive behavior is guaranteed in ActiveSheets as the spreadsheets are bounded in size by construction. First, the number of steps required to compute an update depends on the number of cells in which formulae are entered. This number of steps is finite and fixed whenever an update has to be computed. Second, access to historic values is achieved through a `latch` construct. Latches store exactly one value, making them inherently bounded in size. Windows in ActiveSheets always contain a fixed number of steps, and take the form of a contiguous row or column of cells. This offers boundedness, but limits expressivity. In PARTE, only the upper bound on window size is fixed, and the pattern programmer needs not manually determine that size.

Real-Time Event Monitoring

Mellin [101] describes Solicitor, an event monitoring system which guarantees limits on resource usage. The work in many ways tackles similar problems as the ones described in this dissertation. However, the event correlation semantics in their work is different. Chakravarthy and Mishra [29] defines different kinds of *event contexts*, which decide whether two events may be correlated. Examples are a "Most Recent" context, which only considers the most recent event of a given type; or "Continuous" context, which may reuse an event that was previously matched in combination with the latest match, as long as an uninterrupted series of matching events exists for e.g. determining a trend in the data. The "General" context is however considered as too broad to be applicable, as it entails a Cartesian product. For this reason, Solicitor [101] omits support for such *contexts*.

PARTElang does support matching events in the "General" context. The size and time cost of the Cartesian product is capped by a combination of two factors: the event arrival rate (see page 65) and the semantic windows (see page 27) defined by the concrete event pattern.

Static Analysis of Actor Interaction

Stiévenart et al. [130] describes a static analysis for actor languages. The work aims to support proving upper bounds on actor inboxes. They do so even for arbitrary input, but without taking into account the elapsed real-world time. The static analysis is not sufficiently powerful to identify the relation between subsequent event arrivals, nor the relation between event arrival and event expiration in the PARTE model. Lacking this domain-knowledge, the static analysis would over-approximate the inbox size of PARTE's entry nodes as infinite.

4.6.2 Composition of Temporal Constraints in Event Algebras

Event algebras by necessity deal with temporal constraints: events inherently have a temporal component. Still, different algebras offer different expressivity for certain temporal concerns.

Whole-Pattern Constraints vs. Constraints Per Individual Composition

For instance, the SASE event language by Wu et al. [141] cannot express a pattern which correlates three events E_1, E_2, E_3 such that E_1 occurs at most three seconds before E_2 and such that E_2 occurs at most three seconds before E_3. SASE offers a `within` clause to impose temporal constraints, but such a clause imposes a time window on the entire complex event. In our event algebra the pattern could be expressed as follows:

$$\begin{aligned}
P =& (EventType1 \text{ as } E_1) \\
& \wedge (EventType2 \text{ as } E_2) \\
& \text{WHERE } (E_1.timestamp < E_2.timestamp) \\
& \text{WHERE } (E_1.timestamp + 3\ seconds \geq E_2.timestamp) \\
& \wedge (EventType3 \text{ as } E_3) \\
& \text{WHERE } (E_2.timestamp < E_3.timestamp) \\
& \text{WHERE } (E_2.timestamp + 3\ seconds \geq E_3.timestamp)
\end{aligned}$$

Open vs. Closed Temporal Intervals

Conversely, there also exist patterns which are expressible in some formal models, but not in our event algebra. Consider the seminal interval algebra by Allen [6]. Allen's interval algebra enables expressing how two events correlate in time. For instance, the expression

$$E_1 < E_2$$

denotes that event E_1 occurs before event E_2. Similarly, the expression

$$E_1 \circ E_2$$

denotes that event E_1 overlaps with event E_2. Our event algebra is not capable of expressing Allen's event algebra, for two reasons:

1. Allen's algebra allows for half-open temporal intervals, i.e., unbounded temporal distances. An expression like "$E_1 < E_2$" places no limit on how long after the occurrence of E_1 the system must wait for a potential E_2. To be usable in a CEP context with the requirement of strong reactivity, PARTE is limited to expressing

time windows as closed intervals. There is hence no point in supporting unbounded time windows in our event algebra, barring the exceptions mentioned in section 4.5.

2. Allen's algebra depends on placing events in time using both a start and an end time. PARTE uses only a single timestamp per event. This keeps the formalism shorter and clearer, yet still allows to express realistic patterns. However, the core constructs enabling strong reactivity in PARTElang — discussed in the next chapters — can be applied to time intervals as well as to timestamps. The concepts introduced by this thesis are therefore compatible with an approach to complex event processing in which events occur during some time interval instead of at some timestamp.

Constraints Per Individual Composition vs. Constraints Per Pair of Patterns

A variation of our event algebra recording both start and end times of events — and a variation of PARTE built on top — fully contains the event algebra with interval constraints.[10] The opposite is not the case: some patterns can be expressed in PARTE, but not in the event algebra with interval constraints. Consider the notation

$$(P_1; P_2)[n]$$

for a sequence of two patterns P_1 and P_2, restricted to a temporal window of size n. The compositional nature of this algebra entails that if $(P_1; P_2)[n]$ is a valid pattern, then substituting any valid pattern for either P_1 or P_2 yields a valid pattern too. Hence, both

$$(P_1; (P_2; P_3)[n_A])[n_B]$$

and

$$((P_1; P_2)[n_C]; P_3)[n_D]$$

are valid patterns too. Consider now a query of this form for events such that P_2 occurs within twenty seconds after P_1, and that P_3 occurs within thirty seconds after P_2, in such a way that the entire pattern takes no more than forty seconds. The forty second requirement can be encoded in the interval n_B or n_D. The twenty second interval can be encoded in n_C. The thirty second interval can be encoded in n_A. However, the twenty second interval and the thirty second interval cannot be both encoded in the same pattern. The model only allows restricting the duration of the whole pattern, and of either $(P_2; P_3)$

[10]By "the event algebra with interval constraints" we refer to an event algebra using Allen's interval algebra but imposing finite time windows

or $(P_1; P_2)$. Our event algebra overcomes this limitation of the event algebra with interval constraints, as shown by the following pattern:

$$
\begin{aligned}
P =& (EventType1 \text{ as } E_1)\\
& \wedge (EventType2 \text{ as } E_2)\\
& \text{WHERE } (E_1.timestamp < E_2.timestamp)\\
& \text{WHERE } (E_1.timestamp + 20\ seconds \geq E_2.timestamp)\\
& \wedge (EventType3 \text{ as } E_3)\\
& \text{WHERE } (E_2.timestamp < E_3.timestamp)\\
& \text{WHERE } (E_2.timestamp + 30\ seconds \geq E_3.timestamp)\\
& \text{WHERE } (E_1.timestamp + 40\ seconds \geq E_3.timestamp)
\end{aligned}
$$

Unlike EVA, TESLA [41] offers the abstraction of a rule, and allows for correlating more than two events at a time. Within a rule, individual events' temporal properties can be used, much like in PARTE.

Associativity of Temporal Constraints

A very similar problem problem exists in EVA [72], even when no time windows are imposed.

EVA handles composition of events differently from PARTE. In PARTE, a rule programmer has to manually code the way in which multiple events are abstracted over: to generate a new complex event, an `emit` statement has to be written, specifying how the values of the different events combine to form the new complex event. A new template may have to be defined. In EVA, composition and abstraction happens automatically: a sequence $(E_1; E_2)$ of events E_1 and E_2 is itself a (complex) event. As a result of this choice for implicitness, EVA had to provide a number of sane defaults. For instance, in EVA the timestamp of a complex event is always the timestamp of the last of its constituents. Individual events' timestamps are lost in the pairwise composition, only one of both is retained. In PARTE, the timestamp of a complex event has to be explicitly computed when the new event is created, but that timestamp can be based on any of the constituents' timestamps. The ease of composition offered by EVA has a more significant drawback: its means of composition are not associative. Consider the set of events $\{E_a, E_b, E_c\}$, where E_a occurs at timestamp 2, E_b occurs at timestamp 1, E_c occurs at timestamp 3. As depicted in figure 4.2, these events satisfy the sequence pattern $(E_a; (E_b; E_c))$, but not the sequence pattern $((E_a; E_b); E_c)$, which is equivalent modulo associativity.

In TESLA [41], too, complex events' timestamps are implicitly inherited from the events they are composed of. In TESLA, the timestamp of the first event — which they refer to as the "anchor point" [41] — is used. As mentioned above, TESLA offers the abstraction of a rule, and allows for correlating more than two events at a time. Within a rule, individual events' temporal properties can be used, much like in PARTE. It is only in between rule,

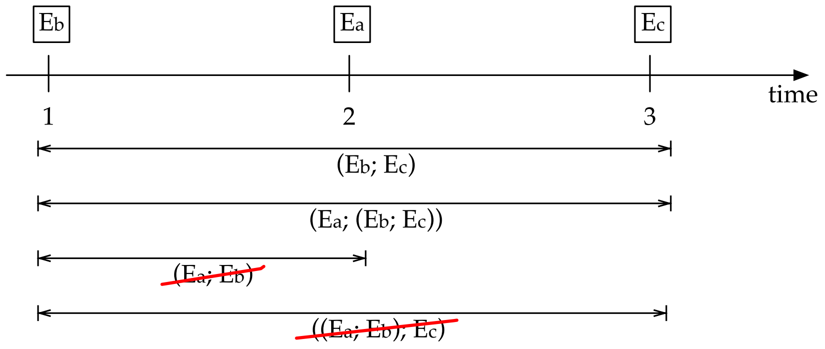


Figure 4.2: Non-associativity of event composition as done in, e.g., EVA — Since E_b [at $t = 1$] occurs before E_c [at $t = 3$], the first pattern holds. Since E_a [at $t = 2$] occurs before $(Eb; Ec)$ [at $t = 3$], the second pattern holds. E_a [at $t = 2$] does not occur before E_b [at $t = 1$]. Hence, $(Ea; Eb)$ is empty, and cannot be composed. In such a semantics, $(E_a; (E_b; E_c))$ differs from $((E_a; E_b); E_c)$, i.e., *follows-by* composition is not associative.

after abstracting into a composite event, that TESLA's automatic event timestamp selection comes into play.

These issues are partially prevented in systems that model events as occurring during a certain interval [41, 47, 93], instead of at a certain timestamp. Merging multiple events' intervals by taking their union does not throw away as much information as merging multiple events' singular timestamps.

4.6.3 Event Capture Semantics in Event Algebras

EVA [72] is an event algebra and programming language for CEP. Instead of using declarative rules, EVA programmers can express sequences of events they are interested in using so-called *profiles* or *subscriptions*. Such subscriptions can make use of operators for temporal disjunction (one of both events has to occur), logical disjunction (one of both constraints has to hold), temporal conjunction (both events have to occur), logical conjunction (both constraints have to hold), temporal sequence (like temporal conjunction, but imposing order), temporal negation (explicitly disallowing an event from occurring), and temporal selection (selecting the ith occurrence satisfying the constraints).

EVA defines the notion of a *trace* as "a semi-ordered sequence of events $e \in \mathbb{E}$ with start- and end-points" [72]. They note that because "a trace behaves essentially as a list, we can use" list operations such as indexing, such that "$tr[i], i \in \mathbb{N}$ refers to the ith event of the trace tr". Therein lies the major semantical difference between EVA and PARTE: EVA expects to match all events (and optionally consume them). A sequence of events is matched one-by-one, and any event that needs to be skipped must explicitly be skipped by the *subscription*. This can for instance be accomplished using a wildcard instead of an event variable. In contrast, skipping an uninteresting event is trivial in PARTE: events that do not match are simply ignored by that rule. On the other hand, detecting the very next event after a certain event is relatively difficult in PARTE: one must explicitly specify a rule asking for a first event, a second event occurring later in time, and the absence of an event that temporally occurs in between the two.

The runtime of the TESLA language, T-REX [41] implements the rules as a set of finite state automata. Unlike normal regular expressions, T-REX does support event capture semantics where events are not *consumed* when they match one pattern. The same events can hence match multiple patterns. T-REX achieves this by storing many automata instead of once: instead of having one single automaton per rule, there is a baseline "template" automaton for the initial, empty state of a rule. In response to the occurrence of an event, all automata which could transition are cloned, and the clone is transitioned according to the event. The automata with the previous state are kept around for future matching with other events. In essence, T-REX stores intermediate results for its incremental matching by storing automata which partially matched a pattern.

The event capture semantics of Cayuga [47] closely match those of TESLA, as their detection model is very similar.

4.7 Conclusion

We introduced *Logic Reactive Programming*, a novel reactive programming paradigm for correlating complex events. We formulated five requirements for Logic Reactive Programming languages. We presented PARTElang, a first LRP language. Its informal semantics and syntax were presented, and an event algebra underlying its semantics was defined. From these, it followed that PARTElang satisfies the first three requirements of LRP: PARTElang programs are sets of declarative CEP patterns ($\mathbf{R_{LRP}}$ **1**) with support for reasoning over the events' temporal aspects ($\mathbf{R_{LRP}}$ **2**). Event data is automatically managed based on semantic time windows ($\mathbf{R_{LRP}}$ **3**).

To show that the last two requirements for LRP languages are satisfied by PARTElang too, it must be shown that PARTElang programs are *strongly reactive* (definition 1) given a certain *event arrival rate* (definition 2). To this end, we stated that constant-time per-event processing is required. To be able to impose an upper bound on the time it takes to fully process an event, the processing must be void of unbounded loops. PARTElang achieves this by enforcing *stratification* (definition 3).

To show that PARTElang is scalable ($\mathbf{R_{LRP}}$ **4**) and strongly reactive ($\mathbf{R_{LRP}}$ **5**), the actual evaluation model must be defined. We do this in the following chapters.

5 Operational Semantics for a Logic Reactive Programming Language

In the previous chapter we introduced the novel Logic Reactive Programming paradigm. In section 4.2, we defined an LRP language, PARTElang. We provided the formal foundations for PARTElang in section 4.3, in the form of an event algebra. In this chapter, we formally define the evaluation model for programs written in PARTElang, so that they implement the semantics of the event algebra.

The architecture of a Logic Reactive Programming system takes the form depicted in figure 5.1. This chapter defines an operational semantics for the "LRP runtime" component. The model must hence support

- receiving primitive events from multiple *event sources* external to the model,
- filtering them based on the conditions specified in a PARTElang program,
- correlating them based on the conditions specified in the implemented logic reactive program,
- generating *compound events* generated in accordance with `emit` constructs specified in a PARTElang program, and
- exposing a distinct point at which a *match* to the program's patterns can be detected, enabling notification.

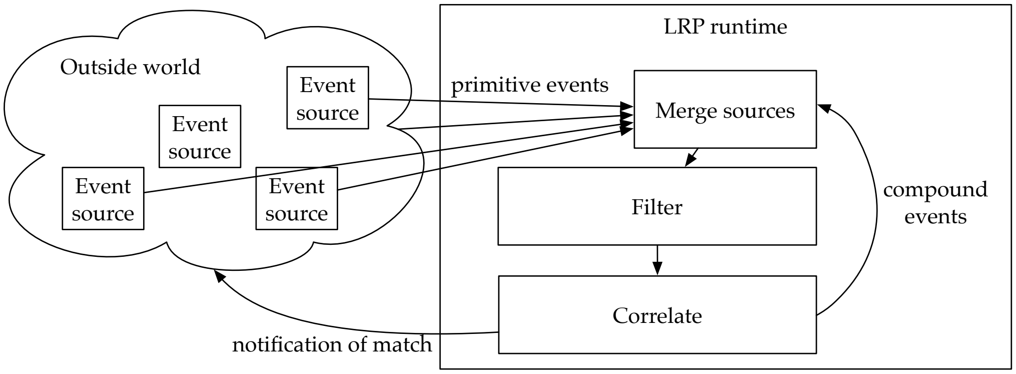


Figure 5.1: The architecture of a Logic Reactive Programming runtime — Primitive events gathered by event sources are merged, filtered, correlated, and either abstracted over as compound events, or as matches.

Of course, the model must ensure that *strong reactivity* is maintained during evaluation, as defined in definition 1 (page 65). This requires constant-time processing. To enable this constant-time processing, constant-space processing must be enforced, since some operations in the event algebra have the complexity of relational joins. Indexing and/or enumerating results of a relational joins take time proportionate to the size of the *instance of the schema* over which they join. The model defined in this chapter therefore ensures an upper bound on state size throughout evaluation.

For readability, we introduce the model incrementally. We first discuss a rudimentary model in section 5.2. The rudimentary model is able to detect most patterns specified in the PARTE language, but falls short by not correctly handling their temporal component. As a result, the rudimentary model fails to implement negation correctly, and fails to expire old data, precluding strong reactivity. In section 5.3 we provide a framework for reasoning about what is missing from the rudimentary model. With that framework in place, we move on to a full-fledged model: the *Featherweight PARTE* model. We define Featherweight PARTE, fwPARTE for short, in section 5.4. A number of possible improvements beyond fwPARTE are discussed in section 5.6. The formal validation in chapter 8 will build on the Featherweight PARTE model, unless explicitly stated otherwise.

This chapter concludes by defining how a program in PARTElang can be mapped onto the fwPARTE model.

5.1 Background: the Rete Algorithm

The fwPARTE model defines how to construct a directed acyclic graph (DAG) in which nodes represent filtering or relational (anti-)joining operations, and edges between nodes represent data dependencies. The use of such a DAG for guiding evaluation and for storing intermediate results is based on the Rete algorithm by Forgy [59]. Therefore, before defining our novel model, this chapter starts by introducing the Rete algorithm, and outlining the modifications necessary for the Logic Reactive Programming context.

5.1.1 Rete: A Forward Chaining Inference Engine Algorithm

The Rete algorithm was proposed by Forgy [59] as an improvement to existing pattern matching algorithms used in expert systems. Rete essentially preprocesses a declarative rule set to enable incremental matching: it compiles a set of patterns (and their reaction logic) into a directed acyclic graph. Nodes in the graph represent part of the work necessary to match the pattern, e.g., verifying whether conditions hold, and unifying sets of logical variables. The edges in the graph represent the data dependencies. To match a new logical fact to a rule set compiled by Rete, one starts at the root, and descends through the DAG, executing the operations encountered. If one of the terminal nodes is activated, a rule is known to be matched, and its reaction logic is executed.

Optimizing Pattern Matching using a Fixed Search Graph

The Rete algorithm achieves part of its efficiency by preparing the Rete graph only once, and optimizing this graph by e.g., merging common prefixes of the graph (commonly referred to as *node reuse* [91]), thereby preventing duplicate work. After this initial graph construction phase, a *recognize-act cycle* is initialized, where the Rete algorithm alternates between matching new data to patterns, and executing the actions specified as the consequent of matched rules.

```
rule rule-1 where
  a : A, b : B, c : C
then
  make D with {}

rule rule-2 where
  a : A, b : B, e : E
then
  make F with {}

rule rule-3 where
  c : C, no d : D
then
  make G with {}
```

Listing 5.1: Example of a rule set

Consider for instance the rule set in listing 5.1, written in a PARTElang-like pseudocode. Rule #1 states that if a compound logical fact matching the simple logical facts a, b, and c is detected, then a fact of the relation D must to be created. Rule #2 states that if a complex logical fact matching the simple logical facts a, b, and e is detected, then a logical fact of the relation F must be created. Finally, rule #3 states that if a compound logical fact consisting

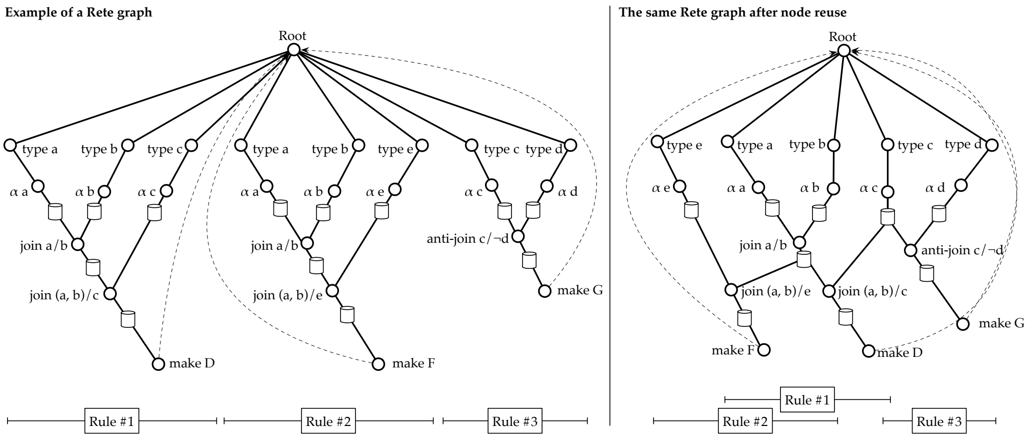


Figure 5.2: Example of a Rete graph and of the application of *node reuse* — Both graphs depict a Rete graph for the rule set listed in listing 5.1. Circles depict Rete graph nodes, cylinders depict storages for partial matches. Solid lines depict the dependency between nodes. Dashed arrows depict dependencies though explicit construction of complex logical facts.

of a simple logical fact *c* and no logical fact *d* is detected, a logical fact of the relation *G* must be created. Figure 5.2 shows what the Rete graph for such a rule set might look like.

Optimizing Pattern Matching by Storing Partial Matches

In addition to precomputing a fixed search graph, the Rete algorithm achieves efficiency by providing the means for incrementally computing matches to patterns as new facts arrive. To this end, Rete prescribes how to store *partial matches*, i.e., matches to a part of the pattern represented by the Rete graph. Partial matches that were previously computed in a Rete graph need not be recomputed when new facts arrive. Along any edge in a Rete graph, partial matches computed by the source node can be stored. When a new fact arrives, it can efficiently be matched to all previous facts which might match a pattern with the new fact, as partial matches to the rest of the pattern are stored along the Rete graph. When facts are explicitly retracted from the fact base, the corresponding partial matches must be retracted too.

One of the advantages of Rete's way of handling partial matches is that it can store arbitrarily many of them. Compare for instance to a finite state automaton. A finite state automaton is at any point in time in at most one state. A Rete graph can be on its way to matching arbitrarily many concurrent sequences to patterns: each possible sequence has its own set of partial matches. The flip side of this behavior is that the storage requirement of a traditional Rete graph is unbounded.

5.1.2 Varying the Amount of State Stored along the Rete Graph

In addition to the traditional Rete algorithm by Forgy [59], some variations were construed to balance the storage size requirements with execution time. A thorough exploration of this topic is outside the scope of this dissertation, though we deem it sufficiently important to sketch the general ideas, as restricting how much data is stored is a core part of the thesis described in this dissertation.

Recall that in Rete, the first two patterns in a rule are joined, and each subsequent pattern is joined one-by-one with the results. In traditional Rete, intermediate results are stored at each join in *partial match histories*. Evidently, it is possible to store fewer intermediate results (saving storage space, but requiring recomputing) or to store more intermediate results (storing not just along edges of the graph, but for every pair of patterns, for every three patterns, etc.). We depicted this concept in figure 5.3.

Storing Fewer Partial Matches

Using traditional Rete as a baseline, fewer partial matches can be stored by (dynamically) omitting some partial match histories (depicted in figure 5.3 by rendering the storage semi-transparent) as done by Rete∗ [140]. Even more storage space can be saved by not saving partial matches between activations at all, as done in TREAT [105]. As a lower bound, there is the option of not saving any previous partial matches, and instead simply applying a Cartesian product involving all data.

Storing More Partial Matches

Conversely, on the right-hand side of the spectrum, more intermediate results can be stored by saving partial matches between every pair of patterns, every three patterns, etc. We found no related work which applies this approach in the context of Rete, since this approach deviates from the DAG structure prescribed by Rete. In the related field of *Incremental View Maintenance*, this approach has been used by DbToaster [4]. Idris et al. [76] show that such approaches do not necessarily pay off. They "presume that DBToaster performs badly [on the dataset of the experiment by Idris et al.] because [DbToaster] needs to actually materialize the results to sub-queries, [but they] did not confirm this". The time it takes to keep the intermediate results up-to-date can outweigh the time that is saved when a query can be answered from the cache of intermediate results.

Finally, Miranker et al. [106] and Batory [18] have explored the possibility of delaying the computation of intermediate results by evaluating them lazily.

Our work builds on the baseline Rete algorithm, since our focus lays with guarantees of strong reactivity and the predictability that this brings. The variations tend to improve average-case performance at the cost of worst-case performance, and at the cost of increased unpredictability.

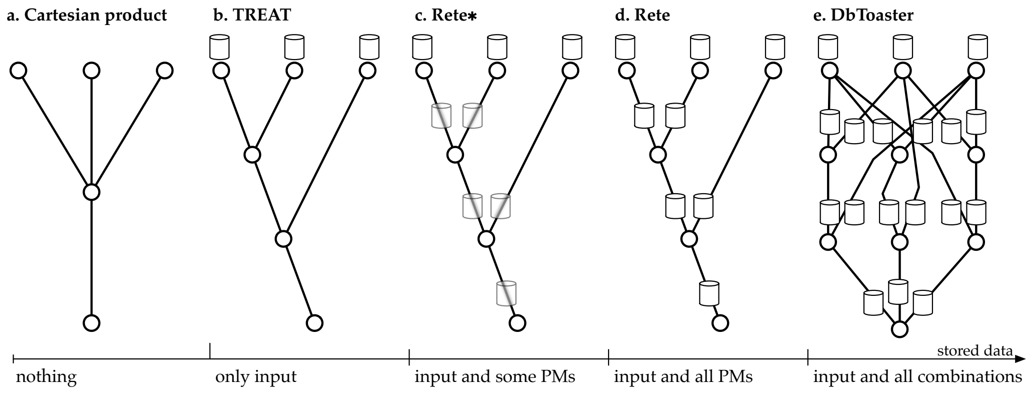


Figure 5.3: Different degrees to which partial matches can be stored in production systems — Cylinders represent storage. From left to right, the amount of data stored increases. A lower bound is offered by the option of not storing anything. The baseline — around the middle of the spectrum — is the seminal work by Forgy [59]: Rete.

5.1.3 Adapting Rete for Strongly Reactive Evaluation

Even in its basic form, the Rete algorithm offers the means to satisfy requirement $R_{LRP}1$ of Logic Reactive Programming (see definition 4 on page 72): Rete supports declarative pattern specifications and registration of reaction logic. To satisfy the other requirements, some adaptations are needed. For instance, consider the use of incremental computation in Rete. This incremental computation speeds up processing of new data. Still, it is not strongly reactive. The number of partial matches stored inside the graph can grow unboundedly. Processing time is proportionate to the number of partial matches stored in the nodes involved in the join. To achieve requirement $R_{LRP}5$**, an upper bound on the number of partial matches stored in the graph must be enforced**. This adaptation ties in closely with $R_{LRP}3$: the automatic management of stored events. Achieving this *managed storage* and the strong reactivity that it enables, is one of the main adaptations we make to Rete. The details explained further in this chapter.

5.1.4 Adapting Rete for Events

When the data fed into a Rete graph are not merely logical facts, but *events*, some changes can be made to the algorithm. Foremost is the idea of *temporal reasoning*, i.e., $R_{LRP}2$, i.e., having join-criteria based on the timestamps of the events. Compared to general data, event data is well-behaved in a very specific, but useful way: timestamps are non-decreasing. If the system can preserve the order of events (at least up to causality), then it can use that knowledge. Consider for instance a rule detecting the occurrence of a complex event *a*, followed by the occurrence of a complex event *b* at a later timestamp.

In a baseline Rete engine, if an instance of *b* occurs, but no instance of *a* occurs, then the instance of *b* would indefinitely sit idly in the knowledge base. A Rete enhanced with temporal reasoning, though, can immediately detect that no *a* corresponding to that instance of *b* can ever take place, since its temporal window of opportunity is already past. Such an enhanced Rete engine can then discard each partial match related to that instance of *b* from its memory.

A Rete engine can also be adapted for reasoning about events by introducing *time windows*. Instead of keeping an arbitrary number of partial matches for each join, a fixed number of potential joins can be specified in an event-aware rule. Events are then only correlated with events that occurred within the window duration.

As we discussed in section 2.3.1, the idea of time windows can be extended to *semantic windows* [55]. Semantic windows are like other time windows, but do not explicitly store a fixed number of events. Instead, they may store a dynamically determined number of events, where the events which are concurrently in the window with respect to some other events, are determined by runtime conditions extracted from the semantics specified in the rules. Discarding events from semantic windows happens when a runtime analysis determines the events irrelevant for the further matching. The work by Teodosiu and Pollak [133] provides the means to do so, by demonstrating how a combination of static analysis of the event patterns and a runtime analysis of events' timestamps. Jointly, those are able to determine whether an event may still be part of a complex event, either in combination with other events being processed, or even with events that are still to occur.

In this chapter, we will extend the approach of Teodosiu and Pollak [133]. We will show how **semantic windows can be applied in a system where the reasoning engine consist of many, largely independent, concurrent units of computation**, and how **the need to synchronize timing information can be localized**. To maintain strong reactivity, our semantic windows will have a determined maximum size, though the minimum size will be variable, and the maximum size will be automatically inferred by the model.

5.1.5 Adapting Rete for a Distributed Context

To enable processing larger workloads on a distributed computer cluster (**$\mathbf{R_{LRP}}$ 4**), one must deviate from the traditional implementations of the Rete algorithm. Such implementations assume that the whole fact base resides in a shared memory space, where all data can be accessed and mutated from each step of the rule processing.[1] A large body of work exists on the topic of disentangling the concepts of Rete from the ample optimizations which only make sense in the context of sequential execution [113, 64, 92, 65, 79, 14, 97].

[1]In fact, PARTE's original vision included building a parallel version of the CLIPS [142] implementation of the Rete algorithm. It was soon revealed that it would be less effort to reimplement Rete with parallelism and distribution in mind, than to untangle the sequential, shared memory assumptions that permeate through CLIPS. Of course, in CLIPS' logic programming paradigm, sequentiality is expected at the language level. Constructs like the activation agenda and salience of rules depend on evaluation happening piecemeal.

Keeping Track of Time in a Distributed Event-Aware Rete

Moving to a distributed context leads to a whole new set of problem domains.

A model for Rete using a distributed memory space must offer some form of fault-tolerance. Furthermore, in a distributed setting, atomic *recognize-act cycles* are no longer cheap to implement. While cycles at a per-node level are cheap (cf. *turns* in actors [43]), orchestrating a cycle of the entire system entails setting up an expensive synchronization point. As we shall demonstrate further in this chapter, such a globally synchronized cycle must not be maintained in an event-aware Rete. A weaker form of synchronization suffices, where distributed Rete nodes maintain a sort of *logical clock* [90] as part of the exchange of event data. As long as in-order propagation of event data can be guaranteed, the temporal constraints can be leveraged to determine whether events must be joined, and whether old partial matches can be discarded [132]. A global view on the knowledge base is not necessary.

Storing Partial Matches in a Distributed Memory Space

In addition to these high-level considerations, the move to a distributed memory model also requires rethinking common implementation idioms. For instance, it is common [116] to implement partial matches as a linked list, where partial matches to a join refer to the two parent partial matches from which they derive, stored higher up in the graph. This is no longer possible if the location where those parent partial matches are stored, is in a different memory space. Similarly, downward pointers to speed up retraction and reinsertion [116] cannot be implemented if the downward pointer would point into storage managed by another concurrently executing entity living in a different memory space.

To minimise the need for synchronization across Rete nodes, the nodes should be as self-contained as possible. For this reason we decided to break with one tradition in Rete systems: instead of considering the stores of partial matches to belong to the node that produced the partial match, we consider this storage to belong to the node which uses those results, i.e., the successor node in the Rete graph. While this means that partial matches can be stored redundantly, this also entails consumers can synchronously access them, as they need only access non-shared memory. Since partial matches are written only once, but can be read multiple times, this leads to an asymptotic decrease in the required synchronization points.

Rete Nodes Plus their Memories as Concurrent Units of Computation

Abstractly, the change we propose looks as follows: the process of pattern matching can be implemented by a group of concurrent actors whose communication pattern mimics a Rete DAG. The semantics of the Rete algorithm can be preserved, while a large degree of parallelism is introduced.[2] At the same time, the structure of the Rete graph offers guarantees which are beneficial to parallelism and distribution.

Our approach has the following beneficial properties with respect to distribution:

- The use of concurrent actors means **different parts of the rule set can be evaluated in parallel**. This applies both to different rules, as well as to different parts of the same rule, owing to the emergence of a pipeline of rule elements processed in their own thread of control.
- The Rete algorithm is stateful, and all data is stored in one of two ways: as *tokens* explicitly sent between the nodes, or as private data uniquely owned by one node. Synchronization for the latter is trivial: its owner's thread of control can **access private data without need for any synchronization**.
- At any time, only one node needs access to any given token: first the node which creates and sends the node, and – once sent – the receiver. This entails that **synchronizing access to tokens is cheap**: it suffices that the sender stops referring to the token once the token has been enqueued.[3]
- No more than two nodes ever contend for the sending side of a queue: two for join- and anti-join-nodes, one for other types of nodes. At most one node contends for the receiving end — namely the owner of the *inbox*. The Rete graph hence **keeps contention low by design**.

  Synchronization across different machines can happen by message passing. Nodes which happen to be scheduled on the same machine can be optimized even further: nonblocking [69, 104, 45] or even wait-free queues [80], coupled with a fence between the last write to the token and the enqueueing, suffice to enforce linearizability [70], and are cheap in the absence of contention.
- Since the Rete graph is a directed acyclic graph (DAG), all tokens-passing happens downwards. There are no tokens "swimming up the stream". Implementations hence **need only offer the synchronization mechanisms common in stream processing** as described in chapter 2.

5.2 A First, Rudimentary Formal Model for PARTE

Section 4.2 presented a language for specifying declarative rules. In section 5.1 we indicated that such declarative rules can be translated into a search graph by the Rete algorithm. This section and section 5.4 present two models for a Rete-like directed

[2]Even for small rule sets — consisting of only a couple of rules — this approach give rise to more concurrent actors than physical processors on a single node of current day hardware.

[3]A receiver trivially cannot access tokens that have not been sent to it yet.

acyclic graph of nodes into which the PARTE language can be compiled. The concrete transformation steps to follow are given in section 5.5.

As we explained at the start of this chapter, we introduce the formal PARTE model incrementally. We first present a rudimentary model of PARTE. This rudimentary model models a system that is mostly able to detect rules specified in the PARTE language, but fails to fully incorporate the notion of time. As a result, the models prescribes accumulating *all* data it encounters, leading to arbitrarily large space consumption. Since evaluation of the model takes a number of steps proportionate to the amount of data stored, evaluation of the rudimentary model grows slower over time. As such, this model cannot possibly generate a strongly reactive system as defined in definition 1. Still, a large part of the complexity can be introduced by means of the rudimentary model — hopefully — without overwhelming the reader.

The remainder of this section introduces a rudimentary model of PARTE. The model is defined formally, though the behavior described by the model fails to accurately implement the PARTE language from section 4.2. The model is defined in a top-down fashion: we begin from the high-level constructs that make up the "PARTE graph", then define their constituent elements, then define those elements' constituents, etc. until we reach the atomic data types. Next, we define the *global evaluation language*, $\xrightarrow{g}$, which defines how tokens traverse the distributed PARTE graph. Finally, we define the *local evaluation language*, $\xrightarrow{l}$, which defines how tokens are processed locally at[4] specific nodes in the graph.

5.2.1 Graph Nodes

At its outermost level, the Rete-like graph encoding a rule set is represented by a set of PARTE graph nodes. The edges between the nodes are encoded in the nodes: each node stores its out-edges. Concretely, the set **N** of PARTE graph nodes consists of tuples of the form

$$\mathcal{N}\langle \iota_n, \bar{\mathfrak{s}}, \overline{m}, \bar{t}, ndata\rangle.$$

Notation
If a is the canonical element of some set, then $\bar{a}$ is a *vector* of elements from that set. A vector has a fixed, finite size. The element at the i^{th} index is typeset as $\bar{a}[i]$.

[4]We refer to an evaluation step by the local evaluation language where the first element of the tuple in the domain of $\xrightarrow{l}$ is a node n as "evaluation *at* node n". At times, we will use terminology somewhat loosely by implying nodes are active elements, i.e., entities which can be the subject of active verbs such as "to send" or "to verify". Formally, in those cases it will be the evaluation of either $\xrightarrow{l}$ or $\xrightarrow{g}$ which enacts those steps, *at* those nodes.

Notation

To disambiguate different types of tuples in the formalization, we prefix a calligraphically typeset version of the set name in front of the tuple. For instance, nodes, which belong to the set **N**, are typeset as a tuple of the form $\mathcal{N}\langle \_, \_, \_, \_, \_ \rangle$.

We use underscores as "wildcards" to indicate that the value of a certain tuple member is inconsequential in the current evaluation rule. Since the tuple member is not bound to a variable, the tuple member cannot be referred to later: different occurrences of underscores in a single formula or rule are not required to refer to the same value. This notational convention, common in some programming languages, hence makes it easier to follow complex formalisms, as the flow of data is easier to follow. In other words: $\mathcal{N}\langle \iota_n, \_, \_, \_, \_ \rangle$ means "a node whose identifier is ι_n, and whose successors, inbox, outbox, and node-local data could be any value that one of those fields of a node may have".

Notation

Sets of identifiers are written as $\mathbf{I}_\_$, where the kind of identifier is indicated by a single letter where the underscore is. Canonical members of this set are written as $\iota_\_$.

For instance, the set $\mathbf{I_N}$ contains node identifiers. A canonical node identifier is ι_n.

A node's tuple elements are the node's identifier, ι_n; a vector of successors, $\bar{\mathfrak{s}}$; a vector of messages, $\overline{m}$; a vector of tokens, $\bar{t}$; and the node-local data *ndata*. This last element is a member of the set **D**. A subset of **D** exists for each conceptual type of PARTE graph node. In the rudimentary PARTE model, these are alpha nodes, join-nodes, not-nodes, terminal nodes, and production nodes.[5] Formally:

$$ndata \in \mathbf{D} = (\mathbf{D}_{alpha} \cup \mathbf{D}_{join} \cup \mathbf{D}_{not} \cup \mathbf{D}_{terminal} \cup \mathbf{D}_{production}).$$

Nodes are identified by node identifiers, the set of which is named $\mathbf{I_N}$ in accordance to our notational convention. Successors are pairs of a node identifier and an activation side, which is either "from the left" (typeset as $\searrow$), "from the top" (typeset as $\downarrow$), or "from the right" (typeset as $\swarrow$). Conceptually, a successor pair $\mathfrak{s}$ specifies an out-edge from the current node, along with an indicator of how the node is activated: for Rete-nodes with two predecessors, a distinction must be made between data arriving from the left predecessor and data arriving from the right predecessor. In PARTE, his data take the form of messages. A message m is a pair of an activation side and a *token*. We define what a token ($t \in \mathbf{T}$) looks like in PARTE on page 109.

The vector of messages $\overline{m}$ represents the node's inbox: $\overline{m}$ contains all tokens that need to be processed at this node, in order, together with the side by which the node is activated

[5]We deviate from the model by Forgy [59] by distinguishing between "side-effecting" terminal nodes, and event-generating production nodes. The semantics of both node types are discussed further in this chapter.

by this token. The vector of tokens $\bar{t}$ plays a complementary role, serving as the node's *outbox*. To percolate tokens through the PARTE graph, messages in a node's outbox $\bar{t}$ have to be appended to the inboxes of all the successor nodes listed in $\bar{\mathfrak{s}}$, using the activation side listed in $\bar{\mathfrak{s}}$ for the new message.

Formally, the introduced sets are the following:

Element ∈ Set	Name
$\iota_n \in \mathbf{I_N}$	Node identifier
$\mathfrak{s} \in \mathbf{I_N} \times \{\searrow, \downarrow, \swarrow\}$	Successor
$m \in \{\searrow, \downarrow, \swarrow\} \times \mathbf{T}$	Message

The sizes of $\overline{m}$ and $\bar{t}$ are fixed, as is usual for vectors, but are not specified in detail here. As we will show later, in the rudimentary PARTE model, any size is too small to deal with arbitrary rule sets. Still, any size greater than zero produces a model that works for some rule sets. Section 5.3.7 defines the required size of the inboxes and outboxes in the full-fledged PARTE model, fwPARTE. In section 8.3 we prove that that size is sufficient to guarantee proper evaluation.

5.2.2 Node-local Data

The node-local data of the five types of PARTE graph nodes in the rudimentary model, is defined as follows:

Element	∈ Set	Name
$\mathcal{D}_{alpha}\langle \iota_t, c_\alpha, \iota_e \rangle$	$\in \mathbf{D}_{alpha}$	Alpha node-local data
$\mathcal{D}_{join}\langle c_\beta, \overline{pm}, \bar{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, dist \rangle$	$\in \mathbf{D}_{join}$	Join-node-local data
$\mathcal{D}_{not}\langle c_\beta, \overline{pm}, \bar{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, dist \rangle$	$\in \mathbf{D}_{not}$	Not-node-local data
$\mathcal{D}_{production}\langle g^{e}_{\Box} \rangle$	$\in \mathbf{D}_{production}$	Production node-local data
$\mathcal{D}_{terminal}\langle\rangle$	$\in \mathbf{D}_{terminal}$	Terminal node-local data

Notation

We refer to a node whose node-local data is in $\mathbf{D}_{alpha}$ as an "alpha node", a node whose node-local data is in $\mathbf{D}_{join}$ a "join-node", etc.

By design, the different types of PARTE graph nodes have a different format of node-local data: a tuple with exactly the data that is relevant to that type of node. As a result, the semantic constructs for the different types of node-local data contain mostly all types of semantic constructs that have not been explained yet. A first, high-level overview is this:

Alpha Nodes implement an individual event pattern. In isolation, a pattern is matched based on the type of event, and its attributes. A successful match results in the binding of the event to a variable defined in the rule-based program. Hence, an alpha node must accept or reject events of a certain type, and provide the identifier to bind successful matches to. That event type is identified by a template identifier ι_t. The template identifier names a relation, as specified in section 4.2.1. Templates themselves are a vector $\overline{\iota_a}$ identifying the attributes which events must have to be adhering to that template. The predicate that accepts or rejects the event is identified by c_α, and the variable to bind to is abstracted into an event identifier ι_e. Such an event identifier is a member of the set $\mathbf{I_E}$, which contains identifiers for all events mentioned in a rule.

Note that ι_es do not uniquely identify elements in **E**. Instead, they uniquely identify an event pattern location. For instance, consider the PARTElang implementation of driver scenario A from listing 7.2. An event identifier might identify "the pattern where `incoming` is bound in rule `FraudPassThrough`", not "the event at timestamp t, bound to `incoming` in rule `FraudPassThrough`".

Join-Nodes and Not-Nodes implement the join or anti-join between multiple patterns. Hence, both join- and not-nodes must check join- criteria. As a Rete-derived model, PARTE stores partial matches, which PARTE stores in the join- and not-nodes. A partial match represents, as the name implies, a partial result of a matching operation. Specifically, a partial match maps a set of event identifiers onto concrete events which can be bound to the variable represented by the event identifiers.

Both join- and not-nodes store the temporal join-criteria (i.e., temporal distances between events) as *dist*, and the other join-criteria as a predicate c_β. Temporal distances are differences between timestamps, such that a *temporal distance function* specifies how far from the *reference event* another event may occur. For a given *reference event pattern* with event identifier ι_{ea}, and another event pattern with event identifier ι_{eb}, $dist(\iota_{eb})$ yields a pair $\langle \Delta_{\theta min}, \Delta_{\theta max} \rangle$ where $\Delta_{\theta min}$ is the minimum temporal distance between the events bound to ι_{ea} and ι_{eb}, and $\Delta_{\theta max}$ is the maximum temporal distance.

Consider for instance a PARTElang fragment `where { a in b [5s, 15s]}`, which requires that the event bound to logical variable `a` occurs at least 5 seconds after the event bound to `b`, and at most 15 seconds after `b`. A call to $dist(b)$ should return a pair $\langle 5s, 15s \rangle$. For a given match to `b` occurring at timestamp t, each event satisfying `a` which occurs in the time interval $[t + 5s, t + 15s]$, matches the compound event. The converse holds too: for a given match to `a` occurring at timestamp t, each event satisfying `b` which occurs in the time interval $[t - 15s, t - 5s]$, matches the compound event. In other words, a call to $dist(a)$ should return a pair $\langle -15s, -5s \rangle$. Temporal distances can be negative. A negative minimum temporal distance means that the event may occur before the reference element. A negative maximum temporal distance means that the event must occur before the reference event.

Finally, the partial matches from left predecessors are stored as a vector of partial matches, $\overline{pm}$; and those of right predecessors as a vector of events, $\overline{e}$. The event identifiers from left and right predecessors are stored as $\overline{\iota_{e_\searrow}}$ and $\iota_{e_\swarrow}$, respectively.

Production Nodes implement the emission of new events in response to matching an entire rule. Event emission must create a new event, based on attributes of the matched events, as specified in section 4.2 (specifically with respect to limitations such as *stratification* as defined on page 67). The construction of a new event based on attributes and temporal data of a partial match is abstracted into an event generation expression, $g^e_\square$.

Terminal Nodes represent the locations where an entire rule program is successfully matched. Since the PARTE model does not define what to do at that point, terminal nodes contain no additional data.

Notation

The type of a mapping, i.e., function, from a type 1 to a type 2 is typeset as $\mathbf{M_{1 \rightharpoonup 2}}$. For instance, a partial match is a mapping from event identifiers (i.e., the set $\mathbf{I_E}$) to events (i.e., the set $\mathbf{E}$), and the set of partial matches is typeset as $\mathbf{M_{I_E \rightharpoonup E}}$.

Element access is typeset using indexing notation. For instance, for a partial match $pm \in \mathbf{M_{I_E \rightharpoonup E}}$, the element bound to event identifier ι_e in pm is typeset as $pm[\iota_e]$.

The semantic constructs we introduced are formally specified as follows:

Element $\in$ Set	Name
$\iota_t \in \mathbf{I_T}$	Template identifier
$\iota_e \in \mathbf{I_E}$	Event identifier
$c_\alpha \in \mathbf{E} \to \mathbb{B}$	Alpha condition
$c_\beta \in (\mathbf{M_{I_E \rightharpoonup E}}) \to \mathbb{B}$	Beta condition
$pm \in \mathbf{M_{I_E \rightharpoonup E}}$	Partial match
$g^e_\square \in (\mathbf{M_{I_E \rightharpoonup E}}) \to \mathbf{E}$	Event generator expression
$\Delta_\theta \in \Delta_\Theta = \mathbf{\Theta}$	Temporal distance
$\text{dist} \in \mathbf{M_{I_E \rightharpoonup (\Delta_\Theta \times \Delta_\Theta)}}$	Temporal distance function

5.2.3 Tokens

When the Rete algorithm [59] exchanges logical facts between Rete nodes, those facts are wrapped in *tokens*. Similarly, PARTE wraps events in tokens when sending them between nodes. Tokens come in two flavors: *alpha tokens* and *beta tokens*. Alpha tokens wrap exactly one event's data. Beta tokens wrap one partial match, which can incorporate one or more events' data. To define tokens, we introduce the following semantic constructs:

Element $\in$ Set		Structure	Name
$t \in \mathbf{T}$	$=$	$\boldsymbol{T_\alpha} \cup \boldsymbol{T_\beta}$	Token
$t_\alpha = \mathcal{T}_\alpha\langle e\rangle \in \boldsymbol{T_\alpha}$	$=$	$\mathbf{E}$	Alpha token
$t_\beta = \mathcal{T}_\beta\langle pm\rangle \in \boldsymbol{T_\beta}$	$=$	$\mathbf{M_{I_E \rightharpoonup E}}$	Beta token

5.2.4 Events

Rules in PARTE reason about streams of events. In line with our definition in the event algebra EA on page 81, events in a stream have a timestamp. PARTE stores the timestamp separately from the non-temporal attributes. Finally, an event must identify its *template*. To accommodate these concepts, the rudimentary PARTE model defines the following semantic constructs:

Element $\in$	Set		Structure	Name
$e = \mathcal{E}\langle\iota_t, attrs, \theta\rangle \in$	$\mathbf{E}$	$=$	$\mathbf{I_T} \times \mathbf{M_{I_A \rightharpoonup V}} \times \mathbf{\Theta}$	Event
$\theta \in$	$\mathbf{\Theta}$	$\subset$	$\mathbb{R}$	Timestamp
$\iota_a \in$	$\mathbf{I_A}$			Attribute identifier
$attrs \in$	$\mathbf{M_{I_A \rightharpoonup V}}$			Attributes
$v \in$	$\mathbf{V}$	$\subset$	$\mathbb{R}$	Value

Events are defined as a three-tuple with tuple constructor $\mathcal{E}$. They contain the identifier of the event's template, values for each of the event's attributes, and the event's timestamp. The time at which an event occurred is represented by an element from $\mathbf{\Theta}$, the set of timestamps. Canonical elements of $\mathbf{\Theta}$ are typeset as θ. A full order is defined on $\mathbf{\Theta}$, such that for any two timestamps $\theta 1, \theta 2 \in \mathbf{\Theta} : (\theta 1 < \theta 2) \veebar (\theta 1 = \theta 2) \veebar (\theta 1 > \theta 2)$.

An event's attributes are represented as a mapping from attribute identifiers to concrete values. Values are the most fundamental type of data present in the PARTE model. **V** contains all primitive value which PARTE can reason about. In the formal model, this is limited to a finite subset of the real numbers. Values can appear as constants in a PARTElang program, or be entered into the model in the role of attributes to events. Canonical elements of the set **V** are typeset as v.

5.2.5 The Global Evaluation Language

The percolation of tokens through the PARTE graph is specified by the global evaluation language $\xrightarrow{g}$.

The global evaluation language consists of two reduction rules: (Proc) and (Prop). (Proc) specifies how each token in a node's inbox is processed using the local evaluation language $\xrightarrow{l}$, whose rules are listed further down in this section. (Prop) specifies how tokens are propagated from a node's outbox to all its successors' inboxes. Both inference rules in $\xrightarrow{g}$ transform a set of PARTE graph nodes to a new set of PARTE graph nodes, and are hence functions in $\mathcal{P}(\mathbf{N}) \to \mathcal{P}(\mathbf{N})$.

Notation

We typeset inference rules using the common notation. For instance, an inference rule named (SomeRule) which describes that a evaluates to b via language c if d equals e and f is greater than g, is typeset as follows:

(SomeRule)

$$\frac{d = e \qquad f > g}{a \xrightarrow{c} b}$$

The Processing Rule

(Proc) specifies that an arbitrary node can be selected from the nodes **N** that make up the PARTE graph, when that node has at least one item in the inbox (i.e., the vector located in third position in the tuple has some first element, $\langle side, t\rangle$). A tuple $\langle data, side, t\rangle$ must then be created. This tuple must be processed by executing a single step of the local inference language $\xrightarrow{l}$, resulting in a tuple $\langle data', \overline{t}'\rangle$. The first element in this tuple represents the updated node-local data, the second element holds all tokens generated in this step of $\xrightarrow{l}$. The outbox of the selected node must have room for the produced tuples. A replacement must be substituted for the selected node, which inherits the node's identifier, successors and inbox. The replacement node's outbox is the selected node's outbox extended with the new tokens $\overline{t}'$. The replacement node's node-local data is provided in $data'$.

(Proc) is specified as follows:

(Proc)

$$\frac{\langle data, side, t\rangle \xrightarrow{l} \langle data', \overline{t'}\rangle \qquad has\text{-}room\text{-}for(\overline{t}, \left|\overline{t'}\right|)}{\{\mathcal{N}\langle \iota_n, \overline{\mathfrak{s}}, \langle side, t\rangle \cdot \overline{m}, \overline{t}, data\rangle\} \sqcup \mathbf{N}_{rest} \xrightarrow{g} \mathbf{N}_{rest} \sqcup \{\mathcal{N}\langle \iota_n, \overline{\mathfrak{s}}, \overline{m}, \overline{t} \diamond \overline{t'}, data'\rangle\}}$$

Notation

We use syntax slightly different from what is common here: $a \cdot \overline{b}$ does not indicate extending the vector $\overline{b}$ with an element a in the front, but rather that $\overline{b}$ is a vector of the same length as the vector $a \cdot \overline{b}$, but where

- each element in $\overline{b}$ occurs in $a \cdot \overline{b}$, in the same order, but at an index one less in b than in $a \cdot \overline{b}$;
- the first element of $a \cdot \overline{b}$ holds the same value as what is named by expression a; and
- $\overline{b}$ has one more empty spot at the end of the vector than $a \cdot \overline{b}$.

For instance, let vector v_a be a vector of length 5, defined as $(1\ 2\ 3\ \emptyset\ \emptyset)$, where empty spots are represented as empty sets. Let $v_a = x \cdot v_b$. Then $x = 1$ and $v_b = (2\ 3\ \emptyset\ \emptyset\ \emptyset)$.

Similarly, the notation $\overline{c} \diamond \overline{d}$ does not result in a vector with a size equal to the sum of the size of both vectors. Instead, the elements in $\overline{d}$ are inserted into the empty spots of $\overline{c}$. The operator $\diamond$ can only be applied if *has-room-for*$(\overline{c}, \left|\overline{d}\right|)$, i.e., if there are enough empty spots in $\overline{c}$ to store all elements of $\overline{d}$. For instance, $(1\ 2\ \emptyset\ \emptyset\ \emptyset) \diamond (3\ 4) = (1\ 2\ 3\ 4\ \emptyset)$.

After (Proc), the PARTE graph consists of all nodes in $\mathbf{N}$ except for the selected node (i.e., $\mathbf{N}_{rest}$), unioned with a singleton set containing the replacement node. Conceptually, this can be thought of as "updating" the selected node to a new state in $\mathbf{N}$. In subsequent rules we occasionally phrase the replacement of a node by a node with the same identity, but different state, as "updating".

The Propagation Rule

The second inference rule in the global evaluation language $\xrightarrow{g}$ specifies that an arbitrary node can be selected from the nodes $\mathbf{N}$ that make up the PARTE graph, when that node has at least one item in the outbox (i.e., the vector located in fourth position in the tuple has a non-empty first element, t). The subset $\mathbf{N}_b$ of $\mathbf{N}$ is defined as all nodes whose identifier ι_b appears in the first position of a tuple in $\overline{\mathfrak{s}_a}$. That is to say, $\mathbf{N}_b$ is the set of *successor* nodes of the selected node, or, equivalently, the set of nodes reachable in a single step through an out-edge of the selected node.

If all successor nodes have room in their inboxes to receive a message, (Prop) continues by specifying that the successor nodes must be updated, yielding $\mathbf{N}'_b$. $\mathbf{N}'_b$ contains an updated node for each node in $\mathbf{N}_b$, inheriting the old node's identifier ι_b, successors $\overline{\mathfrak{s}_b}$, outbox $\overline{m_b}$, and data $data_b$, but appending a tuple $\langle side, t\rangle$ to the inbox, where *side* is the second element in the tuple in $\overline{\mathfrak{s}_a}$ where the first element is that node's identifier. The source node must be updated by removing t from its outbox.

(Prop) is specified as follows:

(Prop)

$$\frac{\begin{array}{c}\mathbf{N}_b = \mathbf{N} \cap \{\mathcal{N}\langle \iota_b, \_, \_, \_, \_\rangle \mid \langle \iota_b, \_\rangle \in \overline{\mathfrak{s}_a}\} \\ \forall \mathcal{N}\langle \_, \_, \overline{m_b}, \_, \_\rangle \in \mathbf{N}_b : \textit{has-room-for}(\overline{m_b}, 1) \\ \mathbf{N}'_b = \{\mathcal{N}\langle \iota_b, \overline{\mathfrak{s}_b}, \overline{m_b} \cdot \langle side, t\rangle, \overline{t_b}, data_b\rangle \mid \mathcal{N}\langle \iota_b, \overline{\mathfrak{s}_b}, \overline{m_b}, \overline{t_b}, data_b\rangle \in N_b, \langle \iota_b, side\rangle \in \overline{\mathfrak{s}_a}\}\end{array}}{\{\mathcal{N}\langle \iota_a, \overline{\mathfrak{s}_a}, \overline{m_a}, t \cdot \overline{t_a}, data_a\rangle\} \sqcup \mathbf{N}_b \sqcup \mathbf{N}_{rest} \xrightarrow{g} \{\mathcal{N}\langle \iota_a, \overline{\mathfrak{s}_a}, \overline{m_a}, \overline{t_a}, data_a\rangle\} \sqcup \mathbf{N}'_b \sqcup \mathbf{N}_{rest}}$$

After (Prop), the PARTE graph consists of all nodes in $\mathbf{N}_{rest}$, unioned with both a singleton set containing the updated version of the selected node, and with the set of updated destination nodes, $\mathbf{N}'_b$.

5.2.6 The Local Evaluation Language

We just defined the global evaluation language $\xrightarrow{g}$. Evaluation steps in $\xrightarrow{g}$ transform a set of PARTE graph nodes into an updated set of PARTE graph nodes, by either propagating tokens to successors (through (Prop)), or by having a token processed by the local evaluation language $\xrightarrow{l}$ (through (Proc)). From the usage of $\xrightarrow{l}$ in $\xrightarrow{g}$, it is evident that a step in $\xrightarrow{l}$ takes a three-tuple in $\mathbf{D} \times \{\searrow, \downarrow, \swarrow\} \times \mathbf{T}$, and produces a pair in $\mathbf{D} \times \mathcal{P}(\mathbf{T})$. That is to say: some node-local data, an activation side, and a token gets transformed into an updated state for the node-local data, and zero or more tokens.

The set $\mathbf{D}$ in the rudimentary PARTE model was defined to be $\mathbf{D}_{alpha} \cup \mathbf{D}_{join} \cup \mathbf{D}_{not} \cup \mathbf{D}_{terminal} \cup \mathbf{D}_{production}$ (see section 5.2.2). We now describe the semantics of each of those node types by defining the evaluation at each type of node by the local evaluation language $\xrightarrow{l}$.

Alpha Nodes

Alpha nodes filter events based on their template and attributes.

Remember from section 5.2.2 that alpha tokens whose event's template does not match the one specified in the alpha token are to be discarded. Alpha tokens whose events' attributes satisfy the constraints encoded in c_α are to be wrapped in beta tokens and propagated to successors of the alpha node. Unlike the traditional Rete model [59], the PARTE model does not store partial matches that satisfy all constraints in an "alpha memory". Instead, a beta token containing the events' data is propagated to all successors, who are responsible for storing the data they require, and discarding it as appropriate. Refer again to figure 5.2 on page 98. Instead of storing partial matches along the edges, PARTE includes the storage with the successor nodes.

Also unlike the traditional Rete model, alpha nodes in the PARTE model form the boundary between the alpha and the beta network. Some traditional Rete implementations have two-input nodes' right-hand-side input be alpha tokens, and let the two-input nodes be the boundary between the alpha and beta network. The traditional approach requires that the first two-input node of any rule performs a special "dummy" join between "nothing" on the left, and the first pattern on the right. We did not depict this in our figure on page 98, as this conceptually only complicates matters, and PARTE does not follow that approach anyway. In PARTE, alpha nodes convert their input to beta tokens, and two-input nodes always operate on streams of beta tokens. As such, the uppermost two-input nodes need not be special-cased. Beta tokens encode events in a mapping from event identifier to event. Since an alpha node is responsible for binding an event to a logical variable in a pattern in a rule, a one-on-one mapping exists between alpha nodes and event identifiers. The event identifier ι_e to which matching events can be bound, is thus stored in the alpha nodes.

Evaluation at an alpha node by the local evaluation language $\xrightarrow{l}$ proceeds as follows:

(Alpha-NoMatchTemplate)

$$\frac{e = \mathcal{E}\langle \iota_{te}, \_, \_\rangle \qquad \iota_{tn} \neq \iota_{te}}{\langle \mathcal{D}_{alpha}\langle \iota_{tn}, c_\alpha, \iota_e\rangle, \downarrow, \mathcal{T}_\alpha\langle e\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{alpha}\langle \iota_{tn}, c_\alpha, \iota_e\rangle, \emptyset\rangle}$$

(Alpa-NoMatchConstraint)

$$\frac{e = \mathcal{E}\langle \iota_{te}, \_, \_\rangle \qquad \iota_{tn} = \iota_{te} \qquad c_\alpha(e) \xrightarrow{\alpha} \text{false}}{\langle \mathcal{D}_{alpha}\langle \iota_{tn}, c_\alpha, \iota_e\rangle, \downarrow, \mathcal{T}_\alpha\langle e\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{alpha}\langle \iota_{tn}, c_\alpha, \iota_e\rangle, \emptyset\rangle}$$

(Alpha-Match)

$$\frac{e = \mathcal{E}\langle \iota_{te}, \_, \_\rangle \qquad \iota_{tn} = \iota_{te} \qquad c_\alpha(e) \xrightarrow{\alpha} \text{true}}{\langle \mathcal{D}_{alpha}\langle \iota_{tn}, c_\alpha, \iota_e\rangle, \downarrow, \mathcal{T}_\alpha\langle e\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{alpha}\langle \iota_{tn}, c_\alpha, \iota_e\rangle, \{\mathcal{T}_\beta\langle [\iota_e \mapsto e]\rangle\}\rangle}$$

Evaluation of an alpha node can commence whenever an alpha token $\mathcal{T}_\alpha\langle e\rangle$ is available. If the alpha token's event conforms to a different event template than the template of the alpha node, the token is discarded without further consideration, as described in (Alpha-NoMatchTemplate). If the event template matches the one stored, the event is subjected to the alpha predicate c_α, which verifies the alpha condition. If $c_\alpha(e)$ evaluates to false, the alpha token is discarded without further consideration, as described in (Alpha-NoMatchConstraint). If instead $c_\alpha(e)$ evaluates to true, a beta token binding the event's data to the event identifier ι_e is produced (as seen in (Alpha-Match)), to be sent to the successors by the (Prop) rule of the global evaluation language $\xrightarrow{g}$.

Join-Nodes

Join-nodes perform a relational join between the stream of beta tokens arriving from their left predecessor, and the stream of beta tokens arriving from their right predecessor.

The creation of these joint relations happens incrementally as new tokens arrive. To enable the incremental joining, previous tokens must be remembered. Join-nodes are hence stateful. Join-nodes store partial matches they receive. Tokens are stored in either a *left partial match history* or a *right partial match history*, depending on whether the token arrived from the left or the right predecessor. As a result, join-nodes behave significantly differently from alpha nodes: whereas alpha nodes generate zero or one token per token they receive, join-nodes respond to the reception of a token by producing between zero and $|\overline{pm}|$ tokens for a right activation, and between zero and $|\overline{e}|$) tokens for a left activation. A boolean predicate over the attributes of the events being joined, c_β, represents the non-temporal join-conditions, if any. The temporal join-conditions define a lifetime, and through that a semantic window.

Evaluation at a join-node by the local evaluation language $\xrightarrow{l}$ proceeds as follows:

(Join-$\searrow$)

$$\frac{\begin{array}{c} \textit{has-room-for}(\overline{pm}, 1) \qquad\qquad \overline{pm}' = \overline{pm} \cdot pm_{t_\beta} \\ \textit{window}_{\swarrow} = \textit{semantic-window}(\{[\iota_{e_\swarrow} \mapsto e] \mid e \in \overline{e}\}, \textit{lifetime}_{\searrow}(\textit{dist}, \textit{timestamps}(pm_{t_\beta}))) \\ \overline{pm_{\text{new}}} = \{pm_{t_\beta} \cdot pm_{\swarrow} \mid pm_{\swarrow} \in \textit{window}_{\swarrow}, c_\beta(pm_{t_\beta} \cdot pm_{\swarrow}) \xrightarrow{\beta} \text{true}\} \end{array}}{\begin{array}{c} \langle \mathcal{D}_{join}\langle c_\beta, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \searrow, \mathcal{T}_\beta\langle pm_{t_\beta}\rangle\rangle \\ \xrightarrow{l} \langle \mathcal{D}_{join}\langle c_\beta, \overline{pm}', \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \{\mathcal{T}_\beta\langle pm\rangle \mid pm \in \overline{pm_{\text{new}}}\}\rangle \end{array}}$$

(Join-$\swarrow$)

$$\frac{\begin{array}{c} \textit{has-room-for}(\overline{e}, 1) \qquad\qquad \overline{e}' = \overline{e} \cdot e_{t_\beta} \\ \textit{window}_{\searrow} = \textit{semantic-window}(\overline{pm}, \textit{lifetime}_{\swarrow}(\textit{dist}, [\iota_{e_\swarrow} \mapsto \textit{timestamp}(e_{t_\beta})])) \\ \overline{pm_{\text{new}}} = \{pm_{\searrow}[\iota_{e_\swarrow} \mapsto e_{t_\beta}] \mid pm_{\searrow} \in \textit{window}_{\searrow}, c_\beta(pm_{\searrow}[\iota_{e_\swarrow} \mapsto e_{t_\beta}]) \xrightarrow{\beta} \text{true}\} \end{array}}{\begin{array}{c} \langle \mathcal{D}_{join}\langle c_\beta, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \swarrow, \mathcal{T}_\beta\langle [\iota_{e_\swarrow} \mapsto e_{t_\beta}]\rangle\rangle \\ \xrightarrow{l} \langle \mathcal{D}_{join}\langle c_\beta, \overline{pm}, \overline{e}', \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \{\mathcal{T}_\beta\langle pm\rangle \mid pm \in \overline{pm_{\text{new}}}\}\rangle \end{array}}$$

Evaluation of (Join-$\searrow$) can only commence if and only if the partial match history *has-room-for* at least one more partial match. If it does, left-activation of a join-node by a beta token $\mathcal{T}_\beta\langle pm_{t_\beta}\rangle$ proceeds by appending the token's partial match pm_{t_β} to the node's left partial match history $\overline{pm}$, yielding $\overline{pm}'$. Subsequently, the lifetime for the new partial match is calculated. Using this lifetime, a semantic window $\textit{window}_{\swarrow}$ is constructed: the set of all partial matches from the right partial match history (i.e., $\overline{e}$) which need to be joined with pm_{t_β}, based only on their timestamps. A set of new partial matches $\overline{pm_{\text{new}}}$ is

$$\text{intersection } \odot : \mathbf{\Theta}^2 \to \mathbf{\Theta}^2 \to \mathbf{\Theta}^2$$
$$\langle a, b\rangle \odot \langle c, d\rangle = \langle \max\{a, c\}, \min\{b, d\}\rangle$$

$$\text{lifetime}_{\searrow} : (\mathbf{I_E} \to \Delta_{\mathbf{\Theta}}{}^2) \to \mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{\Theta}} \to \mathbf{\Theta}^2$$
$$\text{lifetime}_{\searrow}(dist, \ell map) = \bigodot_{\iota_e \in \text{dom}(\ell map)} \langle \ell map[\iota_e] + dist(\iota_e)_{min}, \ell map[\iota_e] + dist(\iota_e)_{max}\rangle$$

$$\text{lifetime}_{\swarrow} : (\mathbf{I_E} \to \Delta_{\mathbf{\Theta}}{}^2) \to \mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{\Theta}} \to \mathbf{\Theta}^2$$
$$\text{lifetime}_{\swarrow}(dist, [\iota_e \mapsto \ell]) = \langle \ell - dist[\iota_e]_{max}, \ell - dist[\iota_e]_{min}\rangle$$

$$\mathit{timestamp} : \mathbf{E} \to \mathbf{\Theta}$$
$$\mathit{timestamp}(\mathcal{E}\langle \_, \_, \theta\rangle) = \theta$$

$$\mathit{timestamps} : \mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{E}} \to \mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{\Theta}}$$
$$\mathit{timestamps}(pm) = \{(\iota_e \mapsto \theta) \mid pm[\iota_e] = \mathcal{E}\langle \_, \_, \theta\rangle\}$$

$$\text{semantic-window} : \mathcal{P}(\mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{E}}) \to \mathbf{\Theta}^2 \to \mathcal{P}(\mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{E}})$$
$$\mathit{semantic\text{-}window}(\overline{pm}, \langle \theta_{start}, \theta_{end}\rangle) = \{pm \mid pm \in \overline{pm}, \forall \mathcal{E}\langle \_, \_, \theta\rangle \in pm : (\theta \geq \theta_{start}) \wedge (\theta \leq \theta_{end})\}$$

Figure 5.4: The auxiliary functions for the rudimentary PARTE model — Lines in green list the types of the functions, while lines in black list their behavior.

constructed as all sets $pm_{t_\beta} \cdot pm_{\swarrow}$ for each $pm_{\swarrow} \in \mathit{window}_{\swarrow}$, for which the condition c_β holds. Finally, each match in $\overline{pm}_{\text{new}}$ is wrapped in a beta token. The set of all beta tokens so generated forms the set tokens produced by this step, which is placed in the second place of the resulting tuple.

Right-activation of a join-node proceeds similarly if the node's right partial match history *has-room-for* at least one more partial match: the token's partial match here is the singleton mapping from the left event identifier $\iota_{e_{\swarrow}}$ to an event e_{t_β}. This mapping must be appended to the right partial match history $\overline{e}$, yielding the updated partial match history $\overline{e}'$. A window $\mathit{window}_{\searrow}$ is constructed, consisting of all partial matches from the left partial match history ($\overline{pm}$) which are in the semantic window with respect to the new partial match. A set of matches $\overline{pm}_{\text{new}}$ is constructed as all sets $pm_{\searrow} \cdot pm_{t_\beta}$ for each $pm_{\searrow} \in \mathit{window}_{\searrow}$, for which the condition c_β holds. Finally, each of these $\overline{pm}_{\text{new}}$ is wrapped in a beta token, and presented as the set of produced tokens.

The auxiliary functions used in the inference rules above are defined in figure 5.4.

Not-Nodes

Not-nodes or anti-join-nodes implement *negation* in the PARTE model.

Not-nodes behave largely similar to join-nodes, with the crucial difference that not-nodes propagate tokens when the relational join fails, and discards the tokens for which a join succeeds.

Whereas in join-nodes both predecessors are in principle interchangeable,[6] not-nodes attribute a different semantics to both predecessors: the left predecessor provides the positive tokens, and the right predecessor provides the negated tokens. Upon receipt of a positive token, this token gets stored into the left partial match history ($\overline{pm}$). The right partial match history ($\overline{e}$) is searched for matching tokens. When a matching negated token is found, i.e., when the negated pattern is satisfied, the anti-join failed for that positive token. The positive token hence is discarded. Inversely, upon receipt of a negated token, that token gets stored into the right partial match history. The left partial match history is searched for matching tokens. When a matching positive token is found, the anti-join failed for that positive token, and the positive token gets discarded. The negated token is kept indefinitely, as it can still match with future positive tokens.

The general structure of the rules for evaluation at not-nodes is similar to that of the rules at join-nodes. The major difference is in how evaluation proceeds when successful matches are found. Instead of collecting all valid matches, and wrapping them in beta tokens for propagation, the rules perform the following actions, depending on the activation side, and on whether a match exists:

(Not-$\searrow$-Match) : when left-activated by a token for whose partial match a match exists on the right, the negation failed. Nothing is produced;

(Not-$\searrow$-NoMatch) : when left-activated by a token for whose partial match no match exists on the right, the negation succeeded. The incoming beta token is propagated;

(Not-$\swarrow$-Match) : when right-activated by a token for whose partial match a match exists on the left, the negation failed. Nothing is produced;

(Not-$\swarrow$-NoMatch) : when right-activated by a token for whose partial match no match exists on the left, the negation succeeded. All left partial matches in the correct semantic window, but which do not match, get wrapped in a beta token and propagated.

Note that (Not-$\swarrow$-NoMatch) can propagate partial matches that have been propagated before. This is one of the shortcomings of the rudimentary PARTE model, and is solved in the full-fledged PARTE model defined in section 5.4.

[6]A relational join is associative and commutative when the resulting projection is considered as a mapping, not an ordered tuple.

In the rudimentary model of PARTE, evaluation at a not-node by the local evaluation language $\xrightarrow{l}$ is nearly identical to the evaluation at a join-node. It proceeds as follows:

(Not-$\searrow$-Match)

$$\frac{\begin{array}{c} \textit{has-room-for}(\overline{pm},1) \qquad \overline{pm}' = \overline{pm} \cdot pm_{t_\beta} \\ \textit{window}_{\swarrow} = \textit{semantic-window}(\{[\iota_{e_\swarrow} \mapsto e] \mid e \in \overline{e}\}, \textit{lifetime}_{\searrow}(\textit{dist}, \textit{timestamps}(pm_{t_\beta}))) \\ \exists\, e_\swarrow \in \textit{window}_\swarrow : c_\beta(pm_{t_\beta}[\iota_{e_\swarrow} \mapsto e_\swarrow]) \xrightarrow{\beta} \text{true} \end{array}}{\langle \mathcal{D}_{not}\langle c_\beta, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \searrow, \mathcal{T}_\beta\langle pm_{t_\beta}\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{not}\langle c_\beta, \overline{pm}', \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \emptyset\rangle}$$

(Not-$\searrow$-NoMatch)

$$\frac{\begin{array}{c} \textit{has-room-for}(\overline{pm},1) \qquad \overline{pm}' = \overline{pm} \cdot pm_{t_\beta} \\ \textit{window}_{\swarrow} = \textit{semantic-window}(\{[\iota_{e_\swarrow} \mapsto e] \mid e \in \overline{e}\}, \textit{lifetime}_{\searrow}(\textit{dist}, \textit{timestamps}(pm_{t_\beta}))) \\ \forall\, e_\swarrow \in \textit{window}_\swarrow : c_\beta(pm_{t_\beta}[\iota_{e_\swarrow} \mapsto e_\swarrow]) \xrightarrow{\beta} \text{false} \end{array}}{\begin{array}{c} \langle \mathcal{D}_{not}\langle c_\beta, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \searrow, \mathcal{T}_\beta\langle pm_{t_\beta}\rangle\rangle \\ \xrightarrow{l} \langle \mathcal{D}_{not}\langle c_\beta, \overline{pm}', \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \{\mathcal{T}_\beta\langle pm_{t_\beta}\rangle\}\rangle \end{array}}$$

(Not-$\swarrow$-Match)

$$\frac{\begin{array}{c} \textit{has-room-for}(\overline{e},1) \qquad \overline{e}' = \overline{e} \cdot e_{t_\beta} \\ \textit{window}_{\searrow} = \textit{semantic-window}(\overline{pm}, \textit{lifetime}_{\swarrow}(\textit{dist}, [\iota_{e_\swarrow} \mapsto \textit{timestamp}(e_{t_\beta})])) \\ \exists\, pm_\searrow \in \textit{window}_\searrow : c_\beta(pm_\searrow[\iota_{e_\swarrow} \mapsto e_{t_\beta}]) \xrightarrow{\beta} \text{true} \end{array}}{\begin{array}{l} \langle \mathcal{D}_{not}\langle c_\beta, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \swarrow, \mathcal{T}_\beta\langle [\iota_{e_\swarrow} \mapsto e_{t_\beta}]\rangle\rangle \\ \xrightarrow{l} \langle \mathcal{D}_{not}\langle c_\beta, \overline{pm}, \overline{e}', \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \emptyset\rangle \end{array}}$$

(Not-$\swarrow$-NoMatch)

$$\frac{\begin{array}{c} \textit{has-room-for}(\overline{e},1) \qquad \overline{e}' = \overline{e} \cdot e_{t_\beta} \\ \textit{window}_{\searrow} = \textit{semantic-window}(\overline{pm}, \textit{lifetime}_{\swarrow}(\textit{dist}, [\iota_{e_\swarrow} \mapsto \textit{timestamp}(e_{t_\beta})])) \\ \forall\, pm_\searrow \in \textit{window}_\searrow : c_\beta(pm_\searrow[\iota_{e_\swarrow} \mapsto e_{t_\beta}]) \xrightarrow{\beta} \text{false} \\ \overline{t'_\beta} = \{\mathcal{T}_\beta\langle pm\rangle \mid pm \in \textit{window}_\searrow\} \end{array}}{\begin{array}{l} \langle \mathcal{D}_{not}\langle c_\beta, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \swarrow, \mathcal{T}_\beta\langle [\iota_{e_\swarrow} \mapsto e_{t_\beta}]\rangle\rangle \\ \xrightarrow{l} \langle \mathcal{D}_{not}\langle c_\beta, \overline{pm}, \overline{e}', \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \overline{t_\beta}'\rangle \end{array}}$$

Production Nodes

Production nodes effectuate reactions when a rule's entire pattern is successfully matched.

They do so by producing a new event. The new event's attributes are derived from the values bound in the token that activates the production node. In addition to straightforwardly copying values, the model supports basic arithmetic operations involving the attributes of events, as well as constants. For instance, part of the driver scenarios (page 12) could be implemented by summing the amount of money transferred by a number of fraudulent transfers, and emitting a new compound event that contains the sum. The exact scope of operations needed to implement the language is described in section 4.2. Here, it suffices to know that these transformations take the form of expressions dependent on a partial match (which themselves are a mapping from event identifier to event), which produces an event. We typeset those event generator expressions as $g^e_\square$.

In the rudimentary model of PARTE, the evaluation at a production node by the local evaluation language $\xrightarrow{l}$ proceeds as follows:

(Prod)

$$\frac{g^e_\square(pm) \longrightarrow e'}{\langle \mathcal{D}_{production}\langle g^e_\square\rangle, \downarrow, \mathcal{T}_\beta\langle pm\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{production}\langle g^e_\square\rangle, \{\mathcal{T}_\alpha\langle e'\rangle\}\rangle}$$

Terminal Nodes

Terminal nodes represent the point at which an entire program is successfully matched.

When the PARTE model is used in a rule engine that is deployed in the real world, activation of terminal nodes should produce side-effects in the outside world, e.g., print something to a computer screen, send a network package, etc. In PARTE, tokens are consumed at terminal nodes by merely discarding them. The behavior of a terminal node is hence as follows:

(Term)

$$\frac{}{\langle \mathcal{D}_{terminal}\langle\rangle, \downarrow, \mathcal{T}_\beta\langle \_\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{terminal}\langle\rangle, \emptyset\rangle}$$

5.3 Formal Concepts for an Operational Semantics of the PARTE Language

The rudimentary model of PARTE presented in the previous section has a number of shortcomings that prevent it from correctly implementing the logic that can be expressed in the PARTE language from section 4.2. Furthermore, it fails to provide the guarantees that section 4.1.2 identified as necessary for strongly reactive behavior. This section defines the concepts necessary to reason about the shortcomings of the rudimentary model. It further defines how the concerns identified by those concepts can be addressed.

5.3.1 Opposite Activation Side and Opposite Partial Match History

Two-input nodes have, as the name implies, two inputs. One of them is referred to as the “left” input, the other is referred to as the “right” input. Tokens from the left input *left-activate* the two-input node, and tokens from the right input *right-activate* the two-input node. We typeset everything related to left-activations with “$\searrow$”, and everything related to right-activation with “$\swarrow$”.[7]

In some cases, values from both inputs need to be combined in some way. To ease further discussions, we introduce the concept of *opposite activation sides*. While rather obvious in context, we define it here to preclude possible ambiguity: the opposite activation side of a left-activation, is a right-activation, and vice versa. There is no opposite activation side to a top-activation (“$\downarrow$”).

More interesting is the notion of an *opposite partial match history*. We refer to the vector of previous partial matches that are stored in a two-input node as a *partial match history*. When a new partial match arrives, it has to be combined with the partial matches in the *opposite partial match history*, defined as follows:

Definition 13: Opposite partial match history

The opposite partial match history of the left *partial match history* of a two-input node is the right partial match history of that node.

The opposite partial match history of the right *partial match history* of a two-input node is the left partial match history of that node.

The opposite partial match history of a token that left-activates a two-input node is the right partial match history of that node.

The opposite partial match history of a token that right-activates a two-input node is the left partial match history of that node.

5.3.2 Partial Match Arrival Rate

We have previously touched upon the notion of “the rate of events” in our event algebra (section 4.3) and in the discussion of related work (section 4.6.1). In the latter we identified a problem with bounding input size: the input for a PARTElang program is potentially

[7]Activation from the top only happens in one-input nodes. As such, no disambiguation is necessary, so arrows from the top are rarely necessary.

unbounded, and an analysis that does not take into account the interleaving of arrival of input and discarding of results based on those inputs, can only conclude that resource usage grows without bounds. Central to the shortcoming of a such a general analysis, is that such analyses attempt to relate arrival of new events with evaluation steps taken elsewhere in the program under analysis. These assumptions do not hold for event processing system like PARTE, where new events do not arrive because of the progress of the program under analysis, but instead new events arrive because some event occurred in the outside world.

Evidently, in an event processing setting, a program (or any part of a program running in parallel) had better finished processing an event before the next event reaches that program (part), for otherwise the new event cannot yet be processed. The subsequent event might have to wait for the previous two events to be processed. The next event might then be delayed even more, and so on. At this point, the systems flow of control is no longer dictated by the arrival of events, but by the internal control structure. The system is — in other words — no longer strongly reactive. To address this, some notion of "the progression of real elapsed time" must be included into the model. New events arrive because time has progressed. The model executed another step because time has progressed. The former is handled by our definition for *event arrival rate* on page 65, which defines the link between the progression of time, and the arrival of new events.

In PARTE, event arrival rate is defined per event source. An event arrival rate for a certain type of event can hence for instance be "500 events per second". This does not only mean that in any second at most 500 events may be emitted. It also means that during any interval of 2 milliseconds, at most one event may be emitted. For instance: while receiving 5 events in a 1 millisecond interval, followed by 999 milliseconds where no event arrives, has an average rate of only 5 events per second, we consider the event arrival rate of such a setup to be at least 5 events per millisecond.

The concept of *event arrival rate* can be generalized: we define the rate at which (anti-)joins are generated as the *partial match arrival rate*.

- For **entry nodes**, the partial match arrival rate is equal to the sum of the event arrival rates of the event sources.
- The partial match arrival rate of **alpha nodes** is determined by the rate at which entry nodes propagate events, i.e., alpha nodes' partial match arrival rate is equal to the sum of the event arrival rates for the events of the type of the pattern represented by the alpha node.
- For the **left-most join-node in a graph**, i.e., the node joining the first two patterns in a PARTElang rule, the partial match arrival rates are equal to the partial match arrival rates of both alpha nodes whose result is joined by the join node.
- For **join-nodes and not-nodes lower in the graph**, as well as for **production nodes and terminal nodes**, the partial match arrival rates are determined by the rate at which predecessor (anti-)join-nodes produce partial matches.

Upper bounds on the *partial match production rates* of these predecessors can be computed as follows:

- A join node whose left partial match arrival rate is *r_ml*, whose right partial match arrival rate is *r_mr*, whose left partial match history has size $s_{\searrow}$, and whose right partial match history has size $s_{\swarrow}$, has a partial match production rate of $r_{\searrow} \times s_{\swarrow} + r_{\swarrow} \times s_{\searrow}$. Plainly: for every new partial match arrival, at most the number of partial matches in the opposite partial match history can be produced through joining.

- Anti-joining produces only partial matches which originate from the left predecessor. Hence, only the partial matches stored in the left partial match history need to be considered. An upper bound on the partial match production rate of not-nodes is hence $r_{\swarrow} \times s_{\searrow}$.

Definition 14: Partial match arrival rate

The rate at which partial matches arrive at a PARTE graph node from a predecessor node, i.e., the *partial match arrival rate* is expressed as a number of partial matches per time unit.

Like the *event arrival rate* (definition 2), partial match arrival rate is an absolute upper bound, not an average upper bound. The partial match arrival rate is defined by the minimum time between subsequent arrivals of partial matches.

5.3.3 Stale Partial Matches

The PARTE language defined in section 4.2 requires all event patterns in a rule to be temporally related to each other by fixed, finite minimum and maximum temporal distances. For any two event patterns p_1 and p_2 in a rule, a finite upper bound $\Delta_{\theta}^{p_1,p_2}$ must exist, such that any event that matches p_2 may occur at a timestamp at most $\Delta_{\theta}^{p_1,p_2}$ later than an event matching p_1, if those events are to satisfy the temporal join-criteria.

When processing an event e_{new}'s data at a two-input node, the temporal constraints place each event e_{other} from the opposing partial match history either before the minimum allowed timestamp, between the minimum and maximum allowed timestamps, or after the maximum allowed timestamp. We refer to those situations as being respectively *before the semantic time window*, *in the semantic time window*, or *after the semantic time window*:

Definition 15: Semantic time window of events

Consider two event patterns $p_1, p_2 \in P_{EA}$. Let the minimum temporal distance between p_1 and p_2 be $\Delta^{p_1,p_2}_{\theta\ min}$, and the maximum temporal distance between both be $\Delta^{p_1,p_2}_{\theta\ max}$. Consider two events $e_1, e_2 \in \mathbf{E}$ which match p_1 and p_2, respectively.

With respect to event e_2, event e_1 is said to be

$$\begin{cases} \textbf{before the semantic window} \text{ if } e_1.\theta < e_2.\theta + \Delta^{p_1,p_2}_{\theta\ min} \\ \textbf{in the semantic window} \text{ if } e_1.\theta \geq e_2.\theta + \Delta^{p_1,p_2}_{\theta\ min} \text{ and } e_1.\theta < e_2.\theta + \Delta^{p_1,p_2}_{\theta\ max} \\ \textbf{after the semantic window} \text{ if } e_1.\theta \geq e_2.\theta + \Delta^{p_1,p_2}_{\theta\ max} \end{cases}$$

Only partial matches of which all events are in the semantic time window with respect to the event being processed, can lead to successful matches. An event occurs "in the semantic time window of a partial match" if the event is in the semantic time window of each event in the partial match. Conversely, an event outside of the semantic time window of at least one event in a partial match, occurs outside of the semantic time window of the the partial match. We generalize this notion for pairs of partial matches:

Definition 16: Semantic time window of partial matches

Two partial matches are in each other's *semantic time window* if their events are in each other's semantic time windows.

It follows that a partial match outside the semantic time window of all partial matches in the *opposite* partial match history of a two-input node does not influence the creation of new partial matches in the current evaluation step. A partial match outside of the semantic time window of all partial matches that are currently in the opposite partial match history, and those that will be in the opposite match history, will hence not influence the creation of partial matches in any evaluation step. We call such partial matches *stale*:

Definition 17: Stale partial match

Consider a partial match history $\overline{pm}$ of a PARTE graph node n, and a partial match pm which is not in $\overline{pm}$.

The partial match pm is a *stale partial match* in the context of $\overline{pm}$, as well as a *stale partial match* in the context of $\overline{pm}$ extended with pm, if the set of all tokens produced by any future sequence of evaluation steps of both the global evaluation language $\xrightarrow{g}$ and the local evaluation language $\xrightarrow{l}$ at n, would be identical between the case *a.)* that $\overline{pm}$ is extended with pm (growing the size of $\overline{pm}$ by one); and in the case *b.)* that pm is not added to $\overline{pm}$.

5.3.4 Expiration

The Rete algorithm [59] defines the notion of assertion and retraction of logical facts. Assertion and retraction are handled by means of positive and negative tokens respectively. We sketched in section 5.1 how both kinds of tokens percolate through a Rete graph. Positive tokens contribute to the construction of partial matches in the knowledge base, whereas negative tokens cause the destruction of all partial matches depending on the data contained in the negative token.

In PARTE, logical facts are replaced by events. For events, assertion takes the form of event emission. However, events need not be retracted: events that occurred do so indefinitely. Events that occurred cannot not retroactively not have occurred.[8] Semantically, no event that entered the system is ever retracted from it. However, removing stale partial matches from the system has — by definition — no impact on which complex events are detected by the model.[9] Teodosiu and Pollak [133] introduced the notion of partial matches whose "lifetime has expired". Their paper shows how individual inter-event temporal distances between pairs of event patterns can be used to construct a complete graph of indirect distances between all event patterns. Using this graph, a reachability analysis can efficiently be implemented for determining which partial matches are stale. Partial matches that have been determined to be stale can then cheaply be removed: when no logical negation is involved, it suffices to remove the partial match from the partial match history data structure. When logical negation is involved, the removal of a fact can still necessitate the reevaluation of a negated condition.

To prevent ambiguity, we explicitly define expiration as follows:

Definition 18: Expiration

The removal of stale partial matches from a node's partial match histories is called *expiration*.

By definition of staleness, expiring a stale partial match does not remove information that is necessary for matching the streams of input events to the rule set implemented by the PARTE graph.

One of the core contributions of the PARTE model is enabling the detection of stale partial matches in a distributed complex event processing model. Crucial to this is the fact that staleness of a partial match in a node can be determined using only node-local data and a single token, i.e., the input provided to a single evaluation step of the local evaluation language $\xrightarrow{l}$. In the majority of the cases, this means no additional messages must

[8]An obvious exception to this is in the context of reasoning with uncertainty. In that context, new information may indicate a previously held believe was unjustified, necessitating the retraction of that belief. In PARTE, events are considered factually true: as far as the model is concerned, every event that is said to have occurred, is considered as undeniably having occurred.

[9] An *implementation* can be impacted, though, as without the removal of stale partial matches, the system eventually runs out of memory.

be sent between the PARTE graph nodes to implement expiration, i.e., no additional synchronization has to take place.

The principle of expiration in a distributed setting is as follows: despite the absence of a global clock, each external event source knows its own local time. Each external event source can hence determine a lower bound on the timestamps of events that it will still generate: the current time of its local clock. This lower bound is included in alpha tokens, such that recipients inside the PARTE graph can reason about these lower bounds. Each node in the PARTE graph is responsible for keeping track of the oldest timestamp that its predecessors may still send; a sort of *logical clock* [90] we hinted at in section 5.1.5. Each node also informs successors of the lower bound on timestamps they can still expect from them, by including the lower bounds in the tokens they produce. When a node is informed by its predecessor that no more events older than some timestamp t can still arrive for a certain event pattern, the node can run its reachability analysis to expire all partial matches which cannot match with corresponding events younger than t. The correctness of the distributed staleness analysis is proven in section 8.2.

5.3.5 Rule Matching

To detect complex events described in a rule set, an inferencing algorithm is required. Pattern detection algorithm like Rete differ from other detection techniques such as finite state automata in the way they model the notion of *matching* the input. For instance, DFAs indicate that their input matched the pattern they implement by entering an *accepting state*. Pattern detection models such as Rete do not have an accepting state, since evaluation is not over once an input is matched: more inputs may match or fail, possibly even without requiring additional input.[10] Still, even though it is not a unique point, the point at which a rule is matched, is important for real-world use cases: reaction logic must be executed, notifications must be sent to external systems, etc.

We define *rule matching* as follows:

Definition 19: Rule matching

A rule is considered *matched* whenever the rule's terminal node pops a token with a partial match from its inbox.

Rule matching is a transient occurrence, taking place while the local evaluation language $\xrightarrow{l}$ evaluates at a terminal node. The PARTE model does not react in any way to the matching of rules.

[10]The last event that was input, might match with multiple partial matches stored inside the Rete network. This entails that multiple matches can be generated in response to a single new input event.

5.3.6 Node States

Events can be emitted to a PARTE system, and the PARTE system will expire events when they become stale. This by itself is not sufficient to prevent the storage requirements from growing without bounds. As explained in chapter 4, this is necessary for PARTE to remain strongly reactive.

We previously introduced one concept that helps imposing limits on storage requirements: the event arrival rate. Without a bounded event arrival rate, the storage requirements are trivially without limit: when events arrive faster than the system can process them, events keep piling up. But even with a fixed event arrival rate, limiting storage size inside a PARTE graph is far from trivial because of nondeterminism due to scheduling of the concurrent evaluation at PARTE graph nodes. The global evaluation language $\xrightarrow{g}$ describes how some PARTE graph node can be selected for evaluating a step at, but does not specify which one.

In a real-world implementation, selecting which node to process at would be done by a scheduling algorithm. Evaluation steps involving distinct subsets of nodes could be executed in parallel (see section 6.1.8). Scheduling can cause temporary imbalance in execution resources,[11] which means that some subsets of the PARTE graph can make progress while another subset does not. The side which received most compute resources must store more data than ideal, since the lack of progress of the resource-starved side prevents partial matches at the faster side from becoming stale, and hence prevents those partial matches from being expired. Additionally, the resource-starved side must store more data than ideal, in the form of messages stored into the inboxes which could not be processed yet by lack of compute resources.

To tackle the problem of imbalance in allocated compute resources, the PARTE model was presented with node inboxes, node outboxes, and partial match histories of fixed, finite size. This entails that the size of a PARTE network is finite and fixed at PARTE graph construction time. To accommodate this restriction, both $\xrightarrow{g}$ and $\xrightarrow{l}$ prevent evaluation at a node when sufficient storage is not available at the node to execute the evaluation step. What remains to be shown, then, is that a PARTE graph whose node inboxes, node outboxes, and partial match histories are limited to a certain maximum size, still implements the semantics of the rule set from which the graph was compiled.

The rules outlined in section 5.4 do not allow evaluation to produce incorrect tokens (as we prove in chapter 8). Hence, there are only two conceivable ways in which a PARTE graph can fail to correctly implement the semantics of the rule set from which the graph was compiled: *a.)* either a node takes on an erroneous configuration from which it cannot recover, or *b.)* multiple nodes take on configurations which jointly form an erroneous configuration from which they cannot recover. In both cases, the result is that progress can no longer be made by the PARTE graph, i.e., that neither $\xrightarrow{g}$ nor $\xrightarrow{l}$ define evaluation

[11]A fair scheduler will continuously balance the load to overcome these imbalances, though imbalances will intermittently exist.

steps of which the preconditions hold. We refer to the first situation as a *blocked node*, and the second as a *blocked subgraph*.

Since PARTE disallows circular dependencies by enforcing stratification (as defined on page 67), blocked subgraphs which are not caused by blocked nodes cannot occur. When evaluation at a set of nodes is only temporarily impossible, but will become possible again once other nodes make progress, those nodes are said to be in a *waiting state*.[12] At any point in time, a node is either *evaluating*, *blocked*, *waiting*, or *suspended*.

Definition 20: Evaluating state

A PARTE graph node $n \in \mathbf{N}$ is in an *evaluating state* during the evaluation of any rule of $\xrightarrow{g}$ or $\xrightarrow{l}$.

Definition 21: Blocked state

A PARTE graph node $n \in \mathbf{N}$ is in a *blocked state* when it is not in an evaluating state, but the number of tokens that would be put in its outbox by a single step of the (Proc) rule of $\xrightarrow{g}$ is larger than the capacity of the outbox, and the number of tokens that would be put in its inbox by a single step of the (Prop) rule of $\xrightarrow{g}$ is larger than the capacity of the inbox.

Additionally, a two-input PARTE graph node $n \in \mathbf{N}$ is also in a *blocked state* when both its partial match histories are filled with non-stale partial matches, i.e., when none of the partial matches can be expired (as that would mean valid matches are not detected by the system), yet there is no room for another match, such that none of the stored partial matches can be expired.

Definition 22: Waiting state

A PARTE graph node $n \in \mathbf{N}$ is in a *waiting state* when n is not evaluating or blocked, but either *a.)* the evaluation of $\xrightarrow{l}$ at n cannot be initiated by the (Proc) rule in $\xrightarrow{g}$ because of the conditions in (Proc); *b.)* the conditions of the (Prop) rule in $\xrightarrow{g}$ cannot be satisfied with n as the source node; or *c.)* there is no evaluation rule in $\xrightarrow{l}$ that allows evaluating at n.

Definition 23: Suspended state

A PARTE graph node $n \in \mathbf{N}$ is in a *suspended state* when it is not in a blocked state, nor in a waiting state, nor in an evaluating state.

[12]Crucially, those other nodes must be able to make progress despite this node's waiting state. Otherwise, the nodes would be deadlocked. Deadlocked nodes would constitute a blocked subgraph, not a waiting subgraph.

When evaluation at a node cannot proceed because storage is lacking, that node is either waiting or blocked. When the situation can be resolved, the node is waiting for other nodes to resolve the problem. Otherwise, it is indefinitely blocked: the problem lies with the node that is blocked, and can hence only be resolved at that node, but since no evaluation at that node is possible, the situation cannot be resolved.

When global progress in the PARTE network becomes impossible because storage is lacking in some nodes, the entire network is blocked. It suffices that one node is blocked for the entire network to become blocked: being blocked is irrecoverable, and global process in a PARTE graph is not possible if parts are irrecoverably stuck.

Definition 24: Blocked PARTE network

A PARTE network is said to be *blocked* if at least one node is blocked.

5.3.7 Minimum Viable Size

From the notion of a *blocked node* we can derive the notion of a *minimum viable size* for the partial match histories, for the outboxes, and for the inboxes of nodes. The *minimum viable size* for one of these vectors is a node-specific amount that indicates how large that vector must be to prevent the node from entering a blocked state. We prove in section 8.3 that the equation defining the minimum viable size we define here for partial match histories indeed prevents two-input nodes from reaching a blocked state because of filled partial match histories.

Definition 25: Minimum viable partial match history size

The *minimum viable partial match history size* of a partial match history $\overline{pm}$ of a node n is the smallest integer i for which holds that n cannot become blocked by lack of storage space in that partial match history $\overline{pm}$ if and only if $|\overline{pm}| \geq i$.

Every (anti-)join defines temporal (anti-)join-constraints. These constraints define a time window. Furthermore, for each activation side of a (anti-)join, a partial match arrival rate is defined (definition 14, on page 121). Consider a two-input PARTE graph node with a time window of W time units, and a partial match arrival rate of r per time unit for one of the activation sides. We prove in section 8.3 that the minimum viable partial match history size for the opposite activation side of that node is $r \times W + 1$.

Definition 26: Minimum viable outbox size

The *minimum viable outbox size* of a node n is the smallest integer i for which holds that n cannot become blocked by lack of space in its outbox $\overline{t}$ iff $|\overline{t}| \geq i$.

For inboxes and outboxes, the proofs are sufficiently simple to be included in the running text below.

Statement

The minimum viable outbox size of a two-input node is the size of the largest partial match history.

Proof.

(i.) The (Proc) rule of $\xrightarrow{g}$ transfers messages to an outbox at the granularity of one step of $\xrightarrow{l}$, the minimum viable outbox size is at least the maximal size of the set of tokens generated in one step of $\xrightarrow{l}$ at n.

(ii.) The (Prop) rule of $\xrightarrow{g}$ transfers tokens out of a node's outbox one by one, so to ensure this can be done, the minimum viable outbox size is at least one.

(iii.) From $(i.)$ and $(ii.)$ it follows that the minimum viable outbox size of a node n is the maximum of one and the cardinality of the set of tokens generated by a single step of $\xrightarrow{l}$ at n. This latter is bounded by the size of the partial match histories of n: a relational (anti-)join between one element and a set of x elements produces at most the Cartesian product of both sides, i.e., at most x elements.

From $(iii.)$ it follows that the minimum viable outbox size is dictated by the partial match history size. □

The minimum viable outbox size of an entry node, an alpha node, and production node is trivially one: for each received token, at most one token can be generated. The minimum viable outbox size of a terminal node is trivially zero: terminal nodes cannot produce new tokens.

Definition 27: Minimum viable inbox size

The *minimum viable inbox size* of a node n is the smallest integer i for which holds that n cannot become blocked by lack of space in its inbox $\overline{m}_n$ iff $|\overline{m}_n| \geq i$.

Statement

The inbox of n, the minimum viable inbox size of all nodes is $|\{\searrow, \downarrow, \swarrow\}|$, i.e., three.

Proof.

(i.) The (Prop) rule of $\xrightarrow{g}$ transfers messages to an inbox one by one.

(ii.) Every node except entry nodes have at most one predecessor per activation side.

(iii.) As per definition 21, a blocked two-input node whose outbox and partial match history is not blocked, requires a single token from one of both activation sides to become unstuck.

From (*i.*) through (*iii.*), it suffices to have one slot in a node's inbox per activation side. □

5.4 The Featherweight PARTE Model

The rudimentary model defined in section 5.2 sketches the foundations of a model for reactive, distributed complex event detection. As we announced at its introduction, the rudimentary model has a number of shortcomings. In section 5.3, we provided the foundations for reasoning about those shortcomings. This section will now make the shortcomings more concrete, and then define a model that tackles those shortcomings: the Featherweight PARTE model.

5.4.1 Shortcomings of the First, Rudimentary PARTE Model

With the relevant concepts introduced in section 5.3, the shortcomings of the rudimentary PARTE model can concisely be summarized:

- Stale partial matches are not expired.
- Negation spuriously signals that no match is found, when it already signaled that before for the same event, or when a match can still occur.
- The size of the inbox, the outbox, and the partial match histories of PARTE graph nodes is not specified. In the rudimentary model there is no way of knowing what the minimum viable sizes for these vectors are. At any given static size, the vectors might be too small to perform the correct matching. In the case of dynamically resized vectors, the size can grow arbitrarily large, preventing strongly reactive enumeration of the elements inside.
- Evaluation in the rudimentary model uses expressions of the form *has-room-for*$(\_, |\_|)$. Such steps are not really evaluation steps: in the rudimentary model, evaluation cannot know how many results will be generated by an evaluation step, before taking that step. The model can hence not determine whether there will be sufficient room. An implementation could optimistically start evaluation, but if it then discovers that the evaluation should enter a waiting state, issues could arise in one of two ways: the execution stack of the evaluation either is stored, or it is discarded. The

former case introduces a hidden spatial cost. The latter case introduces a hidden temporal cost from repeatedly starting evaluation, and stopping when the unmet condition is reached. Either solution undermines strong reactivity. A proper model only depends on conditions which can be checked up front.

The remainder of this section defines the Featherweight PARTE model, fwPARTE for short. As with the definition of the rudimentary PARTE model in section 5.2, we define all semantic constructs in detail. We reuse the notational conventions introduced in that section. Since the big picture should be clear from that section, we structure the definition of fwPARTE in a bottom-up fashion: we specify the semantic constructs that form the events, then move to tokens and graph nodes. Finally, we define the local evaluation language and the global evaluation language.

5.4.2 Events

The most basic level of the semantics remains unchanged from the rudimentary model of PARTE: events consist of a template identifier, a mapping of attributes to values, and the timestamp at which the event occurred:

Element	$\in$	Set		Structure	Name
v	$\in$	$\mathbf{V}$	$\subset$	$\mathbb{R}$	Value
θ	$\in$	$\boldsymbol{\Theta}$	$\subset$	$\mathbb{R}$	Timestamp
ι_t	$\in$	$\mathbf{I_T}$			Template identifier
ι_a	$\in$	$\mathbf{I_A}$			Attribute identifier
$attrs$	$\in$	$\mathbf{M_{I_A \rightharpoonup V}}$	$=$	$\mathbf{I_A} \rightharpoonup \mathbf{V}$	Attributes
$e = \mathcal{E}\langle \iota_t, attrs, \theta \rangle$	$\in$	$\mathbf{E}$	$=$	$\mathbf{I_T} \times \mathbf{M_{I_A \rightharpoonup V}} \times \boldsymbol{\Theta}$	Event

Again, events are written as a tagged tuple $\mathcal{E}\langle \iota_t, attrs, \theta \rangle$ whenever their constituents are relevant to the evaluation rule, or abbreviated as e when they are not. Events consist of a template identifier ι_t, a set of values modeled as a mapping from attribute identifier to value, and the timestamp at which the event occurred.

5.4.3 Tokens

To deal with the shortcomings of the rudimentary PARTE model, fwPARTE keeps track of the possible timestamps of future events at any point in the PARTE graph. Since the fwPARTE model assumes a distributed memory model, only asynchronous communication is available. Timing data must be communicated by means of messages. To keep the messaging load low, timing data should be piggybacked on existing message send as much as possible. Concretely, the ideal situation is one where all timing information necessary for the correct evaluation of a rule set can be communicated by adding only a constant number of timestamps to alpha and beta tokens that would have been sent in a system following the rudimentary PARTE model. The Featherweight PARTE model partially achieves this goal, but cannot fully achieve it in the context of negation.[13]

In fwPARTE, events are often wrapped in tokens. Tokens normally wrap one or more events. Two kinds of tokens exist: alpha tokens and beta tokens. Unlike in the rudimentary PARTE model, in fwPARTE not all tokens wrap an event. Instead, some are merely sent to keep the logical clocks up to date. Such tokens contain a "no change" indicator ($\perp$) instead where the event data would otherwise be. Hence, alpha tokens wrap zero or one event, and beta tokens wrap zero, one, or more events. "No change" tokens constitute a bookkeeping overhead compared to the rudimentary model of PARTE, but are necessary to guarantee correctness in the context of negation, as we show in section 8.4.

In addition to either the data of exactly one event, or a "no change" indicator $\perp$, each alpha token contains metadata consisting of a lower bound on timestamps ℓ, as well as the unique identifier of the source of this information ι_s. Event sources are entities that can either be part of a PARTE graph (production nodes, see below), or lie outside of the PARTE model. In the latter case, the only part of the event source known to PARTE, is their identifier. The PARTE model only needs to know which source produces a certain event, i.e., the event source identifier $\iota_s \in \mathbf{I_S}$. In turn, event sources need only regularly generate alpha tokens with non-decreasing lower bounds on event timestamps.

Each beta token wraps metadata consisting of a lower bound on timestamps ℓmap, in addition to either a partial match, or $\perp$. A partial match pm maps an event identifier to an event. Each beta token with a partial match contains data for one or more events. The semantic constructs related to tokens in fwPARTE are defined as follows:

Element	$\in$ Set		Structure	Name
$\iota_e \in$	$\mathbf{I_E}$			Event identifier
$\iota_s \in$	$\mathbf{I_S}$			Source identifier
$pm \in$			$\mathbf{M_{I_E \rightharpoonup E}}$	Partial match
$\perp \in$				"No change" indicator
$\ell \in$	$\mathbf{L}$	$\subseteq$	$\mathbf{\Theta}$	Temporal lower bound
$\ell map \in$			$\mathbf{M_{I_E \rightharpoonup L}}$	Lower bound map
$p_\alpha \in$			$\mathbf{E} \cup \{\perp\}$	Alpha payload
$p_\beta \in$			$\mathbf{M_{I_E \rightharpoonup E}} \cup \{\perp\}$	Beta payload
$t \in$	$\mathbf{T}$	$=$	$\boldsymbol{T_\alpha} \cup \boldsymbol{T_\beta}$	Token
$t_\alpha = \mathcal{T}_\alpha\langle \iota_s, \ell, p_\alpha \rangle \in$	$\boldsymbol{T_\alpha}$	$=$	$\mathbf{I_S} \times \mathbf{L} \times (\mathbf{E} \cup \{\perp\})$	Alpha token
$t_\beta = \mathcal{T}_\beta\langle \ell map, p_\beta \rangle \in$	$\boldsymbol{T_\beta}$	$=$	$\mathbf{M_{I_E \rightharpoonup L}} \times (\mathbf{M_{I_E \rightharpoonup E}} \cup \{\perp\})$	Beta token

[13]Section 5.6 discusses how the extra message sends can be omitted when no negation is involved.

5.4.4 Graph Nodes

As in the rudimentary model of PARTE, the Featherweight PARTE model represent the graph encoding a rule set as a set of nodes. The nodes store the out-edges as part of their state. In the Featherweight PARTE model, the set $\mathbf{N}$ consists of tuples of the form $\mathcal{N}\langle \iota_n, \bar{\mathfrak{s}}, \overline{m}, \bar{t}, ndata\rangle$. The Featherweight PARTE model defines an additional type of node: entry nodes. Hence, the set $\mathbf{D}$ of node local data is defined as $\mathbf{D}_{entry} \cup \mathbf{D}_{alpha} \cup \mathbf{D}_{join} \cup \mathbf{D}_{not} \cup \mathbf{D}_{terminal} \cup \mathbf{D}_{production}$. Unchanged are the node identifier, successors, and messages, which still take the following forms:

Element ∈ Set	Name
$\iota_n \in \mathbf{I_N}$	Node identifier
$\mathfrak{s} \in \mathbf{I_N} \times \{\searrow, \downarrow, \swarrow\}$	Successor
$m \in \{\searrow, \downarrow, \swarrow\} \times \mathbf{T}$	Message

Nodes are uniquely identified by their node identifier ι_n. The set of successors, $\bar{\mathfrak{s}}$, refers to a node by its identifier, and indicates from which activation side the successor relation to that node is enacted. A message m represents a chunk of communication between nodes in the graph, and indicates from which side the recipient is activated, as well as the token to be processed.

All tokens that need to be processed at this node are contained in the node's inbox $\overline{m}$, in order of arrival, together with the side by which the node is activated by the message. The vector of tokens $\bar{t}$ plays a complementary role, serving as the node's outbox. A rule of the global evaluation language $\xrightarrow{g}$ makes use of these to percolate tokens through the graph: messages in a node's outbox $\bar{t}$ are appended to the inboxes of all the successor nodes listed in $\bar{\mathfrak{s}}$, using the activation side listed in $\bar{\mathfrak{s}}$ for the new message. The size of $\overline{m}$ must be set to the minimum viable inbox size. We proved in section 5.3.7 that a size of three is a proper minimum inbox size for all nodes. The size of $\bar{t}$ must be set to the minimum viable outbox size, i.e., sufficient room to accept all tokens produced by a step in the local evaluation language at the node. For join-nodes this means the largest of both minimum viable partial match history sizes. For not-nodes this is the minimum viable left partial match history size. For other types of nodes, $|\bar{t}| = 1$.

5.4.5 Node-local Data

Six types of nodes exist in fwPARTE. The semantic constructs of the six types of node-local data are defined as follows:

Element	$\in$ Set	Name
$\mathcal{D}_{entry}\langle \ell map, \iota_t\rangle$	$\in \mathbf{D}_{entry}$	Entry node-local data
$\mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle$	$\in \mathbf{D}_{alpha}$	Alpha node-local data
$\mathcal{D}_{join}\langle c_\beta, \ell map, \ell, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, dist\rangle$	$\in \mathbf{D}_{join}$	Join-node-local data
$\mathcal{D}_{not}\langle c_\beta, \ell map, \ell, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, dist\rangle$	$\in \mathbf{D}_{not}$	Not-node-local data
$\mathcal{D}_{terminal}\langle\rangle$	$\in \mathbf{D}_{terminal}$	Terminal node-local data
$\mathcal{D}_{production}\langle \iota_s, g^e_\square, g^\ell_\square\rangle$	$\in \mathbf{D}_{production}$	Production node-local data

These are largely identical to the ones defined by the rudimentary PARTE model on page 106, with four exceptions:

- node-local data for the new "entry node" is defined,
- the template identifiers of alpha nodes are moved to their predecessor entry nodes,
- node-local data of join-nodes and not-nodes keeps track of lower bounds for their left and right event identifiers, and
- the partial match histories of join-nodes and not-nodes are defined to be limited in size: they can not exceed their minimum viable size.

As with the definition of the rudimentary PARTE model higher up in this chapter, a more detailed description of these node-local data elements will be given in the next section. The next section describes how the local evaluation language $\xrightarrow{l}$ transforms tokens and node-local data to new tokens and a new state for node-local data.

For completeness, we first define the remaining semantic constructs:

Element $\in$ Set	Name
$c_\alpha \in \mathbf{E} \to \mathbb{B}$	Alpha condition
$c_\beta \in (\mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{E}}) \to \mathbb{B}$	Beta condition
$g^e_\square \in (\mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{E}}) \to \mathbf{E}$	Event generator expression
$g^\ell_\square \in (\mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{L}}) \to \mathbf{L}$	Temporal lower bound generator expression
$\Delta_\theta \in \Delta_\Theta = \mathbf{\Theta}$	Temporal distance
$dist \in \mathbf{M}_{\mathbf{I_E} \rightharpoonup (\Delta_\Theta \times \Delta_\Theta)}$	Temporal distance function

These elements, too, are largely identical to those defined in the rudimentary PARTE model on page 108. One new element was added: a temporal lower bound generator expression. In the order listed above: alpha and beta conditions can be applied to events or partial matches, respectively, to verify whether the condition holds. Event generator functions create new events based on a partial match, e.g., summing up amounts in a set of financial transactions. The novel temporal lower bound generator expressions calculate a new lower bound on timestamps, based on a set of lower bounds on timestamps. Temporal distances are differences between timestamps, and temporal distance functions specify how far from a reference event pattern another event pattern can match, as explained in the description of join- and not-nodes in section 5.2.2. Note again that a temporal distance may be negative: the order in which event patterns are specified in PARTElang does not dictate the order in which the events must occur to match; order is only determined by the explicit temporal distances.

5.4.6 The Local Evaluation Language

This section defines the local evaluation language $\xrightarrow{l}$ of the Featherweight PARTE model. A step in $\xrightarrow{l}$ takes a three-tuple in $\mathbf{D} \times \{\searrow, \downarrow, \swarrow\} \times \mathbf{T}$, and produces a pair in $\mathbf{D} \times \mathcal{P}(\mathbf{T})$. $\mathbf{D}$ is $\mathbf{D}_{entry} \cup \mathbf{D}_{alpha} \cup \mathbf{D}_{join} \cup \mathbf{D}_{not} \cup \mathbf{D}_{terminal} \cup \mathbf{D}_{production}$. The directions $\{\searrow, \downarrow, \swarrow\}$ indicate the side from which the node gets activated, and $\mathbf{T}$ are tokens.

For instance, the notation $\langle ndata, \searrow, t\rangle \xrightarrow{l} \langle ndata', \bar{t}'\rangle$ signifies a step in $\xrightarrow{l}$ where a node (represented by its node-local data *ndata*) gets activated from the left ($\searrow$) by token t, resulting in the new state for node-local data *ndata*, namely $ndata'$, and the production of all tokens in $\bar{t}'$. All tokens in $\bar{t}'$ are to be sent to the successors of the node of which *ndata* was the node-local data, and that node has to be removed and replaced by a node with identical identifier, successors, inbox, and outbox, but with $ndata'$ as the node-local data. These last few constraints are tackled by the global evaluation language $\xrightarrow{g}$, defined in the section after this one. The local evaluation language $\xrightarrow{l}$ is responsible for the generation of $ndata'$ and $\bar{t}'$.

We now define the local evaluation language $\xrightarrow{l}$, organized by the type of the node-local data it acts on.

Entry Nodes

Entry nodes manage the influx of all events of the same type from multiple sources.

One entry node must exist per event template. Entry nodes only process events of the correct template; the event sources must send events to the correct entry nodes. Entry nodes consolidate the distributed view held by the different event sources for a certain event type. Specifically, they determine the global lower bound on future timestamps of events for their type. The lower bound must be non-decreasing between consecutive events. Events' timestamps must be at least as large as the lower bound stored in the previous alpha token sent along the same path of the graph. If this constraint is violated, the entry node (as well as the entire fwPARTE graph) is in an invalid configuration, and processing cannot continue.

The node-local data of entry nodes is stateful, as they need to keep track of the non-decreasing lower bound on timestamps. They must guarantee that tokens they receive from a certain source are propagated in order. However, no order must be maintained between the reception of tokens from different sources. Indeed, the order in which tokens from multiple event sources are received, is not defined by the PARTE model up to the point at which the interleaving between the event streams is serialized by an entry node, thereby explicitly imposing an order. This entails that events are not guaranteed to be processed in order when multiple sources of a single event type exist. Since the lower bound on the timestamps is non-decreasing, however, progression of time can be tracked. Since event correlation (and importantly: negation) must specify temporal constraints, presence or absence of matches does not depend on the order in which events arrive. We prove this in section 8.1.

Three invariants are assumed to hold at entry nodes:

A lower bound must be known for each event source that can activate the entry node, i.e., the entry node must be aware of the existence of the source. If an alpha token generated by event source ι_s activates an entry node, it must hold that $\iota_s \in dom(s\ell map)$.

The lower bound for a source must not be larger than the token's lower bound, i.e., the lower bound must not retroactively be lowered. If an alpha token $\mathcal{T}_\alpha\langle \iota_s, \ell, \_\rangle$ activates an entry node, it must hold that $s\ell map[\iota_s] \leq \ell$.

The lower bound for a source must not be larger than the token's event's timestamp, i.e., the event must have occurred at a later timestamp than the timestamp up to which the event source asserted to have presented all events. If an alpha token $\mathcal{T}_\alpha\langle \_, \_, e\rangle$ activates an entry node, it must hold that $s\ell map[\iota_s] \leq e.\theta$.

These invariants are enforced initially by construction (section 5.5), and afterwards by the rules of $\xrightarrow{l}$ defined below. We prove that these rules uphold the invariants in chapter 8.

Evaluation of an entry node's data in the local evaluation language $\xrightarrow{l}$ proceeds as follows:

(Entry-NoMatchTemplate)

$$\frac{e = \mathcal{E}\langle \iota_{te}, \_, \_\rangle \qquad \iota_{tn} \neq \iota_{te}}{\langle \mathcal{D}_{entry}\langle s\ell map, \iota_{tn}\rangle, \downarrow, \mathcal{T}_\alpha\langle \iota_s, \ell, e\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{entry}\langle s\ell map, \iota_{tn}\rangle, \emptyset\rangle}$$

(Entry)

$$\frac{\begin{array}{cc} e = \mathcal{E}\langle \iota_{te}, \_, \_ \rangle & \iota_{tn} = \iota_{te} \\ s\ell map' = s\ell map[\iota_s \mapsto \ell] & \overline{t_\alpha} = \{\mathcal{T}_\alpha\langle \iota_s, \min(codom(s\ell map')), e\rangle\} \end{array}}{\langle \mathcal{D}_{entry}\langle s\ell map, \iota_{tn}\rangle, \downarrow, \mathcal{T}_\alpha\langle \iota_s, \ell, e\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{entry}\langle s\ell map', \iota_{tn}\rangle, \overline{t_\alpha}\rangle}$$

(Entry-$\bot$)

$$\frac{s\ell map' = s\ell map[\iota_s \mapsto \ell] \quad \overline{t_\alpha} = \{\mathcal{T}_\alpha\langle \iota_s, \min(codom(s\ell map')), \bot\rangle\}}{\langle \mathcal{D}_{entry}\langle s\ell map, \iota_{tn}\rangle, \downarrow, \mathcal{T}_\alpha\langle \iota_s, \ell, \bot\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{entry}\langle s\ell map', \iota_{tn}\rangle, \overline{t_\alpha}\rangle}$$

In brief, activation of an entry node can proceed in one of three ways, depending on whether the token contains an event of the wrong template, an event of the right template, or a "no change" indicator $\bot$. When an event of the wrong template is encountered, it is ignored without modifying the entry node's node-local data (Entry-NoMatchTemplate). When the token carries an event e of the correct template, evaluation proceeds as per the rule (Entry). First, an updated source-to-lower-bound map is computed: $s\ell map'$. $s\ell map'$ records the new lower bound ℓ for ι_e on top of the information stored previously in $s\ell map$. Second, the singleton set $\overline{t_\alpha}$ is created containing an alpha token sharing the input token's source and event, but carrying the minimum of the lower bounds stored in $s\ell map'$. Finally, a pair combining the updated state of the entry node's local data $\mathcal{D}_{entry}\langle s\ell map', \iota_{tn}\rangle$ and $\overline{t_\alpha}$ is produced.

When the token carries a "no change" indicator $\bot$ instead of an event, evaluation proceeds as per rule (Entry-$\bot$). Since no actual event is propagated, no constraint must be enforced. (Entry-$\bot$) proceeds analogously to (Entry): the new lower bound for ι_s is recorded into $s\ell map$, $\overline{t_\alpha}$ is constructed carrying the data of the incoming alpha token, but with the lower bound replaced by the lowest lower bound in $s\ell map$, and a pair containing the updated node-local data and $\overline{t_\alpha}$ is produced.

Alpha Nodes

Alpha nodes filter events based on the event's attributes.

Alpha nodes are the direct successors of entry nodes. As such, the type of events that can enter an alpha node is known up front. Alpha nodes enforce a set of constraints encoded in c_α. Alpha tokens whose events conform to c_α are wrapped in beta tokens to be propagated to successors. Events whose attributes fail at least one of the constraints are discarded.

As stated in the section on the rudimentary PARTE model above, PARTE differs from the traditional Rete model Forgy [59] in two ways with respect to alpha nodes. First, instead of having alpha and beta memories, events and partial matches are explicitly stored in the PARTE graph nodes that need the data. Second, alpha nodes produce beta tokens instead of alpha tokens. As such, an alpha node is responsible for checking all constant constraints on events (e.g., that an attribute is larger than five), as well as intra-event constraints (e.g., that one attribute is larger than another). All these requirements are abstracted over in the model, and must jointly be encoded in c_α.

Evaluation at an alpha node by the local evaluation language $\xrightarrow{l}$ proceeds as follows:

(Alpha-$\bot$)

$$\frac{}{\langle \mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle, \downarrow, \mathcal{T}_\alpha\langle \_, \ell, \bot\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle, \{\mathcal{T}_\beta\langle [\iota_e \mapsto \ell], \bot\rangle\}\rangle}$$

(Alpha-NoMatchConstraint)

$$\frac{c_\alpha(e) \xrightarrow{\alpha} \text{false}}{\langle \mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle, \downarrow, \mathcal{T}_\alpha\langle s, \ell, e\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle, \emptyset\rangle}$$

(Alpha-Match)

$$\frac{c_\alpha(e) \xrightarrow{\alpha} \text{true}}{\langle \mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle, \downarrow, \mathcal{T}_\alpha\langle s, \ell, e\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle, \{\mathcal{T}_\beta\langle [\iota_e \mapsto \ell], [\iota_e \mapsto e]\rangle\}\rangle}$$

The fwPARTE evaluation rules at alpha nodes differ from those in the rudimentary model in three ways. First, more data is contained in alpha and beta tokens.[14] Second, a new rule (Alpha-$\bot$) is added. (Alpha-$\bot$) defines that "no change" alpha tokens are converted into "no change" beta tokens, and propagated. Third, the rule (Alpha-NoMatchTemplate) is removed, as alpha tokens of non-matching templates can no longer arrive, since they would have been filtered out by the alpha node's predecessor entry node. The remainder of the semantics is the same. To reiterate, for tokens with event data, evaluation proceeds by subjecting the event e to the alpha condition c_α. Iff $c_\alpha(e)$ evaluates to true, a corresponding beta token is constructed. Otherwise, when $c_\alpha(e)$ evaluates to false, no beta token is constructed.

Join-Nodes

Join-nodes still perform an incremental relational join between the stream of beta tokens arriving from their left predecessor, and the stream of beta tokens arriving from their right predecessor.

In addition to storing the partial matches they receive, join-nodes track the lower bound on event times for both the left and right predecessor.

Since join-nodes merge two streams of events, they have a non-trivial job with respect to managing timestamps. Beta tokens created by a join-node must contain temporal

[14]Alpha tokens additionally contain a source identifier (omitted as a wildcard "_" in the rule) and a lower bound ℓ. Beta tokens additionally contain a mapping from event identifier to lower bound.

data of both the left partial match and the right partial match it is joining. The first part hereof was already formalized in the rudimentary model of PARTE: the mapping from event identifier to events originating from left partial matches (a $pm_{\searrow}$), is extended with a mapping from the right event identifier to the right event's timestamp (a $pm_{\swarrow}$, e.g., $[\iota_{e_{\swarrow}} \mapsto e_{\swarrow}]$).

In addition, though, in the Featherweight PARTE model the lower bound on event times must be computed and propagated to successors whenever a join is successfully made. As mentioned before, a join-node's behavior is exceptional in that in response to the reception of a token from one of the predecessors, possibly multiple stored tokens that were received from the predecessor at the opposite side must be taken into account. New tokens generated by a join-node can depend on data from two sources:

- tokens stored in the partial match histories; and
- tokens still to receive from predecessors.

Both data sources must hence be made available.

The first is trivially available in the form of the partial match histories themselves (i.e., in $\overline{pm}$ or in $\overline{e}$). For tokens still to receive from predecessors, the lower bounds are stored as $\ell map_{\searrow}$ for left predecessors, and $\ell_{\swarrow}$ for right predecessors.[15] The lower bounds on event timestamps for each event identifier are then computed as the minimum of the lower bound declared by the predecessor (i.e., one of the lower bounds stored in $\ell map_{\searrow}$ or the lower bound stored in $\ell_{\swarrow}$), and the oldest concrete event time of an event whose data is stored in one of the join-node's partial match histories.[16]

Evaluation at a join-node by the local evaluation language $\xrightarrow{l}$ proceeds as follows in the Featherweight PARTE model:

(Join-$\searrow$-$\bot$)

$$\frac{\begin{array}{c}\overline{e}' = expire_{\swarrow}(\overline{e}, expiration\text{-}time(dist, \ell map_{t_\beta})) \\ \ell map_{\text{implicit}} = min(\overline{timestamps}(\overline{pm}))[\iota_{e_{\swarrow}} \mapsto min(timestamps(\overline{e}'))] \\ \ell map_{\text{prop}} = min(\ell map_{\text{implicit}}, \ell map_{t_\beta}[\iota_{e_{\swarrow}} \mapsto \ell_{\swarrow}])\end{array}}{\begin{array}{c}\langle \mathcal{D}_{join}\langle c_\beta, \ell map_{\searrow}, \ell_{\swarrow}, \overline{pm}, \overline{e}, \overline{\iota_{e_{\searrow}}}, \iota_{e_{\swarrow}}, dist\rangle, \searrow, \mathcal{T}_\beta\langle \ell map_{t_\beta}, \bot\rangle\rangle \\ \xrightarrow{l} \langle \mathcal{D}_{join}\langle c_\beta, \ell map_{t_\beta}, \ell_{\swarrow}, \overline{pm}, \overline{e}', \overline{\iota_{e_{\searrow}}}, \iota_{e_{\swarrow}}, dist\rangle, \{\mathcal{T}_\beta\langle \ell map_{\text{prop}}, \bot\rangle\}\rangle\end{array}}$$

[15]Any complexity caused by the existence of multiple event sources for a single event identifier can be abstracted over thanks to the work done by entry nodes.

[16]Because of nondeterminism in the interleaving of messages from the left and right predecessor nodes, join-nodes can generate tokens out-of-order. A new token arriving from e.g., the left predecessors might match with an old token that was previously received on the right side, even when an earlier left token only matched younger right tokens. The computation of the lower bound on timestamps sketched in the preceding paragraph takes this into account. Since PARTE's correctness is based on the lower bounds on event timestamps instead of on concrete timestamps, this is acceptable behavior.

(Join-$\swarrow$-$\bot$)

$$\begin{array}{c}
\overline{pm}' = \mathit{expire}_{\searrow}(\overline{pm}, \ell_{t_\beta}) \\
\ell map_{\text{implicit}} = \mathit{min}(\overline{\mathit{timestamps}}(\overline{pm}'))[\iota_{e_\swarrow} \mapsto \mathit{min}(\mathit{timestamps}(\overline{e}))] \\
\ell map_{\text{prop}} = \mathit{min}(\ell map_{\text{implicit}}, \ell map_{\searrow}[\iota_{e_\swarrow} \mapsto \ell_{t_\beta}]) \\
\hline
\langle \mathcal{D}_{join}\langle c_\beta, \ell map_{\searrow}, \ell_{\swarrow}, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \swarrow, \mathcal{T}_\beta\langle[\iota_{e_\swarrow} \mapsto \ell_{t_\beta}], \bot\rangle\rangle \\
\xrightarrow{l} \langle \mathcal{D}_{join}\langle c_\beta, \ell map_{\searrow}, \ell_{t_\beta}, \overline{pm}', \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \{\mathcal{T}_\beta\langle \ell map_{\text{prop}}, \bot\rangle\}\rangle
\end{array}$$

(Join-$\searrow$)

$$\begin{array}{c}
\mathit{has\text{-}room\text{-}for}(\overline{pm}, 1) \qquad\qquad \overline{pm}' = \overline{pm} \cdot pm_{t_\beta} \\
\overline{e}' = \mathit{expire}_{\swarrow}(\overline{e}, \mathit{expiration\text{-}time}_{\swarrow}(\mathit{dist}, \ell map_{t_\beta})) \\
\mathit{window}_{\swarrow} = \mathit{semantic\text{-}window}(\{[\iota_{e_\swarrow} \mapsto e] \mid e \in \overline{e}'\}, \mathit{lifetime}_{\searrow}(\mathit{dist}, \mathit{timestamps}(pm_{t_\beta}))) \\
\overline{pm_{\text{new}}} = \{pm_{t_\beta} \cdot pm_{\swarrow} \mid pm_{\swarrow} \in \mathit{window}_{\swarrow}, c_\beta(pm_{t_\beta} \cdot pm_{\swarrow}) \xrightarrow{\beta} \text{true}\} \\
\ell map_{\text{implicit}} = \mathit{min}(\overline{\mathit{timestamps}}(\overline{pm}'))[\iota_{e_\swarrow} \mapsto \mathit{min}(\mathit{timestamps}(\overline{e}'))] \\
\ell map_{\text{prop}} = \mathit{min}(\ell map_{\text{implicit}}, \ell map_{t_\beta}[\iota_{e_\swarrow} \mapsto \ell_{\swarrow}]) \\
\hline
\langle \mathcal{D}_{join}\langle c_\beta, \ell map_{\searrow}, \ell_{\swarrow}, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \searrow, \mathcal{T}_\beta\langle \ell map_{t_\beta}, pm_{t_\beta}\rangle\rangle \\
\xrightarrow{l} \langle \mathcal{D}_{join}\langle c_\beta, \ell map_{t_\beta}, \ell_{\swarrow}, \overline{pm}', \overline{e}', \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \{\mathcal{T}_\beta\langle \ell map_{\text{prop}}, pm\rangle \mid pm \in \overline{pm_{\text{new}}}\}\rangle
\end{array}$$

(Join-$\swarrow$)

$$\begin{array}{c}
\overline{pm}' = \mathit{expire}_{\searrow}(\overline{pm}, \ell_{t_\beta}) \\
\mathit{has\text{-}room\text{-}for}(\overline{e}, 1) \qquad\qquad \overline{e}' = \overline{e} \cdot e_{t_\beta} \\
\mathit{window}_{\searrow} = \mathit{semantic\text{-}window}(\overline{pm}', \mathit{lifetime}_{\swarrow}(\mathit{dist}, [\iota_{e_\swarrow} \mapsto \mathit{timestamps}(e_{t_\beta})])) \\
\overline{pm_{\text{new}}} = \{pm_{\searrow} \cdot pm_{t_\beta} \mid pm_{\searrow} \in \mathit{window}_{\searrow}, c_\beta(pm_{\searrow} \cdot pm_{t_\beta}) \xrightarrow{\beta} \text{true}\} \\
\ell map_{\text{implicit}} = \mathit{min}(\overline{\mathit{timestamps}}(\overline{pm}'))[\iota_{e_\swarrow} \mapsto \mathit{min}(\mathit{timestamp}(\overline{e}'))] \\
\ell map_{\text{prop}} = \mathit{min}(\ell map_{\text{implicit}}, \ell map_{\searrow}[\iota_{e_\swarrow} \mapsto \ell_{t_\beta}]) \\
\hline
\langle \mathcal{D}_{join}\langle c_\beta, \ell map_{\searrow}, \ell_{\swarrow}, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \swarrow, \mathcal{T}_\beta\langle[\iota_{e_\swarrow} \mapsto \ell_{t_\beta}], [\iota_{e_\swarrow} \mapsto e_{t_\beta}]\rangle\rangle \\
\xrightarrow{l} \langle \mathcal{D}_{join}\langle c_\beta, \ell map_{\searrow}, \ell_{t_\beta}, \overline{pm}', \overline{e}', \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \{\mathcal{T}_\beta\langle \ell map_{\text{prop}}, pm\rangle \mid pm \in \overline{pm_{\text{new}}}\}\rangle
\end{array}$$

In contrast to the semantics of the rudimentary model, in fwPARTE the rules for evaluation at join-nodes must handle "no change" tokens. The first two rules shown above, handle such tokens. The rules update the state of the join-nodes based on the temporal lower bounds, and propagate "no change" to the join-nodes' successors. Following our notational conventions, the rule for left activation by "no change" tokens at join-nodes is called (Join-$\searrow$-$\bot$), while the rule for right activation by "no change" tokens at join-nodes is called (Join-$\swarrow$-$\bot$).

(Join-$\searrow$-$\bot$) defines that evaluation of $\xrightarrow{l}$ can proceed at a join-node if a beta token $\mathcal{T}_\beta\langle \ell map_{t_\beta}, \bot \rangle$ is present for left activation, and room exists in the left partial match history. Since this rule handles a left activation, it can render partial matches from the opposite partial match history, i.e., from the right partial match history, *stale*. Evaluation hence takes the following steps:

- First, an expiration time is determined: timestamp such that each right partial match stored in $\overline{e}$ older than that expiration time is stale. The expiration time is determined by the join-node's static temporal distances (i.e., *dist*) and the concrete temporal lower bounds of the new token (i.e., ℓmap_{t_β}). What remains of the partial match history after expiring the stale matches is called $\overline{e}'$.
- Second, the implicit temporal lower bounds are computed. Implicit temporal lower bounds are the temporal lower bounds due to the partial matches that are stored in the partial match histories. Every non-stale partial match can — by definition — still contribute to a valid partial match in a next step. Their timestamps are hence possible timestamps for partial matches that will be generated by the join-node, i.e., the temporal lower bounds at that join-node are at least as small as the stored partial matches' timestamps.
- Third, the propagated temporal lower bounds ℓmap_{prop} are computed. These are computed as the minimum of the explicit lower bounds (i.e., the updated $\ell map_{\searrow}$ and the updated $\ell_{\swarrow}$) and the implicit lower bounds for each event identifier. Finally, a single "no change" token is produced wrapping ℓmap_{prop}.

(Join-$\swarrow$-$\bot$) mirrors this behavior, but for right activation. Since expiration has to happen based on the singular right event's timestamp, computing the expiration time is trivial, so no call to *expiration-time* is needed.

Since the rules of the rudimentary model are not concerned with temporal lower bounds, and since (Join-$\searrow$-$\bot$) and (Join-$\swarrow$-$\bot$) are concerned only with temporal lower bounds, (Join-$\searrow$) and (Join-$\swarrow$) are largely just the combination of a rudimentary rule and the "no change" rule for one of both activation sides. As such, evaluation of (Join-$\searrow$) proceeds as follows if and only if there is room in the partial match history $\overline{pm}$:

- First, the partial match from the beta token is added to the partial match history, yielding $\overline{pm}'$.
- Second, stale matches in $\overline{e}$ expire, yielding $\overline{e}'$.
- Third, the lower and upper bound on timestamps for partial matches matching the incoming token is computed from the temporal constraints. All partial matches in $\overline{e}'$ between those bounds form the *semantic window*.
- Fourth, the new partial matches $\overline{pm_{\text{new}}}$ are computed as the set of all partial matches formed by the partial match that arrived, extended with all partial matches in the semantic window, for which the non-temporal join-criteria c_β hold.

$$\textit{expiration-time}_{\swarrow} : (\mathbf{I_E} \to \Delta_{\mathbf{\Theta}}{}^2) \to \mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{\Theta}} \to \mathbf{\Theta}$$

$$\textit{expiration-time}_{\swarrow}(\mathit{dist}, \ell \mathit{map}) = \mathit{max}(\{-\infty\} \cup \{\ell \mathit{map}(\iota_e) + \mathit{dist}(\iota_e)_{\mathit{min}} \mid \iota_e \in \mathrm{dom}(\ell \mathit{map})\})$$

$$\textit{expire}_{\searrow} : \mathcal{P}(\mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{E}}) \to \mathbf{\Theta} \to \mathcal{P}(\mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{E}})$$

$$\textit{expire}_{\searrow}(\overline{pm}, \theta_{\mathit{expire}}) = \{pm \mid pm \in \overline{pm},\ \forall \iota_e \in \mathit{dom}(pm) : pm(\iota_e).\theta \geq \theta_{\mathit{expire}}\}$$

$$\textit{expire-negated}_{\searrow} : \mathcal{P}(\mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{E}}) \to \mathbf{\Theta} \to (\mathcal{P}(\mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{E}}) \times \mathcal{P}(\mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{E}}))$$

$$\begin{aligned}&\textit{expire-negated}_{\searrow}(\overline{pm}, \theta_{\mathit{expire}}) = \\ &\quad \langle\ \{pm \mid pm \in \overline{pm},\ \forall \iota_e \in \mathit{dom}(pm) : pm(\iota_e).\theta \geq \theta_{\mathit{expire}}\}, \\ &\qquad \{pm \mid pm \in \overline{pm},\ \forall \iota_e \in \mathit{dom}(pm) : pm(\iota_e).\theta < \theta_{\mathit{expire}}\}\rangle\end{aligned}$$

$$\textit{expire}_{\swarrow} : \mathcal{P}(\mathbf{E}) \to \mathbf{\Theta} \to \mathcal{P}(\mathbf{E})$$

$$\textit{expire}_{\swarrow}(\overline{e}, \theta_{\mathit{expire}}) = \{e \mid e \in \overline{e},\ e.\theta \geq \theta_{\mathit{expire}}\}$$

Figure 5.5: The auxiliary functions that are specific to the Featherweight PARTE model — These are used in addition to those of figure 5.4. Lines in green list the types of the functions, while lines in black list their behavior.

- Fifth, the implicit temporal lower bounds on timestamp are computed as the timestamps of the stored partial matches.
- Sixth, the propagated lower bounds are computed as the minimum of the implicit and the explicit lower bounds.
- Finally, for each partial match in $\overline{pm_{\mathrm{new}}}$, a beta token is produced, communicating the propagated temporal lower bounds.

(Join-$\swarrow$) proceeds similarly, but handles a right activation instead of a left activation.

The auxiliary functions *lifetime*, *timestamps*, *semantic-window*$_{\searrow}$, and *semantic-window*$_{\swarrow}$ used in the inference rules above are the same as those defined in figure 5.4 for the rudimentary PARTE model. The functions *expiration-time*$_{\swarrow}$, *expire*$_{\searrow}$, *expire*$_{\swarrow}$, and *expire-negated*$_{\searrow}$ are defined in figure 5.5. Note that the definitions entail that $\forall x, y : \mathit{lifetime}(x, y)_{\mathit{min}} = \textit{expiration-time}(x, y)$. In other words: the expiration time of a partial match is equal to the time at which the lower temporal bound of the opposite side irrevocably falls outside of its lifetime. This is in line with the definition of staleness from definition 17.

Not-Nodes

Remember from page 116 that not-nodes or anti-join-nodes implement *negation* in the PARTE model. We refer to that section for an explanation of their goal. We only repeat the crucial difference between join-nodes and not-nodes: not-nodes propagates tokens when the relational join fails, and discards tokens for which the join succeeds.

In fwPARTE evaluation at a not-node by the local evaluation language $\xrightarrow{l}$ is also somewhat similar to the evaluation at a join-node, though many subtle differences exist.

It proceeds as follows:

(Not-$\searrow$-$\bot$)

$$\frac{\begin{array}{c}\overline{e}' = \mathit{expire}_{\swarrow}(\overline{e}, \mathit{expiration\text{-}time}(\mathit{dist}, \ell \mathit{map}_{t_\beta}))\\ \ell \mathit{map}_{\mathit{implicit}} = \mathit{min}(\mathit{timestamps}(\overline{\mathit{pm}}))\\ \ell \mathit{map}_{\mathit{result}} = \mathit{min}(\ell \mathit{map}_{\mathit{implicit}}, \ell \mathit{map}_{t_\beta})\end{array}}{\begin{array}{l}\quad\langle \mathcal{D}_{\mathit{not}}\langle c_\beta, \ell \mathit{map}_{\searrow}, \ell_{\swarrow}, \overline{\mathit{pm}}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \searrow, \mathcal{T}_\beta\langle \ell \mathit{map}_{t_\beta}, \bot\rangle\rangle\\ \xrightarrow{l} \langle \mathcal{D}_{\mathit{not}}\langle c_\beta, \ell \mathit{map}_{t_\beta}, \ell_{\swarrow}, \overline{\mathit{pm}}, \overline{e}', \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \{\mathcal{T}_\beta\langle \ell \mathit{map}_{\text{result}}, \bot\rangle\}\rangle\end{array}}$$

(Not-$\swarrow$-$\bot$)

$$\frac{\begin{array}{c}\langle \overline{\mathit{pm}'}, \overline{\mathit{pm}_{\text{expired}}}\rangle = \mathit{expire\text{-}negated}_{\searrow}(\overline{\mathit{pm}}, \ell_{t_\beta})\\ \ell \mathit{map}_{\mathit{implicit}} = \mathit{min}(\mathit{timestamps}(\overline{\mathit{pm}'}))\\ \ell \mathit{map}_{\mathit{result}} = \mathit{min}(\ell \mathit{map}_{\mathit{implicit}}, \ell \mathit{map}_{\searrow})\\ \overline{t} = \{\mathcal{T}_\beta\langle \ell \mathit{map}_{\mathit{result}}, \mathit{pm}\rangle \mid \mathit{pm} \in \overline{\mathit{pm}_{\text{expired}}}\} \sqcup \{\mathcal{T}_\beta\langle \ell \mathit{map}_{\mathit{result}}, \bot\rangle\}\end{array}}{\begin{array}{l}\quad\langle \mathcal{D}_{\mathit{not}}\langle c_\beta, \ell \mathit{map}_{\searrow}, \ell_{\swarrow}, \overline{\mathit{pm}}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \swarrow, \mathcal{T}_\beta\langle [\iota_{e_\swarrow} \mapsto \ell_{t_\beta}], \bot\rangle\rangle\\ \xrightarrow{l} \langle \mathcal{D}_{\mathit{not}}\langle c_\beta, \ell \mathit{map}_{\searrow}, \ell_{t_\beta}, \overline{\mathit{pm}'}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \overline{t}\rangle\end{array}}$$

(Not-$\searrow$-Match)

$$\frac{\begin{array}{c}\mathit{has\text{-}room\text{-}for}(\overline{\mathit{pm}}, 1)\\ \overline{e}' = \mathit{expire}_{\swarrow}(\overline{e}, \mathit{expiration\text{-}time}(\mathit{dist}, \ell \mathit{map}_{t_\beta}))\\ \mathit{window}_{\swarrow} = \mathit{semantic\text{-}window}(\overline{e}', \mathit{lifetime}_{\swarrow}(\mathit{dist}, \mathit{timestamps}(\mathit{pm}_{t_\beta})))\\ \exists e \in \mathit{window}_{\swarrow} : c_\beta(\mathit{pm}_{t_\beta} \cdot [\iota_{e_\swarrow} \mapsto e]) \xrightarrow{\beta} \text{true}\}\end{array}}{\begin{array}{l}\quad\langle \mathcal{D}_{\mathit{not}}\langle c_\beta, \ell \mathit{map}_{\searrow}, \ell_{\swarrow}, \overline{\mathit{pm}}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \searrow, \mathcal{T}_\beta\langle \ell \mathit{map}_{t_\beta}, \mathit{pm}_{t_\beta}\rangle\rangle\\ \xrightarrow{l} \langle \mathcal{D}_{\mathit{not}}\langle c_\beta, \ell \mathit{map}_{t_\beta}, \ell_{\swarrow}, \overline{\mathit{pm}}, \overline{e}', \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \mathit{dist}\rangle, \emptyset\rangle\end{array}}$$

(Not-↘-NoMatch)

$$\frac{\begin{array}{c} \textit{has-room-for}(\overline{pm}, 1) \\ \overline{e}' = \textit{expire}_{\swarrow}(\overline{e}, \textit{expiration-time}(\textit{dist}, \ell map_{t_\beta})) \\ \textit{window}_{\swarrow} = \textit{semantic-window}(\overline{e}', \textit{lifetime}_{\swarrow}(\textit{dist}, \textit{timestamps}(pm_{t_\beta}))) \\ \forall e \in \textit{window}_{\swarrow} : c_\beta(pm_{t_\beta} \cdot [\iota_{e_\swarrow} \mapsto e]) \xrightarrow{\beta} \text{false}\} \\ \overline{pm}' = \overline{pm} \cdot pm_{t_\beta} \end{array}}{\begin{array}{l} \langle \mathcal{D}_{not}\langle c_\beta, \ell map_{\searrow}, \ell_{\swarrow}, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \searrow, \mathcal{T}_\beta\langle \ell map_{t_\beta}, pm_{t_\beta}\rangle\rangle \\ \xrightarrow{l} \langle \mathcal{D}_{not}\langle c_\beta, \ell map_{t_\beta}, \ell_{\swarrow}, \overline{pm}', \overline{e}', \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \emptyset\rangle \end{array}}$$

(Not-↙)

$$\frac{\begin{array}{c} \textit{has-room-for}(\overline{e}, 1) \\ \overline{pm}' = \{pm \mid pm \in \overline{pm}, c_\beta(pm \cdot [\iota_{e_\swarrow} \mapsto e_{t_\beta}]) \xrightarrow{\beta} \text{false}\} \\ \langle \overline{pm}'', \overline{pm_{\text{expired}}}\rangle = \textit{expire-negated}_{\searrow}(\overline{pm}', \ell_{t_\beta}) \\ \ell map_{implicit} = \min(\textit{timestamps}(\overline{pm}'')) \\ \ell map_{result} = \min(\ell map_{implicit}, \ell map_{\searrow}) \\ \overline{t} = \{\mathcal{T}_\beta\langle \ell map_{result}, pm\rangle \mid pm \in \overline{pm_{\text{expired}}}\} \sqcup \{\mathcal{T}_\beta\langle \ell map_{result}, \bot\rangle\} \end{array}}{\begin{array}{l} \langle \mathcal{D}_{not}\langle c_\beta, \ell map_{\searrow}, \ell_{\swarrow}, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \swarrow, \mathcal{T}_\beta\langle [\iota_{e_\swarrow} \mapsto \ell_{t_\beta}], [\iota_{e_\swarrow} \mapsto e_{t_\beta}]\rangle\rangle \\ \xrightarrow{l} \langle \mathcal{D}_{not}\langle c_\beta, \ell map_{\searrow}, \ell_{t_\beta}, \overline{pm}', \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, \textit{dist}\rangle, \overline{t}\rangle \end{array}}$$

Evaluation rules (Not-↘-Match) and (Not-↘-NoMatch) jointly uphold the invariant that left partial matches are stored in the left partial match history if and only if there exists no right partial match which satisfied the anti-join-condition c_β when joined with the new right partial match. Additionally, evaluation rule (Not-↙) filters the left partial matches (i.e., $\overline{pm}$) by removing those that satisfy the anti-join-condition c_β when joined with the new right partial match (yielding $\overline{pm}'$). This marks the first significant difference with evaluation at join-nodes.

Upon activation from the right, expiration from the left partial match history has to happen. Unlike in the case of join-nodes, evaluation at not-nodes uses an *expire-negated*$_\searrow$ function. This function returns not just the non-stale partial matches, but instead splits the partial matches into a stale and a non-stale subset: the second and first element in the produced tuple. Due to the invariants described in the previous paragraph, the stale partial matches that were still in the left partial match history, are stale matches for which no matching right partial match exists. By definition of staleness, no matching right partial matches can still occur later. Hence, these stale left partial matches are guaranteed to not match with anything, or in other words: negation-as-failure succeeded for these stale left partial matches. These matches are hence to be added to the outbox, to be propagated to the successors. Note that this happens both in (Not-↙) and in (Not-↙-$\bot$): the results are sourced only from the left partial match history, so even a $\bot$ token on the

Side	Matches in opposite?	Do to $\overline{pm}$	Do to $\overline{e}$	Output
$\searrow$	matches in $\overline{e}$	-	expire old	-
$\searrow$	no matches in $\overline{e}$	append new pm	expire old	-
$\swarrow$	matches in $\overline{pm}$	remove matching, expire old	append new e	all expired pms
$\swarrow$	no matches in $\overline{pm}$	expire old	append new e	all expired pms

Figure 5.6: A summary of the behavior of not-nodes in fwPARTE — For each combination of activation side and presence of matches in the opposite partial match history, changes to the partial match histories and the tokens to produce are listed.

right suffices. Conversely, note how the rules for left activation cannot produce any tokens containing actual partial matches: receipt of positive partial matches cannot prove the validity of other positive partial matches. This marks the second significant difference with evaluation at join-nodes.

The behavior of evaluation of the rules of $\xrightarrow{l}$ at not-nodes is summarized in figure 5.6.

Production Nodes

As explained in section 5.2.6 on page 118, production nodes emit a new compound event when a rule's entire pattern is successfully matched.

Since production nodes produce events, they act as event sources. As indicated in the definition of tokens (section 5.4.3), the Featherweight PARTE model requires knowledge of event sources. Event sources must be identifiable by a unique identifier, thus production nodes are allocated a unique event source identifier $\iota_s \in \mathbf{I_S}$. Similarly, to accommodate the fwPARTE model, a lower bound ℓ must be computed, based on the lower bounds of the tokens received by the production node. The expression that extracts a temporal lower bound from the partial matches is typeset as $g^{\ell}_{\square}$, in analog to $g^{e}_{\square}$. The temporal lower bound generator expression must ensure that it generates non-decreasing values over consecutive calls. It can assume that each of its arguments will receive a value at least as large as in the previous call, since that invariant is upheld by the rest of the PARTE graph.

Of course, production nodes must have a rule for dealing with "no change" indicators as well. To provide for all the changes listed above, the evaluation at a production node by the local evaluation language $\xrightarrow{l}$ proceeds as follows:

(Prod-$\bot$)

$$\frac{g^{\ell}_{\square}(\ell map_{t_\beta}) \longrightarrow \ell}{\langle \mathcal{D}_{production}\langle \iota_s, g^{e}_{\square}, g^{\ell}_{\square}\rangle, \downarrow, \mathcal{T}_\beta\langle \ell map_{t_\beta}, \bot\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{production}\langle \iota_s, g^{e}_{\square}, g^{\ell}_{\square}\rangle, \{\mathcal{T}_\alpha\langle \iota_s, \ell, \bot\rangle\}\rangle}$$

(Prod)

$$\frac{g^{e}_{\square}(pm) \longrightarrow e' \qquad g^{\ell}_{\square}(\ell map_{t_\beta}) \longrightarrow \ell}{\langle \mathcal{D}_{production}\langle \iota_s, g^{e}_{\square}, g^{\ell}_{\square}\rangle, \downarrow, \mathcal{T}_\beta\langle \ell map_{t_\beta}, pm\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{production}\langle \iota_s, g^{e}_{\square}, g^{\ell}_{\square}\rangle, \{\mathcal{T}_\alpha\langle \iota_s, \ell, e'\rangle\}\rangle}$$

Terminal Nodes

As explained in section 5.2.6 on page 118, terminal nodes represent the point at which an entire LRP program is successfully matched. In the formal model, tokens are consumed at terminal nodes by merely discarding them. Terminal nodes hence behave as follows:

(Term)

$$\frac{}{\langle \mathcal{D}_{terminal}\langle\rangle, \downarrow, \mathcal{T}_\beta\langle \_, \_\rangle\rangle \xrightarrow{l} \langle \mathcal{D}_{terminal}\langle\rangle, \emptyset\rangle}$$

5.4.7 The Global Evaluation Language

The previous sections defined the semantic constructs that make up the events, tokens, and graph nodes, and defined how evaluation at a certain graph node takes place by means of the local evaluation language $\xrightarrow{l}$. This section now defines how tokens percolate through the PARTE graph. The way tokens percolate through the PARTE graph is specified the two inference rules of the global evaluation language $\xrightarrow{g}$: (Proc) and (Prop). The graph nodes in previous section were structurally identical to those in the rudimentary PARTE model, save for modifications to the node-local data and the tokens. Since the global evaluation language does not touch the internals of these structures, the rules remain largely unchanged from those in section 5.2.5. One difference exists in (Prop): tokens are only propagated if all recipients have room for a token of that activation side, to prevent exhausting space in the inboxes. This effectively introduces *back-pressure* (see section 2.3.2) into the system.

(Proc)

$$\frac{\langle data, side, t\rangle \xrightarrow{l} \langle data', \overline{t'}\rangle \qquad \textit{has-room-for}(\bar{t}, \left|\overline{t'}\right|)}{\{\mathcal{N}\langle \iota_n, \bar{\mathfrak{s}}, \langle side, t\rangle \cdot \overline{m}, \bar{t}, data\rangle\} \sqcup \mathbf{N}_{rest} \xrightarrow{g} \mathbf{N}_{rest} \sqcup \{\mathcal{N}\langle \iota_n, \bar{\mathfrak{s}}, \overline{m}, \bar{t} \diamond \overline{t'}, data'\rangle\}}$$

(Prop)

$$\frac{\begin{array}{c}\mathbf{N}_b = \mathbf{N} \cap \{\mathcal{N}\langle \iota_b, \_, \_, \_, \_\rangle \mid \langle \iota_b, \_\rangle \in \overline{\mathfrak{s}_a}\} \\ \forall \mathcal{N}\langle \_, \_, \overline{m_b}, \_, \_\rangle \in \mathbf{N}_b : \textit{has-room-for}(\overline{m_b}, 1), \nexists \langle side, \_\rangle \in \overline{m_b} \\ \mathbf{N}'_b = \{\mathcal{N}\langle \iota_b, \overline{\mathfrak{s}_b}, \overline{m_b} \cdot \langle side, t\rangle, \overline{t_b}, data_b\rangle \mid \mathcal{N}\langle \iota_b, \overline{\mathfrak{s}_b}, \overline{m_b}, \overline{t_b}, data_b\rangle \in \mathbf{N}_b, \langle \iota_b, side\rangle \in \overline{\mathfrak{s}_a}\}\end{array}}{\{\mathcal{N}\langle \iota_a, \overline{\mathfrak{s}_a}, \overline{m_a}, t \cdot \overline{t_a}, data_a\rangle\} \sqcup \mathbf{N}_b \sqcup \mathbf{N}_{rest} \xrightarrow{g} \{\mathcal{N}\langle \iota_a, \overline{\mathfrak{s}_a}, \overline{m_a}, \overline{t_a}, data_a\rangle\} \sqcup \mathbf{N}'_b \sqcup \mathbf{N}_{rest}}$$

5.5 Compiling PARTElang Programs into Featherweight PARTE Graphs

As mentioned repeatedly in this chapter, fwPARTE is inspired heavily by the Rete algorithm by Forgy [59]. The Rete algorithm prescribes two phases: the first phase analyses a set of declarative rules and compiles a DAG representing the constraints. The second phase incrementally matches data against the rule set's constraints by percolating tokens through the DAG. Section 5.4 described PARTE's version of the second, dynamic, phase: the fwPARTE operational semantics. However, fwPARTE does not merely ascribe different semantics to the elements of a Rete graph. Instead, fwPARTE defines a slightly different set of elements. As such, Rete's first, static phase cannot be reused as-is.

This section defines PARTE's version of the static rule set analysis and graph construction phase. We give a declarative description specifying which constraints a constructed graph should conform to, instead of an imperative description specifying how the graph should be created, as PARTE's approach to constructing a DAG is neither novel nor important.

The remainder of this section separates this first phase — the construction of the graph — into a series of logical phases. Figure 5.7 shows the skeleton of a rule written in PARTElang, and depicts the state of the fwPARTE network after each of the four phases is completed.

5.5.1 Registering Event Templates

As a first step towards compiling the PARTE graph, the types of events referred to in the rule set must be gathered. For each template mentioned in some rule of the rule set, an entry node must be constructed. The entry nodes must know of all event sources that may send events to them. For completeness, part 1 of figure 5.7 depicts the graph at that point in the graph construction, though the graph is still just a set of disconnected entry nodes at that point.

Consider this event source configuration:

Source #1 emits events adhering to template A;

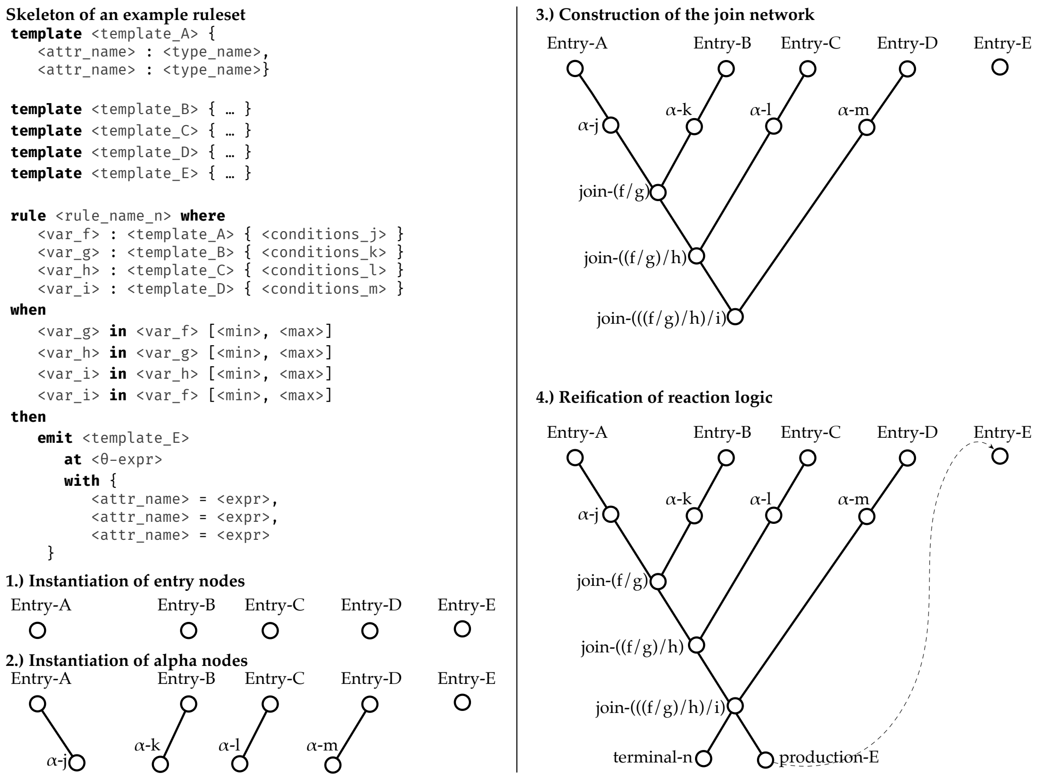
Source #2 emits events adhering to template B, C and D; and

Source #3 emits events adhering to template B and E.

The node labeled "Entry-A" in figure 5.7 must then be initialized to the following fwPARTE semantic constructs:

$$\mathcal{N}\langle \iota_{\text{Entry-A}}, \emptyset, \emptyset, \emptyset, \mathcal{D}_{entry}\langle [\#1 \mapsto -\infty], \iota_{\text{template-A}} \rangle \rangle,$$

where $\iota_{\text{Entry-A}}$ is a fresh node identifier for the node labeled "Entry-A". The following three elements hold the successors, inbox, and outbox of the entry node. These are initially


Skeleton of an example ruleset

```
template <template_A> {
   <attr_name> : <type_name>,
   <attr_name> : <type_name>}

template <template_B> { … }
template <template_C> { … }
template <template_D> { … }
template <template_E> { … }

rule <rule_name_n> where
   <var_f> : <template_A> { <conditions_j> }
   <var_g> : <template_B> { <conditions_k> }
   <var_h> : <template_C> { <conditions_l> }
   <var_i> : <template_D> { <conditions_m> }
when
   <var_g> in <var_f> [<min>, <max>]
   <var_h> in <var_g> [<min>, <max>]
   <var_i> in <var_h> [<min>, <max>]
   <var_i> in <var_f> [<min>, <max>]
then
   emit <template_E>
      at <θ-expr>
      with {
         <attr_name> = <expr>,
         <attr_name> = <expr>,
         <attr_name> = <expr>
   }
```

1.) Instantiation of entry nodes
Entry-A
Entry-B
Entry-C
Entry-D
Entry-E
2.) Instantiation of alpha nodes
α-j
α-k
α-l
α-m
3.) Construction of the join network
join-(f/g)
join-((f/g)/h)
join-(((f/g)/h)/i)
4.) Reification of reaction logic
terminal-n
production-E

Figure 5.7: Sketch of a rule and the phases of PARTE graph construction for that rule — Circles represent fwPARTE graph nodes. Lines indicate predecessor-successor relations. For solid lines, nodes higher up are the predecessors. Dashed lines depict implicit edges over which abstracted, compound events travel. All labels apply to nodes.

empty, typeset as ∅. The entry node's node-local data holds a mapping from event source identifier to the lower bound on timestamps still to receive from that event source. For "Entry-A", the sole event source is #1, which can still send events from arbitrarily early points in time, typeset as $-\infty$. Finally, the entry node lists the identifier of the template of events it may receive, which for "Entry-A" is template A.

The nodes labeled "Entry-B", "Entry-C", "Entry-D", and "Entry-E" are constructed similarly. For instance, the entry node labeled "Entry-B", which accepts tokens from two event sources, is initialized to

$$\mathcal{N}\langle \iota_{\text{Entry-B}}, \emptyset, \emptyset, \emptyset, \mathcal{D}_{entry}\langle [\#2 \mapsto -\infty, \#3 \mapsto -\infty], \iota_{\text{template-B}}\rangle\rangle.$$

5.5.2 Implementing Individual Event Patterns

Once all entry nodes are created, for each unique event pattern,[17] an alpha node is generated. These alpha nodes are instantiated with alpha conditions c_α which implement the conditions of the rule set. For instance, the alpha node labeled "α-j" implements condition `<conditions_j>`. The alpha nodes are registered as successors of their entry nodes, such that e.g., the alpha node labeled "α-j" is a successor of the entry node for events adhering to template A, since the event pattern identified by variable `<var_f>` requires events of template A which satisfy `<conditions_j>`.

The fwPARTE semantic constructs for the entry nodes must thus be modified to register the successor relations. For instance, the entry node for events adhering to template A must be changed to the following:

$$\mathcal{N}\langle \iota_{\text{Entry-A}}, [\iota_{\textit{ff}\text{-j}} \mapsto \downarrow], \emptyset, \emptyset, \mathcal{D}_{entry}\langle [\#1 \mapsto -\infty], \iota_{\text{template-A}}\rangle\rangle,$$

where $\iota_{\text{Entry-A}}$ is a fresh node identifier for the node labeled "α-j". In addition, the alpha nodes themselves must be constructed. The node labeled "α-j" is implemented by this fwPARTE construct:

$$\mathcal{N}\langle \iota_{\textit{ff}\text{-j}}, \emptyset, \emptyset, \emptyset, \mathcal{D}_{alpha}\langle c_{\alpha\,\text{<condition\_j>}}, \iota_{\text{<var\_f>}}\rangle\rangle.$$

The node labeled "α-k" is implemented by this fwPARTE construct:

$$\mathcal{N}\langle \iota_{\textit{ff}\text{-k}}, \emptyset, \emptyset, \emptyset, \mathcal{D}_{alpha}\langle c_{\alpha\,\text{<condition\_k>}}, \iota_{\text{<var\_g>}}\rangle\rangle.$$

In this phase too — as in the next two — *node reuse* can be leveraged by merging identical nodes into one.

5.5.3 Combining Event Patterns: Constructing the Join Network

Once an entry node and alpha node exist for each pattern in a rule set, the implicit conjunction between the patterns of a rule must be made. This conjunction is achieved by means of join- and anti-join-nodes. The first pattern is used as the left predecessor of the first (anti-)join-node. Then, each subsequent pattern is used as the right predecessor of a join-node, using the accumulated join graph as the left predecessor. Each (anti-)join-node encodes the constructs of the conditions that depend on the newly joined pattern, both temporal and non-temporal. For instance, the node labeled "join-(f/g)" contains any inter-event constraints specified in `condition_k`, as well as the temporal constraints specified between the events bound to `var_g` and `var_f` in the `when` clause.

[17]When an identical event pattern exists in multiple locations in a rule set — potentially even in different rules — the same alpha node can represent those patterns, preventing duplicate work. The successors of this alpha node must then consist of the union of all successors of the different alpha nodes that would have been emitted otherwise. This notion of *node reuse* is prevalent in Rete and derived systems [91].

Formally, for each join or anti-join-node, a new fwPARTE node must be constructed, and its left and right predecessors must be modified to register that relation. For instance, the node labeled "α-j" must be changed to the following, to register it as the left predecessor of the node labeled "join-(f/g)":

$$\mathcal{N}\langle \iota_{ff\text{-j}}, [\iota_{\text{join-(f/g)}} \mapsto \searrow], \emptyset, \emptyset, \mathcal{D}_{alpha}\langle c_{\alpha\,\text{<condition\_j>}}, \iota_{\text{<var\_f>}}\rangle\rangle,$$

where $\iota_{\text{join-(f/g)}}$ is a fresh node identifier for the node labeled "join-(f/g)". That join-node's right predecessor — the node labeled "α-k" — must be changed to the following form to register it as the right predecessor of the join-node:

$$\mathcal{N}\langle \iota_{ff\text{-k}}, [\iota_{\text{join-(f/g)}} \mapsto \swarrow], \emptyset, \emptyset, \mathcal{D}_{alpha}\langle c_{\alpha\,\text{<condition\_k>}}, \iota_{\text{<var\_g>}}\rangle\rangle.$$

The join-node itself is implemented thusly:

$$\mathcal{N}\langle \iota_{\text{join-(f/g)}}, \emptyset, \emptyset, \emptyset, \mathcal{D}_{join}\langle c_{\beta}, [\iota_{\text{<var\_f>}} \mapsto -\infty], -\infty, \emptyset, \emptyset, \{\iota_{\text{<var\_f>}}\}, \iota_{\text{<var\_g>}}, dist\rangle\rangle,$$

where c_β implements the inter-event constraints specified in `<conditions_j>` and `<conditions_k>`, and where *dist* is constructed from the relations specified in the `where` clause of `<rule_name>`, using the technique pioneered by Teodosiu and Pollak [133]. Summarizing their work, temporal distances between every two event patterns in a rule can be determined by constructing a graph of event patterns, whose edges are labeled by a pair of minimum and maximum distance. Initially, all event patterns are disconnected, with the exception of self-references labeled with the weights $[0, 0]$, i.e., every pattern can take place at least zero timesteps before itself, and at most zero timesteps after itself. For every explicit dependency in the `when` clause, a weighted edge is added, specifying the minimum and maximum temporal distance specified in the rule. Subsequently the transitive and reflexive closure of the distances can be computed as follows: repeatedly update every edge in the graph (edge$_{i,j}$) to the intersection of the current distances (edge$_{i,j}$) and the addition of edge$_{i,k}$ and edge$_{k,j}$, where intersection is defined as

$$\odot : \mathbf{\Theta}^2 \to \mathbf{\Theta}^2 : [a, b] \odot [c, d] = [\max\{a, c\}, \min\{b, d\}]$$

and addition is defined as

$$\oplus : \mathbf{\Theta}^2 \to \mathbf{\Theta}^2 : [a, b] \oplus [c, d] = [a + c, b + d]$$

The branches extending from the nodes labeled as "Entry-C" and "Entry-D" are linked into the graph similarly, using new join-nodes. At each step, the previous join-node's successor set is modified to register it as the left predecessor of the next join-node.

5.5.4 Representing the Reaction Logic

Finally, the PARTE graph construction implements the reaction logic specified by the rules, i.e., the `then` clause. Neither PARTElang nor the underlying models support any other action than emitting new compound events.

Still, fwPARTE represents the place where a rule is successfully matched as a node in the graph. The node which represents a successful match to the rule named "`rule_name_n`" is the terminal node labeled "terminal n", which is initialized to the following form:

$$\mathcal{N}\langle \iota_{\text{terminal-n}}, \emptyset, \emptyset, \emptyset, \mathcal{D}_{terminal}\langle\rangle\rangle.$$

The fwPARTE model prescribes no other semantics to terminal nodes. The node will never be updated with successors, with tokens in its outbox, or with changes to the null-ary node-local data.

The event emission that is described by the reaction logic of the rule, is implemented by a production node: the node labeled "production-E". The semantic construct of the production node itself is straightforward:

$$\mathcal{N}\langle \iota_{\text{production-E}}, [\iota_{\text{Entry-E}} \mapsto \downarrow], \emptyset, \emptyset, \mathcal{D}_{production}\langle \#4, g^{e}_{\square}, g^{\ell}_{\square}\rangle\rangle,$$

where #4 is a fresh event source identifier. The $\theta - expr$ (from the `at` clause) and the assignments to the attributes of the `emit`ted event of template E (from the `with` clause) get translated to an event generation expression $g^{e}_{\square}$. For each pm which is placed in a token in the inbox of node "production-E", an event of template E can be obtained by evaluating $g^{e}_{\square}(pm)$. Those events are then sent to the successors of the production node. At this point in the graph construction, the set of successors is still the empty set $\emptyset$.

Also based on $\theta - expr$, the temporal lower bound generator expression $g^{\ell}_{\square}$ is created. Here, for each ℓmap that is placed in a token in the inbox of the node, $g^{\ell}_{\square}(\ell map)$ produces a new lower bound, which can be propagated to the successors.

In contrast to other nodes, production nodes are created in a later PARTE graph construction phase than their successors. To restore the internal consistency in the graph, the affected entry node must still be amended to incorporate its new predecessor. After the fourth PARTE graph construction phase completes, the semantic construct representing the node labeled "Entry-E" will look as follows:

$$\mathcal{N}\langle \iota_{\text{Entry-E}}, \emptyset, \emptyset, \emptyset, \mathcal{D}_{entry}\langle [\#3 \mapsto -\infty, \#4 \mapsto -\infty], \iota_{\text{template-E}}\rangle\rangle.$$

5.6 Limitations of Featherweight PARTE and Future Work

Featherweight PARTE is — by design — a reduced subset of what is necessary for realizing a practical strongly reactive complex event detection system. This is evident from for instance the restrictions on the data types used, e.g., fwPARTE does not support text strings or custom data structures as attributes to events. As mentioned in the sections discussing terminal nodes, the only reaction logic explicitly supported by fwPARTE is the emission of compound events, or the detection of a rule matching. Practical implementations will want to register callbacks that are invoked in response to the matching of a rule.

The Featherweight PARTE model — in its current state — has a number of limitations beyond those superficial ones.

Transfers/5min.	PARTE size	Cartesian size	Triangle query size
1	244G	246140G	1G
2	1953G	1969120G	2G
3	6593G	6645780G	4G
4	15629G	15752961G	7G
5	30525G	30767501G	10G
6	52747G	53166243G	14G
7	83760G	84426025G	17G
8	125029G	126023688G	21G
9	178019G	179436071G	25G

Figure 5.8: A demonstration of how fwPARTE's current formula yields high minimum viable partial match history sizes — The magnitude of these sizes is inherent to the Cartesian product underlying arbitrary joining

5.6.1 Limitations of the Minimum Viable Size Calculation

First, the formula which defines the minimum viable partial match history sizes in Featherweight PARTE yields a correct viable size: partial match histories at at least that size, are sufficiently large. Additionally, the minimum viable partial match history sizes are a tight bound around the theoretical minimum required sizes: fwPARTE prescribes $r \times W + 1$ (maximum event arrival rate times window duration, plus one; see definition 25 on page 127), while at least $r \times W$ is required for an arbitrary pattern. However, tighter bounds have recently been proven on specific patterns, based on fractional edge cover [111]. Whereas PARTE would currently assume that a "triangle query" [111] on a dataset of size n has a cost in n^3, Gottlob et al. [63] showed that such queries can be fully materialized in $n^{3/2}$. To make that more concrete: a triangle query joining a dataset of 1000 elements would appear to yield 10^{12} results, whereas only 10^6 can exist.

Porting these findings to PARTE should be feasible, but is not trivial. The Featherweight PARTE model joins each pattern individually with one other pattern, and adds a notion of maximum event rate to the equation. It is at this time unclear how that can best be reconciled with fractional edge cover.

This limitation makes it impractical to apply PARTE to some large applications, including the full-blown versions of our driver scenarios. The driver scenarios are kept simplistic, and hence match a very broad set of events. With PARTE's current formula, the largest of the minimum viable sizes for driver scenario C grows as shown in the first two columns of figure 5.8. The third column compares this to the size of a naive Cartesian product, i.e., the non-stale money-transfer database size (rate times duration, i.e., the value in the first column times the number of minutes in a two-week period) raised to the power 3 (one for each occurrence of MoneyTransferred in the pattern), times the non-stale claim database size (again the number of minutes in a two-week period). In the fourth column, we depict what the largest minimum viable partial match history size would be if the three MoneyTransferred event patterns would form a triangle query, and if PARTE were able to detect this and adapt its minimum viable size formula to reflect this.

Transfers/5min.	PARTE size	PARTE size with aggregation
1	122G	3k
2	976G	7k
3	3296G	11k
4	7814G	15k
5	15262G	19k
6	26373G	23k
7	41880G	27k
8	62514G	31k
9	89009G	35k

Figure 5.9: A demonstration of how fwPARTE's lack of aggregation constructs severely harms performance when aggregation semantics are required

5.6.2 Limitations Inherited from PARTElang and Event Algebra EA

Evidently, the limitations of the language discussed in section 4.5 apply to the model as well: fwPARTE does not offer abstractions for dealing with aggregation and fwPARTE's support for temporal reasoning is limited to defining bounded relative time windows.

These limitations are not superficial. Consider the difference in the largest minimum viable partial match size when expressing driver scenario B in fwPARTE (depicted in the first two columns of figure 5.9) with the largest minimum viable partial match history size when the same three-way join could be expressed as an aggregation, i.e., as "three times an outgoing `MoneyTransferred` event, whose `amounts` `SUM` to approximately the `amount` of the incoming" (depicted in the third column). Instead of making a 4-way Cartesian product between `MoneyTransferred` events, a 2-way Cartesian product is made between `MoneyTransferred` events and ordered sets of up to three `MoneyTransferred` events. Supporting aggregation is hence an important avenue for future research.

5.6.3 Limited Interoperability with Order-Dependent Code

Second, while fwPARTE is an operational semantics for a Rete-derived forward-chaining interpreter for rule-based languages, it lacks the notion of *rule salience* that is prevalent in Rete-derived rule-based systems such as CLIPS [142].

Rule salience is a way of prioritizing the activation of some rules in favor of others. Featherweight PARTE, in contrast, does not even define an *agenda* for rule activations. Instead, fwPARTE fires rules as soon as a token reaches the terminal node or production node by asynchronous message passing. This is well in line with the behavior we ascribed to Logic Reactive Programming languages: an LRP program needs only guarantee that it emits the correct complex events; the order in which different declarative rules emit those events is not specified. However, this does mean that fwPARTE is not compatible with rules and logical facts from a non-reactive logic program.

Future work that enables PARTElang to reason about logical facts outside of the managed event storage will need to explore how fwPARTE's agendaless approach can be combined with the needs of non-event data. Of particular interest will be the means by which such a system can be kept strongly reactive and decentralized, since the ideas of salience and an ordered agenda are by design centralized and sequentializing.

5.6.4 Lack of Failure Handling

Third, fwPARTE lacks support for distributed failure handling and fault tolerance. As a model for distributed computing, PARTE would benefit from the means to deal with partial failure of the computer cluster on which it is deployed. As discussed in chapter 2, the many ways to provide this are well-researched. Many paradigms and frameworks offer out of the box support for checkpointing, replication, replayable logs, etc. A relatively under-researched aspect is how to reconcile these with strong reactivity.

For instance, the cost of periodic checkpointing can be incorporated into the cost model, and restoring from a checkpoint once takes constant time and space, but restoring an arbitrary number of times clearly cannot take only a fixed, finite amount of time. Replaying logs suffers from the same problem: the cost of logging can be incorporated in the cost model, and replaying once takes only time proportionate to the number of partial matches retained, but replaying repeatedly takes time proportionate in the number of replays. Other options, such as replication, fail to uphold strong reactivity out of the box, too: replication cannot guarantee that a partial failure will not impact all replicas of some data, and hence needs to be backed up by techniques such as replayable logs or checkpointing.

Resolving the lack of failure handling in PARTElang is not just an engineering effort: whichever failure handling technique we pursue, concepts from this technique will need to be exposed to the event algebra. Much in the way how the meta-concern of the event arrival rate had to be exposed to fwPARTE, a fault tolerant PARTE model will likely have to specify the meta-concern of a maximum *failure occurrence rate*.

5.6.5 Optimizations Lacking from the Formal Model

Fourth, there are a number of optimization opportunities which are not adopted by the fwPARTE model, but which an implementation would likely apply. These optimizations are in fact implemented in our prototypical implementation described in chapter 6, but — since they are not described in the model — we still consider them to be missing from the model. A brief overview of the most important optimization opportunities is this:

Indexes in partial match histories The bulk of the operations executed on partial match histories requires selecting partial matches based on their timestamp. The model abstracts over how this is done, but prescribes the use of an ordered vector. Clearly, replacing these with a data structure which allows for $O(log(n))$ access and eviction, e.g. a B-Tree, can significantly improve the performance of the system. In similar vein, determining the current lower bounds on timestamps stored in a partial match history can be sped up. For instance, keeping a (counting) sorted multiset of event times that appear for each event pattern, i.e., a $\mathbf{M}_{\mathbf{I}_{\mathbf{E}} \rightharpoonup \Theta \to \mathbb{N}}$.

We explored the option of adding multiple layers of indexes, e.g. to index based on attributes used in the join-conditions. We also investigated the effect of introducing the ideas described in [76], in collaboration with one of the authors of that work. We were unable to reliably achieve improvements, and in many cases the performance dropped noticeably. Our initial investigation suggests that the bin-size after indexing by timestamp was too small for additional indexing to help. Phrased differently, a simple index on timestamp captured such a large fraction of the join-criteria that any concession on that front caused more harm than could be recovered elsewhere.

In general, indexing schemes have a non-negligible overhead. The storage requirement for conjunctive queries itself is roughly proportionate in the size of the Cartesian product of all inputs — though tighter bounds exist as discussed in section 5.6.1. Materializing the indices themselves adds storage costs. Outputting all matches still takes time at least proportionate to the number of matches. These challenges will need to be resolved by future work. Fortunately, such modifications are largely tangential to the requirement of strong reactivity, as the costs are proportionate to sizes that we prove constant for fwPARTE in chapter 8.

Above-minimum storage sizes The model specifies that inboxes, outboxes, and partial match histories should be at their minimum viable sizes. In practice, any constant size greater than or equal to the minimum viable size can be used. Using larger buffers significantly increases the worst-case costs, but may in practice also significantly decrease average-case cost, by changing the message-passing from essentially nearly rendezvous-semantics to a more flexible message-passing style.

Batching of multiple matches The formal operational semantics allows multiple tokens to be created in a single step of $\xrightarrow{l}$, but only propagates those tokens one-by-one in the (Prop)rule of $\xrightarrow{g}$. Since all these tokens have to arrive at the same destination, they could be sent jointly, if the successors have enough room in their inboxes. If the optimization described above is implemented, it becomes possible to batch up multiple tokens, decreasing the amount of events that need sending. Concretely, if a node has verified that the inboxes of all its successors do not contain a token from a certain activation side, it could send all tokens produced in a single step of $\xrightarrow{l}$ for that activation side, since the number of tokens thus produced is finite and statically known. Adapting the model and the proofs would be nontrivial, and is therefore left as future work.

Coarsening the granularity in the fwPARTE graph Many of the nodes in a Featherweight PARTE graph have a trivial or near-trivial task, e.g., comparing a single attribute to a constant, or joining two event patterns where no join-condition applies. For practical implementation, spawning separate concurrent units of computation for each of these nodes is inefficient, even when they are implemented as, e.g., lightweight actors. Instead, specialized actors could be created which perform the job of multiple conceptual fwPARTE nodes. Future work could look into how this can be enacted. Of special interest is the means by which the correct granularity of parallel decomposition can be achieved, and how a running system can be made to reconfigure itself in the face of changing workloads, while remaining strongly reactive.

Limiting $\perp$-tokens There is no need to pass $\perp$-tokens on paths through the graph that do not reach a negation. As we show in section 8.4, "no change" tokens must be sent on paths that do reach not-nodes, but other paths can never become blocked awaiting the absence of events. A simple extension of fwPARTE statically determines which nodes are on a path to a not-node, and only propagate $\perp$-tokens to those nodes.

5.7 Related Work

5.7.1 Formal Models of Production Rule Systems

Our work is not the first to introduce a formal model for production rule systems such as the Rete algorithm. Of particular interest are the work by Snyder and Schmolze [129] and the work by Cirstea et al. [34]. Snyder and Schmolze formalized their production rule system as rewrite semantics for a working memory of facts, implementing the OPS5 [58] production rules. Schmolze and Snyder [125] showed how the formal model by Snyder and Schmolze [129] can be used as the basis for the detection of redundancy among production rules.

A crucial distinction between our work and those earlier formal models of production rule system algorithms, is that the earlier models represent the algorithm as a tree-traversal, with (reads from and) writes to some abstraction of a working memory. Such formalizations implicitly assume a shared memory space, where operations at any Rete node can write to the state of any other Rete node or fragment of working memory. Consequently, these models do not lend themselves to easy parallelization — let alone distribution — as the models do not prescribe how multiple steps can be in progress concurrently without interfering with each other.

5.7.2 Time and Event-Management in Rete-derived Systems

The incremental nature of the Rete algorithm makes it an obvious fit for systems reasoning over online data streams. The presence of a temporal component in the data can be leveraged better when the processing engine understands time. Multiple publications describe extensions to the Rete algorithm which add support for temporal reasoning.

Maloof and Kochut [96] modify Rete to reason temporally. In their extension, events have a duration, and event patterns have built-in support for relative temporal order (`before`, `during` or `after`) between events. Berstel [22] similarly extends the Rete algorithm with operators to for event management. Their extension also offers built-in support for relative temporal order between events, though only `before` or `after`: like PARTE, Berstel [22] uses timestamps, not time durations. Also like PARTE, they use this temporal knowledge to discard unneeded events. Their approach however depends on the ability of the system to atomically inspect the global state of the system, and on the notion of recursively asking predecessor Rete memories to remove event data. Such an approach does not work in a distributed context, and modifying this approach to work without introducing race conditions between the different distributed components would be nontrivial. Additionally, the algorithm devised by Berstel [22] was not envisioned for use in settings where strong reactivity is necessary. Their algorithm hence makes no attempts to guarantee constant time and space bounds.

Walzer et al. [138] describe an extension to Rete exposing all thirteen temporal relations from Allen's interval algebra [6]. Walzer et al. [138] too use this temporal knowledge to discard unneeded events. Like Berstel [22]'s solution, the solution by Walzer et al. [138] depend on global timers which remove intermediate results from globally accessible partial match storage. Their solution does not work without considerable changes in a distributed context. Since their algorithm was not envisioned for use in settings where strong reactivity is necessary, their algorithm makes no attempts to guarantee constant time and space bounds.

5.7.3 Distributed Tracking of Temporal Lower Bounds

PARTE keeps track of temporal lower bounds on events, to enable it to automatically manage the event storage. Some other systems which process event-data in a distributed context, and keep some form of managed event storage, exist. These, too, need to keep track of the progress of time, and relate that to the events still stored. We distinguish between two cases: first, systems which ingest events from possibly multiple, distributed event sources, but whose processing is not itself distributed, and second, systems whose processing can be distributed across multiple computers.

The first category contains systems like EVA [72]. Like PARTE, EVA handles semantic simultaneity of events from multiple event sources by assuming a global system of reference exists for specifying relative order. While a distributed system cannot be assumed to have a synchronized global clock, one can opt to ascribe the semantics that any event source has the valid clock for events whose occurrence it registers. EVA, like PARTE, includes a component responsible for merging the temporal lower bounds of the different event sources. Using this conservative lower bound on actual clocks, the correct results can be computed despite the lack of global clock shared by the event sources.

The second category contains systems like MillWheel [5] and Naiad [108]. Neither are strictly CEP systems, though they are designed to work with data tagged with timestamps, i.e., events. MillWheel explicitly keeps track of *low watermarks*. Low watermarks are somewhat analogous to the lower bounds (ℓ) in fwPARTE, in that both represent a temporal threshold below which events are no longer relevant for the computation. In Naiad's *timely dataflow*, the logical timestamps are an event closer analogue to PARTE's temporal lower bound. Naiad's logical timestamps however include both a purely temporal aspect, and a *loop iteration*.

5.8 Conclusion

We introduced the Featherweight PARTE model, an operational semantics for PARTElang.

We showed how Featherweight PARTE incrementally matches events to complex patterns using a matching algorithm derived from the Rete algorithm. Featherweight PARTE does not depend on a shared address space for its event storage, and is able to automatically manage that distributed event storage: fwPARTE can determine which partial matches are *stale*. Stale partial matches are guaranteed to *expire* before the node they are stored at reaches a *blocked node state*. A PARTElang implementation implementing the Featherweight PARTE model can guarantee correct operation within a fixed, finite memory budget by pre-allocating its buffers at their statically determinable *minimum viable size*.

Featherweight PARTE thus defines a strongly reactive way for correlating complex events specified in PARTElang ($\mathbf{R_{LRP}5}$).

6 Implementation

Up until this point in the dissertation we focused on discussing the theoretical aspects of Logic Reactive Programming. In this chapter we turn LRP into practice. The goal of this chapter is to give a more detailed description of lower-level design decisions which may impact the results in section 7.4.2, for the sake of reproducibility of those results.

The PARTE research artifact went through a number of iterations. The versions described in this chapter differ from the one described in Marr et al. [97] in a number of ways. The most important difference is that the final versions offer the features required to guarantee *strong reactivity* (see definition 1 on page 65): using the Featherweight PARTE model, the size of intermediate storage like inboxes, outboxes, and partial match histories needs not be arbitrarily large, which places an upper bound on the required processing time. This in turn allows it to guarantee a constant-time update cost per new event.

We describe two prototypical implementations of the Featherweight PARTE model. Since both prototypes implement the same model, their high-level workings are covered by the information from chapter 5. The distinction between both versions is as follows:

PARTE$^{\textbf{Rust}}$ is a multi-threaded implementation of the fwPARTE model, which simulates a distributed memory space by modeling the different PARTE nodes as independent units of computation which communicate only by means of message passing.

PARTE$^{\textbf{Elixir}}$ wraps the sequential PARTE nodes of PARTE$^{\text{Rust}}$ inside actors, which can be distributed over a cluster of computers. PARTE$^{\text{Elixir}}$ includes rudimentary support for handling network issues.

6.1 PARTE^{Rust}: a Single-Machine PARTE Prototype

6.1.1 Overview

The multi-threaded prototypical implementation of fwPARTE goes by the name PARTE^{Rust}. PARTE^{Rust} supports launching a PARTE runtime, defining event templates for that runtime, creating a graph of PARTE nodes from a PARTElang program, and spawning those nodes on a thread pool managed by the PARTE runtime.

Once a PARTE runtime is initialized, it is ready to receive events. Events are represented as data structures mimicking the tuples described in section 5.4: they store a reference to their template, the values for each attribute defined by the template, and the event's timestamp. Received events are wrapped in alpha tokens, and placed in the inbox of the entry node for the event's event template. Next, PARTE^{Rust} executes the behavior prescribed by the rules of the global and local evaluation languages defined by the PARTE model. Evaluation at different nodes can happen in parallel, as each node is a distinct unit of concurrency, scheduled in the thread pool. When no evaluation steps are applicable, PARTE^{Rust} implicitly enters a waiting state (where all nodes are *waiting* as defined on page 126) until the external program either provides new input events, or calls for the termination of the PARTE runtime.

6.1.2 Rust Primer

PARTE^{Rust} is implemented in roughly 3000 lines of Rust [98] code. In-depth knowledge of Rust syntax is not needed to follow our description of PARTE^{Rust}. Rust code can broadly be understood if one is familiar with features of ML-like languages, and the syntax of C++ or Java. For the sake of self-containedness of this chapter, we provide a short primer of Rust semantics and syntax. Anyone familiar with Rust can safely skip this subsection.

Values and References

Values in Rust are not by default behind a reference, as they are in e.g. Java or Python, or as `class`es are in C#. Hence, binding a value to a variable does not mean the variable is set to a reference to that value. Instead, binding means that the value's bits are copied to the variable's location.

A reference to a value of type `T` is written as `&T`. Each reference has *lifetime*. A lifetime is a static type-level construct which represents the context in which a value is valid. Lifetimes can be used with references, where lifetime-checking can prevent use-after-free bugs. For instance, if a reference has a lifetime that outlives a method invocation, that method may return the reference. If a reference's lifetime does not outlive a method invocation, that method may not return that reference, as the reference might not be valid when used by the callee. The syntax `&'a T` is used to denote a reference with lifetime `'a`, which refers to a value of type `T`.

Ownership Model

An important feature of Rust is its ownership model, which gets statically checked by the "borrow-checker". A value is normally *owned* by exactly one variable. When a value is passed to a function or assigned to a variable, the ownership of the value is transferred to the new location. If the source of the value was a variable, that variable cannot subsequently be read from: the variable's value has been *moved*. Types can opt out of this behavior by implementing the `Clone` trait. Primitive types (integers, booleans, and characters) are `Clone`, as are shared references.

Shared references always refer to immutable values. Mutable references cannot alias, except by referring to types with *internal mutability*, such as mutex locks or cells. Cells and mutex locks prevent concurrent mutable access at run time. All of these features combine to offer a system for ownership-tracking, data-race-freedom, and automatic memory management without need for a tracing garbage collector.[1]

Closures

The syntax of closures in Rust is "`|arg-list| body`". The `arg-list` is a comma-separated list of argument names, optionally followed by a type annotation ("`arg-name : ArgType`"). The `body` consists of one expression, whose result is returned from the closure. A sequence of expressions can be generated by enclosing a semicolon-separated list of expressions in a pair of curly braces. The result of the last expression is returned as the result of the sequence.

Closures take references to variables in their lexical scope which are used in the body of the closure. This behavior can be overridden by writing the keyword **`move`** in front of the closure syntax; in that case the variables are *moved* into the closure instead being of captured by reference.

Pattern Matching

Rust offers *pattern matching* on variables. Wherever a `struct` is bound to a variable, the variable name can be replaced with a pattern listing the type constructor and the fields of

[1]The reader might notice in the code samples below that no manual memory management is performed. Nor are any of the code's multi-threading safety guarantees upheld manually. The synchronization mechanism used in PARTE$^{\text{Rust}}$ is simple yet safe: data is sent to other threads using multi-producer/single-consumer queues. Once data is handed off to another threads' queue, the original thread loses ownership over the data. The original thread can at that point not even syntactically express touching the data again without invoking a static error from the borrow-checker. Obviously, a receiving thread cannot touch the data before receiving it. Since sending and receiving is causally linked, at no point can a thread mutate data accessible by another thread.

that `struct` and variable names. The values of the fields of that `struct` are then bound to the variables. Pattern matching can also be used to *match* on the variants of an *enum*.

6.1.3 Differences between PARTE$^{\text{Rust}}$ and the Formal Model

Since PARTE$^{\text{Rust}}$ schedules the PARTE nodes on threads in a thread pool, the nodes technically share a single memory space. However, since they only communicate using asynchronous message passing, they are behaviorally equivalent to properly distributed units of computation. Some subtle but insignificant differences exist between the formalism presented in chapter 5 and the code shown below. For instance, instead of maintaining a list of identifiers of successor nodes, PARTE nodes in PARTE$^{\text{Rust}}$ maintain a list of message-passing channels. The implementation further makes no distinction between terminal and production nodes. Additionally, evaluation at terminal nodes deviates slightly from what the formal model prescribes. In the model, matching a rule is a transient state which does not "do" anything. In PARTE$^{\text{Rust}}$, a callback function is invoked whenever a rule is matched. This callback function receives as an argument the partial match that activated the terminal node. This deviation from the model is necessary for practical use cases.

The remainder of this section describes the relevant bits of the implementation in more detail. We focus specifically on the points where the implementation differs slightly from a naive translation of the formal model, and explain the reason for the difference, as well as the implications of our choices. The code is reformatted to fit the width of the pages. Error handling code is removed where it is not relevant to the discussion. Code that simplifies debugging and profiling — e.g. by setting up descriptive names for the threads — is removed as well. Overall we follow the bottom-up structure we introduced in section 5.4. Section titles hence match those found there.

6.1.4 Implementing Values and Events

The most basic type offered by our model and event algebra alike are values. Values in PARTE$^{\text{Rust}}$ are represented as an `enum`, i.e., a tagged union of all primitive types. Type safety of PARTElang programs is checked at run time using those tags. A number of basic operations are implemented for values, e.g., equality checks, inequality checks, addition, multiplication, etc.

Timestamps are implemented as an opaque type, but identify a number of milliseconds. The difference between two timestamps can be computed. This difference can be added to another timestamp, yielding a new timestamp. A strict order is defined on timestamps, but a universally highest timestamp exists (cf. an IEEE floating point's positive infinity), as well as a universally lowest timestamp (cf. an IEEE floating point's negative infinity).

```
 1 #[derive(Debug)]
 2 pub struct Slot {
 3   pub name: String,
 4   pub type_name: &'static str,
 5 }
 6
 7 #[derive(Debug)]
 8 pub struct Template {
 9   name: String,
10   id: TemplateId,
11   slots: Vec<Slot>,
12 }
13
14 impl Template {
15   pub fn new(name: &str, id: TemplateId, slots: Vec<Slot>) -> Result<Self, String> {
16     // ... check for duplicate names, empty names, and empty types
17
18     Ok(Template {name: name.to_owned(), id: id, slots: slots})
19   }
20
21   pub fn name(&self) -> &str { &self.name }
22
23   pub fn id(&self) -> TemplateId { self.id }
24
25   pub fn num_slots(&self) -> usize { self.slots.len() }
26
27   pub fn slot_at_index(&self, idx: SlotIndex) -> Option<&Slot> { self.slots.get(idx) }
28
29   pub fn index_of_slot(&self, slot_name: &str) -> Option<SlotIndex> {
30     self.slots.iter()
31       .position(|&Slot { name: ref n, .. } | { n == slot_name })
32       .map(|i| i as SlotIndex)
33   }
34 }
```

Listing 6.1: PARTE$^{\text{Rust}}$'s implementation of event templates

The implementation of event templates is depicted in listing 6.1, starting at line 14. As evidenced by the accessors, event templates are verified to be valid at construction time, and are immutable afterwards. Note the closure syntax on line 32: this closure transforms a single argument `i` by casting it to a `SlotIndex`. The line before it contains another closure. This closure also takes a single argument — a reference to a `Slot` — but pattern matches on it to *destructure* it into its constituents. Since destructuring a pattern is the inverse of constructing a value, operators have their inverse meaning: in a value expression, a reference operator (an ampersand) takes a reference to a value. In a pattern, that same operator "strips off" a reference, effectively causing the argument to be dereferenced.

As depicted in listing 6.2, events store a `TemplateId` specifying the template, as well as the event's data: the values for the attributes and the time at which the event occurred. At construction time, events are verified to conform to the specification imposed by the template. Events are immutable, exposing only accessors for their attribute values and their timestamp of occurrence. We implement the trait `Index<usize>` for `Events`. This means that for a given event `e` and a given unsigned pointer-sized integer `idx`, the syntax `e[idx]` resolves to the value returned by the `index` method defined on line 32, i.e., `e[idx]` gets the attribute value at index `idx` of event `e`.

```rust
#[derive(Debug, Eq, Ord, PartialOrd, PartialEq)]
pub struct Event {
  template_id: TemplateId,
  values: Vec<Value>,
  timestamp: Time,
}

impl Event {
  pub fn new(template: &Template, values: Vec<Value>, timestamp: Time)
      -> Result<Self, String>
  {
    if values.len() != template.num_slots() {
      Err(/* ... create error message */)
    } else {
      for (slot, value) in template.slots.iter().zip(values.iter()) {
        if slot.type_name != value.type_name() {
            return Err(/* ... create error message */);
        }
      }
      Ok(Event { template_id: template.id(),
                 values: values,
                 timestamp: timestamp })
    }
  }

  pub fn timestamp(&self) -> Time { self.timestamp }
}

impl Index<usize> for Event {
  type Output = Value;

  fn index(&'e self, idx: usize) -> &'e Value {
    self.values.get(idx).unwrap()
  }
}
```

Listing 6.2: PARTE$^{\text{Rust}}$'s implementation of events

6.1.5 Implementing Tokens

Tokens are either alpha tokens or beta tokens. Beta tokens carry multiple events' data: a partial match.

Partial Matches

Our implementation of partial matches in PARTE$^{\text{Rust}}$ is depicted in listing 6.3. In essence, a `PartialMatch` is a vector of atomically reference counted `Events`, with a number of methods defined on the structure. This detail of the implementation shows that PARTE$^{\text{Rust}}$ exploits the fact that all nodes exist in the same address space. This entails that measurements of the working set size of PARTE$^{\text{Rust}}$ programs might show lower numbers than what might be expected from a naive translation of the fwPARTE model. In turn, this introduces the possibility for contention, as multiple threads may have to mutate the atomic integers in which the reference counts are tracked.

```
#[derive(Clone, Debug, PartialEq, Eq, PartialOrd)]
pub struct PartialMatch {
    pub events: Vec<Arc<Event>>,
}

impl PartialMatch {
  pub fn new_by_joining_assuming_consistent(&self,
                                             right_pm: &SingletonPartialMatch) -> Self {
    let mut events = Vec::with_capacity(self.events.len() + 1);
    events.extend_from_slice(&self.events);
    events.push(right_pm.cloned_event());
    PartialMatch { events: events }
  }
}
```

Listing 6.3: PARTE$^{\text{Rust}}$'s core implementation of partial matches

Since a partial match may consist of multiple events and each event may have values for multiple attributes, values in partial matches are identified by a two-dimensional index. Our implementation introduces an auxiliary type `QualifiedSlot` (see line 2 in listing 6.4 on page 166), which identifies a slot qualified by an event index into a partial match. A `QualifiedSlot` is an opaque pair of two indices, together with a number of auxiliary methods. Its fields are not publicly accessible. Instead, it must be used as a whole: we implement the `Index` trait for `PartialMatch`es using `QualifiedSlot`s or references to them, as shown from line 31 onwards. This has the result that for a partial match `pm` and a qualified slot `qs`, the syntax `pm[qs]` evaluates to `pm.events[qs.event_idx][qs.slot_idx]`. This gives rise to a difference in notation between fwPARTE and PARTE$^{\text{Rust}}$, though the difference is inconsequential.

Alpha Tokens

We have now covered the implementation of partial matches. We move on to alpha and beta tokens themselves.

Listing 6.5 depicts relevant excerpts of the implementation of alpha tokens. Alpha tokens store an identifier of the entity that generated them: either a primitive source (i.e., an external event source), or a complex source (i.e., a terminal node of the PARTE$^{\text{Rust}}$ graph). The lower bound on future event timestamps sent along the edge the `AlphaToken` travels, or ℓ for short, is represented as a timestamp named `oldest_timestamp_to_expect`. The alpha token's payload (see page 130) is encoded as an option type: instead of considering it either an event or a "no change" indicator $\perp$, we implement it as either `None` or `Some(event)`. This change makes the code more concise, and is more idiomatic in Rust. The behavior is unaltered: both fwPARTE and PARTE$^{\text{Rust}}$ model the payload as a sum type, of which one variant refers to an event, and the other carries no additional data.

To support *node reuse* (see figure 5.2 on page 98), it must be possible to duplicate alpha tokens so that they may be sent to multiple successors. As in the case of partial matches, we accomplish this by making the payload `Event` atomically reference-counted by wrapping it: `Arc<Event>`. The Rust type for fields optionally containing an atomically reference-

```rust
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub struct QualifiedSlot {
  event_idx: usize,
  slot_idx: usize,
}

impl QualifiedSlot {
  pub fn new(event_idx: usize, slot_idx: usize) -> Self {
    QualifiedSlot { event_idx, slot_idx }
  }

  pub fn transform_from_global_to_right(self, num_left_events: usize) -> Result<Self,
       String> {
    if self.event_idx < num_left_events {
      Err(/* ... create error message */)
    } else if self.event_idx > num_left_events {
      Err(/* ... create error message */)
    } else {
      Ok(QualifiedSlot { event_idx: 0, slot_idx: self.slot_idx })
    }
  }

  pub fn is_within_bounds(&self, num_events: usize) -> bool {
    self.event_idx < num_events
  }

  pub fn is_in_event(&self, event_idx: usize) -> bool {
    self.event_idx == event_idx
  }
}

impl Index<QualifiedSlot> for PartialMatch {
  type Output = Value;

  fn index<'p>(&'p self, qs: QualifiedSlot) -> &'p Value {
    self.events[qs.event_idx][qs.slot_idx]
  }
}

impl<'q> Index<&'q QualifiedSlot> for PartialMatch {
  type Output = Value;

  fn index<'p>(&'p self, qs: &'q QualifiedSlot) -> &'p Value {
    self.events[qs.event_idx][qs.slot_idx]
  }
}
```

Listing 6.4: PARTE$^{\text{Rust}}$'s core implementation of qualified slots, an auxiliary structure for indexing into partial matches

counted `Event` is `Option<Arc<Event>>`, which is hence the type of the `payload` field of an `AlphaToken`. Since events are immutable after construction, sharing does not modify the behavior.

Beta Tokens

Beta tokens carry not just one event's data, but possibly multiple. We have previously introduced the `PartialMatch` type to store those events' attributes and timestamps. One more field constitutes a beta token: a list of temporal lower bounds per event pattern.

```rust
#[derive(Clone, Debug)]
pub struct AlphaToken {
  generator_id: GeneratorId,
  oldest_timestamp_to_expect: Time,
  payload: Option<Arc<Event>>,
}

impl AlphaToken {
  pub fn new_with_event(generator_id: GeneratorId,
                        oldest_timestamp_to_expect: Time,
                        event: Arc<Event>
  ) -> Self {
    let payload = Some(event);
    AlphaToken { generator_id, oldest_timestamp_to_expect, payload }
  }

  pub fn new_no_change(generator_id: GeneratorId,
                       oldest_timestamp_to_expect: Time
  ) -> Self {
    let payload = None;
    AlphaToken { generator_id, oldest_timestamp_to_expect, payload }
  }

  pub fn generator_id(&self) -> &GeneratorId {
    &self.generator_id
  }

  pub fn oldest_timestamp_to_expect(&self) -> Time {
    self.oldest_timestamp_to_expect
  }

  pub fn payload(&self) -> &Option<Arc<Event>> {
    &self.payload // Return a reference to the payload
  }

  pub fn into_payload(self) -> Option<Arc<Event>> {
    self.payload // Return the payload, consuming the AlphaToken
  }
}
```

Listing 6.5: PARTE$^{\text{Rust}}$'s implementation of alpha tokens

This is in line with the formal model. Listing A.3 (on page 243 in the appendix) depicts the relevant parts of the implementation of beta tokens.

Remember from section 5.4.4 (on page 132) that the formal model associates an activation side with beta tokens, forming the *message* abstraction. This abstraction is not present in PARTE$^{\text{Rust}}$. PARTE$^{\text{Rust}}$ uses `ActivationSides` only when describing how PARTE graph nodes should be linked up. We encode `ActivationSides` straightforwardly as an enumaration of three variants: *a.)* `Activate` for successor relations activated from the top, cf. $\downarrow$; *b.)* `LeftActivate` for successor relations activated from the left, cf. $\searrow$; and *c.)* `RightActivate` for successor relations activated from the right, cf. $\swarrow$.

```
1 pub enum Type {
2   Entry, Alpha, Join, Not, Terminal,
3 }
4
5 pub trait Node : Send {
6   type InToken : Token;
7   type OutToken : Token;
8
9   fn node_id(&self) -> NodeId;
10  fn node_type(&self) -> Type;
11  fn is_on_path_to_negation(&self) -> bool;
12  fn enter_processing_loop(self) -> JoinHandle<Result<(), String>>;
13 }
```

Listing 6.6: PARTE$^{\text{Rust}}$'s implementation of PARTE nodes: an enumeration of types and a trait.

6.1.6 Implementing Graph Nodes

PARTE$^{\text{Rust}}$ supports entry nodes, alpha nodes, join-nodes, not-nodes, and terminal nodes.[2] An `enum` is defined for the different types of PARTE nodes (listing 6.6). Additionally, the `Node` trait is defined.

The Node Trait

The `Node` trait abstracts over the different node types, enabling polymorphism. Since different types of PARTE nodes consume and produce different types of tokens, the nodes must specify what kind of tokens they produce. We do so using two *associated types*: `InToken` and `OutToken`. The trait bound "`: Token`" on `InToken` and `OutToken` constrains what an implementation of `Node` can look like: only types which implement the `Token` trait can be used as a node's `InToken` or `OutToken` type.

Apart from the associated types, the `Node` trait specifies 4 methods: one to retrieve the node's globally unique identifier (line 9), one to retrieve the node's `Type` (line 10), a predicate which indicates whether the node is an (indirect) predecessor of a not-node [3] (line 11), and a method which consumes the `Node` and starts the node's execution loop (line 12). Nodes are scheduled as coroutines on a thread pool. The coroutine runtime replaces a coroutine with another one on the same thread whenever the former coroutine performs asynchronous I/O or explicitly yields. PARTE$^{\text{Rust}}$ nodes yield when they find their inbox empty. Hence, though `Node`s seem to follow the approach of *event handling by waiting* (see section 3.1 on page 35), the approach is implicitly transformed into *event handling by callbacks* on a thread pool. Hence, the runtime needs not have as many OS-level threads as there are PARTE nodes.

[2]PARTE$^{\text{Rust}}$'s terminal nodes perform the job of both fwPARTE terminal nodes and fwPARTE production nodes.

[3]Whether or not a node precedes a not-node in the PARTE graph determines whether "no change" indicators need to be propagated, as explained in section 8.4.

```
use std::error::Error;

struct AlphaSuccessorCollection {
  successors: Vec<Sender<AlphaToken>>,
}

impl AlphaSuccessorCollection {
  fn new(successors: Vec<Sender<AlphaToken>>) -> Self {
      AlphaSuccessorCollection { successors }
  }

  fn is_empty(&self) -> bool { self.successors.is_empty() }

  fn send_token(&self, token: AlphaToken) -> Result<(), String> {
      let mut i = self.successors.iter();
      if let Some(succ) = i.next() {
          for succ in i {
            self.send_token_internal(succ, token.clone())?;
          }

          self.send_token_internal(succ, token)?;
      } else {
          Ok(()) // No successors, sending is trivially done
      }
  }

  fn send_token_internal(&self, succ: &mut SyncSender<AlphaToken>, token: AlphaToken) ->
       Result<(), String> {
   loop {
     match succ.try_send(token_to_send) {
       Ok(()) => { break; /* Sent succesfully */ },
       Err(TrySendError::Full(unsent_token)) => {
         token_to_send = unsent_token;
         Scheduler::sched(); // Yield coroutine
       },
       Err(TrySendError::Disconnected(unsent_token)) => {
         return Err(format!("Failed to send {:?}: channel disconnected",
                            unsent_token));
       }
     }
    }
  }
}
```

Listing 6.7: PARTE$^{\text{Rust}}$'s implementation of successor collections ($\bar{\mathfrak{s}}$)

Successors

Each node keeps track of a set of successors. Those successors accept either alpha tokens or beta tokens. PARTE$^{\text{Rust}}$ abstracts over sets of successors as using either `AlphaSuccessorCollection` or `BetaSuccessorCollection`. Both these types are wrappers around a vector of `Senders`. A `Sender` is the sending side of a channel between coroutines. Both types of successor collection feature a constructor, a predicate method to verify whether the successor collection contains any successor, and a method which takes a token and sends one copy of it to each successor. Listing 6.7 shows the implementation of `AlphaSuccessorCollection`. The code is nontrivial since it makes only $n-1$ copies to serve n successors. The first successor instead receives the original token.

The formal model abstracts over the notion of making copies. Furthermore, as demonstrated by line 33, PARTE$^{\text{Rust}}$ deviates from the formal model by potentially yielding during the transmission of tokens to successors. In contrast, fwPARTE uses an intermediate storage: the outbox. Featherweight PARTE ensures there is room in the outbox before starting (Proc), and that there is room in all the successor nodes' inboxes before starting (Proc). Clearly, the amount of unsent tokens after processing an activation is at most the total number of tokens produced during the processing of the activation. The variable size of the coroutine when it is yielded is hence limited to a constant factor of the *minimum viable outbox size* (see page 127). Our proofs on maximum memory usage hence apply to PARTE$^{\text{Rust}}$ as they do to fwPARTE.

In conclusion, the representation of PARTE graph nodes in PARTE$^{\text{Rust}}$ differs significantly from that in the formal model. Nodes in PARTE$^{\text{Rust}}$ are not constructed as a tuple $\mathcal{N}\langle \iota_n, \bar{\mathfrak{s}}, \overline{m}, \bar{t}, ndata\rangle$ of node identifier, set of successors, inbox, outbox, and node-local data. The role of an inbox and outbox is taken over by the `Receiver` and `Sender` sides of a bounded channel. The logic of successor collections is taken over by the types introduced in the previous paragraph. That leaves only a node identifier and the node-type-specific node-local data. PARTE$^{\text{Rust}}$'s implementation incorporates those into the node-local data-structure of each node type. While all types of nodes share a unified interface — the trait `Node` — they are all implemented as separate types. This entails that we need to duplicate the implementation of an accessor for the node identifier in each node, but that evidently comes at only a small cost.

6.1.7 Implementing Node-local Data

As a representative example, consider the slightly simplified implementation of alpha nodes in listing 6.8. A node's implementation consists of three main parts: a `struct` holding the node's data, a constructor and typically some auxiliary methods, and an implementation of the `Node` trait for that `struct`.

A node's constructor has five tasks:

- it establishes the invariants of the node's state — in the case of alpha nodes this merely means checking that successors exist;
- it builds the successor collection from the provided list of successors;
- it creates the multi-producer/single-consumer channel by which tokens are sent to this node. A channel's constructor returns a tuple whose left element is a cloneable object through which messages can be sent (`Sender<T>`), and whose right side is a non-cloneable object from which messages can be received (`Receiver<T>`). Additionally;
- it composes the `struct` itself;
- it starts the node's processing loop on a background thread; and

```rust
pub struct AlphaNode {
    inbox: Receiver<AlphaToken>,
    successors: BetaSuccessorCollection,
    condition: Box<for<'e> Fn(&'e Event) -> bool + Send>,
    is_on_path_to_negation: bool,
    node_id: NodeId,
}

impl AlphaNode {
    pub fn new(node_id: NodeId,
               successors: Vec<(Sender<BetaToken>)>,
               condition: Box<Fn(&Event) -> bool + Send>,
               is_on_path_to_negation: bool,
    ) -> (JoinHandle<Result<(), String>>, Sender<AlphaToken>) {
        assert!(!successors.is_empty(), "An AlphaNode must have successor nodes.");
        let (sender, inbox) = sync_channel(1);
        let successors = AlphaSuccessorCollection::new(successors);
        let node = AlphaNode {
            inbox,
            successors,
            condition,
            is_on_path_to_negation,
            node_id,
        };
        (node.enter_processing_loop(), sender)
    }
}

impl Node for AlphaNode {
    type InToken = AlphaToken;
    type OutToken = BetaToken;

    fn node_id(&self) -> NodeId { self.node_id }

    fn node_type(&self) -> Type { Type::Alpha }

    fn is_on_path_to_negation(&self) -> bool { self.is_on_path_to_negation }

    fn enter_processing_loop(mut self) -> JoinHandle<Result<(), String>> {
        Scheduler::spawn(move || -> Result<(), String> {
            while let Ok(Some(token)) = self.inbox.recv() {
                let oldest_timestamp_to_expect = token.oldest_timestamp_to_expect();

                if let Some(Event(e)) = token.into_payload() { // If token contains event
                    if (*self.condition)(&e) { // If event satisfies the alpha condition
                        let outgoing_token = BetaToken::new_from_single_event(
                            oldest_timestamp_to_expect,
                            e.clone());
                        self.successors.send_token(outgoing_token)?
                    } else {
                        // Nothing to do: event failed test
                    }
                } else { // Process "no change"
                    let outgoing_token = BetaToken::new_from_single_no_change(
                         oldest_timestamp_to_expect);
                    self.successors.send_token(outgoing_token)?
                }
            }
        })
    }
}
```

Listing 6.8: PARTE$^{\text{Rust}}$'s implementation of alpha nodes

- it returns a handle to the spawned coroutine, as well as the sending side of the channel to the node. Note that the `Sender` returned here is used when the node is registered as a successor of its predecessor. This entails that in PARTE$^{\text{Rust}}$ the PARTE graph is constructed from the terminal nodes up. To make this more concrete, consider again the case of an alpha node. When an alpha node is created, a channel of alpha tokens is constructed. The receiving side, a `Receiver<AlphaToken>`, is stored in the alpha node's `inbox` field. The sending side, a `Sender<AlphaToken>`, is stored in the `AlphaSuccessorCollection` that serves as the `successors`-field for some entry node. Similarly, each `Sender<BetaToken>` in the alpha node's `successors`-field has been returned by the constructor of a successor node such as a join-node, not-node, or terminal node.

The last three aspects of the implementation of nodes which warrant discussion in detail are the following: *a.)* the computation of lower bounds per event source in entry nodes; *b.)* the processing of a left activation in a join-node; and *c.)* the handling of event emission and calls to side-effecting code from terminal nodes.

a.) Computing Temporal Lower Bounds

Entry nodes keep track of the lowest timestamp for each event source they know. Section 5.4.3 describes how fwPARTE stores this data as a mapping from source identifier to temporal lower bound, in a field named ℓmap. In the implementation, that trivially maps onto a `HashMap<GeneratorId, Time>`. The way this mapping is maintained, as well as how the lower bound is extracted, is depicted in listing 6.9. As entry nodes are located at the roots of the PARTE graph, they have to deal with invalid input data. Most of the code in `update_stored_timestamps` concerns data validation. The core logic consists of getting a mutable reference to the correct entry in the map, and assigning the new oldest timestamp to it. `compute_oldest_timestamp_for_any_event_source` retrieves the lower bound by taking the minimum of the values stored in the map.

b.) Processing Left-Activation of a Join-Node

Evaluation at join-nodes must achieve the behavior specified by (Join-$\searrow$-$\perp$), (Join-$\swarrow$-$\perp$), (Join-$\searrow$), and (Join-$\swarrow$) (defined from page 138 onwards). To this end, any two-input node which is in a *waiting state* (page 126) asynchronously selects from two inboxes: a left inbox for left activations, and a right inbox for right activations. When either partial match history is at capacity, the node instead only tries to receive tokens from the opposite inbox. Beyond this, the process closely matches what the formal model prescribes by the rules of $\xrightarrow{l}$.

Listing A.4 (on page 244 in the appendix) depicts the code handling a left activation of a join-node. First, some sanity-checks are performed on the received data. Next, the left temporal lower bounds stored at the join-node is updated (line 16). With those lower

```rust
// With field oldest_timestamp_per_event_source: HashMap<GeneratorId, Time>

impl EntryNode {
   // .. constructor etc.

   fn update_stored_timestamps(&mut self,
                               generator_id: &GeneratorId,
                               oldest_timestamp_to_expect_from_generator: Time,
   ) -> Result<(), String> {
      // Determine oldest timestamp previously known for this generator
      let mut oldest = self.oldest_timestamp_per_event_source.get_mut(generator_id)
      if let Some(oldest_t_for_generator_in_map) = oldest {
         // Ensure the promise on the lower bound is not invalidated,
         // as that would be a logic error.
         if oldest_timestamp_to_expect_from_generator < *oldest_t_for_generator_in_map {
             Err(/* ... create error message */)
         } else {
            // If all is well, update the lower bound.
            *oldest_t_for_generator_in_map = oldest_timestamp_to_expect_from_generator;
            Ok(())
         }
      } else {
         Err(format!("Received token from unknown generator {:?}.", generator_id))
      }
   }

   fn compute_oldest_timestamp_for_any_source(&self) -> Time {
      self.oldest_timestamp_per_event_source
          .values()
          .min()
          .unwrap("Expected at least one event source")
   }
}
```

Listing 6.9: Computing temporal lower bounds in entry nodes in PARTE$^{\text{Rust}}$

bounds updated, stale partial matches can be expired from the right partial match history (line 17). The stored and explicit lower bounds are combined, and stored as `ls`. Next, PARTE$^{\text{Rust}}$ checks whether it should proceed as specified in (Join-$\searrow$-$\perp$), or as specified in (Join-$\searrow$). In the former case, evaluation continues from line 46 onwards: "no change" token is propagated to the successors. Otherwise, the lifetime of the newly arrived partial match is computed (line 23). The relational join between the new partial match and those in the right partial match storage is computed. Temporal join-constraints are checked using the `left_lifetime` in combination with the concrete timestamp of each right partial match: only valid matches are produced. The `RightPartialMatchStorage` keeps its collection of partial matches sorted, which makes this selection a cheap operation. The `RightPartialMatchStorage` additionally encodes the non-temporal join-constraints. Partial matches failing the non-temporal constraints are filtered out (line 33). For successful matches a new token is created and propagated to the successors (line 39). Finally, the partial match with which the join-node was activated is stored in the left partial match history.

c.) Representing Reaction Logic

Whenever a terminal node is activated in fwPARTE, it means that some rule was matched. PARTE$^{\text{Rust}}$ exposes the fact that a rule matched by calling some user-specified function.

Whenever a production node is activated in fwPARTE, a new complex event is created from the activating partial match. Since PARTE$^{\text{Rust}}$ merges the notion of a production node and a terminal node, it are PARTE$^{\text{Rust}}$'s terminal nodes which implement this behavior.

Both emitting new events and calling user-specified functions, require that there exists a notion of "some task to be executed, based on the values of a partial match". We implement this in PARTE$^{\text{Rust}}$ as a type `Action`. `Action` is an enumeration — i.e., a sum type — of either `EmitAlphaToken` or `CallForeignAction`. Both variants have a number of attributes. Their definition is listed on lines 6 to 13 in listing A.5 (on page 245 in the appendix). Each `Action` can be executed with either a partial match or with "no change".

Alpha token emission actions respond to partial matches as outlined from line 22 onwards. First, the values for the new complex event are computed. This is accomplished using a list of `ValueProcedure`; one procedure for each attribute of the new complex event. Each function takes an iterator over (references to) the values in the partial match, and produces a new value (line 26). Next, the new complex event's inherited timestamp is computed by a `TimestampProcedure`, a function which takes an iterator over the timestamps of the partial match, and returns a new timestamp for the complex event. The `TimestampProcedure` is also used to determine the inherited lower bound on timestamps guaranteed by this terminal node. Note that this procedure must uphold the constraints outlined in section 5.4.6 for the program to behave correctly. To reiterate: when used to determine temporal lower bounds, PARTE ensures that the arguments provided to this procedure are non-decreasing across successive calls. In turn, the procedure must ensure its return value is non-decreasing across successive calls. Upholding this invariant is the responsibility of the parser of the PARTE rule. The invariant is not verified at run time by our prototypical implementation. Assuming a valid temporal lower bound is computed, an alpha token is generated to wrap the new complex event. This event is sent to all entry nodes which consume events of the template produced by this terminal node (line 41).

Foreign call actions have a relatively simple response to partial matches, as shown on line 44: they merely invoke an `ActionProcedure`. An `ActionProcedure` takes (a reference to) a partial match, and returns either nothing (technically: `()`, i.e., unit, i.e., an empty tuple) or an error message. If the foreign action returns an error message, this error is propagated upwards, shutting down the prototype.

Handling of activation by "no change" indicators is depicted on lines 49 to 65. If a terminal node has an emit action, it has successors which need to be informed of the "no change". The `EmitAlphaToken` action hence takes care of this. Calls to foreign action do not interact with "no change". They hence take no further action.

6.1.8 Implementing the Global Evaluation

In the formal model, the global evaluation language $\xrightarrow{g}$ consists of two rewrite rules: (Proc) and (Prop), as defined on page 145. (Proc) defines how a PARTE node invokes the local evaluation language $\xrightarrow{l}$ when a token is available for processing. In PARTE$^{\text{Rust}}$, this happens implicitly in the `enter_processing_loop` method of nodes. The propagation rule (Prop) defines how produced tokens are moved from a node's outbox to its successors' inboxes. This, too, is handled differently in PARTE$^{\text{Rust}}$. We explained in section 6.1.6 that PARTE$^{\text{Rust}}$ omits explicit outboxes. Furthermore, being a practical implementation, PARTE$^{\text{Rust}}$ can employ potential parallel execution based on the number of hardware threads available. The formal model allows multiple steps of $\xrightarrow{g}$ to be taken at the same time, as long as they evaluate at separate nodes. PARTE$^{\text{Rust}}$ relaxes that requirement: since the inboxes are thread safe, it is possible for, e.g., evaluation of (Join-$\searrow$) to take place at a join-node, while simultaneously the evaluation of (Prop) takes place at both direct predecessors of the join-node. The necessary synchronization on the inbox is handled at a much finer granularity than evaluation steps of $\xrightarrow{l}$ and $\xrightarrow{g}$.

6.2 PARTE$^{\text{Elixir}}$: a Truly Distributed PARTE Prototype

The second prototypical implementation of the Featherweight PARTE model goes by the name PARTE$^{\text{Elixir}}$. PARTE$^{\text{Elixir}}$ is a combined Rust/Elixir program which schedules PARTE$^{\text{Rust}}$ nodes as Elixir [53] actors on a BEAM virtual machine [19]. [4]

The main goal of PARTE$^{\text{Elixir}}$ is to demonstrate that PARTE properly works in a distributed setting such as a computer cluster. PARTE$^{\text{Elixir}}$ demonstrates that the formal model does not inadvertently depend on a shared global address space, a global clock, or other features one might accidentally underspecify in a mathematical model and an implementation like PARTE$^{\text{Rust}}$.

As we addressed in section 5.6.4, handling partial failures in a PARTE graph is still an unsolved problem: existing tools for fault tolerance and failure recovery do not provide the strong guarantees PARTE requires, and PARTE's guarantees cannot express the uncertainty of hardware failure. As such, the failure handling in PARTE$^{\text{Elixir}}$ is limited to the initial graph-construction phase.

6.2.1 Overview

PARTE$^{\text{Elixir}}$ consists of a number of core parts:

[4] A version of the tools that implement PARTE$^{\text{Elixir}}$ is available at `https://soft.vub.ac.be/~trenaux/PARTE/implementation.zip`.

- the core implementation of PARTE$^{\text{Rust}}$, with some minor modifications (see section 6.2.4);
- the Rustler library, which simplifies combining Rust code with Elixir code (see section 6.2.3);
- an Elixir stub for each type of PARTE node (see section 6.2.5); and
- some Elixir code which reads a JSON representation of a PARTE graph — created by PARTE$^{\text{Rust}}$ — and instantiates and links up an instance of the correct Elixir stub for each node in the graph.

6.2.2 Elixir Primer

To understand this section, it suffices to know that Elixir is a dynamically typed actor-based language which runs on BEAM virtual machine, i.e., the virtual machine originally developed for the Erlang programming language. Elixir aims to be a successor to Erlang. Elixir is homoiconic and has a powerful macro system, through which it offers language features which are more "modern" than what Erlang offers. For our case, the distinction between Erlang and Elixir is negligible. We only need a means of scheduling PARTE graph nodes as distributed actors on a BEAM VM, with the minimal wrapper code that enables this. Still, since we discuss some Elixir code, a brief look at Elixir syntax and semantics is in place.

Program Structure

At the top level, an Elixir program consists of a module, defined using `defmodule the_module_name do the_module_content end`. In a module, functions can be defined using `def the_function_name(a, comma-separated, list, of, arguments) do the_function_body end`. Additionally, closures can be constructed using `fn an_argument_pattern -> the_function_body end`, which supports variable pattern matching on the arguments: A closure of the form `fn {:sym, a, b} -> body end` expects to receive a three-tuple as its only argument, of which the first element is the symbol `:sym`, and of which the second and third elements are bound to new local variables `a` and `b`, respectively. Values can be bound to variables using `var_name = expression`. No keyword is required, and no distinction is made between introducing a new variable, or shadowing a variable with a new value.

Actors and Message Sending

Since Elixir is built on top of the BEAM VM, its main form of concurrency and parallelism is offered by actors, or "processes". Actors can communicate with each other by sending messages asynchronously using `send the_recipient_pid, the_message`. The two arguments to `send/2` [5] are the BEAM process identifier of the recipient actor, and the message to send. The message can transparently be serialized and sent over a computer network. Actors do not share state, and conceptually live in a different address space.

Complementary to `send/2` is `receive/1`. Actors can process messages that were sent to them by explicitly entering a `receive` call. The `receive` construct takes as argument a set of patterns, the expressions to evaluate when a message in the actor's inbox matches that pattern, and optionally a timeout delay and the expression to evaluate when no matching message was received within the timeout delay.

6.2.3 Interfacing Rust with Elixir

The responsibility of our interface between Rust and Elixir code is to make the code of the PARTE$^{\text{Rust}}$ implementation described in section 6.1 available to Elixir code. This is facilitated by Erlang's foreign function interface, which is based on *NIF*s. NIFs — which stands for "Natively Implemented Functions" — form the API between the internals of the BEAM VM — which is implemented in C — and any language which can interface with C-language APIs. NIFs can be linked to Erlang functions, enabling Erlang or Elixir code to call the function. The NIF API provides the functions to extract data from the Erlang data types which are passed in as arguments to the NIF, and the functions to construct results, possibly in the managed BEAM heap, possibly pinned to some address. Obviously, in the latter case it is the programmer's responsibility to later invoke a NIF which deallocates that result.

Using NIFs, PARTE$^{\text{Elixir}}$ can instantiate PARTE$^{\text{Rust}}$ nodes, and register functions which, e.g., retrieve a node's node identifier. However, it would be cumbersome and error-prone to manually register the constructors and other methods of all node types, as well as the means of translating tokens to and from an Elixir representation. We therefore made use of the Rustler library [82]. Rustler automates wrapping Rust types in BEAM VM resources, exposing Rust methods as NIFs, and inserting serialization and deserialization methods around message-sends of Rust values.

6.2.4 Modifications to PARTE$^{\text{Rust}}$ Code

PARTE$^{\text{Rust}}$ code could not be integrated as-is. Two main modifications were required: (de-)serialization code had to be added, and the ability to yield to the thread pool had to be adapted to cooperate with Elixir's runtime instead.

[5] Elixir does not in general support overloading functions, but functions with the same name but of different arity can be distinguished. The arity of a function is commonly appended to the name, separated by a forward slash.

```
 1 impl NifEncoder for Time {
 2   fn encode<'a>(&self, env: NifEnv<'a>) -> NifTerm<'a> {
 3     self.0.encode(env)
 4   }
 5 }
 6 impl<'a> NifDecoder<'a> for Time {
 7   fn decode(term: NifTerm<'a>) -> NifResult<Time> {
 8     Ok(Time(term.decode()?))
 9   }
10 }
```

Listing 6.10: (De-)serialization of PARTE$^{\text{Rust}}$ timestamps for use in PARTE$^{\text{Elixir}}$

Serialization and Deserialization of PARTE Tokens

The first is straightforwardly related to PARTE$^{\text{Elixir}}$'s main novelty compared to PARTE$^{\text{Rust}}$: since PARTE$^{\text{Elixir}}$ operates in a distributed address-space setting, messages must be serialized and deserialized. As a result, in PARTE$^{\text{Elixir}}$ events are no longer wrapped as `Arc<Event>`, as they are in PARTE$^{\text{Rust}}$. Instead, some new code was added to the different data types to implement serialization and deserialization.

A manual implementation of serialization and deserialization of timestamps is depicted in listing 6.10. Given a `Time` object and an BEAM NIF environment, `encode` produces a NIF term which is valid while the NIF environment is valid. Rustler captures this using Rust *lifetimes*. The first example hereof starts on line 2: the method `encode` is generic over some lifetime `'a`. Any call-site of `encode` determines the duration of lifetime `'a` for that call. Since the second argument — a NIF environment — specifies the lifetime `'a`, that lifetime is bound to the lifetime of the NIF environment. Since the code specifies that return value must also have lifetime `'a`, it is impossible to return a term that has been generated in some other NIF environment. Additionally, since the only information available about lifetime `'a` is that `env` has lifetime `'a`, it would be a compile-time error to save the NIF environment — or any term created in that environment — in some Rust variable that outlives the execution of `encode`, since Rust cannot show that `'a` outlives the function call. As such, the requirement that the Erlang developers could only express in plain English in the documentation of the NIF FFI, can be enforced by Rust's type checker.

The manual implementation depicted in listing 6.10 is more complicated than what is typically necessary. For most PARTE$^{\text{Rust}}$ types, Rustler can automate the process further. As a representative example, consider again the PARTE$^{\text{Rust}}$ implementation for `Events` in listing 6.2. To add the capability of serialization and deserialization to events, it suffices to add the line `#[derive(NifMap)]` in front of the definition of the struct. This line instructs Rust to auto-derive the `NifMap` trait specified by Rustler. Auto-deriving causes Rustler to generate the code for translating events to and from an Erlang map. Attribute names get translated to Erlang atoms, values get recursively serialized and deserialized by Rustler.

Replacing the Thread Pool with the Elixir Actor Runtime

The second modification to PARTE$^{\text{Rust}}$ code that is required for PARTE$^{\text{Elixir}}$, is the result of replacing the coroutine framework with an Elixir-based actor framework: nodes no longer enter a processing loop on their coroutine. Instead, they expose an API through which they can be informed of activations. The loop and message reception are encoded in Elixir (as we will discuss in section 6.2.5). Concretely, this means that nodes lose their `enter_processing_loop` method, and instead define either `activate` or `left_activate` and `right_activate`. These activation-methods gather the tokens to propagate to successors — i.e., the outbox of the nodes — and return them to their caller. Using Rustler, the methods are exposed to Elixir code as NIFs.

6.2.5 Linking PARTE$^{\text{Elixir}}$ to PARTE$^{\text{Rust}}$

```
call_successors = fn (msg) -> call_node_successors(processes, n, msg) end
activate = case activation_procedures do
  {activate} ->
    fn token -> call_successors.(activate.(node,token)) end
    {activate_left, activate_right} ->
    fn {:left, token} -> call_successors.(activate_left.(node,token))
        {:right, token} -> call_successors.(activate_right.(node,token))
    end
      end
forever(fn -> receive do t -> activate.(t) end end)
```

Listing 6.11: PARTE$^{\text{Elixir}}$'s Elixir wrappers around PARTE$^{\text{Rust}}$ nodes

The final piece of PARTE$^{\text{Elixir}}$ to discuss, are the Elixir wrappers for each node. Listing 6.11 depicts the Elixir code accomplishing this. Each of the wrappers consists of a function which `forever` loops trying to `receive` a token. Upon receipt of a token, the function activates the underlying PARTE$^{\text{Rust}}$ node by calling the function registered through Rustler. The tokens returned by the NIF are propagated to the successor nodes.

Excluding project configuration files and the code inside the Rustler library, PARTE$^{\text{Elixir}}$ is implemented less than 200 lines of Elixir.

6.3 Revisiting the Limitations of Featherweight PARTE

In section 5.6 we enumerated a number of limitations of fwPARTE. Since PARTE$^{\text{Rust}}$ and PARTE$^{\text{Elixir}}$ are implementations of the fwPARTE model, these limitations apply to PARTE$^{\text{Rust}}$ and PARTE$^{\text{Elixir}}$ too. In this section we briefly discuss how these limitations manifest themselves in both prototypical implementations, and how some of these limitations could be approached in PARTE$^{\text{Rust}}$ or PARTE$^{\text{Elixir}}$ once fwPARTE defines how to handle them.

First, we mentioned that fwPARTE is — by design — a reduced subset of what would be necessary for a practical strongly reactive complex event detection system. Featherweight PARTE offers only a restricted set of data types. This manifests itself by the rather spartan implementation of `Values`. The only supported types of `Values` are 64-bit signed integers and 64-bit signed IEEE floating-point numbers.

The only reaction logic supported by fwPARTE is the emission of newly created compound events, or the detection of a rule matching. As explained in section 6.1.7 PARTE$^{\mathrm{Rust}}$ actually supports registering callbacks that are invoked in response to the matching of a rule. Though this behavior goes beyond what the formal model prescribes, it is useful for the prototype. It is self-evident that using this functionality to bypass the model by, e.g., emitting events from the callback in a way that is not compatible with prescribed event source semantics, undoes the guarantees offered by the fwPARTE model.

Distributed failure handling is absent from PARTE$^{\mathrm{Rust}}$ for the simple reason that PARTE$^{\mathrm{Rust}}$ runs as a single process, which means that partial hardware failures cannot occur. PARTE$^{\mathrm{Elixir}}$ does potentially run on multiple hardware nodes, opening the option for handling failure of a subset of them. However, since the Featherweight PARTE model does not define how to handle partial failure in a strongly reactive manner, PARTE$^{\mathrm{Elixir}}$ lacks failure handling mechanisms. More accurately, existing mechanisms shared by the other distributed Rete-based systems developed at the lab were removed from the code base of PARTE$^{\mathrm{Elixir}}$ since they could not guarantee strong reactivity.

Section 5.6 ended with an enumeration of optimization opportunities which are missing from the fwPARTE model, but which an implementation would likely apply. PARTE$^{\mathrm{Rust}}$ is such an implementation, and can hence apply some of those optimizations. We revisit each element from the original list here. It should be noted that all these optimizations can at most offer a constant factor speedup, since the baseline algorithm already runs in constant time.

Indexes in partial match histories As shown by listing A.4, join-nodes abstract over the lists of partial matches they have received before: their left partial match history is no $\mathcal{P}(\mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{E}})$ as the formal model proposes, but some opaque type `LeftPartialMatchStorage`. Similarly, the right partial match history is no $\mathcal{P}(\mathbf{T})$, but some opaque type `RightPartialMatchStorage`. These partial match storages store a representation of the join-conditions of the join-node (or not-node). This enables them to handle storage and lookup of matches in a more intelligent way. As discussed in sections 5.6 and 6.1.7, one optimization this enables, is to keep the partial matches sorted by lifetime, which reduces the cost of looking up potential matches based on the temporal join-condition. We have not managed to demonstrate additional gains from indexing the data to speed up matching based on the non-temporal join-conditions.

Above-minimum storage sizes The model specifies that inboxes, outboxes, and partial match histories should be at their minimum viable sizes. In practice, any constant size greater than or equal to the minimum viable size can be used, as long as the implementation takes this into account.

When using an inbox at its minimum viable size, back-pressure can prevent overloading the system while still guaranteeing correctness, but chances are that the system spends a lot of its run time in a state where all the concurrent components are running in lock-step: all components wait for the slowest component to finish, then perform another step and all wait for whichever component is slowest this time, and so on.

Throughput can be improved by giving the different components some leeway, in the form of an above-minimum sized buffer. That way, components only have to wait if a certain component consistently was the slowest for multiple activations in a row. Such optimizations are ubiquitous in distributed systems, to the point where it could even be considered the default. However, the non-determinism in event arrival and in scheduling mean that this optimization does not improve worst-case performance by even a constant factor. It hence has little value in the context of this dissertation.

Batching of multiple matches This optimization is dependent on the one of the previous item, and offers the same style of advantages and disadvantages. Propagating an entire batch of matches in a single message obviously decreases the overhead, but does not improve the worst-case scenario critical to PARTE's strong reactivity. Since batching up partial matches makes it more difficult to satisfy the constraints on inbox sizes, we did not implement this optimization in PARTE$^{\text{Rust}}$.

Coarsening the granularity in the Featherweight PARTE graph Consider a PARTElang program which contains the pattern `incoming : MoneyTransferred {}`. The pattern specifies no alpha conditions. Consider the case where this is the only pattern in the LRP program subscribed to `MoneyTransferred` events.

When such a program is executed on our prototype, PARTE$^{\text{Rust}}$ will construct an entry node for this pattern. This entry node's only job is to 'merge' a single timestamp. Instead, the node will maintain a hashmap (of size one), do lookups in it, construct new alpha tokens (which are identical to the one it received), and propagate it to its singular successor using asynchronous message passing.

The successor — an alpha node — will serve this alpha token from its inbox, verify whether some always-true predicate holds for the token's event, construct a beta-token wrapping that single event, and propagate it to its successors.

The ratio of overhead to useful work is huge here. In this specific case, an improved system would omit the useless entry node, and propagate `MoneyTransferred` events straight to the alpha node. That reduces the overhead somewhat, but still leaves room for improvement. The alpha node cannot however be removed, as it is responsible for converting the alpha token to a beta token. Still, in the absence of an alpha condition, there is little point in executing the alpha node's work concurrently: the advantage of possible parallel computation does not weigh up to the overhead. The overhead could be reduced by making the PARTE graph more coarse by merging multiple conceptual nodes into a single, multi-purpose node. A two-input node of which a predecessor is an alpha-node without an alpha condition, could be specialized to receive alpha tokens, and do the — essentially no-op — conversion itself.

In a similar vein, more advanced analysis of the load on nodes might enable merging multiple successive join- and/or not-nodes. Alternatively, if a specific join-node or not-node is revealed to be exceptionally computationally expensive, it might be feasible to split of the verification of some join-constraints into a separate node. Such a modification would depend on a strong analysis of the pattern, as delaying filtering based on join-criteria yields a larger set of intermediate results, which has a high chance of inadvertently increasing the load.

Limiting $\perp$-tokens There is no need to pass "no change" tokens on paths through the graph that do not lead to a negation. PARTE$^{\text{Rust}}$ statically checks which nodes are on an (indirect) path to a negation. Nodes which are not, ignore "no change" tokens. It might seem from listing 6.8 that alpha nodes are an exception to this. This is not the case: since they have a single predecessor entry node which is on a path to a negation if and only if the alpha node is, entry nodes ensure that alpha nodes only receive "no change" tokens when they are on a path to a negation. An explicit check in `AlphaNode` is hence redundant, thus omitted.

6.4 Conclusion

We developed two prototypical implementations of the Featherweight PARTE model. One mimics PARTE's distributed memory space on a single machine by implementing nodes as concurrent coroutines which exchange tokens via asynchronous message passing. The second implementation is truly distributed across multiple machines. This ability to distribute workload across multiple computers enables PARTE to achieve the requirement of being scalable ($\mathbf{R_{LRP}\,4}$).

In the next chapter we make use of these implementations. We implement the three driver scenarios in PARTElang, and execute them on top of our prototypical implementations. Our implementations thus enable us to experimentally validate our formal model, and compare it to the state of the art.

7 Experimental Validation

The previous chapter describes two prototypical implementations of the Featherweight PARTE model. In this chapter we show how the 3 driver scenarios introduced in section 2.1 can be expressed as a PARTElang program. To this end, we first explore the driver scenarios in detail (section 7.1). Next, we translate them into PARTElang syntax (section 7.2).

We further describe the challenges in expressing those PARTElang programs using the mature event processing systems described in chapters 2 and 3 (section 7.3). The state of the art either cannot express a limit on the size of the state that is retained — a requirement for strong reactivity — or does so at the cost of missing potential matches. We quantify the latter cost experimentally (section 7.4) to show that the PARTE model offers a measurable benefit over the state of the art.

7.1 Revisiting the Driver Scenarios

We listed 3 driver scenarios in section 2.1 (on page 12). Each of them expressed — in plain English text — a complex event pattern which needed to be detected in event streams that describe financial transactions. The patterns specified *filter conditions* (e.g., "the amount must be at least 100 €"), *joins* (by describing multiple events), and *anti-joins* (e.g., "there was no other transaction"). The joins and anti-joins specified semantic windows (e.g., "2 weeks") and join-conditions (e.g., "the same amount, with a 10% margin").

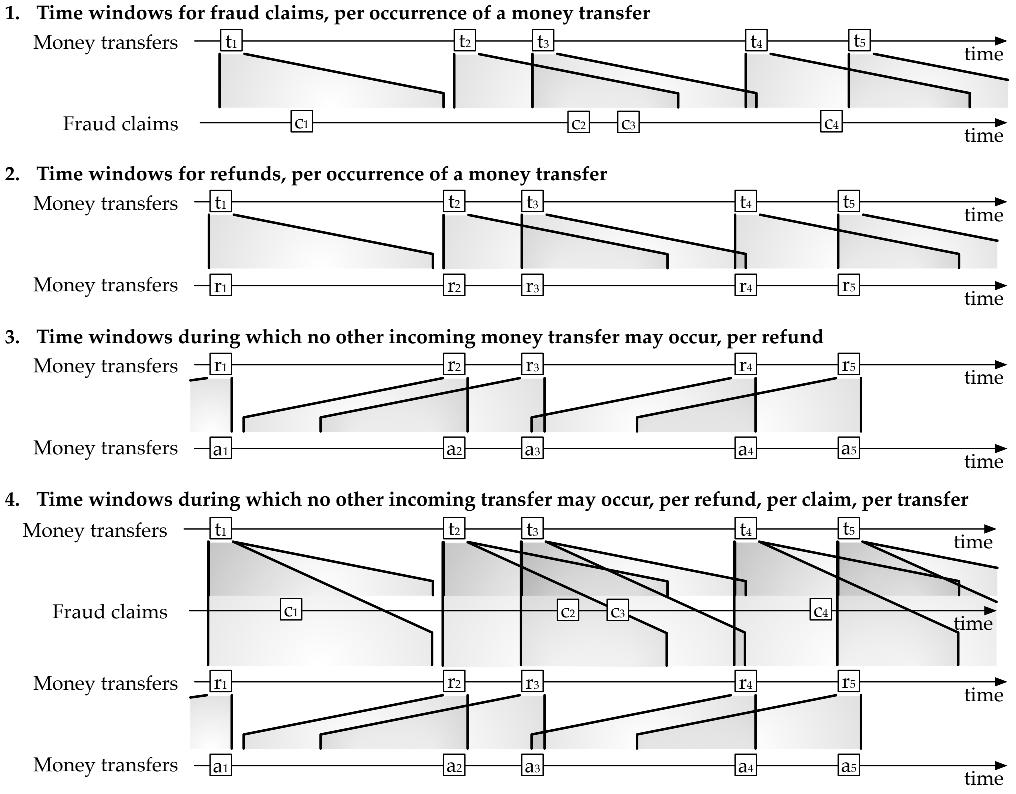

1. Time windows for fraud claims, per occurrence of a money transfer
Money transfers
t1
t2
t3
t4
t5
time
Fraud claims
c1
c2
c3
c4
time
2. Time windows for refunds, per occurrence of a money transfer
Money transfers
t1
t2
t3
t4
t5
time
Money transfers
r1
r2
r3
r4
r5
time
3. Time windows during which no other incoming money transfer may occur, per refund
Money transfers
r1
r2
r3
r4
r5
time
Money transfers
a1
a2
a3
a4
a5
time
4. Time windows during which no other incoming transfer may occur, per refund, per claim, per transfer
Money transfers
t1
t2
t3
t4
t5
time
Fraud claims
c1
c2
c3
c4
time
Money transfers
r1
r2
r3
r4
r5
time
Money transfers
a1
a2
a3
a4
a5
time

Figure 7.1: Events matching part of driver scenario C give rise to semantic windows over the other events — White squares depict event occurrences of the corresponding type. Shaded hexagons depict the extent of a semantic time window from the event on the top, mapped onto the stream below them.

7.1.1 Visualizing the Temporal Aspect of the Driver Scenarios

To better understand how the temporal aspects of the driver scenarios induce time windows on the input event streams, we adapt figure 3.3 from page 59 to depict the situation for driver scenario C, namely the refund-scam. This results in figure 7.1. Two aspects of this figure warrant special attention. First, note that all time windows have the same length. The patterns specify time windows of 2 weeks for any 2 events. Second, note that the same stream of money transfers is used as the source of 3 kinds of transfers:

- as incoming transactions, which are labeled t_i (for $1 \leq i \leq 5$);
- as refund transactions, which as labeled r_i; and
- as another transaction (which the pattern expects to be absent), which is labeled a_i.

Since multiple patterns must match the same stream, the patterns must not *consume* events. This is the case for PARTElang. However, conversely this means that we must, e.g., specify in the pattern that the event matching the refund pattern must be distinct from the event matching the "another refund" pattern. For the case of, e.g., the incoming and refund events, no additional work is required, since a money transfer event cannot be its own refund: a refund's source and destination are the opposite of those of the incoming transaction, and a transaction's destination cannot be the same account as its originator.

Continuing with figure 7.1, part 1 is straightforward: driver scenario C specifies the occurrence of a fraud claim within 2 weeks after a certain money transfer. For each money transfer, there is hence a time window of 2 weeks during which fraud claims can be received. Part 2 of figure 7.1 depicts how the same 2-week time windows apply to the stream of refunds, which is the same stream as the initial transfers. Part 3 depicts the time windows during which the absence of another matching refund has to be ensured; the 2 weeks before each refund. As we mentioned in the context of figure 3.3, the real complexity arises when all time windows are combined. Since driver scenario C only specifies time windows of 2 weeks, all with respect to some money transfer, the combination is still reasonably simple. It is depicted in part 4 of figure 7.1. In short: for each incoming transaction (t_i), all refund and fraud claims of the next 2 weeks are selected. Before each refund thusly matched, a 2-week period is checked for possible other refunding transactions. If any are found, the complex event is ignored.

Of course, the filters and non-temporal join-constraints restrict which of the remaining matches are valid.

7.1.2 Understanding the Join Behavior of the Driver Scenarios

Next, we discuss how the driver scenarios map onto a graph of PARTE nodes. Section 5.5 on page 146 outlined abstractly how a PARTElang program gets compiled into a PARTE graph. We now anticipate what that process should produce for the driver scenarios. Since each of the driver scenarios describes a single complex event pattern, each scenario gives rise to one rule, and thus to one directed subgraph of PARTE nodes.

Driver scenario A describes a pattern of 2 money transfers. In order to match the pattern, both transfers have to satisfy some constraints with respect to the amount of money transferred. Both transfers have to be along an uncommon route. Additionally, the destination of one of the transfers has to be the originator of the other transfer.

This pattern gives rise to the 6 nodes on the left of figure 7.2: for both the incoming ("in") and outgoing ("out") money transfer, an entry node is generated. These nodes merge the event streams of all event sources generating `MoneyTransferred` events. An alpha test node is generated for both the incoming and the outgoing transfers. These nodes verify the aforementioned constraints on the amount of money, and on taking place via an uncommon route. The results of both alpha test nodes are joined. The join conditions are verified as part of the joining process.

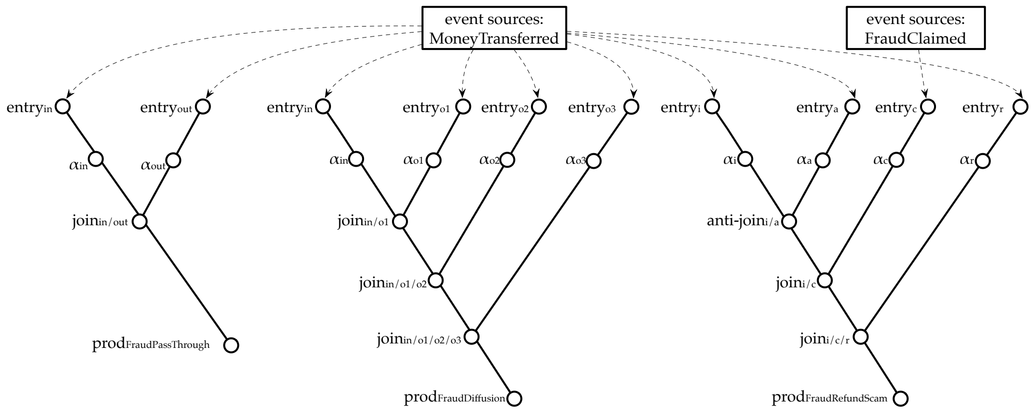


Figure 7.2: The PARTE graph for all 3 driver scenarios — Circles depict PARTE graph nodes. Lines depict predecessor-successor relations. Nodes higher up are the predecessors. All labels apply to nodes.

Finally, the results of the join node are propagated to a production node. When activated with a partial match, this node produces a new event summarizing the fraudulent pattern.

Driver scenario B describes a pattern of 4 money transfers. All 4 have to satisfy some constraints, e.g., that the transfer happened along an uncommon route. Additionally, the 4 transfers have to be joined, satisfying some join conditions.

This pattern gives rise to the 12 nodes in the center of figure 7.2: for all 4 subpatterns, an entry node and a test node are generated. Then, the incoming ("in") transfer is joined with the first outgoing ("o1") transfer by means of a join node. The second outgoing ("o2") money transfer is joined next, using another join node. Similarly, the third outgoing ("o3") money transfer is joined. Finally, the results arrive at a production node, which emits a new event summarizing the fraudulent pattern.

Driver scenario C describes a pattern of 2 money transfers, a fraud claim, and the absence of a fourth. Again, all 4 patterns enforce some constraints. The first 2 transfers have to be joined, satisfying some join conditions. The fraud claim has to be joined to the results. The last transfer has to be anti-joined onto the results of that step. The temporal aspects necessarily must be taken into account here: the absence of the last transfer must hold for the 2-week period after the first event.

This pattern gives rise to the 12 nodes on the right of figure 7.2: for all 4 subpatterns, an entry node and a test node are again generated. The incoming ("i") transfer is joined with the refund ("r") transfer by means of a join node. The fraud claim ("c") is joined with the results by means of another join node. Next, the absence of another ("a") refunding money transfer is enforced by a not-node, i.e., an anti-join. As far as the overall structure of the graph is concerned, joining and anti-joining are identical. Finally, a production node is provided to emit a new event summarizing the fraudulent pattern.

timestamp	id	originator	destination	amount
2018-01-01 08:00:00	1	AAA-AAA-AAA	BBB-BBB-BBB	453 €
2018-01-01 08:00:00	2	CCC-CCC-CCC	DDD-DDD-DDD	120 €
2018-01-01 08:00:05	3	EEE-EEE-EEE	FFF-FFF-FFF	1254 €
2018-01-01 08:00:10	4	GGG-GGG-GGG	HHH-HHH-HHH	320 €
2018-01-01 08:00:53	5	III-III-III	JJJ-JJJ-JJJ	350 €
2018-01-01 08:01:00	6	HHH-HHH-HHH	KKK-KKK-KKK	105 €
2018-01-01 08:01:30	7	EEE-EEE-EEE	FFF-FFF-FFF	1240 €
2018-01-01 08:02:00	8	JJJ-JJJ-JJJ	LLL-LLL-LLL	120 €
2018-01-01 08:02:15	9	HHH-HHH-HHH	MMM-MMM-MMM	105 €
2018-01-01 08:04:00	10	DDD-DDD-DDD	NNN-NNN-NNN	120 €
2018-01-01 08:05:00	11	OOO-OOO-OOO	PPP-PPP-PPP	320 €
2018-01-01 08:06:00	12	JJJ-JJJ-JJJ	QQQ-QQQ-QQQ	120 €
2018-01-01 08:07:00	13	RRR-RRR-RRR	SSS-SSS-SSS	310 €
...				
2018-01-01 09:00:00	200	HHH-HHH-HHH	TTT-TTT-TTT	100 €
2018-01-01 09:00:01	201	SSS-SSS-SSS	RRR-RRR-RRR	310 €
2018-01-01 09:00:20	202	UUU-UUU-UUU	VVV-VVV-VVV	64 €
...				
2018-01-02 12:00:00	5003	SSS-SSS-SSS	RRR-RRR-RRR	310 €
2018-01-02 12:00:05	5004	FFF-FFF-FFF	WWW-WWW-WWW	1240 €
2018-01-02 12:00:07	5005	OOO-OOO-OOO	JJJ-JJJ-JJJ	200 €
...				
2018-01-17 08:00:00	37001	JJJ-JJJ-JJJ	XXX-XXX-XXX	120 €
2018-01-17 08:01:00	37002	YYY-YYY-YYY	ZZZ-ZZZ-ZZZ	410 €

Figure 7.3: A concrete trace of `MoneyTransferred` events — The excerpt contains one instance of driver scenario A, one instance of driver scenario B, and one instance of driver scenario C if a fraud claim contesting transaction 13 would occur on 2018-01-01 at 14:00:00.

7.1.3 A Concrete Event Trace

Before we translate the patterns into PARTElang, we briefly sketch how the event patterns of the driver scenarios are supposed to behave on a concrete event stream.

Consider the trace of `MoneyTransferred` events in figure 7.3. The table lists the timestamps at which the transfers occurred, the identifier of the transaction, the originator and destination accounts, and the amount transferred. For instance, transaction 1 occurs January 1st, 2018, at 08:00, and transfers 453 € from account `AAA-AAA-AAA` to `BBB-BBB-BBB`. For simplicity, we will write "account A" when referring to the account whose identifier is `AAA-AAA-AAA`, "account B" when the identifier is `BBB-BBB-BBB`, and so on.

For driver scenario A and driver scenario B, the stream of money transfers is all that is needed. Driver scenario C additionally requires a stream of fraud claims. For the sake of this example, consider a single fraud claim contesting transaction 13. The fraud claim occurs on 2018-01-01, at 14:00:00.

With that setup, the following patterns occur in the event stream:

- Transaction 1 is a candidate for the incoming transactions of all 3 driver scenarios: it is a transaction of more than 100 €. However, no other transactions involving accounts A and B occur in the stream, so the occurrence of transaction 1 will eventually expire without producing a match.

- Transaction 2 similarly is a valid candidate for the incoming transactions of all 3 driver scenarios. Four minutes later, transaction 10 occurs, which transfers the same amount of money away from account D as what was transferred to account D by transaction 2. If accounts C and D, and the accounts D and N do not customarily transfer money to each other, transactions 2 and 10 match driver scenario A.

- Similarly, transaction 4 can be an incoming transaction for driver scenario B, with transactions 6, 9, and 200 being the outgoing transactions. Of the 320 € sent to account H in transaction 4, 310 € (105 € + 105 € + 100 €) was diffused to accounts K, M, and T. This is within the 10 percent margin. Of course, this is only a valid match if the involved accounts do not customarily transfer money to each other.

- Transaction 7 transfers 1240 € to account F. Transaction 5004 transfers 1240 € out of account F, some 28 hours later. This pattern would match driver scenario A, were it not for transaction 3. Because of transaction 3, transfers from account E to F are considered common, and therefore not suspicious.

- Transaction 13 transfers 310 € from account R to account S. By means of transaction 201, the owner of account S manually reimburses the 310 € to account R on 2018-01-01 at 09:00:01. However — as mentioned above — transaction 13 is disputed by a fraud claim on 2018-01-01, at 14:00:00. This causes the bank to automatically revert the transaction (on 2018-01-02 at 12:00:00, by means of transaction 5003). This pattern matches driver scenario C, unless there was another transfer for approximately 310 € from R to S, in which case both refunds could be valid.

- Finally, transaction 5 transfers 350 € from account I to account J. Transfers 8, 12, and 37001 transfer 360 € (3 times 120 €) out of account J. Assuming these accounts do not customarily transfer money to each other, this would be a match for driver scenario B, were it not that transfer 37001 occurred more than 2 weeks after transfer 5. The last transfer occurs outside of the time window, and should hence not be detected.

7.2 Expressing the Driver Scenarios in PARTElang

We now express all 3 driver scenarios introduced in section 2.1 on page 12 as a PARTElang program.[1] The overall structure of the program is depicted in figure 7.2 on page 186.

Driver scenario A: Detecting Pass-Through of Money

In its most straightforward form, driver scenario A deals with only two event types: first, as inputs it handles money transfers. Second, as output it produces an event representing a suspicious sequence of transactions. The PARTElang program for driver scenario A hence requires only two template definitions, depicted in listing 7.1.

```
template MoneyTransferred {
  id          : TransactionId,
  originator  : AccountId,
  destination : AccountId,
  amount      : Amount,
}

template SuspiciousSequenceDetected {
  mule_account          : AccountId,
  amount                : Amount,
  incoming_transactions : [AccountId; 3],
  outgoing_transactions : [AccountId; 3],
}
```

Listing 7.1: The event templates required to express the detection of possible mule accounts, used in driver scenario A and driver scenario B, expressed in PARTElang.

The template `MoneyTransferred` introduces a relation name for money transfer events which have an identifier, an originator account, a destination account, and an amount of money. The template `SuspiciousSequenceDetected` introduces a relation name for events abstracting over the detection of suspicious sequences, listing the suspected mule account, the amount of money transferred in the suspicious sequence, and the transactions involved. The base types `TransactionId`, `AccountId`, `Amount`, and the list of up to 3 `AccountId`s, `[AccountId; 3]`, are assumed to be built-in for the sake of this example.

Driver scenario A, i.e., the potentially fraudulent pass-through of money, can be captured by a single rule, listed in listing 7.2. This rule — named `FraudPassThrough` — describes a pattern where money is transferred twice: one incoming transaction named `incoming`, and one outgoing transaction named `outgoing`. The incoming transaction is filtered by the condition that the `amount` of money transferred is at least 100.0 €. Both the incoming and outgoing transactions are filtered by the condition `uncommon_route(originator, destination)`.

[1]The complete PARTElang implementation of the driver scenarios is available at `https://soft.vub.ac.be/~trenaux/PARTE/implemented-driver-scenarios.zip`.

```
rule FraudPassThrough where
  incoming : MoneyTransferred {
    amount >= 100.0,
    uncommon_route(originator, destination)
  }
  outgoing : MoneyTransferred {
    uncommon_route(originator, destination),
    originator = incoming.destination,
    amount = incoming.amount
  }
when
  outgoing in incoming [0 days, 14 days]
then
  emit SuspiciousSequenceDetected
    at incoming.timestamp
    with {
      mule_account          = incoming.destination,
      amount                = incoming.amount,
      incoming_transactions = list(incoming.id),
      outgoing_transactions = list(outgoing.id)
    }
```

Listing 7.2: The PARTElang rule implementing driver scenario A: a suspicious incoming and a suspicious outgoing transaction are matched, and abstracted into a `SuspiciousSequenceDetected` compound event.

Note that `originator` and `destination` refer to the fields of the local event, i.e., to the `incoming` transaction in the first case, and to the `outgoing` transaction in the second. Assume in this example that `uncommon_route(a, b)` is built-in, and implements the domain logic behind uncommon sender/receiver pairs, e.g., that `a` never sent money to `b` before, or that `b` is not a registered recipient for `a`. The pattern binding `outgoing` imposes two more join-constraints: that the `originator` of the outgoing transaction matches the `destination` of the `incoming` transaction, and that the `amount` of both transactions is the same. Finally, the `when`-clause imposes the temporal join-condition that the `outgoing` transaction takes place within 2 weeks after the `incoming` transaction.

The `then`-clause of the rule abstracts the detected pattern into a new compound event of the type `SuspiciousSequenceDetected`. The `SuspiciousSequenceDetected` event inherits the timestamp from the `incoming` transaction. Other attributes are assigned based on the attributes of the matched transactions.

This rule, when applied to the event trace of figure 7.3 on page 187, would emit a `SuspiciousSequenceDetected` event whose timestamp is 2018-01-01 08:00:00 — the timestamp of transaction 2 — and whose `mule_account` attribute refers to account D. The `amount` would of course be 120 €, the `incoming_transactions` a singleton list of transaction id 2, and `outgoing_transactions` a singleton list of transaction id 10.

Driver scenario B: Diffusion of Money

Driver scenario B reuses the event template definitions from driver scenario A, defined in PARTElang in listing 7.1. The potentially fraudulent diffusion of money can be captured by

```
rule FraudDiffusion where
  incoming : MoneyTransferred {
    amount >= 100.0,
    uncommon_route(originator, destination)
  }
  outgoing_1 : MoneyTransferred {
    uncommon_route(originator, destination),
    originator = incoming.destination,
  }
  outgoing_2 : MoneyTransferred {
    uncommon_route(originator, destination),
    originator = incoming.destination,
    id != outgoing_1.id,
  }
  outgoing_3 : MoneyTransferred {
    uncommon_route(originator, destination),
    originator = incoming.destination,
    id != outgoing_1.id,
    id != outgoing_2.id,
    outgoing_1.amount + outgoing_2.amount + outgoing_3.amount >= incoming.amount * 0.9,
    outgoing_1.amount + outgoing_2.amount + outgoing_3.amount <= incoming.amount * 1.1,
  }
when
  outgoing_1 in incoming [0 days, 14 days]
  outgoing_2 in incoming [0 days, 14 days]
  outgoing_3 in incoming [0 days, 14 days]
then
  emit SuspiciousSequenceDetected
    at incoming.timestamp
    with {
      mule_account          = incoming.destination,
      amount                = incoming.amount,
      incoming_transactions = list(incoming.id),
      outgoing_transactions = list(outgoing_1.id, outgoing_2.id, outgoing_3.id)
    }
```

Listing 7.3: The PARTElang rule implementing driver scenario B: a suspicious incoming and 3 suspicious outgoing transactions are matched, and abstracted into a `SuspiciousSequenceDetected` compound event.

a single rule. This rule, named `FraudDiffusion`, can be found in listing 7.3. It captures a pattern involving 4 transactions: one incoming transfer, and 3 outgoing transfers. The events must satisfy the following conditions:

- The incoming `MoneyTransferred` event must transfer at least 100.0 €.
- All `MoneyTransferred` events must be on an `uncommon_route`.
- All outgoing transactions are restricted to transactions whose originator matches the destination of the incoming transaction.
- Distinctness between `outgoing_1`, `outgoing_2`, and `outgoing_3` [2] is established by requiring them to have a different `id`.

[2]Remember that PARTE has a *zero consume* event consumption policy (see section 3.4.1). This entails that a single event can be matched to multiple rules, but also that a single event can be matched to multiple sub-patterns in one rule.

- The pattern of `outgoing_3` enforces that the sum of the amounts of the outgoing transactions is within ten percent of the amount of the incoming transaction.
- The `when`-clause imposes the temporal constraint that the outgoing transactions takes place within 2 weeks after the `incoming` transaction.
- The `then`-clause of the rule abstracts the detected pattern into a new compound event of the type `SuspiciousSequenceDetected`. The new event inherits the timestamp from the `incoming` transaction. Other attributes of the new complex event are assigned based on the attributes of the matched transactions.

This rule, when applied to the event trace of figure 7.3 on page 187, would emit a `SuspiciousSequenceDetected` event whose timestamp is 2018-01-01 08:00:10 — the timestamp of transaction 4 — and whose `mule_account` attribute refers to account H. The `amount` would be 320 €: the amount of the `incoming` transaction. If we had wanted to list the amount of money transferred out of the mule account — in this case 310 € — we would have to replace the expression `incoming.amount` (assigned to the new event's `amount` attribute on line 32) with the expression `outgoing_1.amount + outgoing_2.amount + outgoing_3.amount`. The `incoming_transactions` attribute of the emitted event is of course a singleton list of transaction id 4. The `outgoing_transactions` attribute is the full 3-element list containing transaction identifiers 6, 9 and 200.

Reducing Code Duplication in Driver Scenarios A and B

The code in listings 7.2 and 7.3 shows that the common concern of the occurrence of "money being transferred, while being suspicious" leads to code duplication in the current solution. These events can be abstracted into events of a novel event template named for instance `MoneyTransferredSuspiciously`.[3] With the addition of this rule, the solutions for the first two driver scenarios can be adapted by no longer matching on occurrences of `MoneyTransferred`, but on occurrences of `MoneyTransferredSuspiciously`. The now-redundant pattern conditions involving the constant constraint on the `amount` and those involving the predicate `uncommon_route` can then be removed from the rules `FraudPassThrough` and `FraudDiffusion`.[4]

Driver scenario C: Detecting Refund Scam

A PARTElang implementation of driver scenario C is listed in listings 7.4 and 7.5 on pages 193 and 194. Two new event templates are defined: one to capture claims of fraudulent transactions, and one to abstract the detection of possible refund scam. The

[3]A rule that performs this abstraction is provided in the appendix, in listing A.1 on page 241.
[4]The resulting code is provided in the appendix in listing A.2 on page 242.

former template lists the affected transaction and the reason for the claim. The latter template lists the incoming transaction and the transaction through which it was refunded.

The rule `FraudRefundScam` specifies the reaction to a pattern involving 4 subpatterns. The first two are money transfers, where the second transaction returns a similar amount of money than what was sent in the first. The third subpattern concerns a `claim` of fraud on the transaction matched to the `incoming` event pattern. The fourth and final subpattern is a negated pattern requiring the absence of a third transaction for which the event matched to `refund` could be a valid refund. The `when`-clause imposes the temporal constraints we explored in section 7.1.1 and in figure 7.1 on page 184: the refund and fraud claim must take place within 2 weeks after the `incoming` transaction, and restrict the window during which no additional incoming transaction may take place to the 2 weeks before the refund.

The `then`-clause of this rule emits a new `RefundScamSuspected` compound event. This event inherits the timestamp from the `claim`. Other attributes are assigned based on the attributes of the matched transactions.

This rule, when applied to the event trace of figure 7.3 on page 187 and the fraud claim mentioned in section 7.1.3 on page 187, would emit a `RefundScamSuspected` event whose timestamp is 2018-01-01 08:07:00 — the timestamp of transaction 13. The newly emitted event's `incoming_transaction` would of course be transaction id 13, and its `refund_transaction` attribute would refer to transaction 201.

```
template FraudulentTransactionClaimed {
  transaction_id : TransactionId,
  reason         : String,
}

template RefundScamSuspected {
  incoming_transaction : TransactionId,
  refund_transaction   : TransactionId,
}
```

Listing 7.4: The event templates required to express the detection of possible refund scam, used in driver scenario C in addition to those from listing 7.1, expressed in PARTElang.

7.3 Shortcomings of the State of the Art Revisited

We have now implemented our driver scenarios in PARTElang. PARTElang programs are by design expressive event patterns with reaction logic (satisfying **R_{LRP}1** defined in section 4.1.5 on page 70), which relate events in time (**R_{LRP}2**). Because our implementation (from chapter 6) implements the Featherweight PARTE model, the programs can be matched to a live stream in a scalable (**R_{LRP}4**), strongly reactive fashion (**R_{LRP}5**). The event storage is automatically managed (**R_{LRP}3**).

```
rule FraudRefundScam where
  incoming : MoneyTransferred,
  refund : MoneyTransferred {
    destination = incoming.originator,
    originator = incoming.destination,
    amount >= incoming.amount * 0.9,
    amount <= incoming.amount * 1.1,
  },
  claim : FraudulentTransactionClaimed {
    transaction_id = incoming.id
  },
  no additional_incoming : MoneyTransferred {
    originator = incoming.originator,
    destination = incoming.destination,
    amount >= refund.amount * 0.9,
    amount <= refund.amount * 1.1,
  }
when
  refund in incoming [0 days, 14 days]
  claim in incoming [0 days, 14 days]
  no additional_incoming in refund [-14 days, 0 days]
then
  emit RefundScamSuspected
    at claim.timestamp
    with {
      incoming_transaction = incoming.id,
      refund_transaction   = refund.id,
    }
```

Listing 7.5: The PARTElang rule implementing driver scenario C: a transaction, its refund, a fraud claim on the first transaction, and the absence of another candidate for the refunded transaction are matched, and abstracted into a `RefundScamSuspected` compound event.

One might ask whether all these steps were necessary; whether it would not suffice to merely use the PARTE model to determine the minimum viable sizes (as defined by definitions 25 to 27 on page 127) of partial match storages, inboxes, and outboxes. Instead of using PARTE$^{\text{Rust}}$ or PARTE$^{\text{Elixir}}$, one could consider using the more mainstream technologies described in chapters 2 and 3 as the actual implementation platform. In this section, we demonstrate why that is not a straightforward option.

7.3.1 Issue: No Notion of Minimum Viable Sizes in the Model

We have explained the shortcomings of existing distributed Big Data and stream processing system and event handling systems throughout this dissertation: first in the description of the state of the art (in chapters 2 and 3), then by explaining which additional requirements an LRP language has (in section 4.1), then by highlighting relevant differences in the related work sections (in sections 4.6 and 5.7). At this point, we can summarize the shortcomings of the related work in a different way:

Insight

A major shortcoming of existing streaming event handling systems is that they have no notion of minimum viable sizes.

The set of minimum viable sizes is not a global property of an event pattern; it is a property of an event pattern and an evaluation model. In the LRP paradigm, any event pattern is guaranteed to define a set of finite, fixed minimum viable sizes, and evaluation is guaranteed to terminate successfully within those minimum viable sizes. This property does not hold for arbitrary non-LRP evaluation models.

Consider a hypothetical implementation of driver scenario A on top of a stream processing framework, e.g., Spark Streaming [146]. Let the minimum viable left partial match history size of the single join node in driver scenario A be $\mathit{size}_{\searrow}$. Let partial match histories be represented as windows containing a number of financial transactions, upper-bounded by the minimum viable size. Consider now a case where — due to the scheduling of different computations on different machines in the Spark cluster — the stream of incoming financial transactions processes $\mathit{size}_{\searrow} + 1$ elements, while the processing of the stream of outgoing financial transactions is idle. The first of those $\mathit{size}_{\searrow} + 1$ incoming financial transactions will be pushed out of the window *without ever being joined with outgoing transactions*, leading to potentially missed results.

The remainder of this section discusses how this problem comes about in existing distributed streaming event handling systems. Single-threaded event handling technologies which receive events from at most one source per event type, do not suffer from this shortcoming. However, they only do so because their execution involves no scheduling, and hence no source of nondeterminism. A hypothetical distributed version of those technologies — using state of the art distribution technologies — would suffer the same shortcomings, where either strong reactivity has to be sacrificed, or results are potentially missed.

7.3.2 Case Study: Expressing the Solution in Apache Flink

Apache Flink [8] is one of the Big Data stream processing frameworks discussed in section 2.3.3. As discussed in section 3.3, a CEP extension for Flink, FlinkCEP [9], exists. Flink therefore looks like a promising implementation technology for our driver scenarios. However, none of the approaches offered by Flink can guarantee strong reactivity, as the evaluation model does not support the notion of minimum viable sizes:

FlinkCEP Amongst the different approaches available in Flink, FlinkCEP offers the most expressive API for complex event processing. Sadly, FlinkCEP offers no way of statically limit the number of elements retained in its internal data structures, precluding guaranteed strong reactivity.

Flink's Streaming SQL The Streaming SQL-based option suffers the same shortcoming: its declarative language offers no way of specifying a maximum number of elements retained.

CountWindow In Flink, an upper limit can be placed on the number of elements retained in a window using either `CountWindow`s or *evictors*. The former specifies a tumbling window of a certain size. Once the window is full, a processing function is applied to all elements in the window. This is not the behavior required by our driver scenarios, though: the scenarios require that two streams are joined, and that new events from one stream are matched to a (sliding) window of the other stream. `CountWindow`s do not support joining.

Windowed Join with Evictor Joining two streams in Flink can be done in two ways: either as a *windowed join*, or as an *interval join*. The former entails that two streams are joined on a shared key. In driver scenario B that could take the form of pairwise joining incoming money transfers (keyed by `destination`) to 3 streams of outgoing money transfers (keyed by `destination`).

Windowed joins can use tumbling windows, sliding windows, session windows, or manually triggered windows. CountWindows are not supported for windowed joins, though the addition of an explicit `CountEvictor` can limit the number of elements retained in the time window.

For our driver scenarios, sliding windows are needed. However, Flink's windowed joins with sliding windows do not give rise to the behavior needed for our driver scenarios: the processing function applied to the window takes pairs of elements: the function is called for each pair of elements which is still in the window. What is required, instead, is an API for joining one new arrival with each element of the other stream which is still in the window. As it is, Flink's sliding windows either duplicate matches (when the `slide` parameter is smaller than the window size), or effectively become tumbling windows, thereby missing matches which span multiple windows. Any approach to filter out the duplicates necessarily requires manually managing another window's worth of recent elements, defeating the point of using Flink's windows in the first place.

Interval Join The final approach available in Flink is using *interval joins*. Interval joins are a form of semantic window, where events of one stream are joined with events of another stream based on a time window with respect to individual events on the first stream. The API is fixed, and takes the form `s1.intervalJoin(s2).between( t_min, t_max).process(f)`, where `t_min` and `t_max` are durations. The data structure used to store the events is not limited in size, and not accessible to the programmer: The processing function `f` gets no access to any sort of window data structure, only to both elements' data. For interval joins, too, it is hence not possible to impose a limit on the number of events that is retained in the window, precluding strong reactivity.

7.3.3 Case Study: Expressing the Solution in Spark Streaming

Spark Streaming [146] is another Big Data stream processing frameworks discussed in section 2.3.3. It is less opinionated than Flink, requiring more manual work but offering more flexibility. Still, when using the idiomatic approach it is impossible to limit the state size without potentially missing some matches.

Using Idiomatic Spark Streaming

Consider implementing driver scenario A in Spark Streaming. Idiomatically, joining the stream of incoming financial transactions with the stream of outgoing financial transactions, could be done by using Spark Streaming's `join` operator, joining incoming transactions keyed by `destination` to outgoing transactions keyed by `originator`. To impose an upper limit on the number of elements retained, one cannot use Spark Streaming's `window` operator, as the windows it creates are time-based, not size-based. Additionally, it creates one window for both joined sides, which is still wrong API, as what is needed are two windows; one for each side to be joined. Worse, limiting the size of a window does not suffice here, as the windows are per key, but arbitrarily many keys can exist.

Using Manual Windowing on Top of Spark Streaming

One could bypass these problems by manually implementing the join behavior, essentially reimplementing parts of PARTE on top of Spark Streaming. To this end, one can create a Spark RDD consisting of a tuple containing both windows of elements which may still take part in the joining process — essentially partial match histories. In Scala syntax, the initial Spark RDD could be created as follows:

```
val initialRDD : RDD[(List[List[TransferData]], List[TransferData])]
  = ssc.sparkContext.parallelize(List((List(), List())))
```

Listing 7.6: Initial state, i.e., partial match histories, used in the manual implementation of driver scenario B in Spark Streaming

Driver scenario B could then be implemented [5] by

- reading the money transfers from a socket (as a `DStream[String]`);
- parsing them into instances of the custom `TransferData` class (resulting in a `DStream[TransferData]`;

[5]The actual Scala code implementing this, is available at `https://soft.vub.ac.be/~trenaux/PARTE/evaluation.zip`.

- creating 4 instances of this stream: one stream filtered by the amount and by whether the transfer is on an uncommon route (yielding the stream of incoming money transfers), and 3 streams filtered by whether the transfer is on an uncommon route (yielding the 3 streams of outgoing money transfers);

- assigning an arbitrary, but consistent key to all elements, and some indication of whether this 'token' left-activates or right activates. The incoming stream left-activates, the 3 outgoing streams right-activate (refer back to figure 7.2 on page 186). All 4 streams hence have a type like `DStream[(Int, (ActivationSide, List[TransferData]))]`.

- These streams could then be joined by taking the union of the stream of incoming money transfers and the first stream of outgoing money transfers, then applying `mapWithState`, mapping a function which stores new elements in the correct partial match history, drops old elements past the minimum viable size of the partial match history, checks all events in the opposite partial match history for a match (taking into account the join constraints, namely that `incoming.dest == outgoing1.orig` and that `outgoing1.timestamp >= incoming.timestamp` as well as that `outgoing1.timestamp <= incoming.timestamp + window_size`), then calling `update` on the RDD-state with updated new partial match histories, and yielding the successful matches, prefixed by an `ActivationSide` and the key;

- taking the union of the stream produced above and the second stream of outgoing money transfers, then again applying `mapWithState`, mapping another custom patching procedure updating the RDD-state; and finally

- taking the union of the stream produced above and the third stream of outgoing money transfers, then again applying `mapWithState`, mapping another custom patching procedure updating the RDD-state.

This approach works in principle, but still suffers from the issue stated in section 7.3.1: the evaluation model has no notion of the minimum viable size, and thus does not block execution along one path to allow another path to catch up. Models like PARTE can apply *backpressure* when one side of a join would overshoot its minimum viable space. Models like the one used by Spark Streaming do not apply *backpressure* in those cases. This makes sense: absent the formal model of PARTE, there is no indication that processing data from one side of the join will inevitably free up space at the other side before both sides are full. For general-purpose stream processing libraries, blocking one side of a join because its window is full, could easily lead to deadlocks.

A result of this lack of in-model support for minimum viable sizes, is that possible matches are not detected by this implementation. We quantify this in the next section.

7.4 Quantifying the Problems with the State of the Art

In the previous section we concluded that expressing a strongly reactive implementation of the driver scenarios is not possible in the state of the art, as every solution either fails to guarantee evaluation in constant space (precluding constant-time evaluation), or risks missing matches to patterns by evicting non-stale data from its windows. We now quantify the latter case.

7.4.1 Experimental Setup

To quantify risks of the latter case, we execute both PARTE$^{\text{Rust}}$ and the Spark Streaming implementation described in last section on generated stream of money transfers. For PARTE$^{\text{Rust}}$ we allocate the minimum viable size for all its inboxes, outboxes, and partial match histories, as determined by the Featherweight PARTE model. Our experiments show that the Spark Streaming implementation, when using partial match histories of the same size, misses some results. The number of missed results decreases as the partial match history size increases, until the point where the Spark Streaming implementation also detects all matches.

Generation of Event Traces Containing Matches to the Patterns

We randomly generate a number of scenarios, in the form of event traces. Each trace contains a number of `MoneyTransferred` events. Some sets of these events jointly match the pattern specified by driver scenario B, other events do not, and act as noise. We only generate valid matches for driver scenario B; as explained in section 7.1.2, driver scenario A is merely a simplified version of driver scenario B, and driver scenario C displays a very similar join behavior.

To reduce the time it takes to execute the test runs, as well as to overcome the issues outlined in section 5.6.1, the time window specified by the pattern is lowered from 14 days to 14 milliseconds. The event arrival rate of `MoneyTransferred` events is in turn raised to 10 per millisecond, spread over two event sources. The partial match arrival rates and minimum viable partial match history sizes are depicted in figure 7.4.

The different generated traces differ in the following ways:

Number of 3-way splits: the number of sets of 4 events which jointly constitute a 3-way split of money within the time window;

Number of unrelated events: the number of events which are not part of a 3-way split; and

Run: for each number of 3-way splits and unrelated events, multiple "runs" are generated. Each "run" produces a different random assignment of timestamps to the events.

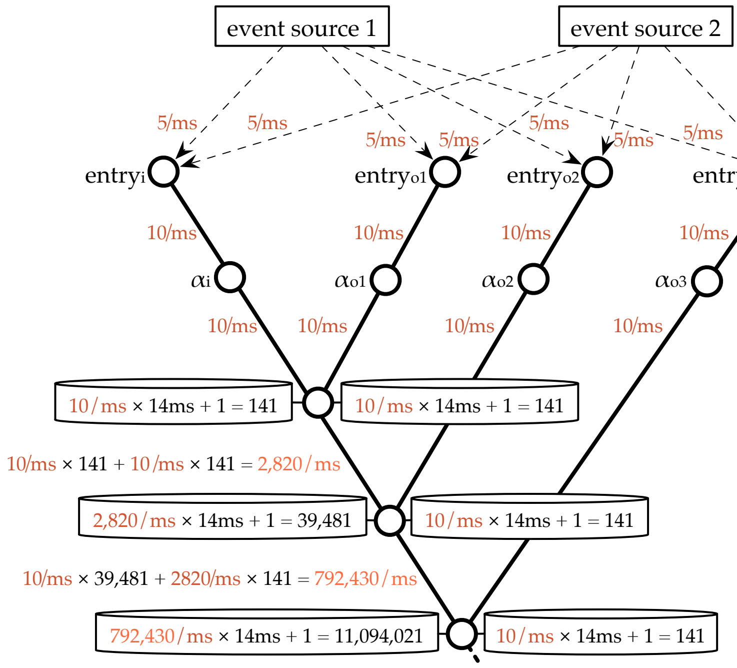

event source 1
event source 2
5/ms
entryi
entryo1
entryo2
entryo3
10/ms
αi
αo1
αo2
αo3
10/ms × 14ms + 1 = 141
10/ms × 141 + 10/ms × 141 = 2,820/ms
2,820/ms × 14ms + 1 = 39,481
10/ms × 39,481 + 2820/ms × 141 = 792,430/ms
792,430/ms × 14ms + 1 = 11,094,021

Figure 7.4: PARTE graph of driver scenario B, with minimum viable partial match history sizes — The partial match arrival rates — which determine the minimum viable partial match history sizes lower in the graph — are depicted in orange.

```
for run_id in 0..100 {
  for num_diffusiony in &[10, 100, 1000] {
    let num_on_common_route = num_diffusiony / 4;
    let num_below_threshold = num_diffusiony / 4;
    let num_full_diffusions = num_diffusiony - num_on_common_route - num_below_threshold;
    for &num_unrelated in &[0, 1, 10, 100, 1000, 10000] {
      let events = generate_events_with_diffusion(num_full_diffusions,
                                                   num_on_common_route,
                                                   num_below_threshold,
                                                   num_unrelated,
                                                   &template_transferred,
                                                   last_start_time);

      let name = format!("d={}-u={}-r={}.txt", num_diffusiony, num_unrelated, run_id);
      let events_and_sources = assign_event_source_to_event(
        &name, num_primitive_events_sources, events.into_iter());
    }
  }
}
```

Listing 7.7: Rust code driving the event trace generation for the evaluation in section 7.4

For completeness, we reproduce the code of the event trace generator in this chapter.

```
 1 use rand::distributions::{Bernoulli, Distribution, Uniform};
 2 let distribution_of_ce_start = Uniform::from(0..last_start_time);
 3 let mut rng = rand::thread_rng(); // Pseudo-random number generator
 4 let mut all_events = BTreeMap::new();
 5 let mut next_id = 0;
 6
 7 for i in 0..num_full_diffusions {
 8   let ce_start_time = distribution_of_ce_start.sample(&mut rng);
 9
10   while let Err(()) = add_new_diffusion_group(&mut next_id,
11                                               ce_start_time,
12                                               true, // is_above_threshold
13                                               false, // is_common_route
14                                               &template_transferred,
15                                               &mut rng,
16                                               &mut all_events);
17 }
18
19 for i in 0..num_common_route { /* ... similarly ... */ }
20
21 for i in 0..num_below_threshold { /* ... similarly ... */ }
22
23 let distribution_of_is_above_threshold = Bernoulli::new(0.5);
24 let distribution_of_is_common_route = Bernoulli::new(0.9);
25 for i in 0..num_unrelated {
26   let ce_start_time = distribution_of_ce_start.sample(&mut rng);
27
28   while let Err(()) = add_new_unrelated_event(
29     &mut next_id,
30     ce_start_time,
31     distribution_of_is_above_threshold.sample(&mut rng),
32     distribution_of_is_common_route.sample(&mut rng),
33     &template_transferred,
34     &mut rng,
35     &mut all_events);
36 }
```

Listing 7.8: Rust code generating the event trace given a configuration

The main driver of the event trace generator is depicted in listing 7.7. In the configuration shown, it prepares 100 different runs (line 1) of each scenario. Of each scenario, a version exists generating 10, 100, and 1000 (line 2) 3-way splits. As shown on lines 3 to 5, the number of 3-way splits is divided into 3 parts: the first quarter (rounded down) form non-fraudulent diffusions of money, which happen along common routes. The next quarter (rounded down) form non-fraudulent diffusions of money, which involve less than 100 €. The remaining splits are generated to satisfy the pattern of a fraudulent diffusion of money. Line 6 cycles through the different number of unrelated events to generate. Finally, the concrete events are generated (line 7), the file name for the benchmark is produced, and the events are assigned to event sources.

The code generating the event traces is depicted in listing 7.8. Depicted from line 7 onwards, the code produces some "full diffusions", i.e. groups of `MoneyTransferred` events which fully satisfy the constraints of a fraudulent diffusion of money. For each group, a start time is randomly selected (on line 8) by sampling a uniform distribution (defined on line 2). The actual creation of the diffusion group is performed by the procedure `add_new_diffusion_group` (called on line 10, and defined in listing A.6

on page 246). If that procedure would generate too many events for a certain time span, thereby surpassing the maximum event arrival rate, the procedure returns an error instead. If such an error is returned, the event trace generator simply calls the procedure again.

After creating the “full diffusions”, diffusions along a common route and diffusions involving less money than the 100 € threshold are generated using a similar loop. Finally, the unrelated events are generated. The call at line 29 adds a single unrelated event to the set of `all_events`, unless the randomly generated event would surpass the maximum event arrival rate, in which case another unrelated event is generated. The probability that an unrelated event is along a common route, or that the event is above the 100 € threshold, are respectively 90% and 50%, sampled from Bernoulli distributions (defined on lines 23 and 24).

For the generation of the actual events, consider listing 7.9 (and listing A.6, provided in the appendix on page 246). For a diffusion group, an incoming `MoneyTransferred` event is randomly generated at a timestamp chosen uniformly within the duration of the scenario. Then, 3 matching outgoing money transfers are randomly generated, uniformly distributed within the semantic time window after the incoming transfer. The generator ensures that the union of the preexisting set of events and the 4 new events is below the maximum event rate, and either adds the new events to the set, or returns an error.

The case for the unrelated events is similar, but simpler. As depicted in listing 7.9, depending on the configuration a random value for the amount of money transferred is generated (on line 1), and the originator and destination of the transfer is determined (on line 2). If the emission of the event would surpass the maximum event rate, the sample is rejected (by returning an error on line 5). Otherwise, the event is constructed and added to the set of events. The unrelated `MoneyTransferred` events operate on an entirely separate set of account identifiers, to ensure they do not accidentally contribute to a 3-way split. However, the pattern matchers are not aware of this distinction, and they cannot discard the unrelated money transfers before attempting to join. The unrelated money transfers hence do contribute to window evictions.

Searching the Event Traces for Matches to the Patterns

Each event trace generated by the process described above is first matched against the PARTElang program for driver scenario B (defined in listing 7.3 on page 191), executed by PARTE$^{\text{Rust}}$. Next, it is ran against the manual implementation in Spark Streaming (in section 7.3.3 on page 197), at increasing multiples of the minimum viable size. First, the Spark Streaming program is ran with windows of exactly the minimum viable partial match size determined by Featherweight PARTE, i.e. with the same memory consumption as the run of PARTE$^{\text{Rust}}$. Next, the Spark Streaming program is ran with windows twice the minimum viable size, then 3 times, and so on, until the point where the Spark Streaming program reliably finds all matches.

```
let amount = if is_above_threshold {rng.gen_range(100, 999)} else {rng.gen_range(1, 99)};
let (orig_acc, dest_acc) = unique_account_pair_from_seed(*next_id, is_common_route);

{ // Ensure it's safe to use this event w.r.t. the `EVENT_RATE`
  if (all_events.get(&time).unwrap_or_default().iter().count() + 1) > EVENT_RATE {
    // This random sample failed: would produce too many events in a certain timeslot
    return Err(());
  }
}

let event = Event::new(template_transferred,
                       vec![ Value::from(*next_id),
                             Value::from(orig_acc),
                             Value::from(dest_acc),
                             Value::from(amount)],
                       Time::from_milliseconds(time as f64)).unwrap();
*next_id += 1;
all_events.entry(time).or_default().push(event);
```

Listing 7.9: Rust code generating a new unrelated event given a configuration

Though this is not a performance benchmark, we minimize measurement noise by executing all benchmarks on the same machine. Interference by other processes are reduced to a minimum: the machine is not used for other purposes during the benchmark. The machine has sufficient RAM to match all event traces, and a sufficiently large heap size is allocated for the JVM in which the Spark Streaming job runs, such that all runs of the programs complete without errors.

7.4.2 Results

Using the experimental setup described in section 7.4.1, we obtained results for event traces with 10, 100, and 1000 diffusion groups (consisting of 4 `MoneyTransferred` events each), each interspersed by either 100, 1000, or 10000 unrelated `MoneyTransferred` events. The results we obtained are summarized in appendix B.[6]

Case: 10 Diffusion Groups

The graph in figure 7.5 plots the fraction of fraud cases which are detected for a given partial match history size. In total, 10 diffusion groups occur in the event trace, of which 6 satisfy all conditions of a fraudulent money diffusion. When the event trace consists of only the 40 `MoneyTransferred` events forming diffusion groups, PARTE$^{\text{Rust}}$ detects all 6 cases of fraud when its storage is limited to the minimum viable size. The Spark Streaming implementation also finds all of the fraud cases when its storage is limited to the same size as PARTE at the minimum viable sizes. The same holds when the 10 diffusion groups are hidden in a stream of 100 or 1000 unrelated events.

[6]The raw output of our experiments, as well as the code used to process it, is available at `https://soft.vub.ac.be/~trenaux/PARTE/evaluation.zip`.

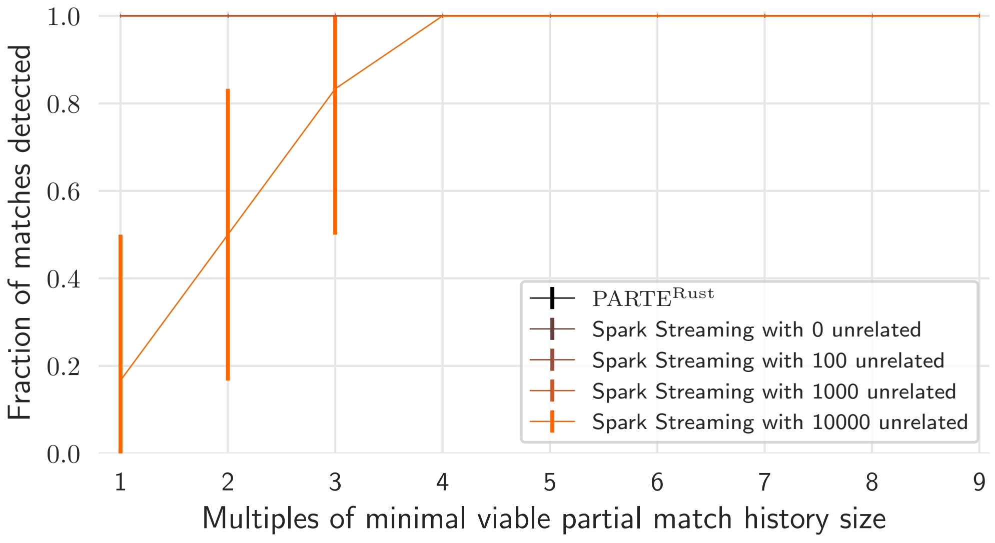


Figure 7.5: Fraction of the 6 fraud cases detected for a given partial match history size — Error bars indicate the 5% and 95% percentile of the measurements. The line connects the medians.

However, when the stream contains 10000 unrelated events, the Spark Streaming implementation only detects between 0 and 3 of the 6 fraud cases (with a median of 1 case) when it is limited to a storage size equal to PARTE's minimum viable size. Increasing the storage size allocated for the Spark Streaming implementation improves the situation: at twice PARTE's minimum viable size, the Spark Streaming implementation detects between 1 and 5 of the cases (with a median of 3 cases). Increasing the storage size again further improves the situation: at 3 times PARTE's minimum viable size, the Spark Streaming implementation finds between 3 and 6 of the 6 cases (with a median of 5 cases). With 4 times PARTE's minimum viable size, the Spark Streaming implementation finds all 6 matches in our experiments. PARTE^{Rust} detects all 6 matches regardless of the amount of noise.

Case: 100 Diffusion Groups

The graph in figure 7.6 similarly plots the fraction of fraud cases which are detected for a given partial match history size. In this case, 100 diffusion groups occur in the event trace, of which 50 satisfy all conditions of a fraudulent diffusion of money. When the event trace consists of only the 400 `MoneyTransferred` events forming diffusion groups, PARTE^{Rust} detects all 50 cases of fraud when its storage is limited to the minimum viable size. The Spark Streaming implementation finds approximately 35 out of the 50 (70%) fraud cases with its storage limited to equal PARTE's the minimum viable sizes. Doubling

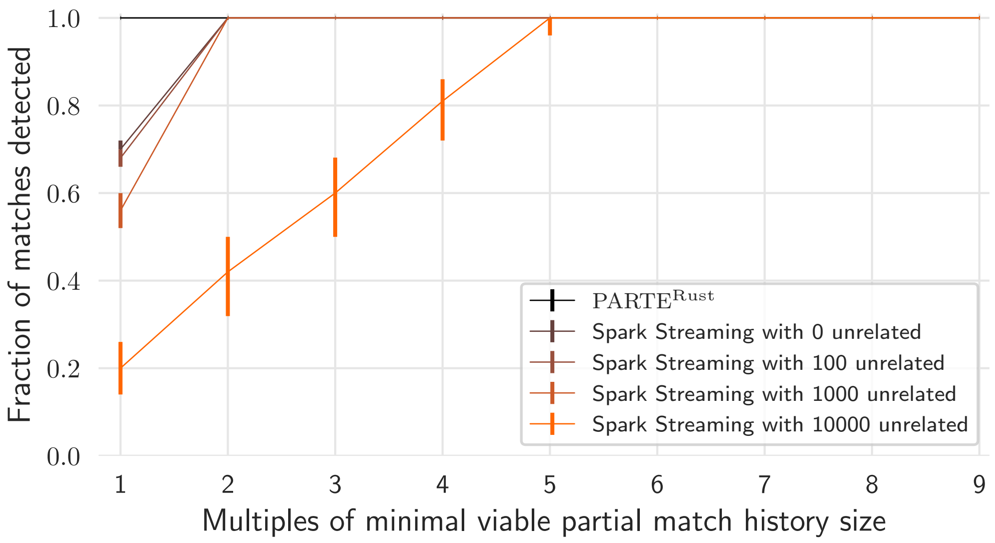


Figure 7.6: Fraction of the 50 fraud cases detected for a given partial match history size — Error bars indicate the 5% and 95% percentile of the measurements. The line connects the medians.

the storage size allocated for the Spark Streaming implementation resolves the issue: all 50 cases are then detected.

The situation is similar when the 100 diffusion groups are hidden in a stream of 100 or 1000 unrelated events: in the former case, a median of 34 out of the 50 (68%) fraud cases are found at the minimum viable size, in the latter case a median of 28 out of 50 (56%). In both cases, doubling the partial match storage size solves the issue: all 50 cases are then detected.

However, in the case of 10000 unrelated `MoneyTransferred` events, detection rates are significantly worse: the Spark Streaming implementation only detects a median of 10 out of 50 (20%) of the cases when limited to PARTE's minimum viable size, and at five times that size still does not always detect all 50 cases. Only at six times PARTE's minimum viable size does the Spark Streaming implementation reliably find all matches. PARTE$^{\text{Rust}}$— in contrast — reliably detects all 50 matches in all cases.

Case: 1000 Diffusion Groups

The graph in figure 7.7 similarly plots the fraction of fraud cases which are detected for a given partial match history size. In this final case, 1000 diffusion groups occur in the event trace, of which 500 satisfy all conditions of a fraudulent diffusion of money. When the event trace consists of only the 4000 `MoneyTransferred` events forming diffusion

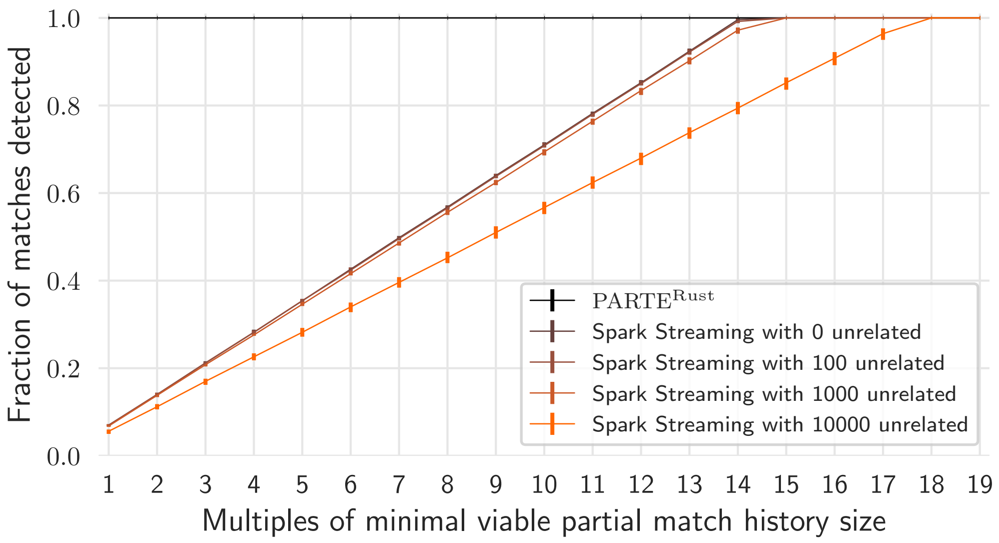


Figure 7.7: Fraction of the 500 fraud cases detected for a given partial match history size — Error bars indicate the 5% and 95% percentile of the measurements. The line connects the medians.

groups, PARTE^{Rust} detects all 500 cases of fraud when its storage is limited to the minimum viable size. The Spark Streaming implementation finds only 35 of the 500 (7%) fraud cases with its storage limited to equal PARTE's the minimum viable sizes. It takes 15 times PARTE's minimum viable storage size before the Spark Streaming implementation reliably detects all fraud cases.

As with the event traces discussed previously, the situation becomes worse as unrelated events are added to the event trace. In the worst case studied — 10000 unrelated events — it takes a storage size 18 times that of PARTE^{Rust} before all fraud cases are reliably matched. Again, PARTE^{Rust} reliably detects all 500 matches in all cases.

7.4.3 Conclusion

Existing systems lack a built-in notion of a minimum viable size. Their execution model hence does not provide the means to operate in a minimum viable size. As a result, forcing state of the art systems to operate in a limited memory size, comes at the cost of correctness: potential matches are missed. The number of missed matches grows as the number of events to match grows. **PARTE does not suffer from this shortcoming, and can hence guarantee correctness with significantly smaller memory sizes.**

PARTE is not only a formal model for finding an upper bound on memory requirements, it is also an execution model. It does not suffice to express a program in PARTElang syntax, and to have the Featherweight PARTE model determine the minimum viable sizes. The actual implementation of the program must be executed according to the Featherweight PARTE model, as existing execution platforms do not support operating within the minimum viable sizes.

7.5 Conclusion

We explored the 3 driver scenarios in detail, and expressed them in PARTElang. We described the difficulty of expressing those PARTElang programs in mature event processing technologies. We identified the core problem to be the lack of a notion of minimum viable sizes in the execution model of those event processing systems: existing systems either cannot express a limit on the size of the state that is retained, or do so at the cost of missing potential matches. We quantified this latter cost by executing a port of the PARTElang program for driver scenario B to Spark Streaming, and measuring how many fraudulent money diffusions went undetected when the program was limited to using only PARTE's minimum viable partial match sizes. We showed that the problem of the Spark Streaming port was due to the size of the storages: by increasing the storage size, the number of missed fraud cases decreased.

Our conclusions from this chapter are hence twofold. First, we conclude that PARTE can be used to express cases like those exemplified by our driver scenarios from section 2.1. Second, we conclude that the PARTE model offers a measurable benefit over the state of the art, by offering an execution model that supports operating within a minimum viable size.

8 Formal Validation

In chapter 5 we defined the Featherweight PARTE model. Featherweight PARTE is an operational semantics for the Logic Reactive Programming language PARTElang defined in section 4.2. In this chapter, we prove that Featherweight PARTE indeed satisfies the property of *strong reactivity* defined in definition 1 on page 65. We go about this as follows:

- First, we prove that our approach implements PARTElang sans *strong reactivity*. Specifically, we show that the logical clocks maintained by Featherweight PARTE are properly maintained in a distributed address-space setting (section 8.1). We then show that the lower bounds tracked by these logical clocks allow for a conservative garbage collection of the managed storage (section 8.2), guaranteeing constant space usage. Finally, we show that those lower bounds ensure that progress is possible during evaluation (section 8.3).

- Second, we prove that our model's use of updates in the absence of new data is fundamentally necessary, despite not appearing in the declarative language, nor in its underlying event algebra (section 8.4).

- Third, we prove that fwPARTE ensures a finite upper bound on spatial (section 8.5) and temporal (section 8.6) resource usage.

- Fourth, combining those proofs, we conclude that fwPARTE does result in a *strongly reactive* evaluation model for PARTElang.

8.1 Correctness of the Lower Bounds on Event Timestamps

Statement

The lower bounds on event timestamps that are stored in nodes of a PARTE graph correctly reflect the state of the PARTE graph.

We prove the correctness of the lower bounds on event timestamps by induction on the steps in the global and local evaluation languages $\xrightarrow{g}$ and $\xrightarrow{l}$ (as defined in sections 5.4.6 and 5.4.7 on pages 134 and 145). First, we establish that four invariants hold in a newly constructed fwPARTE graph, then we demonstrate that every step in $\xrightarrow{g}$ and $\xrightarrow{l}$ maintains these four invariants.

8.1.1 Local Invariants

The four invariants are these:

I.1 — Lower Bounds Stored in Node Reflect State at Predecessors For any node which maintains a lower bound or a vector of lower bounds on timestamps of predecessors in its node local data, these lower bounds correctly reflect the lower bounds of its predecessors.

I.2 — Lower Bounds Produced by Node Reflect State at Node For any token that is produced by a step of the local evaluation language at a node, the lower bounds on timestamps contained in that token reflect the lower bound on timestamps that is known at that originating node.

I.3 — Lower Bounds Stored in Outboxes Reflect State at Node Tokens that are created at a node convey the lower bounds on timestamps from the perspective of that node, at the time the token is created. The lower bound (for alpha tokens) or vector of lower bounds (for beta tokens) of any token stored in the outbox of a node is still a lower bound on event timestamps originating from that node.

I.4 — Lower Bounds Stored in Inboxes Reflect State at Predecessors Any message that appears in the inbox of a node, contains one lower bound (for alpha tokens) or a vector of lower bounds (for beta tokens). This lower bound conveys a lower bound on the timestamps which were correct for that node's predecessor, at the time when the token was created at that predecessor. That lower bound is still a lower bound on event timestamps originating from that predecessor.

At PARTE graph construction time (see section 5.5), all lower bounds on timestamps maintained in node-local data are set to $-\infty$. By definition, $-\infty$ is a lower bound on any timestamp, and hence invariant I.1 trivially holds in the base case. At PARTE graph construction time, no step of $\xrightarrow{l}$ has taken place, and all inboxes and outboxes are empty. As such, invariants I.2, I.3, and I.4 also trivially hold in the base case.

8.1.2 Proof

For this section we remind the reader of the evaluation rules of the global evaluation language $\xrightarrow{g}$ in section 5.4.7 on page 145, and to the evaluation rules of the local evaluation language $\xrightarrow{l}$ in section 5.4.6 from page 134 onwards.

We now prove that invariants I.1 through I.4 hold during evaluation of all evaluation steps of both languages. We first discuss the two rules of the global evaluation language. Next, for each node type we discuss the rules of the local evaluation language.

During (Proc)

Lemma 1:

I.1 is upheld by (Proc).

Proof. By the induction hypothesis, invariants I.1, I.2, I.3, and I.4 hold before a step of (Proc). Also by the induction hypothesis, the invariants are preserved by the single step of $\xrightarrow{l}$. No other transformation touches the node-local data in (Proc). Since timestamps are non-decreasing, any previously correct lower bound on timestamps of predecessors contained in the node-local data, remains a correct lower bound on timestamps regardless of what semantical state-change the predecessor undergoes. Hence, I.1 is upheld throughout (Proc). □

Lemma 2:

I.2 is upheld by (Proc).

Proof. I.2 is trivially upheld, since (Proc) is not a step of $\xrightarrow{l}$. □

Lemma 3:

I.3 is upheld by (Proc).

Proof. By the induction hypothesis, the tokens already in the outbox before evaluation of (Proc), i.e., $\overline{t}$, satisfy I.3. Also by the induction hypothesis, I.2 holds for $\overline{t'}$. From this, it follows that the lower bounds contained in $\overline{t} \diamond \overline{t'}$ are correct lower bounds at n, i.e., that I.3 is upheld. □

Lemma 4:

I.4 is upheld by (Proc).

Proof. I.4 is trivially upheld, since (Proc) does not modify any node's inbox, and correct messages in inboxes remain correct regardless of state updates to the rest of the node, since timestamps are non-decreasing. □

During (Prop)

Lemma 5:

I.1 is upheld by (Prop).

Proof. Since (Prop) does not modify any node's node-local data, and correct lower bounds on timestamps contained there, remain correct regardless of state updates to the rest of the PARTE graph, since timestamps are non-decreasing. Hence, I.1 is trivially upheld during execution of (Prop). □

Lemma 6:

I.2 is upheld by (Prop).

Proof. I.2 is trivially upheld, since (Prop) is not a step of $\xrightarrow{l}$. □

Lemma 7:

I.3 is upheld by (Prop).

Proof. Since (Prop) does not add tokens to nodes' outboxes, but only removes some tokens from them, (Prop) cannot place incorrect lower bound into nodes outboxes. Invariant I.3 is hence upheld throughout execution of (Prop). □

Lemma 8:

I.4 is upheld by (Prop).

Proof. (Prop) moves a token t from the outbox of a node a into the inbox of the nodes in $\mathbf{N}_b$. By the induction hypothesis, I.3 holds for t at node a, since t was in a's outbox. Hence, the lower bounds on timestamps contained in t correctly reflect the lower bounds on timestamps at node a. Since each node in $\mathbf{N}_b$ is in $\overline{\mathfrak{s}_a}$, a is the predecessor of each node in $\mathbf{N}_b$, i.e., the lower bounds on timestamps contained in t correctly reflects the lower bounds at the successor node a, i.e., I.4 holds. □

During Evaluation at Entry Nodes

Lemma 9:

I.1 is upheld by (Entry) and (Entry-$\bot$).

Proof. By the induction hypothesis, the node-local data of the entry node at which $\xrightarrow{l}$ evaluates (Entry) or (Entry-$\bot$) holds correct lower bounds on timestamps for all sources in $s\ell map$. The new lower bound ℓ is a correct lower bound on timestamps the entry node will receive from its predecessor identified by source identifier s, since invariant I.4 of the induction hypothesis ensures this. Replacing the mapping for s in $s\ell map$ by the new correct lower bound ℓ hence results in a new correct source-to-lower-bound mapping $s\ell map'$. Invariant I.1 is hence upheld. □

Lemma 10:

I.2 is upheld by (Entry) and (Entry-$\bot$).

Proof. The alpha token produced by a step of (Entry) or (Entry-$\bot$) wraps exactly one lower bound on timestamps: $min(codom(s\ell map'))$. This lower bound is greater than or equal to any previously communicated lower bound: it is equal if $s\ell map[s]$ previously held the strictly smallest value in $codom(s\ell map)$, and greater than or equal to the previously communicated lower bound otherwise. This lower bound is never greater than any value held in $s\ell map$, by definition of min, and never greater than any timestamp or lower bound on timestamp still to appear in $s\ell map$, by the induction hypothesis. Together, these entail that $min(codom(s\ell map'))$ constitutes a correct lower bound on timestamps from the entry node, and hence that the produced token reflects a correct lower bound. In conclusion, I.2 is upheld during (Entry) and (Entry-$\bot$). □

Lemma 11:

I.3 and I.4 are upheld by (Entry) and(Entry-$\bot$).

As a rule of the local evaluation language, both (Entry) and (Entry-$\bot$) touch neither inbox nor outbox, trivially upholding invariants I.3 and I.4. We will not list I.3 and I.4 for the remaining rules of $\xrightarrow{l}$, as they are always trivially upheld for this same reason.

During Evaluation at Alpha Nodes

Lemma 12:

I.1 is upheld by (Alpha-$\bot$), (Alpha-Match), and (Alpha-NoMatchConstraint).

Proof. Alpha nodes do not store any lower bounds on timestamps in their node-local data, trivially upholding I.1. □

Lemma 13:

I.2 is upheld by (Alpha-$\bot$), (Alpha-Match), and (Alpha-NoMatchConstraint).

Proof. By the induction hypothesis, the node-local data of an alpha node at which $\xrightarrow{l}$ evaluates a step holds correct lower bounds on timestamps for its predecessor. The new lower bound ℓ is a correct lower bound on timestamps the alpha node will receive from its predecessor entry node, since invariant I.4 of the induction hypothesis ensures this. Alpha nodes store no state. Thus, ℓ retains its correctness as the lower bound on timestamps for events for the event pattern handled by this alpha node, i.e., the event pattern identified by ι_e. Invariant I.2 is hence upheld for rules (Alpha-$\bot$) and (Alpha-Match). The rule (Alpha-NoMatchConstraint) produces no tokens, trivially upholding I.2. □

During Evaluation at Join-Nodes

Lemma 14:

I.1 is upheld by (Join-$\searrow$-$\bot$), (Join-$\swarrow$-$\bot$), (Join-$\searrow$), and (Join-$\swarrow$).

Proof.

(i.) By the induction hypothesis, $\ell map_{\searrow}$, $\ell_{\swarrow}$ are correct lower bounds on timestamps.

(ii.) On left-activation, the stored $\ell map_{\searrow}$ is overwritten by the ℓmap_{t_β}, which by the induction hypothesis is a correct lower bounds on timestamps arriving from the left predecessor.

(iii.) On right-activation, the stored $\ell_{\swarrow}$ is overwritten by the singular lower bound mapped to the right event identifier $\iota_{e_{\swarrow}}$ in the ℓmap_{t_β}, i.e, $\ell map_{t_\beta}[\iota_{e_{\swarrow}}]$. This is — by the induction hypothesis — guaranteed to be a correct lower bound on future timestamps arriving from the right predecessor.

(iv.) It follows from *(ii.)* and *(iii.)* that I.1 is upheld. □

Lemma 15:

I.2 is upheld by (Join-$\searrow$-$\bot$), (Join-$\swarrow$-$\bot$), (Join-$\searrow$), and (Join-$\swarrow$).

Proof.

(i.) The partial matches that can be generated by a join-node, receive their data from two origins: either from partial matches that still have to be sent to it by the node's predecessors (i.e., in tokens that are in its inbox, or will still arrive in its inbox), or from partial matches that have already been sent to the node (i.e., in partial matches already stored in its partial match histories).

(ii.) It follows from (*i.*) that the lower bound on timestamps for a certain event identifier is determined by two factors in join-nodes: it is the minimum of the lower bounds of partial matches that still have to arrive, and the lower bounds of partial matches that are still stored.

(iii.) The former are communicated explicitly by the predecessors, and are stored explicitly by the join-node. By I.1, these are correct lower bounds for the predecessors, and hence for the join-node itself.

(iv.) For the latter, it holds that the explicit lower bounds are the minima of the timestamps actually stored for each event identifier.

(v.) It follows from (*iii.*) and (*iv.*) that I.2 is upheld. □

During Evaluation at Not-Nodes

Lemma 16:

I.1 is upheld by (Not-$\searrow$-$\bot$), (Not-$\swarrow$-$\bot$), (Not-$\searrow$-Match), (Not-$\searrow$-NoMatch), and (Not-$\swarrow$).

Proof. Invariant I.1 is upheld at not-nodes for the same reason it is upheld at join-nodes. □

Lemma 17:

I.2 is upheld by (Not-$\searrow$-$\bot$), (Not-$\swarrow$-$\bot$), (Not-$\searrow$-Match), (Not-$\searrow$-NoMatch), and (Not-$\swarrow$).

Proof. Invariant I.2 is upheld at not-nodes in a similar way as at join-nodes, with one difference: only the temporal lower bounds for event identifiers bound in the left predecessor of the not-node are produced, since the event identifier from the right predecessor is not bound, and is not used in the successors of the not-node. This does not affect the reasoning on correctness of the lower bounds for the left event identifiers, and hence I.2 is upheld. □

During Evaluation at Production Nodes

Lemma 18:

I.1 is upheld by (Prod-$\bot$) and (Prod).

Proof. Production nodes do not store any lower bounds on timestamps in their node-local data, trivially upholding I.1. □

Lemma 19:

I.2 is upheld by (Prod-$\bot$) and (Prod).

Proof. Invariant I.2 must be upheld during (Prod-$\bot$) and (Prod) by the concrete definition of lower bound generator expression: if and only if $g^{\ell}_{\square}$ generates correct lower bounds based on a mapping of event identifiers to correct temporal lower bounds, by the induction hypothesis I.2 holds. The constraint on $g^{\ell}_{\square}$ can be achieved by ensuring that $\forall \ell map_1, \ell map_2 \in \mathbf{M}_{\mathbf{I_E} \rightharpoonup \mathbf{L}} : \ell map_1 < \ell map_2 \implies g^{\ell}_{\square}(\ell map_1) < g^{\ell}_{\square}(\ell map_2)$, where $\ell map_1 < \ell map_2$ means ℓmap_1 is a correct lower bound mapping of mapping ℓmap_2. □

During Evaluation at Terminal Nodes

Lemma 20:

I.1 and I.2 are upheld by (Term).

Proof. Terminal nodes neither store any lower bounds on timestamps in their node-local data, nor produce any tokens, trivially upholding all invariants. □

8.1.3 Conclusion

It follows from lemmas 1 to 20 that invariants I.1 through I.4 hold during all evaluation steps on a Featherweight PARTE graph. Hence, the propagation of lower bounds on timestamps within fwPARTE is correct.

8.2 Expiration does Not Discard Valid Matches

The semantics of expiration differ little from those described by Teodosiu and Pollak [133]. In their paper, they show that their algorithm only discards "unused temporal information", which coincides with our definition of *stale partial matches* on page 122.

In this section we demonstrate only that our migration from a centralized setting to a distributed memory space model does not affect their conclusion.

We described in section 5.5 how the work by Teodosiu and Pollak [133] consists of a static and a dynamic phase, and how PARTE does not deviate from their approach for the static phase. We hence refer to their paper for the correctness of the static phase. In the dynamic phase, they established that discarding partial matches does not affect which valid matches are found, if and only if the “temporal information” which is discarded, is indeed “unused”, i.e., within our nomenclature and definitions: *expiration* does not discard valid matches if and only if the partial matches which are expired, are indeed *stale* partial matches.[1] To prove that expiration does not discard valid matches, it hence suffices to prove the following:

Statement
Expiration in fwPARTE behaves identical to the sequential shared-memory case described by Teodosiu and Pollak [133].

8.2.1 Proof

Proof.

(i.) Partial matches are only stored at join-nodes and not-nodes, and only inspected and manipulated in the steps of $\xrightarrow{l}$ at those nodes. In the operational model of fwPARTE, *expiration* hence takes the form of removing partial matches from a partial match history in the steps of $\xrightarrow{l}$ at join-nodes or not-nodes. More specifically, a partial match is “removed” when that partial match occurs in a partial match history on the left-hand side of a step in $\xrightarrow{l}$, but does not occur in the corresponding partial match history on the right-hand side of that step.

(ii.) It follows trivially from the definition of functions *expire* and *expire_negated* from section 5.4.6 that they only remove partial matches that fall outside of the semantic window, assuming that the semantic window is computed correctly.

(iii.) It follows from (*i.*) and (*ii.*) that we can phrase the statement to prove another way: to prove that expiration does not discard valid matches, it suffices to prove that that the semantic window is computed correctly.

(iv.) The semantic windows computed by the shared-memory-space approach by Teodosiu and Pollak [133] are correct, as shown in their paper.

[1]Of course, by definition 17 and definition 18, *expiration* is defined to only affect *stale* partial matches. In this section, we demonstrate that the operational semantics of fwPARTE indeed ascribe the correct expiration behavior to joining and anti-joining.

(v.) It follows from (*iii.*) and (*iv.*) that expiration does not discard valid matches in PARTE if and only if the semantic windows computed by PARTE are identical to those computed by the approach by Teodosiu and Pollak.

(vi.) It follows trivially from their definition that the uses of the dynamically computed temporal lower bounds in evaluation steps of $\xrightarrow{l}$ at join-nodes and not-nodes are equivalent to those of the approach by Teodosiu and Pollak: the computation is performed using local data, mimicking the shared-memory-space approach. The only remaining source of difference, is hence the propagation of the temporal lower bounds between the nodes.

(vii.) It follows from our proof in section 8.1 that the requirement of (*vi.*) is satisfied: the distributed-memory-space propagation of temporal lower bounds correctly propagates temporal lower bounds. □

8.2.2 Conclusion

The proof by Teodosiu and Pollak [133] depends only on elements unmodified by the PARTE model, and on correctly determining “unused”, i.e, *stale*, partial matches. It follows from the reasoning sketched in previous subsection that expiration as implemented by the fwPARTE model abides by the definition of expiration on page 123, i.e., expiration only affects stale partial matches. Hence, expiration of stale partial matches does not cause valid matches to be discarded.

8.3 Unattainability of a Blocked PARTE Network

The correctness of the fwPARTE model depends on the evaluation not leading to a *blocked PARTE network*. We must therefore prove that nodes in the fwPARTE graph never reach a *blocked node state*. The bulk of the proof was already given throughout section 5.3.6 and section 5.3.7. We have already established that — by construction — only two-input nodes can become *blocked*. We now prove the part that was not proven yet, namely that the *minimum viable partial match history size* we used in the fwPARTE model is a proper *minimum viable size*:

Statement

The minimum viable partial match history sizes used in fwPARTE is correct, i.e., the minimum viable size ensures that fwPARTE can evaluate without reaching a blocked state.

8.3.1 Proof

Every (anti-)join defines temporal (anti-)join-constraints. These constraints define a time window. Furthermore, for each activation side of a (anti-)join, a partial match arrival rate is defined.

Consider a two-input PARTE graph node with a time window of W time units, a left partial match arrival rate of $r_{\searrow}$ per time unit, and a right partial match arrival rate of $r_{\swarrow}$ per time unit. We first consider the case of left activations.

Lemma 21:

The minimum viable left partial match history size is $W \times r_{\searrow} + 1$.

Proof.

(i.) Consider the case where left partial matches occur at their highest allowed frequency. There are $r_{\searrow}$ left partial matches at time t_1, $r_{\searrow}$ left partial matches at time $t_2, \ldots$ and $r_{\searrow}$ left partial matches at time t_W. These $r_{\searrow} \times W$ left partial matches may be all fit inside the *semantic time window*.

(ii.) From $(i.)$ it follows that $r_{\searrow} \times W$ is a lower bound on the number of left partial matches that need to be stored.

(iii.) Consider the case where $r_{\searrow} \times W + 1$ left partial matches are stored. By definition of $r_{\searrow}$, there can be at most $r_{\searrow}$ left partial matches with times between t_1 and t_2. Similarly, there can be at most $r_{\searrow}$ partial matches with times between t_2 and t_3, etc. Finally, there can be at most $r_{\searrow}$ partial matches between t_{W-1} and t_W. It follows that $\sum_{i=0}^{W-1} |\overline{pm}[t_i, t_{i+1}]| \leq W \times r_{\searrow}$, and hence that the partial match with index $r \searrow \times W + 1$ necessarily must occur at a timestamp $\geq t_{W+1}$.

(iv.) From $(iii.)$ it follows that at least one of these partial matches will fall outside of the time window of the next opposing match the combine with. Hence, either the oldest left partial match will expire when a new right partial match arrives, or the youngest left partial match will be too young for the next right partial match. Phrased differently, when storing $r_{\searrow} \times W + 1$ left partial matches, at least one of those matches is either *stale* or not yet relevant for the matching. In the former case, it can be expired, in the latter case, it can be left in the inbox.

(v.) From $(iv.)$ it follows that $r_{\searrow} \times W + 1$ is an upper bound on the number of left partial matches that need to be stored.

(vi.) In addition to the spots required to store the valid left partial matches, one spot of "maneuvering space" is required. This extra space guarantees that there is always room to serve a token from the inbox. We hence increase the lower bound established in $(ii.)$ from $r_{\searrow} \times W$ to $r_{\searrow} \times W + 1$.

(vii.) From $(v.)$ and $(vi.)$ it follows that $r_{\searrow} \times W + 1$ is both a lower bound and an upper bound to the number of left partial matches that need to be stored to enable evaluation to proceed without blocking two-input nodes upon left activation. □

Lemma 22:

The minimum viable right partial match history size is $W \times r_{\swarrow} + 1$.

Proof. A similar reasoning can be made for the case of right-activations. Note that — while the direction of the semantic window is reversed when considering the opposite activation side — the absolute duration of the semantic window is unchanged. □

8.3.2 Conclusion

From lemmas 21 and 22 it follows that for a two-input PARTE graph node with a time window of W time units, left and right partial match arrival rates of $r_{\searrow}$ per time unit and $r_{\swarrow}$ per time unit, the minimum viable left partial match history size is $W \times r_{\searrow} + 1$ and the minimum viable right partial match history size is $W \times r_{\swarrow} + 1$.

8.4 Necessity of "No Change" in Negated Subgraphs

We prove the necessity of propagating tokens even when no new events are available by demonstrating that not using such tokens may lead to nodes remaining in a *waiting state* longer than they should be. Doing so would open up the possibility of exhausting room in the partial match histories.

Consider a variant of fwPARTE which only propagates tokens when new events are available, i.e., which lacks support for "no change" indicators. We denote this variant as *fwPARTE*$\backslash\perp$. We show that *fwPARTE*$\backslash\perp$ effectively introduces a reachable *blocked state*. Evidently, a reachable blocked state precludes *strong reactivity*, and is hence unacceptable for the proper fwPARTE model. To match negated patterns, the absence of partial matches must be detectable. We prove the following statement:

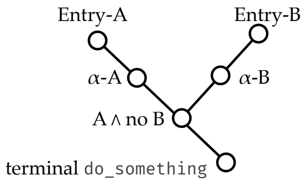
Statement

No minimum viable size can be defined for *fwPARTE*$\backslash\perp$, such that evaluation is guaranteed not to reach a blocked state.

Example rule

```
rule a_rule_with_negation where
    a : A { }
    no b : B { attr = a.attr }
when
    no b in a [0 seconds, 4 seconds]
then
    do_something(a.attr)
```

fwPARTE\⊥ graph of example rule



Event arrival per timestamp

θ	Occurrence with template A	Occurrence with template B
0	A { attr : 's' }	B { attr: 's' }
1	A { attr : 't' }	/
2	A { attr : 'u' }	/
3	A { attr : 'v' }	/
4	A { attr : 'w' }	/
5	A { attr : 'x' }	/
6	A { attr : 'y' }	/
7	A { attr : 'z' }	B { attr: 'z' }

Figure 8.1: Example of a rule with a negated pattern, and a series of events — On the left, the rule is represented both textually in PARTElang, and as a graph of fwPARTE\⊥ nodes. On the right, a table lists which events occur at a certain timestamp.

8.4.1 Proof

Proof. Consider the rule and its fwPARTE\⊥ graph depicted in figure 8.1. Assume that the event arrival rate of both events of type A, and events of type B is 1 event per second.

Using the definitions for *minimum viable size* from section 5.3.7, the minimum viable size of the inbox for the not-node is 3, the minimum viable size of the outbox is 5, and so are the minimal viable sizes of the not node's partial match histories. Consider now the event arrivals depicted in the table in figure 8.1.

At timestamp 4, the left partial match history contains events of template A with values 's', 't', 'u', 'v', and 'w' bound to its single attribute `attr`, and timestamps 0 through 4 respectively. The right partial match history contains only the single event of template B with value 's' bound to its single attribute `attr`, and timestamp 0. When the event of template A for timestamp 5 arrives, there is no room in the left partial match history. However, in the absence of events of template B, none of the events in the left partial match history are detected to be *stale* by the execution of the local evaluation language $\xrightarrow{l}$ (see section 5.4.6), despite the fact that `A { attr: 's' }` is *stale* as per definition 17. This undetected staleness requires the not-node to retain the token until an event of template B arrives, requiring additional spatial resources for each such stale token.

In this example, the situation would be rectified at timestamp 7, when `B { attr: 'z' }` arrives. For this specific event arrival sequence, the minimum viable left partial match history in fwPARTE\⊥ is 7. More generally, the minimum viable partial match history sizes in fwPARTE\⊥ depends linearly on the maximum time between events. This runtime condition inherently preempts the possibility of statically limiting the size. The possibility exists that no event of the template used on the negated side of a not-node ever occurs. This by definition makes the minimum viable partial match history sizes in fwPARTE$\backslash\perp$ non-finite. $\square$

To resolve this, tokens must sometimes be produced even when no real events occurred, which concludes our reasoning that "no change" tokens are needed. The fwPARTE$\backslash\perp$ model extended with the option to propagate "no change" tokens is denoted by fwPARTE$'$. We further must show what the minimum generation rate of "no change" tokens should be to prevent exhausting spatial resources in fwPARTE$'$.

8.4.2 Minimum Required "No Change" Token Generation Rate

From the last proof it follows that a token — e.g. a "no change" token — must arrive at an activation side of a not-node before spatial resources of the *opposite partial match history* are exhausted. A trivially correct rate of "no change" token generation consists of generating a "no change" token at each tick of the static *event arrival rate*. This "no change" token generation rate serves as an upper bound: a higher rate is definitely not required.

The minimum required rate depends on the size of the inboxes, outboxes, and partial match histories of the nodes along the subgraph of the fwPARTE$'$ graph containing not-nodes.[2] Some minimum viable size must be defined for inboxes, outboxes, and partial match histories in fwPARTE$'$. For inboxes and outboxes, fwPARTE$'$ inherits the minimum viable sizes of fwPARTE. For the partial match histories, given a minimum viable size in fwPARTE, denoted m, it holds that $\forall s \geq m : s$ is a viable size in fwPARTE$'$ if a $\perp$ token is produced by all entry nodes for templates used on the negated side of a not-node at least once per semantic time window size. This follows from the observations that exhaustion of spatial resources in fwPARTE$\backslash\perp$ only occurs when an entire semantic time window's worth of events are consecutively absent for the negated event pattern. Hence, fwPARTE is nothing more than fwPARTE$'$ with the tightest possible minimum viable size on partial match histories, and hence the minimum "no change" token generation rate for fwPARTE$'$ is the minimum "no change" token generation rate for fwPARTE.

8.4.3 Conclusion

It follows from the reasoning in this section that "no change" tokens must be generated by nodes producing tokens that might right-activate not-nodes. Such tokens must only be generated when no data-carrying tokens would otherwise be sent within the semantic time window of the successor node in the fwPARTE graph.

[2]By this subgraph, we mean all not-nodes and their direct and indirect predecessors, up to the entry nodes, but not sibling nodes which are not themselves direct or indirect predecessors of not-nodes. Note that this predecessor relation includes event emission by production nodes.

8.5 Boundedness of State Size of fwPARTE Graph

Proving that finite bounds exist on the state size is impossible for the rudimentary PARTE model from section 5.2, as its lack of event expiration trivially leads to infinite storage requirements. In contrast, the Featherweight PARTE model from section 5.4 has a fixed, finite upper bound on state by construction. In the previous sections we showed that the limits on state size of partial match histories, inboxes, and outboxes do not prevent correct execution. In this section we exhaustively list all components found in the fwPARTE model, and demonstrate for each one of them that a finite, static upper limit on the size of their state exists.

8.5.1 Local Invariant

The invariant for the proof in this section is as follows:

I — Fixed, Finite Size of Subelements Each element in a fwPARTE graph consists of a finite, statically limited number of elements, each of which has a finite, statically limited size.

Primitive, atomic elements such as identifiers, values, and timestamps trivially have a constant finite size. Invariant *I* holds for all primitive elements.

8.5.2 Proof

At the top level, a system in the fwPARTE model consists of a set of event templates, and a set of nodes **N**. The set of templates is finite and immutable once the rule set is compiled into a graph. The size of **N** is also limited at construction time: the graph construction algorithm defined in section 5.5 converts a rule set consisting of a finite number of rules into a graph with a finite number of nodes. This entails that the size of a system implementing the PARTE model has a finite, static upper bound if and only if each individual node has a finite, static state size. Individual nodes' state size, in turn, is determined by the data they store in their node-local data, as well as the number of tokens they store in their inboxes and outboxes. By the induction hypothesis, these have a fixed, finite limit on state size.

In the remainder of this section we survey the state size of the elements in a PARTE graph. We do so in the order of their definition in section 5.4. We start at events, and build up to the level of nodes.

Lemma 23:

The size of individual events and tokens has a finite upper bound.

Events As defined in section 5.4 events are tuples of the form $\mathcal{E}\langle \iota_t, attrs, \theta \rangle$.

Proof. By the induction hypothesis, the template identifier ι_t and timestamp θ have a constant size. The size of *attrs* depends on the number of values, and hence on the number of attribute identifiers for which a value is provided. The number of attribute identifiers for an event is fixed, and specified by the template identified by ι_t. In conclusion, individual events have a fixed, finite size. $\square$

Alpha tokens The spatial resource usage of alpha tokens is similar. Remember from section 5.4 that alpha tokens are tuples of the form $\mathcal{T}_\alpha\langle \iota_s, \ell, p_\alpha \rangle$.

Proof. By the induction hypothesis, the source identifier ι_s is of constant size. The lower bound on future event timestamps, ℓ, is a single timestamp, and hence also of constant size. The alpha payload p_α is either the "no change" indicator $\perp$, or an event. The symbol $\perp$ is trivially of constant size. By the induction hypothesis, events have a fixed, finite size too.

In conclusion, individual alpha tokens have a fixed, finite size. $\square$

Beta tokens We now consider beta tokens. Beta tokens take the form of a tuple $\mathcal{T}_\beta\langle \ell map, p_\beta \rangle$.

Proof. In case the beta payload p_β is a "no change" indicator $\perp$, it is trivially of a fixed size. Otherwise, both the map of lower bounds on event timestamps, ℓmap, and the map of partial matches carried as beta payload, are mappings whose domains are the same set of event identifiers. This set of event identifiers represents the event patterns whose data are represented by the beta token. By construction [3] beta tokens constructed in fwPARTE wrap a prefix of the event patterns joined by a certain rule. As a result, beta tokens that occur at any point in the graph have a known, fixed, and limited set of event patterns for which they carry data. Hence, both ℓmap and p_β have fixed number of elements.

What remains to be shown is that the elements in both mappings have a limited size. The elements in ℓmap are lower bounds on timestamps. By the induction hypothesis, these are of a finite, constant size. The elements in the beta payload are events. As established some paragraphs ago, events uphold the induction hypothesis. In conclusion, individual beta tokens have a limited size. $\square$

Lemma 24:

The size of node-local data has a finite upper bound.

The nodes in a PARTE graph can be of a number of distinct types, each with their own type of node-local data. Remember from page 132 that in Featherweight PARTE, graph

[3] Concretely, when beta tokens are created in the evaluation rules (Alpha-$\perp$), (Alpha-Match), (Join-$\searrow$-$\perp$), (Join-$\swarrow$-$\perp$), (Join-$\searrow$), (Join-$\swarrow$), (Not-$\searrow$-$\perp$), (Not-$\swarrow$-$\perp$), (Not-$\searrow$), and (Not-$\swarrow$) of the local evaluation language $\xrightarrow{l}$, as defined in section 5.4.6.

nodes can be either entry nodes, alpha nodes, join-nodes, not-nodes, production nodes, or terminal nodes. Resource usage of these node types' node-local data is as follows:

Entry node-local data takes the form $\mathcal{D}_{entry}\langle \ell map, \iota_t \rangle$, where ℓmap is a mapping from event source identifiers to lower bounds on the timestamps of events that those sources may still produce.

Proof. The set of event source identifiers that make up the domain of this mapping is fixed at the time of PARTE graph construction. Both the event source identifiers and the timestamps themselves are of fixed size by the induction hypothesis. The other constituent of entry node-local data is the template identifier, ι_t, for which I holds. Hence, entry node-local data is of a fixed, finite size. □

Alpha node-local data takes the form $\mathcal{D}_{alpha}\langle c_\alpha, \iota_e \rangle$, where c_α is a predicate function, and ι_e is an event pattern identifier.

Proof. By the induction hypothesis, both elements are of fixed, finite size at PARTE graph construction time, and hence alpha node-local data is of a fixed, finite size too. □

Join-node-local data takes the form $\mathcal{D}_{join}\langle c_\beta, \ell map, \ell, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, dist \rangle$, where c_β is a predicate function, ℓmap is a mapping from event pattern identifier to lower bound on timestamps for that (left) event pattern, ℓ is a lower bound on timestamps for the right event pattern, $\overline{pm}$ is the set of left partial matches, $\overline{e}$ is the set of right partial matches, $\overline{\iota_{e_\searrow}}$ is the set of left event pattern identifiers, $\iota_{e_\swarrow}$ is the right event pattern identifier, and *dist* contains the temporal distances between the left and right event patterns.

Proof. For c_β, ℓ, and $\iota_{e_\swarrow}$, invariant I holds by the induction hypothesis. By construction of the PARTE graph, the set of left event identifiers is finite and fixed at PARTE graph construction time. As a result, ℓmap, $\overline{\iota_{e_\searrow}}$, and *dist* are of a fixed and finite size. For $\overline{pm}$ and $\overline{e}$, the size of individual matches is fixed and finite too: they contain a fixed, finite number of events, which themselves are of fixed and finite size.

The fwPARTE model imposes limits on the partial match histories. Specifically, it restricts $|\overline{pm}|$ and $|\overline{e}|$ from outgrowing their *minimum viable sizes*, as defined in section 5.3.7. Since the *minimum viable sizes* are finite and fixed for a given node's partial match histories, join-node-local data is of a fixed, finite size too. □

Not-node-local data takes the form $\mathcal{D}_{not}\langle c_\beta, \ell map, \ell, \overline{pm}, \overline{e}, \overline{\iota_{e_\searrow}}, \iota_{e_\swarrow}, dist \rangle$, with similar fields as join-node-local data. Following a similar reasoning, we can again conclude that all fields are of a fixed, finite size at PARTE graph construction time.

Production node-local data takes the form $\mathcal{D}_{production}\langle \iota_s, g^e_\Box, g^\ell_\Box \rangle$, where ι_s is an event source identifier, $g^e_\Box$ is an event generation expression, and $g^\ell_\Box$ is a temporal lower bound generation expression.

Proof. For all three elements, invariant I holds from PARTE graph construction time. No rule in either $\xrightarrow{l}$ nor $\xrightarrow{g}$ exists which mutates them, trivially upholding I for production node-local data. □

Terminal node-local data takes the form $\mathcal{D}_{terminal}\langle\rangle$, which contains no fields and is hence trivially of a fixed, finite size.

We conclude from this study of state size of the node-local data of nodes in a fwPARTE graph that a finite upper bound exists on state size of node-local data, and that this bound is fixed at PARTE graph construction time.

Lemma 25:

The size of inboxes and outboxes has a finite upper bound.

The Featherweight PARTE model defined in section 5.4 defines nodes as tuples of the form $\mathcal{N}\langle\iota_n, \bar{\mathfrak{s}}, \overline{m}, \bar{t}, ndata\rangle$. Each node in a fwPARTE graph hence has an inbox $\overline{m}$ of messages, and an outbox $\bar{t}$ of tokens.

Proof. Tokens uphold invariant I, as established in our proof of lemma 23. By the induction hypothesis, messages — being a tuple with a token and an activation side (i.e., $\searrow$, $\downarrow$, or $\swarrow$) — have of a fixed, finite maximum size too. As with partial match histories in two-input nodes, the number of elements in inboxes and outboxes may change during evaluation of a PARTE graph, but — as with partial match histories — the fwPARTE model defines that the inboxes and outboxes may not grow beyond their *minimum viable size*. The *minimum viable sizes* are finite and fixed for a given node's inbox and outbox. This entails that the inboxes and outboxes uphold invariant I. □

8.5.3 Conclusion

It follows from lemmas 23 to 24 that the size of state captured in a fwPARTE graph is finite and fixed at PARTE graph construction time. An upper bound on the size can be computed statically, by inspecting the rule set and the *event arrival rate*, without needing to know the concrete runtime event values.

8.6 A PARTElang Program can be Evaluated in Constant Time per Event

Previous sections showed that the fwPARTE evaluation model correctly implements complex event detection in a distributed setting, with *managed storage* with constant space requirements.

The final requirement before we can claim fwPARTE enables a *strongly reactive* evaluation of PARTElang, is constant time processing of a single event.

Statement

Evaluation of a single event by Featherweight PARTE takes a constant, finite amount of time.

A fwPARTE graph consists of a finite number of fwPARTE nodes, determined at *PARTE graph construction time*. Since the graphs are also acyclic, the graph has a constant, finite depth. Because of *stratification*, evaluation of a PARTElang program requires at most a constant, finite number of descents through the fwPARTE graph for each event. It follows that a PARTElang program can be evaluated in constant time per event if and only if the processing at each node takes at most a constant, finite amount of time.

We prove in this section that that condition holds, by induction over the steps in the global and local evaluation languages of fwPARTE.

8.6.1 Local Invariant

We formalize the invariant I for the different lemmas as follows:

I — Constant-time sub-steps Each individual step in either the global evaluation language or the local evaluation language can be evaluated in constant time.

The invariant holds for all selections, filters, enumerations, and insertions on and into the sub-elements that constitute the node-local of all fwPARTE nodes, since a fixed upper bound on their sizes exists, as we proved in the previous section.

8.6.2 Proof

We again remind the reader of the evaluation rules of the global evaluation language $\xrightarrow{g}$ in section 5.4.7 on page 145, and to the evaluation rules of the local evaluation language $\xrightarrow{l}$ in section 5.4.6 from page 134 onwards.

During (Proc)

Lemma 26:

Invariant I is upheld by (Proc).

Proof. Checking the conditions of the (Proc) rule takes a number of steps. We list all conditions of the inference step of (Proc), and discuss the asymptotical number of steps — which we refer to as "time" — required to perform the check.

(i.) $\{\dots\} \sqcup \mathbf{N}_{rest}$: selecting a node from a set takes at most time proportionate to the size of the set. Since $|\mathbf{N}|$ is fixed at *PARTE graph construction time*, this check takes a constant amount of time;

(ii.) $\mathcal{N}\langle \iota_n, \overline{\mathfrak{s}}, \langle side, t\rangle \cdot \overline{m}, \overline{t}, data\rangle$: verifying whether at least one message is in the inbox, take a constant amount of time;

(iii.) $\langle data, side, t\rangle \xrightarrow{l} \langle data', \overline{t'}\rangle$: by the induction hypothesis, invariants I holds during the evaluation of $\xrightarrow{l}$;

(iv.) *has-room-for*$(\overline{t}, \left|\overline{t'}\right|)$: checking whether there is room in the outbox might take evaluation time proportionate to the size of the outbox. Since fwPARTE imposes a finite, constant upper bound on the size of the outbox, evaluation time has a finite, constant upper bound; and

(v.) $\mathbf{N}_{rest} \sqcup \{\langle \iota_n, \overline{\mathfrak{s}}, \overline{m}, \overline{t} \diamond \overline{t'}, data'\rangle\}$: appending results to the outbox takes at most time proportionate to the fixed size of the outbox. Replacing a node with a new node, in a set of fixed size $|\mathbf{N}|$, takes constant time.

It follows from $(i.)$ through $(v.)$ that invariant I hence holds during (Proc). □

During (Prop)

Lemma 27:

Invariant I is upheld by (Prop).

Proof. We repeat a similar reasoning for the parts of the inference rule (Prop):

(i.) $\{\dots\} \sqcup \mathbf{N}_b \sqcup \mathbf{N}_{rest}$: selecting a node from a set takes at most time proportionate to the size of the set. Since $|\mathbf{N}|$ is fixed at *PARTE graph construction time*, extracting a node takes a constant amount of time. Extracting $\mathbf{N}_b$, once defined, also takes constant time;

(ii.) $\mathcal{N}\langle \iota_a, \overline{\mathfrak{s}_a}, \overline{m_a}, t \cdot \overline{t_a}, data_a\rangle$: verifying whether at least one token is present in the outbox of the selected node take a constant amount of time;

(iii.) $\mathbf{N}_b = \mathbf{N} \cap \{\mathcal{N}\langle \iota_b, \_, \_, \_, \_\rangle \mid \langle \iota_b, \_\rangle \in \overline{\mathfrak{s}_a}\}$: selecting all nodes whose node identifier occurs in the list of successors of node a, takes at most time proportionate in the number of nodes, and the number of successors of a. Both of these are limited by the number of existing nodes, which is finite and fixed at *PARTE graph construction time*;

(iv.) $\forall \mathcal{N}\langle \_, \_, \overline{m_b}, \_, \_\rangle \in \mathbf{N}_b : \textit{has-room-for}(\overline{m_b}, 1)$: verifying whether all nodes in $\mathbf{N}_b$ have room in their inbox, takes time proportionate in the size of $\mathbf{N}_b$, which is constant;

(v.) $\mathbf{N}'_b = \{\mathcal{N}\langle \iota_b, \overline{\mathfrak{s}_b}, \overline{m_b} \cdot \langle side, t\rangle, \overline{t_b}, data_b\rangle \mid \ldots \in \mathbf{N}_b, \langle \iota_b, side\rangle \in \overline{\mathfrak{s}_a}\}$: appending messages to an inbox takes at most time proportionate to the maximum size of the inbox. Selecting nodes from $\mathbf{N}_b$ takes at most time proportionate to $|\mathbf{N}_b|$; and

(vi.) $\{\mathcal{N}\langle \iota_a, \overline{\mathfrak{s}_a}, \overline{m_a}, \overline{t_a}, data_a\rangle\} \sqcup \mathbf{N}'_b \sqcup \mathbf{N}_{rest}$: the union of a singleton set, $\mathbf{N}'_b$, and $\mathbf{N}_{rest}$ is by construction limited by the size of $\mathbf{N}$, which is finite and constant.

The fwPARTE model assumes that a constant upper bound exists on the time it takes to send a message to another node. Failure to do so within that time frame equates failing a hard deadline, i.e., a fatal error necessitating a shutdown of the system. Within the abstraction used by fwPARTE, it follows from $(i.)$ through $(vi.)$ that invariant I holds during (Prop). □

During Evaluation at Entry Nodes

Lemma 28:

Invariant I is upheld by (Entry-NoMatchTemplate), (Entry), and (Entry-$\bot$).

Proof. For the inference rules of $\xrightarrow{l}$, a similar reasoning holds:

(i.) $\langle \mathcal{D}_{entry}\langle s\ell map, \iota_{tn}\rangle, \downarrow, \mathcal{T}_\alpha\langle \iota_s, \ell, e\rangle\rangle$ and $\langle \mathcal{D}_{entry}\langle s\ell map, \iota_{tn}\rangle, \downarrow, \mathcal{T}_\alpha\langle \iota_s, \ell, \bot\rangle\rangle$: apart from determining whether the alpha token contains an event or a "no change" indicator, the precondition holds trivially, so takes constant time to evaluate;

(ii.) $e = \mathcal{E}\langle \iota_{te}, \_, \_\rangle$: extracting the template identifier from an event trivially takes constant time;

(iii.) $\iota_{tn} = \iota_{te}$ and $\iota_{tn} \neq \iota_{te}$: comparing two identifiers for equality trivially takes constant time;

(iv.) $s \in dom(s\ell map)$, $s\ell map[s] \leq \ell$, and $s\ell map[s] \leq e.\theta$: lookups in the source-identifier-to-lower-bound mapping take time proportionate to the size of the source-identifier-to-lower-bound mapping, which has a size limited by the number of sources. The number of sources is finite and fixed at *PARTE graph construction time*, providing a finite, constant bound on the lookup time;

(v.) $s\ell map' = s\ell map[\iota_s \mapsto \ell]$: overwriting an entry in the source-identifier-to-lower-bound mapping is constant for the same reason;

(vi.) $\overline{t_\alpha} = \{\mathcal{T}_\alpha\langle\iota_s, \min(codom(s\ell map')), e\rangle\}$ and $\overline{t_\alpha} = \{\mathcal{T}_\alpha\langle\iota_s, \min(codom(s\ell map')), \bot\rangle\}$: selecting the minimum from a finite, constant-sized mapping take finite, constant time. Constructing a singleton-set takes a finite, constant amount of time; and

(vii.) $\langle\mathcal{D}_{entry}\langle s\ell map', \iota_{tn}\rangle, \emptyset\rangle$ and $\langle\mathcal{D}_{entry}\langle s\ell map', \iota_{tn}\rangle, \overline{t_\alpha}\rangle$: the postcondition trivially takes a constant amount of time.

It follows from (*i.*) through (*vii.*) that invariant *I* holds throughout the steps of the local evaluation language at entry nodes. □

During Evaluation at Alpha Nodes

Lemma 29:

Invariant *I* is upheld by (Alpha-⊥), (Alpha-Match), and (Alpha-NoMatchConstraint).

Proof.

(i.) $\langle\mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle, \downarrow, \mathcal{T}_\alpha\langle\_, \ell, \bot\rangle\rangle$ and $\langle\mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle, \downarrow, \mathcal{T}_\alpha\langle s, \ell, e\rangle\rangle$: apart from determining whether the alpha token contains a "no change" indicator or an event, the precondition holds trivially, so takes constant time to evaluate;

(ii.) $c_\alpha(e) \overset{\alpha-test}{\longrightarrow} false$ and $c_\alpha(e) \overset{\alpha-test}{\longrightarrow} true$: evaluating an alpha test takes constant time by construction: the alpha test is of fixed, finite depth at *PARTE graph construction time*, and consists only of attribute accesses and trivial binary operations; and

(iii.) $\langle\mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle, \{\mathcal{T}_\beta\langle[\iota_e \mapsto \ell], \bot\rangle\}\rangle$, $\langle\mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle, \emptyset\rangle$, and $\langle\mathcal{D}_{alpha}\langle c_\alpha, \iota_e\rangle, \{\mathcal{T}_\beta\langle[\iota_e \mapsto \ell], [\iota_e \mapsto e]\rangle\}\rangle$: constructing a singleton beta token and a result-tuple trivially takes constant time.

It follows from (*i.*) through (*iii.*) that invariant *I* holds throughout the steps of the local evaluation language at alpha nodes. □

During Evaluation at Join-Nodes

Lemma 30:

Invariant *I* is upheld by (Join-↘-⊥), (Join-↙-⊥), (Join-↘), and (Join-↙).

Proof.

(i.) $\langle \mathcal{D}_{join}\langle\ldots\rangle, \searrow, \mathcal{T}_\beta\langle \ell map_{t_\beta}, \bot\rangle\rangle$ and $\langle \mathcal{D}_{join}\langle\ldots\rangle, \swarrow, \mathcal{T}_\beta\langle [\iota_{e_\swarrow} \mapsto \ell_{t_\beta}], \bot\rangle\rangle$: determining whether a "no change" token left-activated or right-activated a join-node trivially takes constant time;

(ii.) $\langle \mathcal{D}_{join}\langle\ldots\rangle, \searrow, \mathcal{T}_\beta\langle \ell map_{t_\beta}, pm_{t_\beta}\rangle\rangle$ and $\langle \mathcal{D}_{join}\langle\ldots\rangle, \swarrow, \mathcal{T}_\beta\langle [\iota_{e_\swarrow} \mapsto \ell_{t_\beta}], [\iota_{e_\swarrow} \mapsto e_{t_\beta}]\rangle\rangle$: determining whether an event-carrying token left-activated or right-activated a join-node trivially takes constant time. Destructuring a tuple — in the case of a right-activation — takes constant time as well, as the mapping is a singleton mapping;

(iii.) $\langle \mathcal{D}_{join}\langle \ldots, \ell map_{t_\beta}, \ell_{t_\beta}, \overline{pm}', \overline{e}', \ldots\rangle, \{\ldots\}\rangle$: replacing individual elements in a tuple is a constant-time operation;

(iv.) $\overline{e}' = expire_\swarrow(\overline{e}, expiration\text{-}time(dist, \ell map_{t_\beta}))$: by the induction hypothesis, I holds for ℓmap_{t_β}, therefore the *expiration-time* can be determined in constant time. The run time of *expire* is proportional in the size of its first argument. By the induction hypothesis, I holds for $\overline{e}$. Hence, $\overline{e}'$ can be constructed in constant time;

(v.) $\overline{pm}' = expire_\searrow(\overline{pm}, \ell_{t_\beta})$: analogously to (*iv.*), I holds for $\overline{pm}$, hence $\overline{pm}'$ can be constructed in constant time;

(vi.) $has\text{-}room\text{-}for(\overline{pm}, 1)$ and $has\text{-}room\text{-}for(\overline{e}, 1)$: by the induction hypothesis, I holds for $\overline{pm}$ and $\overline{e}$, rendering capacity-checks for both vectors constant-time operations;

(vii.) $\overline{pm}' = \overline{pm} \cdot pm_{t_\beta}$ and $\overline{e}' = \overline{e} \cdot e_{t_\beta}$: by the induction hypothesis, I holds for $\overline{pm}$ and $\overline{e}$, rendering insertion into both vectors constant-time operations;

(viii.) $\ell map_{\text{implicit}} = min(\overline{timestamps}(\overline{pm}^{(\prime)}))[\iota_{e_\swarrow} \mapsto min(timestamps(\overline{e}^{(\prime)}))]$: selecting the minimum from the combination of two vectors for which I holds, is a constant-time operation;

(ix.) $lifetime(dist, timestamps(pm_{t_\beta}))$ and $lifetime(dist, timestamp(e_{t_\beta})$: by the induction hypothesis, all selections on events and partial matches are constant-time operations;

(x.) $window_\swarrow = semantic\text{-}window_\swarrow(\overline{e}', \ldots)$ and the induction hypothesis, all selections on partial match histories are constant-time operations;

(xi.) $\overline{pm_{\text{new}}} = \{pm_{\ldots} \cdot pm_{\ldots} \mid pm_\searrow \in window_{\ldots}, c_\beta(pm_{\ldots} \cdot pm_{\ldots}) \xrightarrow{\beta} \text{true}\}$: appending a singleton set to a set for which I holds, is a constant-time operation. The run time of filtering a vector by a constant-time boolean predicate is proportionate to the size of the vector;

(xii.) $\ell map_{\text{prop}} = min(\ell map_{\text{implicit}}, \ell map_{\ldots}[\iota_{e_{\swarrow}} \mapsto \ell^{(\prime)}_{\swarrow}])$: as in step $(viii.)$, selecting the minimum from two vectors for which I holds, takes constant time;

(xiii.) $\{\mathcal{T}_\beta\langle \ell map_{\text{prop}}, \bot\rangle\}$: creating a "no change" beta token takes constant time; and

(xiv.) $\{\mathcal{T}_\beta\langle \ell map_{\text{prop}}, pm\rangle \mid pm \in \overline{pm_{\text{new}}}\}$: enumerating and transforming a vector adhering to I takes constant time. By $(xi.)$, I holds for $\overline{pm_{\text{new}}}$.

It follows from $(i.)$ through $(xiv.)$ that invariant I holds throughout the evaluation steps of $\xrightarrow{l}$ at join-nodes. □

During Evaluation at Not-Nodes

Lemma 31:

Invariant I is upheld by (Not-$\searrow$-$\bot$), (Not-$\swarrow$-$\bot$), (Not-$\searrow$-Match), (Not-$\searrow$-NoMatch), and (Not-$\swarrow$).

Evaluation at node nodes proceeds similarly. The proof is hence mostly analogous, though operating on $\mathcal{D}_{not}\langle\ldots\rangle$ instead of $\mathcal{D}_{join}\langle\ldots\rangle$. The only step deviating significantly from the ones at join-nodes, is the following:

Proof.

(i.) $expire\text{-}negated_{\searrow}(\overline{pm}', \ell_{t_\beta})$: by its definition in figure 5.5, $expire\text{-}negated_{\searrow}$ takes time proportionate to the size of its first argument. By the induction hypothesis, I holds for $\overline{pm}'$. The negated expiration hence takes a finite, constant amount of time, i.e., I holds for $expire\text{-}negated_{\searrow}(\overline{pm}', \ell_{t_\beta})$.

It follows that invariant I holds throughout the evaluation steps of $\xrightarrow{l}$ at not-nodes. □

During Evaluation at Production Nodes

Lemma 32:

Invariant I is upheld by (Prod-$\bot$) and (Prod).

Proof.

(i.) $\langle \mathcal{D}_{production}\langle \iota_s, g^e_\square, g^\ell_\square\rangle, \downarrow, \mathcal{T}_\beta\langle \ell map_{t_\beta}, \ldots\rangle\rangle$: detecting whether a token exists in a production nodes' inbox can trivially be done in constant time;

(ii.) $g^{e}_{\square}(pm) \longrightarrow e'$: the event generator expression can by the induction hypothesis be evaluated in constant time on a partial match;

(iii.) $g^{\ell}_{\square}(\ell map_{t_\beta}) \longrightarrow \ell$: the lower-bound generator expression can by the induction hypothesis be evaluated in constant time on a lower-bound mapping; and

(iv.) $\langle \mathcal{D}_{production}\langle \iota_s, g^{e}_{\square}, g^{\ell}_{\square}\rangle, \{\mathcal{T}_\alpha\langle \iota_s, \ell, \ldots\rangle\}\rangle$: an alpha token can be generated in constant time.

It follows from $(i.)$ through $(iv.)$ that the invariant I holds throughout the evaluation steps of $\xrightarrow{I}$ at production nodes. □

During Evaluation at Terminal Nodes

Lemma 33:

Invariant I is upheld by (TERM).

Proof. At terminal nodes, a single step is taken. This trivially takes constant time. □

8.6.3 Conclusion

We can now prove the original statement from page 227: evaluation of a single event by Featherweight PARTE takes a constant, finite amount of time:

Proof.

(i.) Per chapter 5, a fwPARTE graph is stratified (as defined in definition 3 on page 67) and hence acyclic.

(ii.) Trivially (by construction and by the percolation in (PROC)), evaluation in a fwPARTE graph proceeds strictly downwards.

(iii.) From $(i.)$ and $(ii.)$, evaluation of an event can percolate through the fwPARTE graph at most once.

(iv.) From section 8.5, a fwPARTE graph has a static, finite number of nodes.

(v.) Per lemmas 26 to 33, evaluation takes a constant time at each sub-step in the evaluation in a fwPARTE graph.

It follows from $(iii.)$, $(iv.)$, and $(v.)$ that evaluation takes a static, finite amount of time. □

8.7 Conclusion

This chapter presented the proofs that the Featherweight PARTE model guarantees a strongly reactive evaluation strategy for PARTElang programs. The constant-space and constant-time processing of a single event, combined with a known *event arrival rate*, and a finite depth of the graph (due to the constant, finite depth of an individual PARTE graph, coupled with *stratification*) make it possible to statically determine whether a certain implementation of the fwPARTE model on a physical machine will be able to keep up with a certain stream of events.

The Featherweight PARTE model hence makes it possible to build a verifiably *strongly reactive* language runtime for the PARTElang logic reactive programming language. This entails that PARTElang programs can be deployed in scenarios that require strong guarantees that the system can keep up with incoming event streams, for instance in security monitoring contexts.

9 Conclusion

9.1 Revisiting the Problem Statement

There are many use cases for monitoring large streams of events for occurrences of specific patterns. We explored three specific driver scenarios in the context of monitoring financial transactions for possible cases of fraud. In scenarios such as these, there exists a clear need to process the events immediately: the suspected fraud must be detected as soon as it occurs.

A large research domain is dedicated to providing the tools to detect *complex events*, i.e., events which exist because of the occurrence of multiple sub-events. Similarly, many systems are developed for this purpose by the industry. However, in our driver scenarios, the additional constraint exists that the pattern detection system cannot cease to monitor, even momentarily. In section 1.1 we formulated the problem statement as follows:

Problem Statement

There is a need for platforms which autonomously process complex events in high-throughput streams of events, incrementally in a data-driven, always-listening, *strongly reactive* manner.

We defined what it means to be *strongly reactive* on page 65.

We surveyed the state of the art in event stream processing systems to determine which solutions to that problem already exist.

Since our driver scenarios require scalable, high-throughput solutions, we first surveyed systems which aim to tackle high loads (chapter 2). We identified two main categories of software systems for processing large-scale event streams: Big Data stream processing, and streaming databases. The former lacks support for matching complex event patterns in the monitored streams. The latter lacks the means of enforcing strong reactivity.

We therefore surveyed the remainder of the state of the art of event handling systems (chapter 3). We identified two more categories of software systems which offered features important for our driver scenarios. First, (Functional) Reactive Programming languages enable processing event streams by means of a predefined, constant-depth dependency graph. Second, Complex Event Processing languages enable incrementally matching complex event patterns to event streams. However, FRP lacks support for matching complex event patterns, and CEP lacks guaranteed strong reactivity.

9.2 Revisiting our Contributions

9.2.1 The Logic Reactive Programming Paradigm

In chapter 4 we introduced a novel programming paradigm which solves the problem stated in section 1.1. This novel paradigm — Logic Reactive Programming — enforces 5 requirements:

R_{LRP}1 LRP languages provide the means for **declaratively expressing complex event patterns** and the **reaction logic** to execute in response to matches to those patterns.

R_{LRP}2 LRP languages offer built-in support for **temporal reasoning**, such that the complex event patterns can express how the constituent events relate to each other in time.

R_{LRP}3 LRP languages make use of the constraints specified in the declarative event patterns to **automatically manage** which event data is stored, and how.

R_{LRP}4 LRP languages offer **scalable, incremental matching** of the complex event patterns.

R_{LRP}5 LRP languages are guaranteed to offer **strongly reactive** pattern matching.

Table 9.1 depicts how each of the four categories of the state of the art of event stream processing achieves some of those requirements. Only Logic Reactive Programming achieves all five.

	Distributed Big Data & Stream Processing (Chapter 2)		Event Handling (Chapter 3)		Our Contribution
	Big Data Stream Processing	Streaming Databases	Complex Event Processing	Reactive Programming	Logic Reactive Programming
$R_{\text{LRP}}1$	– no patterns – Callback Hell	✓ relational join – outside of the model	✓✓ expressive patterns – Callback Hell	– no patterns ✓✓ well-supported	✓ restricted patterns ✓ supported (restricted)
$R_{\text{LRP}}2$	– unsupported	✓ supported	✓✓ well-supported	– unsupported	✓ supported
$R_{\text{LRP}}3$	~ no storage	✓✓ large managed storage	✓ managed storage	~ no storage	✓ managed storage
$R_{\text{LRP}}4$	✓✓ very scalable	✓ scalable	– not scalable	– not scalable	✓ scalable
$R_{\text{LRP}}5$	~ not guaranteed	– unsupported	– unsupported	~ not guaranteed	✓ guaranteed

Table 9.1: Requirements attained by the state of the art of event stream processing, and by our contribution: Logic Reactive Programming

9.2.2 PARTElang: a Logic Reactive Programming Language

We developed a first Logic Reactive Programming language: PARTElang (section 4.2). PARTElang programs take the form of sets of declarative rules, encoding how a set of declarative event patterns can be abstracted into new complex events. The formal foundations of PARTElang are twofold. First, the semantics of what constitutes a match to an event pattern are specified by an *event algebra* (section 4.3). Second, the way in which PARTElang programs can be evaluated in a strongly reactive manner is defined by an operational semantics: the Featherweight PARTE model (section 5.4).

9.2.3 Featherweight PARTE: an Operational Semantics for PARTElang

The Featherweight PARTE model defines how a PARTElang program can be compiled into a set of processing nodes. Additionally, the model defines two sets of rewrite rules: a global evaluation language which defines how event data spreads through the system, and a local evaluation language which defines how event data is processed by individual processing nodes. We formally prove that the Featherweight PARTE model guarantees strong reactivity: for any PARTElang program, a fixed, finite *minimum viable size* exists for all places where data is stored in a fwPARTE graph. The minimum viable size is the smallest size for which fwPARTE can guarantee to successfully detect all occurrences of the complex events specified in the PARTElang program. Since the minimum viable size is fixed and finite, the cost of processing an event using the Featherweight PARTE model is constant for a given PARTElang program (chapter 8).

9.2.4 Two Prototypical Implementations of Featherweight PARTE

We developed two prototypical implementations of the Featherweight PARTE model (chapter 6). The first — PARTE$^{\text{Rust}}$ — is a multi-threaded testbed for the model. The second — PARTE$^{\text{Elixir}}$ — schedules the sequential components of PARTE$^{\text{Rust}}$ on an Erlang actor runtime.

To demonstrate how PARTElang can be applied to a real-world use case, we implemented our driver scenarios in full in chapter 7. We experimentally validated the utility of Featherweight PARTE as an execution model: we implement our driver scenarios using a state of the art Big Data stream processing framework, and show that the idiomatic way of using it either precludes strong reactivity, or is unable to operate within the bounds of a minimum viable size.

9.3 Revisiting the Limitations and Future Work

Logic Reactive Programming provides a way of declaratively defining software systems which autonomously monitor high-throughput streams of events in a strongly reactive manner. Four major limitations offer opportunities for future investigation:

More flexible temporal constraints PARTElang and its event algebra EA only support temporal constraints which define a *closed time interval*. This enables strongly reactive evaluation of PARTElang programs. However, there exist other temporal constraints which could be made compatible with strong reactivity.

Future work could explore to what extent this limitation can safely be relaxed. In general, any temporal constraint which adds only a constant run time cost per event, and only a constant space cost per program, can be made compatible with strong reactivity. We describe two possible avenues in section 4.5.1 (on page 86).

Support for aggregation PARTElang and its event algebra EA can only reason about the single event matching an event pattern, not about *the set* of all events which match an event pattern. PARTElang programs hence cannot express patterns involving a variable number of events.

Future work could explore to what extent this limitation can safely be relaxed. As we indicate in section 4.5.2 (on page 86), general aggregation is incompatible with strong reactivity [101], but a restricted form of aggregation can be made compatible. Since semantic windows in PARTE have a fixed, finite upper bound on size, operations with costs proportionate to their size, are compatible with strong reactivity. The addition of support for aggregation over semantic windows is hence mostly a matter of engineering effort.

Strongly reactive failure handling Featherweight PARTE defines an operational semantics which assumes a distributed memory space. However, safely distributing computations over multiple computers, e.g., in a computer cluster, introduces other complexities on top of the distributed memory space. One important issue in distributed computing is dealing with *partial failures*. The Featherweight PARTE model lacks the capability to do so. While the guarantees of strong reactivity and the restricted, declarative language prevent runtime software failures, the possibility of hardware failures persists.

Future work could explore what is required for strongly reactive failure handling, or the extent to which that is possible. As we pointed out in section 5.6.4 (on page 153), we expect that a maximum *failure occurrence rate* will have to be defined to enable this. In contrast to our previous point — on support for aggregation — the addition of strongly reactive failure handling is not just an engineering effort. New formal work will be required.

Language and runtime engineering PARTElang is purposefully built as a minimal language. Similarly, the Featherweight PARTE model is kept "featherweight". Our prototypical implementations of the model, in turn, are evidently just research prototypes. All this contributes to the situation that PARTE lacks some marked features. In section 4.5.3 (on page 87) we listed some language features which are missing from PARTElang, but which would improve the *software engineering aspects* of writing PARTElang programs. Similarly, in section 5.6.5 (on page 153) we listed some *optimizations* which are not defined by our formal model, but which would reduce the average-case execution time.

Future work could explore adding Midas-style [121, 74] language features, most notably support for reusability through modules. Additionally, the optimizations described from page 153 onwards could be added to the model and prototypes. New theoretical work will be necessary to explore how the granularity of a fwPARTE graph can be coarsened, and to determine how multiple results can be batched in such a way that strongly reactive processing remains guaranteed.

9.4 Closing Remarks

In this dissertation, we presented Logic Reactive Programming, a paradigm for programs which monitor high-throughput event streams for matches to complex event patterns. LRP programs are characterized by *1.)* a declarative specification of complex event patterns and reaction logic, *2.)* built-in temporal reasoning, *3.)* automatic management of event data, *4.)* scalable online processing, and *5.)* strong reactivity. Before LRP, there was no event stream processing system which combined these desirable properties.

While a large focus of this dissertation was on the formal, semantical aspects of the Logic Reactive Programming paradigm, we explored the full spectrum from event algebra, through formally proven operational semantics, to practical implementations of real-world driver scenarios.

In conclusion, Logic Reactive Programming presents a significant improvement over the state of the art. We believe that this combination of expressive, declarative complex event patterns with guaranteed strong reactivity constitutes an important new research avenue.

A
Additional Code Snippets

A.1 Additional PARTElang Snippets

We remind the reader that the complete implementation of the driver scenarios in PARTElang is available at `https://soft.vub.ac.be/~trenaux/PARTE/implemented-driver-scenarios.zip`. Some relevant snippets, referred to from the text, are reproduced below:

```
rule AbstractSuspiciousTransfer where
  incoming : MoneyTransferred {
    amount >= 100.0,
    uncommon_route(originator, destination)
  }
when
then
  emit MoneyTransferredSuspiciously
    at incoming.timestamp
    with {
      originator = incoming.originator,
      destination = incoming.destination,
      amount = incoming.amount,
    }
```

Listing A.1: The PARTElang rule implementing the abstraction that reduces code duplication in the first two driver scenarios: an occurrence of `MoneyTransferred` which is suspicious is abstracted into a `MoneyTransferredSuspiciously` compound event.

```
rule FraudPassThrough where
  incoming : MoneyTransferredSuspiciously
  outgoing : MoneyTransferredSuspiciously {
    originator = incoming.destination,
    amount = incoming.amount
  }
when
  outgoing in incoming [0 days, 14 days]
then
  emit SuspiciousSequenceDetected
    at incoming.timestamp
    with {
      mule_account = incoming.destination,
      amount = incoming.amount,
      incoming_transactions = list(incoming.id),
      outgoing_transactions = list(outgoing.id)
    }

rule FraudDiffusion where
  incoming : MoneyTransferredSuspiciously
  outgoing_1 : MoneyTransferredSuspiciously {
    originator = incoming.destination,
  }
  outgoing_2 : MoneyTransferredSuspiciously {
    originator = incoming.destination,
    id != outgoing_1.id,
  }
  outgoing_3 : MoneyTransferredSuspiciously {
    originator = incoming.destination,
    id != outgoing_1.id,
    id != outgoing_2.id,
    outgoing_1.amount + outgoing_2.amount + outgoing_3.amount >= incoming.amount * 0.9,
    outgoing_1.amount + outgoing_2.amount + outgoing_3.amount <= incoming.amount * 1.1,
  }
when
  outgoing_1 in incoming [0 days, 14 days]
  outgoing_2 in incoming [0 days, 14 days]
  outgoing_3 in incoming [0 days, 14 days]
then
  emit SuspiciousSequenceDetected
    at incoming.timestamp
    with {
      mule_account = incoming.destination,
      amount = incoming.amount,
      incoming_transactions = list(incoming.id),
      outgoing_transactions = list(outgoing_1.id, outgoing_2.id, outgoing_3.id)
    }
```

Listing A.2: Alternative PARTElang rules implementing driver scenario A and driver scenario B: where a suspicious transfer is defined as a transfer of 100 € or more along an uncommon route.

A.2 Additional Snippets of the PARTE^{Rust} Implementation

Some relevant snippets — referred to from the text — of PARTE^{Rust}, our prototypical implementation of the Featherweight PARTE model, are reproduced below:

```rust
#[derive(Clone, Debug)]
pub struct BetaToken {
    oldest_timestamp_per_pattern: Vec<Time>,
    payload: Option<PartialMatch>,
}

impl BetaToken {
    pub fn new_from_single_event(oldest_timestamp_to_expect: Time,
                                 event: Arc<Event>,
                                 ) -> Self {
        BetaToken {
            oldest_timestamp_per_pattern: vec![oldest_timestamp_to_expect],
            payload: Some(PartialMatch { events: vec![event] })
        }
    }

    pub fn new_from_events(oldest_timestamp_per_pattern: &[Time],
                           events: Vec<Arc<Event>>,
                           ) -> Self {
        assert!(!events.is_empty());
        assert_eq!(oldest_timestamp_per_pattern.len(), events.len());

        BetaToken {
            oldest_timestamp_per_pattern: Vec::from(oldest_timestamp_per_pattern),
            payload: Some(PartialMatch { events: events })
        }
    }

    pub fn new_from_single_no_change(oldest_timestamp_to_expect: Time) -> Self {
        BetaToken {
            oldest_timestamp_per_pattern: vec![oldest_timestamp_to_expect],
            payload: None,
        }
    }

    pub fn new_by_extending_no_change(oldest_timestamp_per_pattern: Vec<Time>) -> Self {
        BetaToken {
            oldest_timestamp_per_pattern: oldest_timestamp_per_pattern,
            payload: None
        }
    }

    // ... accessors and mutators
}
```

Listing A.3: PARTE$^{\text{Rust}}$'s implementation of beta tokens

A.3 Additional Snippets of the Event Trace

We remind the reader that a complete version of the event trace generator used in chapter 7 is available at `https://soft.vub.ac.be/~trenaux/PARTE/evaluation.zip`. Some relevant snippets, referred to from the text, are reproduced below:

```
// With field left_storage: LeftPartialMatchStorage,
// With field right_storage: RightPartialMatchStorage,
// With field oldest_timestamp_per_left_event_pattern: Vec<Time>,
// With field temporal_distance_to_reference_events: Vec<TemporalDistance>,

impl JoinNode {
  // .. constructor etc.

  fn process_left_activation(&mut self, token: BetaToken) -> Result<(), String> {
    // ... check for out-of-order propagation, or wrong update of the lower bounds

    let (oldest_timestamp_per_left_event_pattern, payload) = token.into_parts();

    // Update the timing info and discard stale matches from the right token storage
    self.oldest_timestamp_per_left_event_pattern
      = oldest_timestamp_per_left_event_pattern;
    self.discard_stale_matches_from_right_storage();

    let ls = self.compute_lower_bound_on_event_times();

    if let Some(left_partial_match) = payload {
      // Compute lifetime for new token by intersecting lifetimes of each event pattern.
      let left_lifetime = self
        .temporal_distance_to_reference_events.iter()
        .zip(left_partial_match.events.iter().map(|e| e.timestamp()))
        .fold(Lifetime::unrestricted(), |lifetime, (distance, e_time)| {
          Lifetime::from_distance_and_absolute_time(distance, e_time)
            .intersection(&lifetime)
        });

      // For every partial match in the right storage, try to unify:
      self.right_storage
        .attempt_combining_with_stored(
          &left_partial_match,
          &left_lifetime,
          &mut |left_pm, right_pm| {
            let es = left_pm.new_by_joining_assuming_consistent(right_pm).events;
            let outgoing_token = BetaToken::new_from_events(&ls, es);
            self.successors.send_token(outgoing_token)
          })?;

      // Store the incoming partial match in the left storage
      self.left_storage.insert_match(left_lifetime, left_partial_match);

      Ok(())
    } else {
      let outgoing_token = BetaToken::new_by_extending_no_change(ls);
      self.successors.send_token(outgoing_token)
    }
  }

  fn discard_stale_matches_from_right_storage(&mut self) {
    // Compute the lifetime of the 'oldest event' we can still expect on the left, by
    // intersecting the lifetimes for of the 'oldest event' for each event pattern.
    let lifetime = self.temporal_distance_to_reference_events.iter()
      .zip(self.oldest_timestamp_per_left_event_pattern.iter())
      .fold(Lifetime::unrestricted(), |lifetime, (distance, &oldest_time)| {
        Lifetime::from_distance_and_absolute_time(distance, oldest_time)
          .intersection(&lifetime)
      });

    self.right_storage.discard_older_than(lifetime.start);
  }
}
```

Listing A.4: Processing left activation of join-nodes in PARTE$^{\text{Rust}}$

```rust
type ValueProcedure = Box<for<'a> Fn(&'a Iterator<Item=&Value>) -> Value + Send>;
type TimestampProcedure = Box<for<'a> Fn(&'a Iterator<Item=Time>) -> Time + Send>;
type ActionProcedure = Box<for<'a> Fn(&'a PartialMatch) -> Result<(), String> + Send>;

pub enum Action {
  EmitAlphaToken {
    entry_nodes: AlphaSuccessorCollection,
    template: Arc<Template>,
    value_procedures: Vec<(ValueProcedure, Vec<QualifiedSlot>)>,
    inherited_timestamp_procedure: TimestampProcedure,
  },
  CallForeignAction ( ActionProcedure )
}

impl Action {
  fn with_partial_match(&self,
                        generator_id: GeneratorId,
                        oldest_timestamp_per_event_pattern: &[Time],
                        pm: &PartialMatch
  ) -> Result<(), String> {
    match self {
      &Action::EmitAlphaToken { ref entry_nodes,
                                ref template,
                                ref value_procedures,
                                ref inherited_timestamp_procedure } => {
        let vals = value_procedures.iter()
          .map(|&(ref procedure, ref selected_slots)| {
            procedure(&selected_slots.iter().map(|qs| &pm[qs]))
          }).collect(),

        let times = pm.events.iter().map(Event::timestamp);
        let times = inherited_timestamp_procedure(&times);

        let oldest_times = oldest_timestamp_per_event_pattern.iter().cloned();
        let oldest_times = inherited_timestamp_procedure(&oldest_times);

        let outgoing_token = AlphaToken::new_with_event(
          generator_id,
          oldest_times,
          Arc::new(Event::new(template, vals, times)?));
        entry_nodes.send_token(outgoing_token)
      },
      &Action::CallForeignAction ( ref action ) => {
        action(pm)
      }
    }
  }

  fn with_no_change(&self,
                    generator_id: GeneratorId,
                    oldest_timestamp_per_event_pattern: &[Time],
  ) -> Result<(), String> {
    match self {
      &Action::EmitAlphaToken { ref entry_nodes,
                                ref inherited_timestamp_procedure, .. } => {
        let ts = oldest_timestamp_per_event_pattern.iter().cloned();
        let ts = inherited_timestamp_procedure(&ts);
        let outgoing_token = AlphaToken::new_no_change(generator_id, ts);
        entry_nodes.send_token(outgoing_token)
      },
      &Action::CallForeignAction { .. } => {
        Ok(()) // Nothing to do, foreign actions don't interact with "no change"
      }
    }
  }
}
```

Listing A.5: Representing reaction logic in PARTE$^{\text{Rust}}$

```rust
let amount_ea = if is_above_threshold {
  rng.gen_range(34, 999)
} else {
  rng.gen_range(1, 33)
};

let (in_acc, shared_acc, out_acc_a, out_acc_b, out_acc_c)
  = unique_account_id_set_from_seed(*next_id, is_common_route);

use rand::distributions::{Distribution, Uniform};
let distribution_of_outgoing_time
  = Uniform::from(start_time .. start_time + MAX_GROUP_DURATION);
let time_a = distribution_of_outgoing_time.sample(rng);
let time_b = distribution_of_outgoing_time.sample(rng);
let time_c = distribution_of_outgoing_time.sample(rng);

{ // Ensure it's safe to use these events w.r.t. the `EVENT_RATE`
  let mut times_of_new_events = BTreeMap::new();
  for time in &[start_time, time_a, time_b, time_c] {
    *times_of_new_events.entry(*time).or_insert(0) += 1;
  }

  for (time, count) in times_of_new_events {
    if (all_events.get(&time).unwrap_or_default().iter().count() + count) > EVENT_RATE {
      // This random sample failed: would produce too many events in a certain timeslot
      return Err(());
    }
  }
}

let i = Event::new(template_transferred,
                   vec![ Value::from(*next_id + 0),
                         Value::from(in_acc),
                         Value::from(shared_acc),
                         Value::from(amount_ea * 3)],
                   Time::from_milliseconds(start_time as f64)).unwrap();
let o_a = Event::new(template_transferred,
                     vec![ Value::from(*next_id + 1),
                           Value::from(shared_acc),
                           Value::from(out_acc_a),
                           Value::from(amount_ea)],
                     Time::from_milliseconds(time_a as f64)).unwrap();
let o_b = Event::new(template_transferred,
                     vec![ Value::from(*next_id + 2),
                           Value::from(shared_acc),
                           Value::from(out_acc_b),
                           Value::from(amount_ea)],
                     Time::from_milliseconds(time_b as f64)).unwrap();
let o_c = Event::new(template_transferred,
                     vec![ Value::from(*next_id + 3),
                           Value::from(shared_acc),
                           Value::from(out_acc_c),
                           Value::from(amount_ea)],
                     Time::from_milliseconds(time_c as f64)).unwrap();

*next_id += 4;

for (time, event) in vec![(start_time, i), (time_a, o_a), (time_b, o_b), (time_c, o_c)] {
  all_events.entry(time).or_insert_default().push(event);
}
```

Listing A.6: Rust code generating a new three-way split given a configuration

B
Experimental Results

This appendix summarizes the results obtained in our experiment described in section 7.4 from page 199 onwards. The concrete results can be acquired from `https://soft.vub.ac.be/~trenaux/PARTE/evaluation.zip`. For each scenario, a fraction of less than 1 for the 100% percentile indicates that the Spark Streaming implementation missed some results at that scale factor. A fraction above 1 would indicate incorrect matches were found. Results for PARTE$^{\text{Rust}}$ are not included here, as PARTE$^{\text{Rust}}$ detected all results (i.e., a fraction of "1.0") for each scenario.

Scenario		Fraction of fraud cases found per percentiles						
# unrelated	Scale factor	0%	5%	25%	50%	75%	95%	100%
0	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	3	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	4	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	6	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	7	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	9	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Scenario		Fraction of fraud cases found per percentiles						
# unrelated	Scale factor	0%	5%	25%	50%	75%	95%	100%
100	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	3	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	4	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	6	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	7	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	9	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	3	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	4	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	6	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	7	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	9	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	1	0.0	0.0	0.17	0.17	0.33	0.5	0.83
10000	2	0.0	0.17	0.33	0.5	0.67	0.83	1.0
10000	3	0.17	0.5	0.67	0.83	1.0	1.0	1.0
10000	4	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	6	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	7	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	9	0.0	1.0	1.0	1.0	1.0	1.0	1.0

Table B.1: **Fraction of the 6 fraud cases detected by the Spark Streaming implementation** — The first column lists the number of unrelated events added in the event stream in addition to the 10 diffusion groups. The second column indicates how many multiples of PARTE's minimum viable partial match history sizes were allocated for the Spark Streaming implementation.

Scenario		Fraction of fraud cases found per percentiles						
# unrelated	Scale factor	0%	5%	25%	50%	75%	95%	100%
0	1	0.7	0.7	0.7	0.7	0.72	0.72	0.74
0	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	3	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	4	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	6	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	7	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	9	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	1	0.64	0.66	0.68	0.68	0.7	0.7	0.72
100	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	3	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	4	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	6	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	7	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	9	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Scenario		Fraction of fraud cases found per percentiles						
# unrelated	Scale factor	0%	5%	25%	50%	75%	95%	100%
1000	1	0.5	0.52	0.54	0.56	0.58	0.6	0.6
1000	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	3	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	4	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	6	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	7	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	9	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	1	0.1	0.14	0.18	0.2	0.24	0.26	0.32
10000	2	0.28	0.32	0.38	0.42	0.46	0.5	0.58
10000	3	0.44	0.5	0.56	0.6	0.64	0.68	0.72
10000	4	0.64	0.72	0.78	0.81	0.84	0.86	0.96
10000	5	0.94	0.96	0.98	1.0	1.0	1.0	1.0
10000	6	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	7	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	9	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table B.2: Fraction of the 50 fraud cases detected by the Spark Streaming implementation — The first column lists the number of unrelated events added in the event stream in addition to the 100 diffusion groups. The second column indicates how many multiples of PARTE's minimum viable partial match history sizes were allocated for the Spark Streaming implementation.

Scenario		Fraction of fraud cases found per percentiles						
# unrelated	Scale factor	0%	5%	25%	50%	75%	95%	100%
0	1	0.07	0.07	0.07	0.07	0.07	0.07	0.07
0	2	0.14	0.14	0.14	0.14	0.14	0.14	0.15
0	3	0.21	0.21	0.21	0.21	0.21	0.22	0.22
0	4	0.28	0.28	0.28	0.28	0.28	0.29	0.29
0	5	0.35	0.35	0.35	0.35	0.36	0.36	0.36
0	6	0.42	0.42	0.42	0.43	0.43	0.43	0.43
0	7	0.49	0.49	0.5	0.5	0.5	0.5	0.5
0	8	0.56	0.56	0.57	0.57	0.57	0.57	0.58
0	9	0.63	0.63	0.64	0.64	0.64	0.64	0.65
0	10	0.7	0.71	0.71	0.71	0.71	0.72	0.72
0	11	0.78	0.78	0.78	0.78	0.78	0.79	0.79
0	12	0.84	0.85	0.85	0.85	0.85	0.86	0.86
0	13	0.92	0.92	0.92	0.92	0.93	0.93	0.93
0	14	0.99	0.99	0.99	1.0	1.0	1.0	1.0
0	15	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	16	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	17	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	18	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0	19	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	1	0.07	0.07	0.07	0.07	0.07	0.07	0.07
100	2	0.14	0.14	0.14	0.14	0.14	0.14	0.14
100	3	0.21	0.21	0.21	0.21	0.21	0.21	0.22
100	4	0.28	0.28	0.28	0.28	0.28	0.29	0.29
100	5	0.35	0.35	0.35	0.35	0.36	0.36	0.36
100	6	0.42	0.42	0.42	0.42	0.43	0.43	0.43
100	7	0.49	0.49	0.49	0.5	0.5	0.5	0.5
100	8	0.56	0.56	0.56	0.57	0.57	0.57	0.57
100	9	0.63	0.63	0.64	0.64	0.64	0.64	0.65

Scenario		Fraction of fraud cases found per percentiles						
# unrelated	Scale factor	0%	5%	25%	50%	75%	95%	100%
100	10	0.7	0.7	0.71	0.71	0.71	0.71	0.72
100	11	0.77	0.77	0.78	0.78	0.78	0.78	0.79
100	12	0.84	0.85	0.85	0.85	0.85	0.86	0.86
100	13	0.91	0.92	0.92	0.92	0.92	0.93	0.93
100	14	0.99	0.99	0.99	0.99	1.0	1.0	1.0
100	15	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	16	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	17	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	18	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	19	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	1	0.06	0.07	0.07	0.07	0.07	0.07	0.07
1000	2	0.13	0.13	0.14	0.14	0.14	0.14	0.14
1000	3	0.2	0.2	0.21	0.21	0.21	0.21	0.22
1000	4	0.27	0.27	0.27	0.28	0.28	0.28	0.29
1000	5	0.34	0.34	0.34	0.35	0.35	0.35	0.36
1000	6	0.41	0.41	0.41	0.42	0.42	0.42	0.43
1000	7	0.48	0.48	0.48	0.49	0.49	0.49	0.5
1000	8	0.55	0.55	0.55	0.56	0.56	0.56	0.57
1000	9	0.62	0.62	0.62	0.62	0.63	0.63	0.64
1000	10	0.68	0.69	0.69	0.69	0.7	0.7	0.71
1000	11	0.75	0.76	0.76	0.76	0.77	0.77	0.78
1000	12	0.82	0.82	0.83	0.83	0.84	0.84	0.85
1000	13	0.89	0.89	0.9	0.9	0.9	0.91	0.92
1000	14	0.96	0.96	0.97	0.97	0.97	0.98	0.99
1000	15	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	16	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	17	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	18	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	19	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	1	0.0	0.05	0.05	0.06	0.06	0.06	0.06
10000	2	0.0	0.11	0.11	0.11	0.12	0.12	0.12
10000	3	0.0	0.16	0.17	0.17	0.17	0.18	0.18
10000	4	0.0	0.22	0.22	0.23	0.23	0.23	0.24
10000	5	0.0	0.27	0.28	0.28	0.29	0.29	0.3
10000	6	0.0	0.33	0.34	0.34	0.34	0.35	0.35
10000	7	0.0	0.38	0.39	0.4	0.4	0.41	0.41
10000	8	0.0	0.44	0.45	0.45	0.46	0.47	0.48
10000	9	0.49	0.5	0.5	0.51	0.52	0.52	0.53
10000	10	0.54	0.55	0.56	0.57	0.57	0.58	0.59
10000	11	0.6	0.61	0.62	0.62	0.63	0.64	0.64
10000	12	0.66	0.66	0.67	0.68	0.69	0.69	0.7
10000	13	0.71	0.72	0.73	0.74	0.74	0.75	0.76
10000	14	0.77	0.78	0.79	0.79	0.8	0.81	0.81
10000	15	0.83	0.84	0.85	0.85	0.86	0.86	0.87
10000	16	0.89	0.89	0.9	0.91	0.91	0.92	0.93
10000	17	0.94	0.95	0.96	0.96	0.97	0.98	0.99
10000	18	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10000	19	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table B.3: Fraction of the 500 fraud cases detected by the Spark Streaming implementation — The first column lists the number of unrelated events added in the event stream in addition to the 1000 diffusion groups. The second column indicates how many multiples of PARTE's minimum viable partial match history sizes were allocated for the Spark Streaming implementation.

Bibliography

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The Design of the Borealis Stream Processing Engine. In *Cidr*, volume 5, pages 277–289, 2005.

[2] Anurag Acharya and Milind Tambe. Collection Oriented Match. In *Proceedings of the Second International Conference on Information and Knowledge Management*, CIKM '93, pages 516–526, New York, NY, USA, 1993. ACM. ISBN 0-89791-626-3. doi: 10.1145/170088.170411.

[3] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient Pattern Matching over Event Streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376634.

[4] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *Proceedings of the International Conference on Very Large Data Bases Endowment*, 5(10):968–979, June 2012. ISSN 2150-8097. doi: 10.14778/2336664.2336670.

[5] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proceedings of the International Conference on Very Large Data Bases Endowment*, 6(11):1033–1044, August 2013. ISSN 2150-8097. doi: 10.14778/2536222.2536229.

[6] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Commun- umications of the ACM*, 26(11):832–843, November 1983. ISSN 0001-0782. doi: 10.1145/182.358434.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1807–1823, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134078.

[8] Apache Software Foundation. Apache Flink® - Stateful Computations over Data Streams. `https://flink.apache.org/`, 2018. Accessed: 2018-08-08.

[9] Apache Software Foundation. FlinkCEP - Complex event processing for Flink. `https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html`, 2018. Accessed: 2018-08-08.

[10] Apache Software Foundation. Hadoop. `https://hadoop.apache.org/`, 2018. Accessed: 2018-08-08.

[11] Apache Software Foundation. Kafka — A Distributed Streaming Platform. `https://kafka.apache.org/`, 2018. Accessed: 2018-08-08.

[12] Apache Software Foundation. ZooKeeper. `https://zookeeper.apache.org/`, 2018. Accessed: 2018-08-08.

[13] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM. ISBN 1-58113-634-X. doi: 10.1145/872757.872854.

[14] Mostafa M Aref and Mohammed A Tayyib. Lana-Match Algorithm: a Parallel Version of the Rete-Match Algorithm. *Parallel Computing*, 24(5–6):763–775, 1998. ISSN 0167-8191. doi: 10.1016/S0167-8191(98)00003-9.

[15] Alexander Artikis, Marek Sergot, and Georgios Paliouras. Run-Time Composite Event Recognition. In *Proceedings of the 6th ACM International Conference on Distributed Event-based Systems*, DEBS '12, pages 69–80, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1315-5. doi: 10.1145/2335484.2335492.

[16] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A Survey on Reactive Programming. *ACM Computing Surveys (CSUR)*, 45(4):52:1–52:34, August 2013. ISSN 0360-0300. doi: 10.1145/2501654.2501666.

[17] Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. Contract-based Load Management in Federated Distributed Systems. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, volume 1 of *NSDI'04*, pages 15–15, Berkeley, CA, USA, 2004. USENIX Association.

[18] Don Batory. The LEAPS Algorithm. Technical report, University of Texas at Austin, Austin, TX, USA, 1994. URL `ftp://ftp.cs.utexas.edu/pub/techreports/tr94-28.ps.Z`.

[19] BEAM. The Erlang Runtime System. `https://happi.github.io/theBeamBook/`, 2018. Accessed: 2018-08-08.

[20] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9):1270–1282, Sep. 1991. ISSN 0018-9219. doi: 10.1109/5.97297.

[21] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, pages 389–448, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. ISBN 978-3-540-39593-5.

[22] B. Berstel. Extending the RETE Algorithm for Event Management. In *Proceedings Ninth International Symposium on Temporal Representation and Reasoning*, pages 49–51, July 2002. doi: 10.1109/TIME.2002.1027472.

[23] Kevin Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohammed Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene Shekita. Jaql: a Scripting Language for Large Scale Semistructured Data Analysis. In H. V. Jagadish, José Blakeley, Joseph M. Hellerstein, Nick Koudas, Wolfgang Lehner, Sunita Sarawagi, and Uwe Röhm, editors, *Proceedings of the 37th International Conference on Very Large Data Bases*, Proceedings of the International Conference on Very Large Data Bases Endowment, pages 1272–1283, Seattle, USA, 2011. VLDB Endowment.

[24] Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. Pause 'n' Play: Formalizing Asynchronous C$^\sharp$. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 233–257, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31057-7.

[25] Krysia Broda, Keith Clark, Rob Miller, and Alessandra Russo. SAGE: a Logical Agent-based Environment Monitoring and Control System. In Manfred Tscheligi, Boris de Ruyter, Panos Markopoulus, Reiner Wichert, Thomas Mirlacher, Alexander Meschterjakov, and Wolfgang Reitberger, editors, *Ambient Intelligence*, pages 112–117, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-05408-2.

[26] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Readings in Hardware/Software Co-Design. chapter Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems, pages 527–543. Kluwer Academic Publishers, Norwell, MA, USA, 2002. ISBN 1-55860-702-1.

[27] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Nesime Tatbul, Stan Zdonik, and Michael Stonebraker. Monitoring Streams — A New Class of Data Management Applications. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 215–226. Elsevier, 2002.

[28] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. FaCT: a Flexible, Constant-Time Programming Language. In *Secure Development Conference (SecDev)*, pages 69–76. IEEE, September 2017. ISBN 978-1-5386-3467-7.

[29] S. Chakravarthy and D. Mishra. Snoop: an Expressive Event Specification Language for Active Databases. *Data & Knowledge Engineering*, 14(1):1–26, November 1994. ISSN 0169-023X. doi: 10.1016/0169-023X(94)90006-X.

[30] S. Chakravarthy and D. Mishra. Snoop: an Expressive Event Specification Language for Active Databases. *Data & Knowledge Engineering*, 14(1):1–26, 1994. ISSN 0169-023X. doi: https://doi.org/10.1016/0169-023X(94)90006-X.

[31] K. Mani Chandy. Theory and Implementation of a Distributed Event based Platform. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 205–213, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4021-2. doi: 10.1145/2933267.2940321.

[32] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 379–390, New York, NY, USA, 2000. ACM. ISBN 1-58113-217-4. doi: 10.1145/342009.335432.

[33] Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S. Miller, Franz Och, Christopher Olston, and Fernando Pereira. Yedalog: Exploring Knowledge at Scale. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 63–78, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-80-4. doi: 10.4230/LIPIcs.SNAPL.2015.63.

[34] Horatiu Cirstea, Claude Kirchner, Michael Moossen, and Pierre-Etienne Moreau. Production Systems and Rete Algorithm Formalisation. Contract A04-R-546 || cirstea04d, INRIA Lorraine, LORIA, Nancy, France, 2004. URL `https://hal.inria.fr/inria-00099850`. Rapport de contrat.

[35] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*, pages 313–328, 2010.

[36] Gregory H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Brown University, 2008. URL `http://cs.brown.edu/people/ghcooper/thesis.pdf`.

[37] Gregory H. Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In Peter Sestoft, editor, *Programming Languages and Systems*, pages 294–308, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-33096-7.

[38] Antony Courtney. Frappé: Functional Reactive Programming in Java. In *Practical Aspects of Declarative Languages*, pages 29–44. Springer, 2001.

[39] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 647–651, New York, NY, USA, 2003. ACM. ISBN 1-58113-634-X. doi: 10.1145/872757.872838.

[40] Gianpaolo Cugola and Alessandro Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Computing Surveys (CSUR)*, 44(3): 15:1–15:62, June 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187677.

[41] Gianpaolo Cugola and Alessandro Margara. Complex Event Processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2012.03.056.

[42] Evan Czaplicki and Stephen Chong. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462161.

[43] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. *43 Years of Actors: a Taxonomy of Actor Models and their Key Properties*, pages 31–40. AGERE! 2016. ACM, 2016. ISBN 978-1-4503-4639-9. doi: 10.1145/3001886.3001890.

[44] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, volume 6 of *OSDI'04*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[45] Damian Dechev and Bjarne Stroustrup. Scalable Nonblocking Concurrent Objects for Mission Critical Code. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '09, pages 597–612. ACM, 2009. ISBN 978-1-60558-768-4. doi: 10.1145/1639950.1639954.

[46] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards Expressive Publish/Subscribe Systems. In Yannis Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust, and Christian Boehm, editors, *Advances in Database Technology - EDBT 2006*, pages 627–644, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32961-9. doi: 10.1007/11687238_38.

[47] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: a General Purpose Event Monitoring System. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 412–422, 2007.

[48] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A Query Language for XML. *Computer Networks*, 31(11):1155–1169, 1999. ISSN 1389-1286. doi: 10.1016/S1389-1286(99)00020-1.

[49] Klaus R. Dittrich and Stella Gatziu. Time Issues in Active Database Systems. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pages 1–6, Arlington, TX, 1993.

[50] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 361–376, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660240.

[51] Jonathan Edwards. Coherent Reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 925–932, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-768-4. doi: 10.1145/1639950.1640058.

[52] Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for Modern AI. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 181–220, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24206-9.

[53] Elixir. Elixir. `https://elixir-lang.org/`, 2018. Accessed: 2018-08-08.

[54] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM. ISBN 0-89791-918-1. doi: 10.1145/258948.258973.

[55] Kyumars Sheykh Esmaili. Reflections on Almost Two Decades of Research into Stream Processing: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 21–23, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5065-5. doi: 10.1145/3093742.3095110.

[56] EsperTech. Esper. `http://www.espertech.com/esper/`, 2006. Accessed: 2018-08-08.

[57] Patrick Eugster and K. R. Jayaram. EventJava: an Extension of Java for Event Correlation. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 570–594, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_26.

[58] Charles L. Forgy. OPS5 User's Manual. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa, July 1981. URL `http://www.dtic.mil/docs/citations/ADA106558`.

[59] Charles L. Forgy. Rete: a Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982. ISSN 0004-3702. doi: 10.1016/0004-3702(82)90020-0.

[60] Eric Friedman, Peter Pawlowski, and John Cieslewicz. SQL/MapReduce: a Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions. *Proceedings of the International Conference on Very Large Data Bases Endowment*, 2(2):1402–1413, August 2009. ISSN 2150-8097. doi: 10.14778/1687553.1687567.

[61] Stella Gatziu and Klaus R. Dittrich. Events in an Active Object-Oriented Database System. In Norman W. Paton and M. Howard Williams, editors, *Rules in Database Systems*, pages 23–39, London, 1994. Springer London. ISBN 978-1-4471-3225-7. doi: 10.1007/978-1-4471-3225-7_2.

[62] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 327–338, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1-55860-151-1.

[63] Georg Gottlob, Stephanie Tien Lee, Gregory Valiant, and Paul Valiant. Size and Treewidth Bounds for Conjunctive Queries. *Journal of the ACM (JACM)*, 59(3): 16:1–16:35, June 2012. ISSN 0004-5411. doi: 10.1145/2220357.2220363.

[64] A. Gupta, C. Forgy, A. Newell, and R. Wedig. Parallel Algorithms and Architectures for Rule-based Systems. In *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 28–37. IEEE Computer Society Press, 1986. ISBN 0-8186-0719-X. doi: 10.1145/17407.17360.

[65] Anurag Prakash Gupta, Charles Lanny Forgy, Dirk Kalp, and Allen Newell. Parallel OPS5 on the Encore Multimax. *Proceedings of the International Conference on Parallel Processing*, pages 271–280, 1988.

[66] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. On Supporting Kleene Closure over Event Streams. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 1391–1393, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-1836-7. doi: 10.1109/ICDE.2008.4497566.

[67] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sep. 1991. ISSN 0018-9219. doi: 10.1109/5.97300.

[68] Philipp Haller and Martin Odersky. Scala Actors: Unifying Thread-based and Event-based Programming. *Theoretical Computer Science*, 410(2-3):202–220, February 2009. ISSN 0304-3975. doi: 10.1016/j.tcs.2008.09.019.

[69] Timothy Harris. A Pragmatic Implementation of Non-blocking Linked-lists. In Jennifer Welch, editor, *Distributed Computing*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer Berlin / Heidelberg, 2001. ISBN 978-3-540-42605-9.

[70] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972.

[71] Annika Hinze and Agnès Voisard. A Parameterized Algebra for Event Notification Services. In *Proceedings of the Ninth International Symposium on Temporal Representation and Reasoning (TIME'02)*, TIME '02, pages 61–63, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1474-X.

[72] Annika Hinze and Agnès Voisard. EVA: an Event Algebra Supporting Complex Event Specification. *Information Systems*, 48:1–25, 2015. ISSN 0306-4379. doi: 10.1016/j.is.2014.07.003.

[73] Steve Hoffman. *Apache Flume: Distributed Log Collection for Hadoop*. Packt Publishing Ltd, 2013. ISBN 978-1784392178.

[74] Lode Hoste. *A Declarative Approach for Engineering Multimodal Interaction*. PhD thesis, Vrije Universiteit Brussel, 2015. URL `https://soft.vub.ac.be/Publications/2015/vub-soft-phd-15-02.pdf`.

[75] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. In Johan Jeuring and Simon L. Peyton Jones, editors, *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures*, pages 159–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-44833-4. doi: 10.1007/978-3-540-44833-4_6.

[76] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1259–1274, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4197-4. doi: 10.1145/3035918.3064027.

[77] Daniel Ignatoff, Gregory H Cooper, and Shriram Krishnamurthi. Crossing State Lines: Adapting Object-oriented Frameworks to Functional Reactive Languages. In *Functional and Logic Programming*, pages 259–276. Springer, 2006.

[78] Facebook Inc. React — A JavaScript library for building user interfaces. `https://reactjs.org/`, 2018. Accessed: 2018-08-08.

[79] Toru Ishida. Parallel Rule Firing in Production Systems. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):11–17, March 1991. ISSN 10414347. doi: 10.1109/69.75883.

[80] Prasad Jayanti and Srdjan Petrovic. Logarithmic-Time Single Deleter, Multiple Inserter Wait-free Queues and Stacks. In *Proceedings of the 25th international Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS '05, pages 408–419. Springer-Verlag, 2005. ISBN 3-540-30495-9, 978-3-540-30495-1. doi: 10.1007/11590156_33.

[81] Q. Jiang, R. Adaikkalavan, and R. Chakravarthy. NFMi: an Inner-Domain Network Fault Management System. In *21st International Conference on Data Engineering (ICDE'05)*, pages 1036–1047, April 2005. doi: 10.1109/ICDE.2005.94.

[82] Hans Elias Bukholm Josephsen. Rustler. `https://github.com/hansihe/Rustler`, 2018. Accessed: 2018-08-08.

[83] Kennedy Kambona, Thierry Renaux, and Wolfgang De Meuter. Efficient Matching in Heterogeneous Rule Engines. In *Proceedings of the 30th International Conference on Industrial, Engineering, Other Applications of Applied Intelligent Systems (IEA/AIE 2017: Advances in Artificial Intelligence: From Theory to Practice)*, volume 10350 of *Lecture Notes in Computer Science*, pages 394–406. Springer, 6 2017. ISBN 978-3-319-60041-3.

[84] Kennedy Kambona, Thierry Renaux, and Wolfgang De Meuter. Harnessing Community Knowledge in Heterogeneous Rule Engines. In Tim A. Majchrzak, Paolo Traverso, Karl-Heinz Krempels, and Valérie Monfort, editors, *Web Information Systems and Technologies*, volume 322, pages 132–160. Springer International Publishing, 2018. ISBN 978-3-319-93527-0. doi: 10.1007/978-3-319-93527-0_7.

[85] Kennedy Kondo Kambona, Thierry Renaux, and Wolfgang De Meuter. Reentrancy and Scoping for Multitenant Rule Engines. In *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST 2017)*, volume 1, pages 59–70. Scitepress, 2017.

[86] Ju-Whan Kim and Tek-Jin Nam. EventHurdle: Supporting Designers' Exploratory Interaction Prototyping with Gesture-based Sensors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems 2013*, CHI '13, pages 267–276, New York, NY, USA, 2013. ACM.

[87] S.P.T. Krishnan and Jose L. Ugia Gonzalez. Google Cloud Dataflow. In *Building Your Next Big Thing with Google Cloud Platform*, pages 255–275. Apress, Berkeley, CA, 2015. ISBN 978-1-4842-1005-5.

[88] Neelakantan R. Krishnaswami. Higher-order Functional Reactive Programming without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 221–232, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500588.

[89] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742788.

[90] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563.

[91] Ho Soo Lee and Marshall I. Schor. Match Algorithms for Generalized Rete Networks. *Artificial Intelligence*, 54(3):249–274, 1992. ISSN 0004-3702. doi: 10.1016/0004-3702(92)90047-2.

[92] Frank Lopez. The Parallel Production System. Master's thesis, University of Illinois at Urbana-Champaign, 1987.

[93] D. C. Luckham and J. Vera. An Event-based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Sept 1995. ISSN 0098-5589. doi: 10.1109/32.464548.

[94] David C. Luckham. *The Power of Events: an Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0201727897.

[95] Samuel Madden and Michael J. Franklin. Fjording the Stream: an Architecture for Queries Over Streaming Sensor Data. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 555–566, 2002.

[96] M. A. Maloof and K. J. Kochut. Modifying Rete to Reason Temporally. In *Proceedings of 1993 IEEE Conference on Tools with AI (TAI-93)*, pages 472–473, Nov 1993. doi: 10.1109/TAI.1993.634008.

[97] Stefan Marr, Thierry Renaux, Lode Hoste, and Wolfgang De Meuter. Parallel Gesture Recognition with Soft Real-Time Guarantees. *Science of Computer Programming*, 98: 159–183, 2 2015. ISSN 0167-6423. doi: doi:10.1016/j.scico.2014.02.012.

[98] Nicholas D. Matsakis and Felix S. Klock, II. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3217-0. doi: 10.1145/2663171.2663188.

[99] Dennis McCarthy and Umeshwar Dayal. The Architecture of an Active Database Management System. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 215–224, New York, NY, USA, 1989. ACM. ISBN 0-89791-317-5. doi: 10.1145/67544.66946.

[100] Erik Meijer. Reactive Extensions (Rx): Curing your Asynchronous Programming Blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 11. ACM, 2010.

[101] Jonas Mellin. *Resource-Predictable and Efficient Monitoring of Events*. PhD thesis, Department of Computer Science, University of Skövde, June 2004.

[102] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-scale Datasets. *Proceedings of the International Conference on Very Large Data Bases Endowment*, 3(1-2): 330–339, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920886.

[103] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640091.

[104] Maged M. Michael. Safe Memory Reclamation for Dynamic Lock-free Objects using Atomic Reads and Writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 21–30. ACM, 2002. ISBN 1-58113-485-1. doi: 10.1145/571825.571829.

[105] Daniel P. Miranker. TREAT: a Better Match Algorithm for AI Production Systems. In *Proceedings of the sixth National Conference on Artificial Intelligence*, volume 1 of *AAAI'87*, pages 42–47. AAAI Press, 1987. ISBN 0-934613-42-7.

[106] Daniel P. Miranker, David A. Brant, Bernie Lofaso, and David Gadbois. On the performance of lazy matching in production systems. In *Proceedings of the eighth National Conference on Artificial Intelligence*, volume 1 of *AAAI'90*, pages 685–692. AAAI Press, 1990. ISBN 0-262-51057-X.

[107] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-based Systems*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 3540326510.

[108] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522738.

[109] Florian Myter, Wolfgang De Meuter, and Christophe Scholliers. Distributed Reactive Programming for Reactive Distributed Systems. *The Art, Science, and Engineering of Programming*, 3(3):1–52, 2019. ISSN 2473-7321. doi: 10.22152/programming-journal.org/2019/3/5.

[110] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed Stream Computing Platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[111] Hung Q. Ngo. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, SIGMOD/PODS '18, pages 111–124, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4706-8. doi: 10.1145/3196959.3196990.

[112] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proceedings of the International Conference on Very Large Data Bases Endowment*, 10(12):1634–1645, August 2017. ISSN 2150-8097. doi: 10.14778/3137765.3137770.

[113] Kemal Oflazer. *Partitioning in Parallel Processing of Production Systems*. PhD thesis, Carnegie Mellon University, 1985.

[114] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a Not-so-Foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376726.

[115] Adrian Paschke and Alexander Kozlenkov. Rule-based Event Processing and Reaction Rules. In Guido Governatori, John Hall, and Adrian Paschke, editors, *Rule Interchange and Applications*, pages 53–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04985-9.

[116] Mark Perlin. Scaffolding the RETE Network. In *[1990] Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence*, pages 378–385, Nov 1990. ISBN 0-8186-2084-6. doi: 10.1109/TAI.1990.130367.

[117] P. R. Pietzuch, B. Shand, and J. Bacon. Composite Event Detection as a Generic Middleware Extension. *IEEE Network*, 18(1):44–55, Jan 2004. ISSN 0890-8044. doi: 10.1109/MNET.2004.1265833.

[118] Mark Proctor. Drools: a Rule Engine for Complex Event Processing. In *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*, AGTIVE'11, pages 2–2, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-34175-5. doi: 10.1007/978-3-642-34176-2_2.

[119] ReactiveX. ReactiveX — An API for asynchronous programming with observable streams. `http://reactivex.io/`, 2018. Accessed: 2018-08-08.

[120] Thierry Renaux, Lode Hoste, Stefan Marr, and Wolfgang De Meuter. Parallel Gesture Recognition with Soft Real-Time Guarantees. In *Proceedings of the 2nd edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! 2012, pages 35–46. ACM, 10 2012. ISBN 978-1-4503-1630-9. doi: 10.1145/2414639.2414646.

[121] Thierry Renaux, Lode Hoste, Christophe Scholliers, and Wolfgang De Meuter. Software Engineering Principles in the Midas Gesture Specification Language. In *Proceedings the 2nd International Workshop on Programming for Mobile and Touch, Portland*, PRoMoTo 2014, pages 9–16, Portland, Oregon, USA, 10 2014. ACM. ISBN 978-1-4503-2295-9.

[122] Thierry Renaux, Lode Hoste, and Wolfgang De Meuter. Logical Reactive Programming. In *REBLS'15: Workshop on Reactive and Event based Systems 2015*, October 2015. URL `http://www.guidosalvaneschi.com/REBLS/REBLS2015/attachments/REBLS15_paper_8.pdf`.

[123] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on Software Engineering*, 43(12):1125–1143, Dec 2017. ISSN 0098-5589. doi: 10.1109/TSE.2017.2655524.

[124] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–36, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2772-5. doi: 10.1145/2577080.2577083.

[125] J.G. Schmolze and W. Snyder. Detecting Redundancy among Production Rules using Term Rewrite Semantics. *Knowledge-Based Systems*, 12(1):3–11, 1999. ISSN 0950-7051. doi: https://doi.org/10.1016/S0950-7051(99)00003-9.

[126] Christophe Scholliers, Lode Hoste, Beat Signer, and Wolfgang De Meuter. Midas: a Declarative Multi-Touch Interaction Framework. In *Proceedings of the Fifth International Conference on Tangible, Embedded, and Embodied Interaction*, TEI '11, pages 49–56, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0478-8. doi: 10.1145/1935701.1935712.

[127] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-665-6. doi: 10.1145/1619258.1619264.

[128] Scarlet Schwiderski-Grosche and Ken Moody. The SpaTeC Composite Event Language for Spatio-temporal Reasoning in Mobile Systems. In *Proceedings of the Third ACM International Conference on Distributed Event-based Systems*, DEBS '09, pages 11:1–11:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-665-6. doi: 10.1145/1619258.1619273.

[129] Wayne Snyder and James G. Schmolze. Rewrite Semantics for Production Rule Systems: Theory and Applications. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction — Cade-13*, pages 508–522, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-68687-3. doi: 10.1007/3-540-61511-3_110.

[130] Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. Mailbox Abstractions for Static Analysis of Actor Programs. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:30, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-035-4. doi: 10.4230/LIPIcs.ECOOP.2017.25.

[131] Mark Sullivan. Tribeca: a Stream Database Manager for Network Traffic Analysis. In *Proceedings of the 22th International Conference on Very Large Data Bases*, volume 96, page 594, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. ISBN 1-55860-382-4.

[132] Janwillem Swalens, Thierry Renaux, Lode Hoste, Stefan Marr, and Wolfgang De Meuter. Cloud PARTE: Elastic Complex Event Processing based on Mobile Actors. In *Proceedings of the 3rd International Workshop on Programming based on Actors, Agents, and Decentralized Control*, AGERE! 2013, pages 1–10. ACM, 10 2013. ISBN 978-1-4503-2602-5.

[133] Dan Teodosiu and Gunter Pollak. Discarding Unused Temporal Information in a Production System. *International Conference on Information and Knowledge Management*, pages 177–184, 1992.

[134] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous Queries over Append-only Databases. pages 321–330, 1992. doi: 10.1145/130283.130333.

[135] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2376-5. doi: 10.1145/2588555.2595641.

[136] Tom Van Cutsem and Wolfgang De Meuter. *Event-driven Mobile Computing with Objects*, pages 324–345. IGI Global, 4 2010. ISBN 978-1-60566-697-6. Annika Hinze and Alejandro P. Buchmann.

[137] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream Processing with a Spreadsheet. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming*, volume 8586, pages 360–384, New York, NY, USA, 2014. Springer-Verlag New York, Inc. ISBN 978-3-662-44201-2. doi: 10.1007/978-3-662-44202-9_15.

[138] Karen Walzer, Alexander Schill, and Alexander Löser. Temporal Constraints for Rule-based Event Processing. In *Proceedings of the ACM First Ph.D. Workshop in CIKM*, PIKM '07, pages 93–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-832-9. doi: 10.1145/1316874.1316890.

[139] Fusheng Wang and Peiya Liu. Temporal Management of RFID Data. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1128–1139. VLDB Endowment, 2005. ISBN 1-59593-154-6.

[140] Ian Wright and James AR Marshall. The Execution Kernel of RC++: RETE*, a Faster RETE with TREAT as a Special Case. *International Journal on Intelligent Games & Simulation*, 2(1):36–48, 2003.

[141] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-Performance Complex Event Processing over Streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142520.

[142] Robert M. Wygant. CLIPS — A Powerful Development and Delivery Expert System Tool. *Computers & Industrial Engineering*, 17(1):546–549, 1989. ISSN 0360-8352. doi: 10.1016/0360-8352(89)90121-6.

[143] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: a System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[144] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. *HotCloud'10: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, 10(10-10):95, 2010.

[145] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: a Fault-tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[146] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522737.

[147] D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)(ICDE)*, pages 392–399, 03 1999. doi: 10.1109/ICDE.1999.754955.