Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Combining Expressiveness and Efficiency in a Complex Event Processing Middleware

Doctoral Dissertation of:
Alessandro Margara

Advisor:
Prof. Gianpaolo Cugola
Tutor:
Prof. Letizia Tanca
Supervisor of the Doctoral Program:
Prof. Carlo Fiorini

2011 – XXIV

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32 I-20133 — Milano

To Alessia

Acknowledgements

I wish to thank my advisor, Prof. Gianpaolo Cugola. He has always been present to discuss with me, giving me the best advice to go on. During these years he has been a real guide for me, always able to direct my efforts in the right direction. I really enjoyed working with him and sharing with him my first steps into research. He helped me growing, teaching me how to improve and defend my ideas. Thanks!

I cannot forget all my colleagues, and in particular my officemates, Alberto, Alessandro, Andrea, Daniel, Leandro, Liliana, Luca, Marco. I spent the most significant moments of this journey with them.

A special thank goes to my old friends, Alfo, Andre, Andre, Cillo, Dado, Pol, Tom, Sammy, Zio, for everything we shared. Thanks to Enzo: I've learn so much from him.

Last but not least, my biggest thank goes to Alessia and to my family. They have always believed in me and they encouraged me to pursue my dreams. Without their help, I would have never become the person I am.

Ringraziamenti

Desidero ringraziare il mio advisor, Prof. Gianpaolo Cugola, per la sua disponibilità e per i suoi innumerevoli e preziosi consigli. In questi anni è stato per me una vera e propria guida, sempre capace di sostenere e indirizzare i miei sforzi. Condividere con lui i miei primi passi nel mondo della ricerca mi ha permesso di crescere, imparando a coltivare e difendere le mie idee. Grazie!

Non posso dimenticare tutti i miei colleghi e in particolare i miei compagni di ufficio, Alberto, Alessandro, Andrea, Daniel, Leandro, Liliana, Luca, Marco, con cui ho condiviso i momenti più significativi di questo percorso.

Un grazie speciale ai miei amici di sempre, Alfo, Andre, Andre, Cillo, Dado, Pol, Tom, Sammy, Zio, per tutto quello che abbiamo vissuto insieme. Grazie a Enzo per i suoi preziosi insegnamenti.

Infine, il ringraziamento più grande va ad Alessia e alla mia famiglia, per aver sempre creduto in me e avermi spronato a inseguire i miei sogni senza arrendermi mai. Senza di voi non sarei mai potuto diventare quello che sono.

Abstract

Several complex systems operate by observing a set of *primitive events* that happen in the external environments, interpreting and combining them to identify higher level *composite events*, and finally sending the notifications about these events to the components in charge of reacting to them, thus determining the overall system's behavior.

Examples of systems that operate this way are sensor networks for environmental monitoring, financial applications, fraud detection tools, and RFID-based inventory management. More in general, as observed in [1], the information system of every complex company can and should be organized around an *event-based core* that realizes a sort of nervous system to guide and control the operation of the other sub-systems.

The task of identifying composite events from primitive ones is performed by the *Complex Event Processing* (*CEP*) *Engine*. It operates by interpreting a set of *event definition rules* that describe how composite events are defined from primitive ones. The CEP engine is usually part of a CEP *system* or *middleware* which also handles the communication with local and remote clients.

To capture all the requirements of the aforementioned applications, a CEP engine has to face several challenges. First, it has to provide a suitable language for rule specification, explicitly tailored to model complex temporal relationships that join together primitive events in composite ones. Second, it has to implement efficient processing algorithms, to detect composite events and deliver notifications to interested parties with the lowest possible delay. Finally, it has to support distributed scenarios, in which the communication parties may be deployed over a wide geographical area.

This thesis first proposes a modelling framework to compare and analyze not only existing CEP systems, but all the systems developed with the aim of processing continuous flows of information according to predeployed processing rules. This allows us to identify the main advantages and limitations of existing approaches, by looking at a wide range of proposals. Moreover, our modelling framework draws a common ground for comparing efforts coming from different research communities, with different background, expertise, and vocabulary. We believe that our work can bridge the gap between different worlds, promoting the communica-

tion and reducing the effort required to compare and merge the results produced so far.

Moving from the issues identified while analyzing existing works, we introduce T-Rex, a new CEP system explicitly designed to combine expressiveness and efficiency. In particular, we first present TESLA, the new event definition language used by T-Rex. TESLA is explicitly designed to model in an easy and natural way the complex relationships that join primitive events and the actions required to aggregate them to obtain composite events.

Then we discuss in details the implementation of T-Rex, studied to efficiently process TESLA rules. First of all we focus on the problem of *matching*, i.e., selecting relevant (primitive) events based on their content, which is one of the fundamental actions present in every event-based system. We propose a novel matching algorithm explicitly designed to take advantage of parallel hardware, including modern Graphical Processing Units (GPUs). This is the first solution that analyzed the adoption of parallel hardware to speed up matching and our evaluation shows impressive results with respect to existing sequential solutions.

Afterward, we focus on complete TESLA rules, and we discuss and compare two different processing algorithms that take two opposite approaches to process incoming events. A comparison with existing products shows the effectiveness of both our proposals and the differences among them.

Independently from the adopted algorithm, T-Rex leverages the presence of multiple processing cores to efficiently evaluate different rules in parallel. To further reduce the time required to handle the most complex rules, i.e., those involving a large number of primitive events, we present and evaluate a third algorithm to process TESLA rules on GPUs.

Our contribution goes beyond the implementation of T-Rex, indeed this is the first work that describes in details how CEP can leverage off-the-shelf parallel hardware: multi-core CPUs and GPUs. Since our analysis is organized around the basic language constructs provided by TESLA, but also present in most of existing CEP languages, our work represents an important contribution to determine how CEP can take advantage of currently available parallel hardware architectures and which processing algorithms are best suited to exploit their processing power.

The last aspect examined by this thesis is how to take advantage of the availability of multiple processing nodes, distributing the processing load over different machines, to better support large-scale distributed scenarios, reducing the delay required to receive results, or the occupation of network resources. To this extent, we present and compare different solutions for a distributed T-Rex, extracting the advantages and limitations

of each of them. They include the protocols to organize available nodes into an overlay network, to partition and distribute event definition rules, and to cooperatively handle event processing and delivery.

Riassunto

Molti sistemi complessi si basano sull'osservazione di *eventi primitivi* provenienti dall'ambiente esterno, interpretandoli e aggregandoli in *eventi compositi*, che offrono una conoscenza del mondo a un livello di astrazione superiore. Infine, inviano notifiche di tali eventi a tutti i componenti adibiti a prendere decisioni e mettere in atto azioni sulla base di essi, determinando in questo modo il comportamento complessivo del sistema.

Esistono svariati sistemi che operano in questo modo: le reti di sensori per il monitoraggio ambientale, le applicazioni finanziarie, i tool per la scoperta di frodi bancarie e i sistemi automatici di gestione dei prodotti basati su tecnologia RFID. Più in generale, come osservato in [1], il sistema informativo di ogni azienda può e dovrebbe essere organizzato intorno a un centro di gestione degli eventi, che realizzi una sorta di sistema nervoso per guidare e controllare il comportamento di ogni altro sotto-sistema.

Il compito di identificare eventi compositi a partire da eventi primitivi è svolto da un motore di *Complex Event Processing* (*CEP*). Esso interpreta una serie di regole che descrivono come gli eventi compositi sono definiti a partire dai primitivi. Il motore di CEP è solitamente parte di un *sistema* o *middleware* di CEP, il quale si occupa anche di gestire la comunicazione con componenti locali e remoti.

Per rispondere a tutte le esigenze delle applicazioni prima descritte, un motore di CEP deve soddisfare alcuni requisiti. In primo luogo, deve fornire un adeguato linguaggio per la definizione delle regole, espressamente studiato per esprimere complesse relazioni temporali. In secondo luogo, deve implementare algoritmi di processing efficienti, per produrre eventi compositi e inviare notifiche ai componenti interessati con una bassa latenza. Infine, deve poter supportare scenari distribuiti, in cui gli attori che prendono parte alla comunicazione potrebbero coprire una vasta area geografica.

In questa tesi viene dapprima proposto un framework di modelli che permette di analizzare e confrontare non solo i sistemi CEP esistenti, ma più in generale tutti i sistemi nati per processare flussi continui di informazioni sulla base di un insieme di regole. Questo ci permette di individuare i principali vantaggi e limitazioni degli approcci proposti sinora, guardando ad un ampio insieme di sistemi. Inoltre, il nostro framework

definisce un terreno comune per il confronto degli sforzi di ricerca intrapresi da diverse comunità, aventi diverse basi teoriche, conoscenze, e terminologie. Crediamo che questo lavoro possa contribuire a ridurre la distanza fra le diverse aree di ricerca, semplificando la comunicazione e agevolando lo scambio di esperienze, per portare a un'unione dei risultati ottenuti nel recente passato.

Partendo dalle problematiche individuate durante l'analisi dei sistemi esistenti, nella tesi viene poi presentato T-Rex un nuovo sistema di CEP, pensato per offrire espressività ed efficienza. In particolare, proponiamo innanzitutto TESLA, il nuovo linguaggio utilizzato da T-Rex per la definizione di eventi. TESLA è espressamente pensato per modellare in modo semplice e naturale le complesse relazioni tra eventi primitivi e le azioni necessarie per aggregarli in eventi compositi.

Successivamente viene presentata in dettaglio l'implementazione di T-Rex, studiato per processare in modo efficiente regole TESLA. La tesi si sofferma prima sul problema del *matching*, ovvero della selezione di eventi (primitivi) sulla base del contenuto degli eventi stessi. Tale problema sta alla base di ogni sistema a eventi. Nella tesi si propone un nuovo algoritmo espressamente progettato per sfruttare hardware parallelo, incluse le moderne schede grafiche (GPU). Questo è il primo lavoro in cui si analizza l'utilizzo di hardware parallelo per velocizzare il processo di matching e la fase di valutazione mostra significativi miglioramenti rispetto agli algoritmi sequenziali presenti in letteratura.

La tesi si concentra quindi sulla valutazione completa di regole TESLA, descrivendo e confrontando due algoritmi di processing che seguono due diversi approcci per l'analisi degli eventi in ingresso. Un confronto con i sistemi esistenti mostra i benefici della proposta presentata.

Indipendentemente dall'algoritmo adottato, T-Rex è in grado di sfruttare la presenza di più core per valutare in maniera efficiente diverse regole in parallelo. Un ulteriore contributo della tesi deriva dalla definizione di un algoritmo di processing espressamente pensato per girare sulle moderne schede grafiche, il quale permette di ridurre significativamente il tempo necessario per gestire singole regole complesse, che richiedono l'analisi di un ingenete numero di eventi.

È nostra convinzione che il contributo di questa tesi vada oltre la semplice progettazione e implementazione di T-Rex. Infatti essa è il primo lavoro ad analizzare in dettaglio come CEP possa trarre vantaggio dalla presenza di hardware parallelo: multi-core CPU e GPU. Sebbene la nostra analisi si concentri sugli operatori offerti da TESLA, questi costituiscono la base di molti linguaggi per la definizione di eventi presenti in letteratura. Per tale ragione il nostro lavoro costituisce un importante contributo per determinare come CEP possa trarre vantaggio dalle

architetture di hardware parallelo oggi disponibili e quali algoritmi di processing siano più adatti per sfruttarne al meglio le caratteristiche.

L'ultimo aspetto esaminato in questa tesi riguarda la possibilità di sfruttare la presenza di diversi nodi, distribuendo il carico di processing tra essi, al fine di offrire un miglior supporto a scenari distribuiti, riducendo la latenza necessaria per la ricezione di risultati e l'occupazione di risorse di rete. In tale ambito presentiamo e confrontiamo diverse soluzioni per un sistema T-Rex distribuito, evidenziando i vantaggi e le limitazioni di ognuna di esse. Tali soluzioni includono i protocolli per organizzare i nodi disponibili in una overlay network, per scomporre e distribuire le regole di definizione degli eventi e per gestire in modo cooperativo l'elaborazione e l'invio di eventi.

Contents

List of Figures

List of Tables

1 Introduction

Several complex systems operate by observing a set of *primitive events* that happen in the external environments, interpreting and combining them to identify higher level *composite events*, and finally sending the notifications about these events to the components in charge of reacting to them, thus determining the overall system's behavior.

Examples of systems that operate this way are sensor networks for environmental monitoring [2, 3]; financial applications requiring a continuous analysis of stocks to detect trends [4]; fraud detection tools, which observe streams of credit card transactions to prevent frauds [5]; RFID-based inventory management, which performs a continuous analysis of registered data to track valid paths of shipments and to capture irregularities [6]. More in general, as observed in [1], the information system of every complex company can and should be organized around an *event-based core* that realizes a sort of nervous system to guide and control the operation of the other sub-systems.

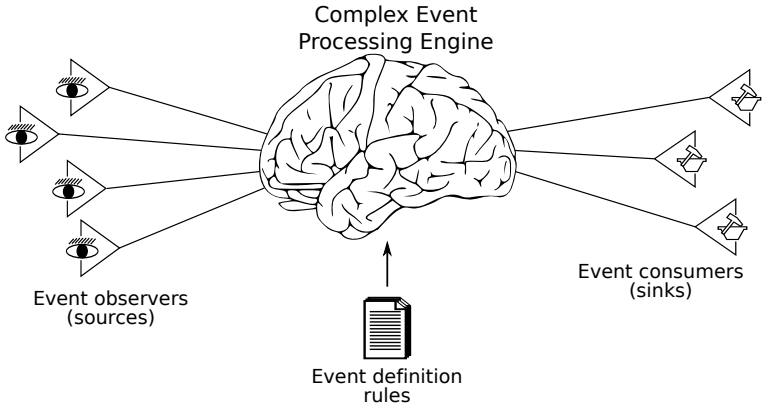


Figure 1.1: The high-level view of an CEP application

The general architecture of such event-based applications is shown in Figure 1.1. At the peripheral of the system are the *sources* and the *sinks*. The former observe primitive events and report them, while the latter receive composite event notifications and react to them. The task of identifying composite events from primitive ones is performed by the

Complex Event Processing (*CEP*) *Engine*. This fundamental component is usually part of a *CEP middleware* (which also includes the client-side libraries to access the engine) and operates by interpreting a set of *event definition rules*, which describe how composite events are defined from primitive ones.

Recently, a number of systems were developed that aim at playing the role of a CEP middleware as identified above. As we will better motivate in Chapter 2, they can be considered part of a wide class of systems, that we callectively define *Information Flow Processing* (*IFP*) systems [7]. IFP systems were developed to solve the general problem of processing continuous flows of information coming from heterogeneous sources to produce new knowledge and deliver it to interested components.

To capture all the requirements of a CEP middleware, IFP systems have to face several challenges.

- They have to provide a suitable language for rule specification. Such a language should be explicitly tailored to model complex temporal relationships that join together primitive events in composite ones; it should be expressive, to satisfy all the needs of above mentioned applications; it should be easy to use, to simplify the task of writing rules in different application scenarios; finally, it should provide a clear and unambiguous semantics.

- They have to implement efficient processing algorithms, to detect composite events and deliver notifications to sinks with the lowest possible delay. These algorithm should be able to scale with the number and the complexity of the rules deployed, to embrace large scale applications.

- They have to support distributed scenarios, in which sources and sinks may be deployed over a wide geographical area.

As we will better argument in the following chapters, existing proposals can hardly fulfill all these requirements. To overcome these limitations, this thesis introduces T-Rex, a new CEP system, explicitly designed to combine a simple yet expressive rule definition language and efficient algorithms for event processing and notification. The main contributions of this work cover five different areas:

- We propose a modelling framework for IFP systems. This allows us to classify and compare a large number of existing systems, to understand the advantages and limitations of each of them. Moreover, as mentioned above, IFP systems come from different communities, with different background and expertise. We believe that

our work can bridge the gap between different worlds, promoting the communication and reducing the effort required to merge the results produced so far.

- We propose a new rule definition language, called TESLA. TESLA is explicitly designed to model in an easy and natural way the complex relationships that join primitive events and the actions required to aggregate them to obtain composite events.

- We propose PCM, a novel *matching* algorithm. Matching is the process that selects relevant (primitive) events based on their content. This is a key functionality, present in all event-based systems, and represents a first, fundamental, step toward the definition of a CEP system. PCM is explicitly designed to take advantage of parallel hardware, including modern Graphical Processing Units (GPUs). A detailed comparison of PCM against state of the art shows significant advantages over existing solutions.

- We discuss the design and implementation of T-Rex, our CEP engine, studied to efficiently evaluate TESLA rules. More in particular, we discuss and compare two different processing algorithms, that take two opposite approaches to process incoming events. A comparison with existing products shows the effectiveness of our proposals. Independently from the adopted algorithm, T-Rex leverages the presence of multiple processing cores to efficiently evaluate different rules in parallel. To further reduce the time required to handle the most complex rules, i.e., those involving a large number of primitive events, we present and evaluate a third algorithm to process TESLA rules on GPUs.

- To support large scale distributed scenarios we discuss and compare different solutions for distributing complex event processing over multiple machines. They include the protocols to organize available nodes into an overlay network, to partition and distribute event definition rules and sinks' interests, and to cooperatively handle event processing and delivery.

1.1 Organization of the Thesis

The thesis is structured as follows:

- Chapter 2 presents our modelling framework for IFP systems. This introduces a precise terminology that is adopted throughout the

thesis. The framework is then used to classify and compare several existing systems, isolating their advantages and limitations. This allows us to draw a precise picture of the research area, identifying the results achieved so far and expliciting the open issues.

- Chapter 3 introduces TESLA, our rule definition language. The chapter presents all TESLA operators, providing a formal definition for each of them using TRIO, a metric temporal logic. Chapter 3 also introduces some motivating examples that will be used throughout the thesis to evaluate our work and compare it with existing proposals.

- Chapter 4 introduces PCM (Parallel Content-Based Matching), our parallel algorithm for content-based matching. Content-Based matching plays a central role in many systems, and it is one of the main building blocks of a CEP system. Chapter 4 first introduces two parallel programming models, naming OpenMP (for multi-core CPUs) and CUDA (for GPUs) and shows how PCM can be efficiently implemented on both of them. Finally, it evaluates the performance of PCM against a state-of-the art matching system.

- Chapter 5 presents the design and implementation of T-Rex, our CEP engine studied to efficiently process TESLA rules. The Chapter presents and compare two separate implementations of T-Rex that adopt two different processing algorithms. Finally, it evaluates the behavior of T-Rex in details, using a large number of workloads, and comparing it with a mature and widely adopted CEP system.

- Chapter 6 shows how one of the processing algorithms used in T-Rex can be implemented on the CUDA parallel architecture to run efficiently on GPUs. It evaluates the performance of this solution in details, extracting some general conclusions about the aspects that make the use of GPUs more profitable for a CEP rule evaluation algorithm.

- Chapter 7 shows how T-Rex can benefit from the presence of multiple processing nodes to distribute the processing load. We present different protocols for distributed event detection in details and we evaluate the benefits and limitations of each of them using a network simulator.

- Finally, Chapter 8 provides some conclusive remarks and indication for future research initiatives.

2 A Modelling Framework for IFP Systems

2.1 Introduction

This chapter introduces a modeling framework for IFP systems, and uses it to classify and compare existing proposals, coming both from the academia and from the industry.

In the following, we collectively call *Information Flow Processing* (*IFP*) *domain* the domain of distributed applications that require processing continuously flowing data from geographically distributed sources at unpredictable rate to obtain timely responses to complex queries. Examples of such applications come from the most disparate fields: from wireless sensor networks to financial tickers, from traffic management to clickstream inspection.

Likewise we call *Information Flow Processing* (*IFP*) *engine* a tool capable of timely processing large amount of information as it flows from the peripheral to the center of the system.

The concepts of "timeliness" and "flow processing" are crucial to justify the need of a new class of systems. Indeed, traditional DBMSs: (i) require data to be (persistently) stored and indexed before it could be processed, and (ii) process data only when explicitly asked by the users, i.e., asynchronously with respect to its arrival. Both aspects contrast with the requirements of IFP applications. As an example, consider the need of detecting fire in a building by using temperature and smoke sensors. On the one hand, a fire alert has to be notified as soon as the relevant data becomes available. On the other, there is no need to store sensor readings if they are not relevant for fire detection, while the relevant data can be discarded as soon as the fire is detected, since all the information they carry, like the area where the fire occurred, if relevant for the application, should be part of the fire alert.

These requirements led to the development of a number of systems specifically designed to process information as a flow (or a set of flows) according to a set of pre-deployed *processing rules*. Despite having a common goal, these systems differ in a wide range of aspects, including architecture, data models, rule languages, and processing mechanisms.

In part, this is due to the fact that they were the result of the research efforts of different communities, each one bringing its own view of the problem and its background for the definition of a solution, not to mention its own vocabulary [8]. After several years of research and development we can say that two models emerged and are today competing: the *data stream processing* model [9] and the *complex event processing* model [1].

As suggested by its own name, the data stream processing model describes the IFP problem as processing *streams of data* coming from different sources to produce new data streams as an output, and views this problem as an evolution of traditional data processing, as supported by DBMSs. Accordingly, *Data Stream Management Systems* (*DSMSs*) have their roots in DBMSs but present substantial differences. While traditional DBMSs are designed to work on persistent data, where updates are relatively infrequent, DSMSs are specialized in dealing with transient data that is continuously updated. Similarly, while DBMSs run queries just once to return a complete answer, DSMSs execute *standing queries*, which run continuously and provide updated answers as new data arrives. Despite these differences, DSMSs resemble DBMSs, especially in the way they process incoming data through a sequence of transformations based on common SQL operators like selections, aggregates, joins, and in general all the operators defined by relational algebra.

Conversely, the complex event processing model views flowing information items as *notifications* of *events* happening in the external world, which have to be filtered and combined to understand what is happening in terms of *higher-level* events. Accordingly, the focus of this model is on detecting occurrences of particular *patterns* of (low-level) events that represent the higher-level events whose occurrence has to be notified to the interested parties. While the contributions to this model come from different communities, including distributed information systems, business process automation, control systems, network monitoring, sensor networks, and middleware in general. The origins of this approach may be traced back to the publish-subscribe domain [10]. Indeed, while traditional publish-subscribe systems consider each event separately from the others, and filter them (based on their topic or content) to decide if they are relevant for subscribers, *Complex Event Processing* (*CEP*) systems extend this functionality by increasing the expressive power of the subscription language to consider complex event patterns that involve the occurrence of multiple, related events.

It is our firm belief that both models have historically brought important contributions in the IFP domain. Accordingly, we define our modeling framework with the aim of comparing the results coming from

the different communities, and we use it to provide an extensive review of the state of the art in the area, with the overall goal of reducing the effort to merge the results produced so far.

The rest of the chapter is organized as follows: Section 2.2 describes the IFP domain in more detail, provides an initial description of the different technologies that have been developed to support it, and explains the need for combining the best of different worlds to fully support IFP applications. Section 2.3 describes our framework to model and analyze the different aspects that are relevant for an IFP engine from its functional architecture, to its data and processing models, to the language it provides to express how information has to be processed, to its run-time architecture. We use this framework in Section 2.4 to describe and compare the state of the art in the field, discussing the results of such classification in Section 2.5. Section 2.6 reviews existing attempt to model the IFP domain and other work that are strictly related with the IFP domain. Finally, Section 2.7 provides some conclusive remarks and a list of open issues that emerge from our analysis of the state of the art.

2.2 Background and Motivation

In this section we characterize the application domain that we call Information Flow Processing and motivate why we introduce a new term. We describe the different technologies that have been proposed to support such a domain and conclude by motivating the need for our modeling framework.

2.2.1 The IFP Domain

With the term *Information Flow Processing* (*IFP*) we refer to an application domain in which users need to collect information produced by multiple, distributed sources, to process it in a timely way, in order to extract new knowledge as soon as the relevant information is collected.

Examples of IFP applications come from the most disparate fields. In environmental monitoring, users need to process data coming from sensors deployed in the field to acquire information about the observed world, detect anomalies, or predict disasters as soon as possible [2, 3]. Similarly, several financial applications require a continuous analysis of stocks to identify trends [4]. Fraud detection requires continuous streams of credit card transactions to be observed and inspected to prevent frauds [11]. To promptly detect and possibly anticipate attacks to a corporate network, intrusion detection systems have to analyze network

traffic in real-time, generating alerts when something unexpected happens [12]. RFID-based inventory management performs continuous analysis of RFID readings to track valid paths of shipments and to capture irregularities [6]. Manufacturing control systems often require anomalies to be detected and signalled by looking at the information that describe how the system behaves [13, 14].

Common to all these examples is the need of processing information as it flows from the periphery to the center of the system without requiring at least in principle the information to be persistently stored. Once the flowing data has been processed, producing new information, it can be discarded, while the newly produced information leaves the system as its output. Notice that complex applications may also need to store data, e.g., for historical analysis, but in general this is not the goal of the IFP subsystem.

The broad spectrum of applications domains that require "information flow processing" in the sense above, explains why several research communities focused their attention to the IFP domain, each bringing its own expertise and point of view, but also its own vocabulary. The result is a typical "Tower of Babel syndrome", which generates misunderstandings among researchers that negatively impacts the spirit of collaboration required to advance the state of the art [15]. This explains why in our framework we decided to adopt our own vocabulary, moving away from terms like "event", "data", "stream", or "cloud", to use more general (and unbiased) terms like "information" and "flow". In so doing, we neither have the ambition to propose a new standard terminology nor we want to contribute to "raising the tower". Our only desire is to help the reader look at the field with her's mind clear from any bias toward the meaning of the various terms used.

This goal lead us to model an IFP system as in Figure 2.1. The *IFP engine* is a tool that operates according to a set of *processing rules*, which describe how incoming *flows of information* have to be processed to timely produce new flows as outputs. The entities that create the information flows entering the IFP engine are called *information sources*. Examples of such information are notifications about events happening in the observed world or, more generally, any kind of data that reflects some knowledge generated by the source. *Information items* that are part of the same flow are neither necessarily ordered nor of the same kind. They are information chunks, with their semantics and relationships (including total or partial ordering, in some cases). The IFP engine takes such information items as input and processes them, as soon as they are available, according to a set of processing rules, which specify how to filter, combine, and aggregate different flows of information, item by item, to

generate new flows, which represent the output of the engine. We call *information sinks* the recipients of such output, while *rule managers* are the entities in charge of adding or removing processing rules. In some situations the same entity may play different roles. For example, it is not uncommon for an information sink to submit the rules that produce the information it needs.

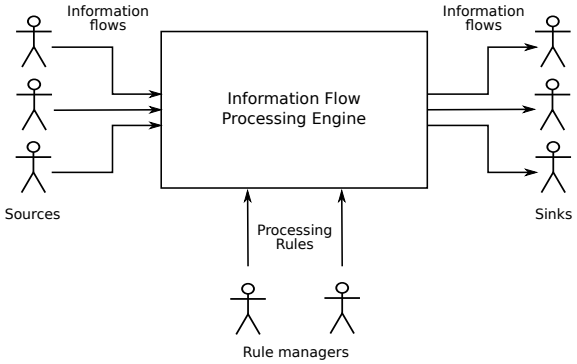


Figure 2.1: The high-level view of an IFP system

While the definition above is general enough to capture every IFP system we are interested in classifying, it encompasses some key points that characterize the domain and come directly from the requirements of the applications we mentioned above. First of all the need to perform real-time or quasi real-time processing of incoming information to produce new knowledge (i.e., outgoing information). Second, the need for an expressive language to describe how incoming information has to be processed, with the ability of specifying complex relationships among the information items that flow into the engine and are relevant to sinks. Third, the need for scalability to effectively cope with situations in which a very large number of geographically distributed information sources and sinks have to cooperate.

2.2.2 IFP: One Name Different Technologies

As we mentioned, IFP has attracted the attention of researchers coming from different fields. The first contributions came from the database community in the form of *active database systems* [16], which were introduced to allow actions to automatically execute when given conditions arise. *Data Stream Management Systems* (*DSMSs*) [9] pushed this idea further, to perform query processing in the presence of continuous data streams.

In the same years that saw the development of DSMSs, researchers with different backgrounds identified the need for developing systems that are capable of processing not generic data, but event notifications, coming from different sources, to identify interesting situations [1]. These systems are usually known as *Complex Event Processing* (*CEP*) *Systems*.

Active Database Systems

Traditional DBMSs are completely passive, as they present data only when explicitly asked by users or applications: this form of interaction is usually called *Human-Active Database-Passive* (*HADP*). Using this interaction model it is not possible to ask the system to send notifications when predefined situations are detected. *Active database systems* have been developed to overcome this limitation: they can be seen as an extension of classical DBMSs where the reactive behavior can be moved, totally or in part, from the application layer into the DBMS.

There are several tools classified as active database systems, with different software architectures, functionality, and oriented toward different application domains; still, it is possible to classify them on the basis of their *knowledge model* and *execution model* [17]. The former describes the kind of active rules that can be expressed, while the latter defines the system's runtime behavior.

The knowledge model usually considers active rules as composed of three parts:

- *Event* defines which sources can be considered as event generators: some systems only consider internal operators, like a tuple insertion or update, while others also allow external events, like those raised by clocks or external sensors;
- *Condition* specifies when an event must be taken into account; for example we can be interested in some data only if it exceeds a predefined limit;
- *Action* identifies the set of tasks that should be executed as a response to an event detection: some systems only allow the modification of the internal database, while others allow the application to be notified about the identified situation.

To make this structure explicit, active rules are usually called *Event-Condition-Action* (*ECA*) rules.

The execution model defines how rules are processed at runtime. Here, five phases have been identified:

- *signaling* i.e., detection of an event;
- *triggering* i.e., association of an event with the set of rules defined for it;
- *evaluation* i.e., evaluation of the conditional part for each triggered rule;
- *scheduling* i.e., definition of an execution order between selected rules;
- *execution* i.e., execution of all the actions associated to selected rules.

Active database systems are used in three contexts: as a database extension, in closed database applications, and in open database applications. As a database extension, active rules refer only to the internal state of the database, e.g., to implement an automatic reaction to constraint violations. In closed database applications, active rules can support the semantics of the application but external sources of events are not allowed. Finally, in open database applications events may come both from inside the database and from external sources. This is the domain that is closer to IFP.

Data Stream Management Systems

Even when taking into account external event sources, active database systems, like traditional DBMSs, are built around a persistent storage where all the relevant data is kept, and whose updates are relatively infrequent. This approach negatively impacts on their performance when the number of rules expressed exceeds a certain threshold or when the arrival rate of events (internal or external) is high. For most applications we classified as IFP such limitations are unacceptable.

To overcome them, the database community developed a new class of systems oriented toward processing large *streams of data* in a timely way: *Data Stream Management Systems* (*DSMSs*). DSMSs differ from conventional DBMSs in several ways:

- differently from tables, streams are usually unbounded;
- no assumption can be made on data arrival order;
- size and time constraints make it difficult to store and process data stream elements after their arrival; one-time processing is the typical mechanism used to deal with streams.

Users of a DSMS install *standing* (or *continuous*) *queries*, i.e., queries that are deployed once and continue to produce results until removed. Standing queries can be executed periodically or continuously as new stream items arrive. They introduce a new interaction model w.r.t. traditional DBMSs: users do not have to explicitly ask for updated information, rather the system actively notifies it according to installed queries. This form of interaction is usually called *Database-Active Human-Passive* (*DAHP*) [18].

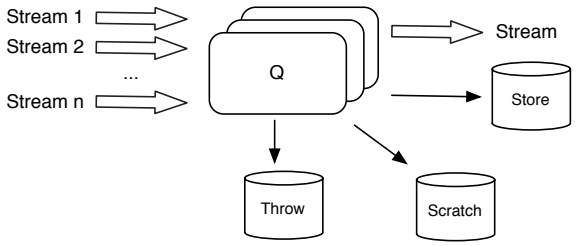


Figure 2.2: The typical model of a DSMS

Several implementations were proposed for DSMSs. They differ in the semantics they associate to standing queries. In particular, the answer to a query can be seen as an append only output stream, or as an entry in a storage that is continuously modified as new elements flow inside the processing stream. Also, an answer can be either exact, if the system is supposed to have enough memory to store all the required elements of input streams' history, or approximate, if computed on a portion of the required history [19, 20].

In Figure 2.2 we report a general model for DSMSs, directly taken from [21]. The purpose of this model is to make several architectural choices and their consequences explicit. A DSMS is modeled as a set of standing queries Q, one or more *input streams* and four possible outputs:

- the *Stream* is formed by all the elements of the answer that are produced once and never changed;
- the *Store* is filled with parts of the answer that may be changed or removed at a certain point in the future. The *Stream* and the *Store* together define the current answer to queries Q;
- the *Scratch* represents the working memory of the system, i.e. a repository where it is possible to store data that is not part of the answer, but that may be useful to compute the answer;

- the *Throw* is a sort of recycle bin, used to throw away unneeded tuples.

To the best of our knowledge, the model described above is the most complete to define the behavior of DSMSs. It explicitly shows how DSMSs alone cannot entirely cover the needs for IFP: being an extension of database systems, DSMSs focus on producing query answers, which are continuously updated to adapt to the constantly changing contents of their input data. Detection and notification of complex patterns of elements involving sequences and ordering relations are usually out of the scope of these systems.

Complex Event Processing Systems

The limitation above originates from the same nature of DSMSs, which are generic systems that leave to their clients the responsibility of associating a semantics to the data being processed. *Complex Event Processing* (*CEP*) *Systems* adopt the opposite approach. As shown in Figure 2.3, they associate a precise semantics to the information items being processed: they are *notifications* of *events* happened in the external world and *observed* by sources. The CEP engine is responsible for filtering and combining such notifications to understand what is happening in terms of *higher-level* events (sometimes also called *composite events* or *situations*) to be notified to sinks, which act as *event consumers*.

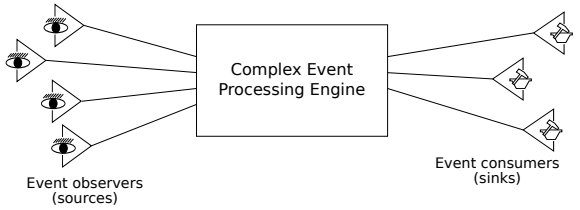


Figure 2.3: The high-level view of a CEP system

Historically, the first event processing engines [22] focused on filtering incoming notifications to extract only the relevant ones, thus supporting an interaction style known as *publish-subscribe*. This is a message oriented interaction paradigm based on an indirect addressing style. Users express their interest in receiving some information by *subscribing* to specific classes of events, while information sources *publish* events without directly addressing the receiving parties. These are dynamically chosen by the publish-subscribe engine based on the received subscriptions.

Conventional publish-subscribe comes in two flavors: topic and content-based [10]. *Topic-based* systems allow sinks to subscribe only to predefined topics. Publishers choose the topic each event belongs to before publishing. *Content-based systems* allow subscribers to use complex *event filters* to specify the events they want to receive based on their content. Several languages have been used to represent event content and subscription filters: from simple attribute/value pairs [23] to complex XML schema [24, 25].

Whatever language is used, subscriptions may refer to single events only [26] and cannot take into account the history of already received events or relationships between events. To this end, CEP systems can be seen as an extension to traditional publish-subscribe, which allow subscribers to express their interest in *composite events*. As their name suggests, these are events whose occurrence depends on the occurrence of other events, like the fact that a fire is detected when three different sensors, located in an area smaller than $100\,m^2$, report a temperature greater than 60^oC, within $10\,sec.$ one from the other.

CEP systems put great emphasis on the issue that represents the main limitation of most DSMSs: the ability to detect complex patterns of incoming items, involving sequencing and ordering relationships. Indeed, the CEP model relies on the ability to specify composite events through *event patterns* that match incoming event notifications on the basis of their content and on some ordering relationships on them.

This feature has also an impact on the architecture of CEP engines. In fact, these tools often have to interact with a large number of distributed and heterogeneous information sources and sinks, which observe the external world and operate on it. This is typical of most CEP scenarios, such as environmental monitoring, business process automation, and control systems. This also suggested, at least in the most advanced proposals, the adoption of a distributed architecture for the CEP engine itself, organized (see Figure 2.4) as a set of *event brokers* (or *event processing agents* [27]) connected in an *overlay network* (the *event processing network*), which implements specialized routing and forwarding functions, with scalability as their main concern. For the same reason, CEP systems research focused on optimizing parameters, like bandwidth utilization and end-to-end latency, which are usually ignored in DSMSs.

CEP systems can thus be classified on the basis of the architecture of the CEP engine (centralized, hierarchical, acyclic, peer-to-peer) [28, 10], the forwarding schemes adopted by brokers [29], and the way processing of event patterns is distributed among brokers [30, 31].

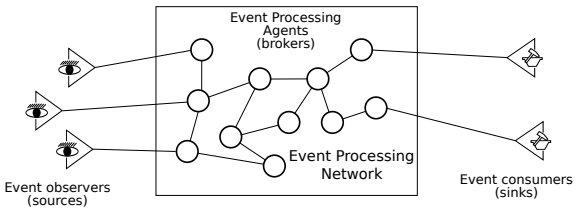


Figure 2.4: CEP as a distributed service

2.2.3 Our Motivation

At the beginning of this section we introduced IFP as a new application domain demanding timely processing information flows according to their content and to the relationships among its constituents. IFP engines have to filter, combine, and aggregate a huge amount of information according to a given set of processing rules. Moreover, they usually need to interact with a large number of heterogeneous information sources and sinks, possibly dispersed over a wide geographical area. Users of such systems can be high-level applications, other processing engines, or end-users, possibly in mobile scenarios. For these reasons expressiveness, scalability, and flexibility are key requirements in the IFP domain.

We have seen how IFP has been addressed by different communities, leading to two classes of systems: active databases, lately subsumed by DSMSs on one side, and CEP engines on the other. DSMSs mainly focuses on flowing data and data transformations. Only few approaches allow the easy capture of sequences of data involving complex ordering relationships, not to mention taking into account the possibility to perform filtering, correlation, and aggregation of data directly in-network, as streams flow from sources to sinks. Finally, to the best of our knowledge, network dynamics and heterogeneity of processing devices have never been studied in depth.

On the other hand, CEP engines, both those developed as extensions of publish-subscribe middleware and those developed as totally new systems, define a quite different model. They focus on processing event notifications, with a special attention to their ordering relationships, to capture complex event patterns. Moreover, they target the communication aspects involved in event processing, such as the ability to adapt to different network topologies, as well as to various processing devices, together with the ability of processing information directly in-network, to distribute the load and reduce communication cost.

As we claimed in Section 2.1, IFP should focus both on effective data processing, including the ability to capture complex ordering relationships among data, and on efficient event delivery, including the ability to process data in a strongly distributed fashion. To pursue these goals, researchers must be able to understand what has been accomplished in the two areas, and how they complement each other in order to combine them. As we will see in Section 2.4, some of the most advanced proposals in both areas go in this direction. This work is a step in the same direction, as it provides an extensive framework to model an IFP system and use it to classify and compare existing systems.

2.3 The Modelling Framework

In this section we introduce our modelling framework to compare the different proposals in the area of IFP. It includes several models that focus on the different aspects relevant for an IFP system.

2.3.1 Functional Model

Starting from the high-level description of Figure 2.1, we define an abstract architecture that describes the main functional components of an IFP engine. This architecture brings two contributions: (i) it allows a precise description of the functionalities offered by an IFP engine; (ii) it can be used to describe the differences among the existing IFP engines, providing a versatile tool for their comparison.

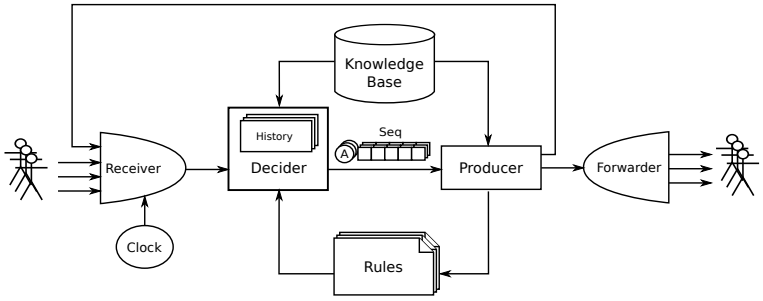


Figure 2.5: The functional architecture of an IFP system

As we said before, an IFP engine takes flows of information coming from different sources as its input, processes them, and produces other flows of information directed toward a set of sinks. Processing rules describe how to filter, combine, and aggregate incoming information to produce outgoing information. This general behavior can be decomposed

in a set of elementary actions performed by the different components shown in Figure 2.5.

Incoming information flows enter the *Receiver*, whose task is to manage the channels connecting the sources with the IFP engine. It implements the transport protocol adopted by the engine to move information around the network. It also acts as a demultiplexer, receiving incoming items from multiple sources and sending them, one by one, to the next component in the IFP architecture. As shown in figure, the *Receiver* is also connected to the *Clock*: the element of our architecture that is in charge of periodically creating special information items that hold the current time. Its role is to model those engines that allow periodic (as opposed to purely reactive) processing of their inputs, as such, not all engines currently available implement it.

After traversing the *Receiver*, the information items coming from the external sources or generated by the *Clock* enter the main processing pipe, where they are elaborated according to the processing rules currently stored into the *Rules* store.

From a logical point of view we find important to consider rules as composed by two parts: $C \rightarrow A$, where C is the *condition part*, while A is the *action part*. The condition part specifies the constraints that have to be satisfied by the information items entering the IFP engine to trigger the rule, while the action part specifies what to do when the rule is triggered.

This distinction allows us to split information processing into two phases: a *detection phase* and a *production phase*. The former is realized by the *Decider*, which gets incoming information from the *Receiver*, item by item, and looks at the condition part of rules to find those enabled. The action part of each triggered rule is then passed to the *Producer* for execution.

Notice that the *Decider* may need to accumulate information items into a local storage until the constraints of a rule are entirely satisfied. As an example, consider a rule stating that a fire alarm has to be generated (action part) when both smoke and high temperature are detected in the same area (condition part). When information about smoke reaches the *Decider*, the rule cannot fire, yet, but the *Decider* has to record this information to trigger the rule when high temperature is detected. We model the presence of such "memory" through the *History* component inside the *Decider*.

The detection phase ends by passing to the *Producer* an action A and a sequence of information items Seq for each triggered rule, i.e., those items (accumulated in the *History* by the *Decider*) that actually triggered the rule. The action A describes how the information items in Seq have to be

manipulated to produce the expected results of the rule. The maximum allowed length of the sequence *Seq* is an important aspect to characterize the expressiveness of the IFP engine. There are engines where *Seq* is composed by a *single* item; in others the maximum length of *Seq* can be determined by looking at the set of currently deployed rules (we say that *Seq* is *bounded*); finally, *Seq* is *unbounded* when its maximum length depends on the information actually entering the engine.

The *Knowledge Base* represents, from the perspective of the IFP engine, a read-only memory that contains information used during the detection and production phases. This component is not present in all IFP engines; usually, it is part of those systems, developed by the database research community, which allow accessing persistent storage (typically in the form of database tables) during information processing. Issues about initialization and management of the *Knowledge Base* are outside the scope of an IFP engine, which simply uses it as a pre-existing repository of stable information.

Finally, the *Forwarder* is the component in charge of delivering the information items generated by the *Producer* to the expected sinks. Like the *Receiver*, it implements the protocol to transport information along the network up to the sinks.

In summary, an IFP engine operates as follows: each time a new item (including those periodically produced by the *Clock*) enters the engine through the *Receiver*, a *detection-production cycle* is performed. Such a cycle first (detection phase) evaluates all the rules currently present in the *Rules* store to find those whose condition part is true. Together with the newly arrived information, this first phase may also use the information present in the *Knowledge Base*. At the end of this phase we have a set of rules that have to be executed, each coupled with a sequence of information items: those that triggered the rule and that were accumulated by the *Decider* in the *History*. The *Producer* takes this information and executes each triggered rule (i.e., its action part). In executing rules, the *Producer* may combine the items that triggered the rule (as received by the *Decider*) together with the information present in the *Knowledge Base* to produce new information items. Usually, these new items are sent to sinks (through the *Forwarder*), but in some engines they can also be sent internally, to be processed again (*recursive processing*). This allows information to be processed in steps, by creating new knowledge at each step, e.g., the composite event "fire alert", generated when smoke and high temperature are detected, can be further processed to notify a "chemical alert" if the area of fire (included in the "fire alert" event) is close to a chemical deposit. Finally, some engines allow actions to change the rule set by adding new rules or removing them.

2.3.2 Processing Model

Together with the last information item entering the *Decider*, the set of deployed rules, and the information stored in the *History* and in the *Knowledge Base*, three additional concepts concur at uniquely determining the output of a single detection-production cycle: the *selection*, *consumption*, and *load shedding* policies adopted by the system.

Selection policy. In presence of situations in which a single rule R may fire more than once, picking different items from the *History*, the selection policy [32] specifies if R has to fire once or more times, and which items are actually selected and sent to the *Producer*.

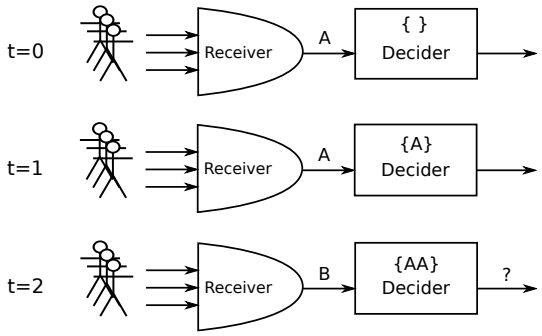


Figure 2.6: Selection policy: an example

As an example, consider the situation in Figure 2.6, which shows the information items (modeled as capital letters) sent by the *Receiver* to the *Decider* at different times, together with the information stored by the *Decider* into the *History*. Suppose that a single rule is present in the system, whose condition part (the action part is not relevant here) is $A \wedge B$, meaning that something has to be done each time both A and B are detected, in any order. At $t = 0$ information A exits the *Receiver* starting a detection-production cycle. At this time the pattern $A \wedge B$ is not detected, but the *Decider* has to remember that A has been received since this can be relevant to recognize the pattern in the future. At $t = 1$ a new A exits the *Receiver*. Again, the pattern cannot be detected, but the *Decider* stores the new A into the *History*. Things change at $t = 2$, when information B exits the *Receiver* triggering a new detection-production cycle. This time not only the condition part of the rule is satisfied, but it is satisfied by two possible sets of items: one includes item B with the first A received, the other includes the same B with the second A received.

Systems that adopt the *multiple* selection policy allow each rule to fire

more than once at each detection-production cycle. This means that in the situation above the *Decider* would send two sequences of items to the *Producer*, one including item B followed by the A received at $t = 0$, the other including item B followed by the A received at $t = 1$.

Conversely, systems that adopt a *single* selection policy allow rules to fire at most once at each detection-production cycle. In the same situation above the *Decider* of such systems would choose one among the two As received, sending a single sequence to the *Producer*: the sequence composed of item B followed by the chosen A. It is worth noting that the single selection policy actually represents a whole family of policies, depending on the items actually chosen among all possible ones. In our example, the *Decider* could select the first or the second A.

Finally, some systems offer a *programmable* policy, by including special language constructs that enable users to decide, often rule by rule, if they have to fire once or more than once at each detection-production cycle, and in the former case, which elements have to be actually selected among the different possible combinations.

Consumption policy. Related with the selection policy is the consumption policy [32], which specifies whether or not an information item selected in a given detection-production cycle can be considered again in future processing cycles.

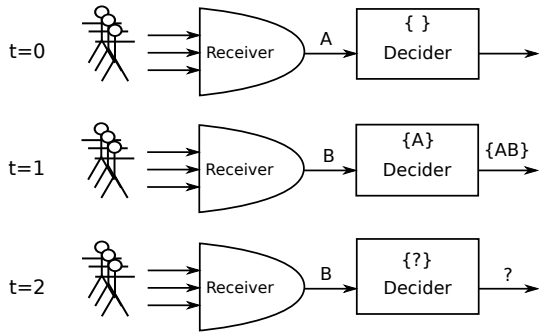


Figure 2.7: Consumption policy: an example

As an example, consider the situation in Figure 2.7, where we assume again that the only rule deployed into the system has a condition part $A \wedge B$. At time $t = 0$, an instance of information A enters the *Decider*; at time $t = 1$, the arrival of B satisfies the condition part of the rule, so item A and item B are sent to the *Producer*. At time $t = 2$, a new instance of B enters the system. In such a situation, the consumption policy determines if pre-existing items A and B have still to be taken into consideration to decide if rule $A \wedge B$ is satisfied. The systems that

adopt the *zero* consumption policy do not invalidate used information items, which can trigger the same rule more than once. Conversely, the systems that adopt the *selected* consumption policy "consume" all the items once they have been selected by the *Decider*. This means that an information item can be used at most once for each rule.

The zero consumption policy is the most widely adopted in DSMSs. In fact, DSMSs' usually introduce special language constructs (windows) to limit the portion of an input stream from which elements can be selected (i.e. the valid portion of the *History*); however, if an item remains in a valid window for different detection-production cycles, it can trigger the same rule more than once. Conversely, CEP systems may adopt either the zero or the selected policy depending from the target application domain and from the processing algorithm used.

As it happens for the selection policy, some systems offer a *programmable* selection policy, by allowing users to explicitly state, rule by rule, which selected items should be consumed after selection and which have to remain valid for future use.

Load shedding. Load shedding is a technique adopted by some IFP systems to deal with bursty inputs. It can be described as an automatic drop of information items when the input rate becomes too high for the processing capabilities of the engine [19]. In our functional model we let the *Decider* be responsible for load shedding, since some systems allow users to customize its behavior (i.e., deciding when, how, and what to shed) directly within the rules. Clearly, those systems that adopt a fixed and pre-determined load shedding policy could, in principle, implement it into the *Receiver*.

2.3.3 Deployment Model

Several IFP applications include a large number of sources and sinks, possibly dispersed over a wide geographical area, producing and consuming a large amount of information that the IFP engine has to process in a timely manner. Hence, an important aspect to consider is the *deployment architecture* of the engine, i.e., how the components that implement the functional architecture in Figure 2.5 can be distributed over multiple nodes to achieve scalability.

We distinguish between *centralized* vs. *distributed* IFP engines, further differentiating the latter into *clustered* vs. *networked*.

With reference to Figure 2.8, we define as *centralized* an IFP engine in which the actual processing of information flows coming from sources is realized by a single node in the network, in a pure client-server archi-

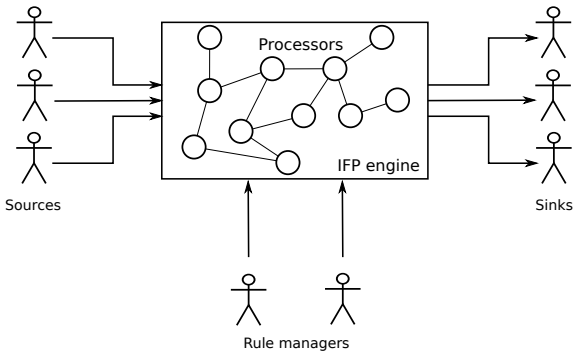


Figure 2.8: The deployment architecture of an IFP engine

tecture. The IFP engine acts as the server, while sources, sinks, and rule managers acts as clients.

To provide better scalability, a *distributed* IFP engine processes information flows through a set of *processors*, each running on a different node of a computer network, which collaborate to perform the actual processing of information. The nature of this network of processors allows to further classify distributed IFP engines. In a *clustered* engine scalability is pursued by sharing the effort of processing incoming information flows among a cluster of strongly connected machines, usually part of the same local area network. In a clustered IFP engine, the links connecting processors among themselves perform much better than the links connecting sources and sinks with the cluster itself. Furthermore, the processors are in the same administrative domain. Several advanced DSMSs are clustered.

Conversely, a *networked* IFP engine focuses on minimizing network usage by dispersing processors over a wide area network, with the goal of processing information as close as possible to the sources. As a result, in a networked IFP engine the links among processors are similar to the links connecting sources and sinks to the engine itself (usually, the closer processor in the network is chosen to act as an entry point to the IFP engine), while processors are widely distributed and usually run in different administrative domains. Some CEP systems adopt this architecture.

In summary, in their seek for better scalability, clustered and networked engines focus on different aspects: the former on increasing the available processing power by sharing the workload among a set of well

connected machines, the latter on minimizing bandwidth usage by processing information as close as possible to the sources.

2.3.4 Interaction Model

With the term *interaction model* we refer to the characteristics of the interaction among the main components that form an IFP application. With reference to Figure 2.1, we distinguish among three different submodels. The *observation model* refers to the interaction between information sources and the IFP engine. The *notification model* refers to the interaction between the engine and the sinks. The *forwarding model* defines the characteristics of the interaction among processors in the case of a distributed implementation of the engine.

We also distinguish between a *push* and a *pull* interaction style [22, 33]. In a pull observation model, the IFP engine is the initiator of the interaction to bring information from sources to the engine; otherwise we have a push observation model. Likewise, in a pull notification model, the sinks have the responsibility of pulling information relevant for them from the engine; otherwise we have a push notification model. Finally, the forwarding model is pull if each processor is reponsible for pulling information from the processor upstream in the chain from sources to sinks; we have a push model otherwise. While the push style is the most common in the observation model, notification model, and especially in the forwarding model, a few systems exist, which prefer a pull model or supports both. We survey them in Section 2.4.

2.3.5 Data Model

The various IFP systems available today differ in how they represent single information items flowing from sources to sinks, and in how they organize them in flows. We collectively refer to both issues with the term *data model*.

As we mentioned in Section 2.2, one of the aspects that distinguishes DSMSs from CEP systems is that the former manipulate generic data items, while the latter manipulates event notifications. We refer to this aspect as the *nature of items*. It represent a key issue for an IFP system as it impacts several other aspects, like the rule language. We come back to this issue later.

A further distinction among IFP systems, orthogonal with respect to the nature of the information items they process, is the way such information items are actually represented, i.e., their *format*. The main formats adopted by currently available IFP systems are *tuples*, either

typed or untyped; *records*, organized as sets of key-value pairs; *objects*, as in object-oriented languages or databases; or *XML* documents.

A final aspect regarding information items is the ability of an IFP system to deal with *uncertainty*, modeling and representing it explicitly when required [34, 35]. In many IFP scenarios, in fact, information received from sources has an associated degree of uncertainty. As an example, if sources were only able to provide rounded data [35], the system could associate such data with a level of uncertainty instead of accepting it as a precise information.

Besides single information items, we classify IFP systems based on the nature of information flows, distinguishing between systems whose engine processes *homogeneous* information flows and those that may also manage *heterogeneous* flows.

In the first case, all the information items in the same flow have the same format, e.g., if the engine organizes information items as tuples, all the items belonging to the same flow must have the same number of fields. Most DSMSs belong to this class. They view information flows as typed data streams, which they manage as transient, unbounded database tables, to be filtered and transformed while they get filled by data flowing into the system.

In the other case, engines allow different types of information items in the same flow; e.g., one record with four fields, followed by one with seven, and so on. Most CEP engines belong to this class. Information flows are viewed as heterogeneous channels connecting sources with the CEP engine. Each channel may transport items (i.e., events) of different types.

2.3.6 Time Model

With the term *time model* we refer to the relationship between the information items flowing into the IFP engine and the passing of time. More precisely, we refer to the ability of the IFP system of associating some kind of *happened-before* relationship [36] to information items.

As we mentioned in Section 2.2, this issue is very relevant as a whole class of IFP systems, namely CEP engines, is characterized by the of event notifications, a special kind of information strongly related with time. Moreover, the availability of a native concept of ordering among items has often an impact on the operators offered by the rule language. As an example, without a total ordering among items it is not possible to define sequences and all the operators meant to operate on sequences.

Focusing on this aspect we distinguish four cases. First of all there are systems that do not consider this issue as prominent. Several DSMSs, in

fact, do not associate any special meaning to time. Data flows into the engine within streams but timestamps (when present) are used mainly to order items at the frontier of the engine (i.e., within the *receiver*), and they are lost during processing. The only language construct based on time, when present, is the windowing operator (see Section 2.3.8), which allows one to select the set of items received in a given time-span. After this step the ordering is lost and timestamps are not available for further processing. In particular, the ordering (and timestamps) of the output stream are conceptually separate from the ordering (and timestamps) of the input streams. In this case we say that the time model is *stream-only*.

At the opposite of the spectrum are those systems, like most CEP engines, which associate each item with a timestamp that represent an absolute time, usually interpreted as the time of occurrence of the related event. Hence, timestamps define a total ordering among items. In such systems timestamps are fully exposed to the rule language, which usually provide advanced constructs based on them, like sequences (see Section 2.3.8). Notice how such kind of timestamps can be given by sources when the event is observed, or by the IFP engine as it receives each item. In the former case, a buffering mechanism is required to cope with out-of-order arrivals [9].

Another case is that of systems that associate each item with some kind of label, which, while not representing an absolute instant in time, can define a partial ordering among items, usually reflecting some kind of causal relationship, i.e., the fact that the occurrence of an event was caused by the occurrence of another event. Again, what is important for our model is the fact that such labels and the ordering they impose are fully available to the rule language. In this case we say that the time model is *causal*.

Finally, there is the case of systems that associate items with an interval, i.e., two timestamps taken from a global time, usually representing: the time when the related event started, the time when it ended. In this case, depending on the semantics associated with intervals, a total or a partial ordering among items can be defined [37, 38, 39].

2.3.7 Rule Model

Rules are much more complex entities than data. Looking at existing systems we can find many different approaches to represent rules, which depend on the adopted rule language. However, we can roughly classify them in two macro classes: *transforming rules* and *detecting rules*.

Transforming rules define an *execution plan* composed of *primitive operators* connected to each other in a graph. Each operator takes several

flows of information items as inputs and produces new items, which can be forwarded to other operators or directly sent out to sinks. The execution plan can be either user-defined or compiled. In the first case, rule managers are allowed to define the exact flow of operators to be executed; in the second case, they write their rules in a high-level language, which the system compiles into an execution plan. The latter approach is adopted by a wide number of DSMSs, which express rules using SQL-like statements.

As a final remark we notice that transforming rules are often used with homogeneous information flows, so that the definition of an execution plan can take advantage of the predefined structure of input and output flows.

Detecting rules are those that present an explicit distinction between a condition and an action part. Usually, the former is represented by a logical predicate that captures *patterns* of interest in the sequence of information items, while the latter uses ad-hoc constructs to define how relevant information has to be processed and aggregated to produce new information. Examples of detecting rules can be found in many CEP systems, where they are adopted to specify how new events originate from the detection of others. Other examples can be found in active databases, which often use detecting rules to capture undesired sequences of operations within a transaction (the condition), in order to output a *roll-back* command (the action) as soon as the sequence is detected.

The final issue we address in our rule model is the ability to deal with *uncertainty* [34, 35, 40]. In particular, some systems allow to distinguish between *deterministic* and *probabilistic* rules. The former define their outputs deterministically from their inputs, while the latter allow a degree of uncertainty (or a confidence) to be associated with the outputs. Notice that the issue is only partially related with the ability of managing uncertain data (see Section 2.3.5). Indeed, a rule could introduce a certain degree of uncertainty even in presence of definite and precise inputs. As an example, one could say that there is a good probability of fire if a temperature higher than 70^oC is detected. The input information is precise, but the rule introduce a certain degree of uncertainty of knowledge (other situations could explain the raising of temperature).

2.3.8 Language Model

The rule model described in the previous section provides a first characterization of the languages used to specify processing rules in currently available IFP systems. Here we give a more precise and detailed description of such languages: in particular, we first define the general classes

into which all existing languages can be divided and then we provide an overview of all the operators we encountered during the analysis of existing systems. For each operator we specify if and how it can be defined inside the aforementioned classes.

Language Type

Following the classification introduced into the rule model, the languages used in existing IFP systems can be divided into the following two classes:

- *Transforming languages* define transforming rules, specifying one or more operations that process the input flows by filtering, joining, and aggregating received information to produce one or more output flows. Transforming languages are the most commonly used in DSMSs. They can be further divided into two classes:
  - *Declarative languages* express processing rules in a declarative way, i.e. by specifying the expected results of the computation rather than the desired execution flow. These languages are usually derived from relational languages, in particular relational algebra and SQL [41], which they extend with additional ad-hoc operators to better support flowing information.
  - *Imperative languages* define rules in an imperative way, by letting the user specify a plan of primitive operators, which the information flows have to follow. Each primitive operator defines a transformation over its input. Usually, systems that adopt imperative languages offer visual tools to define rules.
- *Detecting, or pattern-based languages* define detecting rules by separately specifying the firing conditions and the actions to be taken when such conditions hold. Conditions are usually defined as *patterns* that select matching portions of the input flows using logical operators, content and timing constraints; actions define how the selected items have to be combined to produce new information. This type of languages is common in CEP systems, which aim to detect relevant information items before starting their processing.

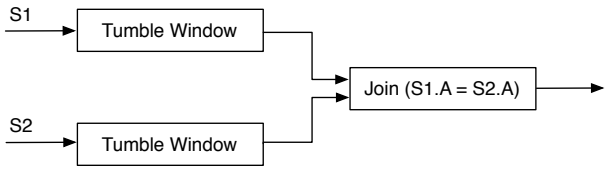
It is worth mentioning that existing systems sometime allow users to express rules using more than one paradigm. For example, many commercial systems offer both a declarative language for rule creation, and a graphical tool for connecting defined rules in an imperative way. Moreover, some existing declarative languages embed simple operators for pattern detection, blurring the distinction between transforming and detecting languages.

As a representative example for declarative languages, let us consider CQL [42], created within the Stream project [43] and currently adopted by Oracle [44]. CQL defines three classes of operators: relation-to-relation operators are similar to the standard SQL operators and define queries over database tables; stream-to-relation operators are used to build database tables selecting portions of a given information flow, while relation-to-stream operators create flows starting from queries on fixed tables. Stream-to-stream operators are not explicitly defined, as they can be expressed using the existing ones. Here is an example of a CQL rule:

(2.1)
```
Select IStream(*)
From F1 [Rows 5], F2 [Rows 10]
Where F1.A = F2.A
```

This rule isolates the last 5 elements of flow F1 and the last 10 elements of flow F2 (using the stream-to-relation operator `[Rows n]`), then combines all elements having a common attribute A (using the relation-to-relation operator `Where`) and produces a new flow with the created items (using the relation-to-stream operator `IStream`).

Imperative languages are well represented by Aurora's Stream Query Algebra (SQuAl), which adopts a graphical representation called *boxes and arrows* [45]. In Figure 2.2 we show how it is possible to express a rule similar to the one we introduced in Example 2.1 using Aurora's language. The tumble window operator selects portions of the input stream according to given constrains, while join merges elements having the same value for attribute A.

(2.2)


Detecting languages can be exemplified by the composite subscription language of Padres [46]. Example 2.3 shows a rule expressed in such language.

(2.3) $\texttt{A(X>0) \& (B(Y=10);}_{\texttt{[timespan:5]}}\ \texttt{C(Z<5))}_{\texttt{[within:15]}}$

A, B, and C represent item types or topics, while X, Y, and Z are inner fields of items. After the topic of an information item has been determined, filters on its content are applied as in content-based publish-subscribe systems. The rule of Example 2.3 fires when an item of type A having an attribute X>0 enters the systems and also an item of type B with Y=10 is detected, followed (in a time interval of 5 to 15 sec.) by an item of type C with Z<5. Like other systems conceived as an evolution of traditional publish-subscribe middleware, Padres defines only the detection part of rules, the default and implicit action being that of forwarding a notification of the detected pattern to proper destinations. However, more expressive pattern languages exist, which allow complex transformations to be performed on selected data.

In the following, whenever possible, we use the three languages presented above to build the examples we need.

Available Operators

We now provide a complete list of all the operators that we found during the analysis of existing IFP systems. Some operators are typical of only one of the classes defined above, while others cross the boundaries of classes. In the following we highlight, whenever possible, the relationship between each operator and the language types in which it may appear.

In general there is no direct mapping between available operators and language expressiveness: usually it is possible to express the same rule combining different sets of operators. Whenever possible we will show such equivalence, especially if it involves operators provided by different classes of languages.

Single-Item Operators. A first kind of operators provided by existing IFP systems are *single-item* operators, i.e. those processing information items one by one. Two classes of single-item operators exist:

- *Selection operators* filter items according to their content, discarding elements that do not satisfy a given constraint. As an example, they can be used to keep only the information items that contain temperature readings whose value is greater than 20^oC.
- *Elaboration operators* transform information items. As an example, they can be used to change a temperature reading, converting from Celsius to Fahrenheit. Among elaboration operators, it is worth mentioning those coming from relational algebra, and in particular:
  - *Projection* extracts only part of the information contained in the considered item. As an example, it is used to process

items containing data about a person in order to extract only the name.

– *Renaming* changes the name of a field in languages based on records.

Single-item operators are always present; declarative languages usually inherit selection, projection, and renaming from relational algebra, while imperative languages offer primitive operators both for selection and for some kind of elaboration. In pattern-based languages selection operators are used to select those items that should be part of a complex pattern. Notably, they are the only operators available in publish-subscribe systems to choose items to be forwarded to sinks. When allowed, elaborations can be expressed inside the action part of rules, where it is possible to define how selected items have to be processed.

Logic Operators. Logic operators are used to define rules that combine the detection of several information items. They differ from sequences (see below) in that they are *order independent*, i.e., they define patterns that rely only on the detection (or non detection) of information items and not on some specific ordering relation holding among them.

- A *conjunction* of items I_1, I_2, .. I_n is satisfied when all the items I_1, I_2, .. I_n have been detected.
- A *disjunction* of items I_1, I_2, .. I_n is satisfied when at least one of the information items I_1, I_2, .. I_n has been detected.
- A *repetition* of an information item I of degree $\langle m, n \rangle$ is satisfied when I is detected at least m times and not more than n times (it is a special form of conjunction).
- A *negation* of an information item I is satisfied when I is not detected.

Usually it is possible to combine such operators with each others or with other operators to form more complex patterns. For example, it is possible to specify disjunctions of conjunctions of items, or to combine logic operators and single-item operators to dictate constraints both on the content of each element and on the relationships among them.

The use of logic operators allows the definition of expressions whose value cannot be verified in a finite amount of time, unless explicit bounds are defined for them. This is the case of repetition and negation, which require elements to remain undetected in order to be satisfied. For this

reason existing systems combine these operators with other linguistic constructs known as *windows* (see below).

Logic operators are always present in pattern-based languages, where they represent the typical way to combine information items. Example 2.4 shows how a disjunction of two conjunctions can be expressed using the language of Padres (A, B, C, and D are simple selections).

(2.4)
```
(A & B) || (C & D)
```

On the contrary, declarative and imperative languages do not provide logic operators explicitly; however they usually allow conjunctions, disjunctions, and negations to be expressed using rules that transform input flows. Example 2.5 shows how a conjunction can be expressed in CQL: the `From` clause specifies that we are interested in considering data from both flows F1 **and** F2.

(2.5)
```
Select IStream(F1.A, F2.B)
From F1 [Rows 50], F2 [Rows 50]
```

Sequences. Similarly to logic operators, *sequences* are used to capture the arrival of a set of information items, but they take into consideration the order of arrival. More specifically, a sequence defines an ordered set of information items I_1, I_2, .. I_n, which is satisfied when all the elements I_1, I_2, .. I_n have been detected in the specified order (see Section 2.3.6).

The sequence operator is present in many pattern-based languages, while transforming languages usually do not provide it explicitly. Still, in such languages it is sometimes possible (albeit less natural) to mimic sequences, for example when the ordering relation is based on a timestamp field explicitly added to information items, as in the following example: [1]

(2.6)
```
Select IStream(F1.A, F2.B)
From F1 [Rows 50], F2 [Rows 50]
Where F1.timestamp < F2.timestamp
```

[1] CQL adopts a stream-based time model. It associates implicit timestamps to information items and use them in time-based windows, but they cannot be explicitly addressed within the language; consequently they are not suitable to define sequences.

Iterations. *Iterations* express possibly unbounded sequences of information items satisfying a given *iterating condition*. Like sequences, iterations rely on the ordering of items. However, they do not define the set of items to be captured explicitly, but rather implicitly using the iterating condition.

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
a.type = 'contaminated' and
b[1].from = a.site and
b[i].from = b[i-1].to }
WITHIN 3 hours
```
(2.7)

Example 2.7 shows an iteration written in the Sase+ language [47]. The rule detects contamination in a food supply chain: it captures an alert for a contaminated site (item a) and reports all possible series of infected shipments (items $b[i]$). Iteration is expressed using the + operator (usually called *Kleene plus* [48]), defining sequences of one or more *Shipment* information items. The iterating condition $b[i].from = b[i-1].to$ specifies the collocation condition between each shipment and the preceding one. *Shipment* information items need not to be contiguous within the input flow; intermediate items are simply discarded ($skip\_till\_any\_match$). The length of captured sequences is not known a priori but it depends on the actual number of shipments from site to site. To ensure the termination of pattern detection, a time bound is expressed, using the *WITHIN* operator (see *Windows* below).

While most pattern-based languages include a sequence operator, it is less common to find iterations. In addition, like sequences, iterations are generally not provided in transforming languages. However, some work investigated the possibility to include sequences and iterations in declarative languages (e.g., [49]). These efforts usually result in embedding pattern detection into traditional declarative languages.

As a final remark, iterations are strictly related with the possibility for an IFP system to read its own output and to use it for recursive processing. In fact, if a system provides both a sequence operator and recursive processing, it can mimic iterations through recursive rules.

Windows. As mentioned before, it is often necessary to define which portions of the input flows have to be considered during the execution of operators. For this reason almost all the languages used in existing systems define *windows*. Windows cannot be properly considered as

operators; rather, they are language constructs that can be applied to operators to limit the scope of their action.

To be more precise, we can observe that operators can be divided into two classes: *blocking operators*, which need to read the whole input flows before producing results, and *non-blocking operators*, which can successfully return their results as items enter the system [9]. An example of a blocking operator is negation, which has to process the entire flow before deciding that the searched item is not present. The same happens to repetitions when an upper bound on the number of items to consider is provided. On the contrary, conjunctions and disjunctions are non-blocking operators, as they can stop parsing their inputs as soon as they find the searched items. In IFP systems information flows are, by nature, unbounded; consequently, it is not possible to evaluate blocking operators as they are. In this context, windows become the key construct to enable blocking operators by limiting their scope to (finite) portions of the input flows. On the other hand, windows are also extensively used with non-blocking operators, as a powerful tool to impose a constraint over the set of items that they have to consider.

Existing systems define several types of windows. First of all they can be classified into *logical* (or *time-based*) and *physical* (or *count-based*) [50]. In the former case, bounds are defined as a function of time: for example to force an operation to be computed only on the elements that arrived during the last five minutes. In the latter case, bounds depend on the number of items included into the window: for example to limit the scope of an operator to the last 10 elements arrived.

An orthogonal way to classify windows considers the way their bounds move, resulting in the following classes [51, 50]:

- *Fixed windows* do not move. As an example, they could be used to process the items received between 1/1/2010 and 31/1/2010.
- *Landmark windows* have a fixed lower bound, while the upper bound advances every time a new information item enters the system. As an example, they could be used to process the items received since 1/1/2010.
- *Sliding windows* are the most common type of windows. They have a fixed size, i.e., both lower and upper bounds advance when new items enter the system. As an example, we could use a sliding window to process the last ten elements received.
- *Pane and tumble windows* are variants of sliding windows in which both the lower and the upper bounds move by k elements, as k elements enter the system. The difference between pane and tumble

windows is that the former have a size greater than k, while the latter have a size smaller than (or equal to) k. In practice, a tumble window assures that every time the window is moved all contained elements change; so each element of the input flow is processed at most once. The same is not true for pane windows. As an example, consider the problem of calculating the average temperature in the last week. If we want such a measure every day at noon we have to use a pane window, if we want it every Sunday at noon we have to use a tumble window.

Example 2.8 shows a CQL rule that uses a count-based, sliding window over the flow F1 to count how many among the last 50 items received has $A > 0$; results are streamed using the IStream operator. Example 2.9 does the same but considering the items received in the last minute.

```
Select IStream(Count(*))
From F1 [Rows 50]
Where F1.A > 0
```
(2.8)

```
Select IStream(Count(*))
From F1 [Range 1 Minute]
Where F1.A > 0
```
(2.9)

Interestingly there exist languages that allow users to define and use their own windows. The most notable case is ESL [52], which provides *user defined aggregates* to allow users to freely process input flows. In so doing, users are allowed to explicitly manage the part of the flow they want to consider, i.e., the window. As an example, consider the ESL rule in Example 2.10: it calculates the smallest positive value received and delivers it as its output every time a new element arrives. The window is accessible to the user as a relational table, called `inwindow`, whose inner format can be specified during the aggregate definition; it is automatically filled by the system when new elements arrive. The user may specify actions to be taken at different times using three clauses: the `INITIATE` clause defines special actions to be executed only when the first information item is received; the `ITERATE` clause is executed every time a new element enters the system, while the `EXPIRE` clause is executed every time an information item is removed from the window. In our example the `INITIATE` and `EXPIRE` clauses are empty, while the `ITERATE` clause removes every incoming element having a negative value, and

immediately returns the smallest value (using the `INSERT INTO RETURN` statement). It is worth noting that the aggregate defined in Example 2.10 can be applied to virtually all kinds of windows, which are then modified during execution in order to contain only positive values.

(2.10)
```
WINDOW AGGREGATE positive_min(Next Real): Real {
  TABLE inwindow(wnext real);
  INITIALIZE : { }
  ITERATE : {
    DELETE FROM inwindow
    WHERE wnext < 0
    INSERT INTO RETURN
    SELECT min(wnext)
    FROM inwindow
  }
  EXPIRE : { }
}
```

Generally, windows are available in declarative and imperative languages. Conversely, only a few pattern-based languages provide windowing constructs. Some of them, in fact, simply do not include blocking operators, while others include explicit bounds as part of blocking operators to make them unblocking (we can say that such operators "embed a window"). For example, Padres does not provide the negation and it does not allow repetitions to include an upper bound. Similarly, CEDR [53] can express negation through the UNLESS operator, shown in Example 2.11. The pattern is satisfied if A is not followed by B within 12 hours. Notice how the operator itself requires explicit timing constraints to become unblocking.

(2.11)
```
EVENT Test-Rule
WHEN UNLESS(A, B, 12 hours)
WHERE A.a < B.b
```

Flow Management Operators. Declarative and imperative languages require ad-hoc operators to merge, split, organize, and process flows of information. They include:

- The *Join* operator, used to merge two flows of information as in traditional DBMS. Being a blocking operator the join is usually

applied to portions of the input flows, which are processed as standard relational tables. As an example, the CQL Rule 2.12 uses the join operator to combine the last 1000 items of flows $F1$ and $F2$ by merging those items that have the same value in field A.

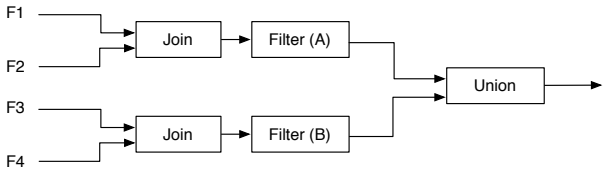
(2.12)
```
Select IStream(F1.A, F2.B)
From F1 [Rows 1000], F2 [Rows 1000]
Where F1.A = F2.A
```

- *Bag operators*, which combine different flows of information considering them as bags of items. In particular, we have the following bag operators:
  - The *union* merges two or more input flows of the same type creating a new flow that includes all the items coming from them.
  - The *except* takes two input flows of the same type and outputs all those items that belong to the first one but not to the second one. It is a blocking operator.
  - The *intersect* takes two or more input flows and outputs only the items included in all of them. It is a blocking operator.
  - The *remove-duplicate* removes all duplicates from an input flow.

  Rule 2.13 provides an example of using the union operator in CQL to merge together two flows of information. Similarly, in Example 2.14 we show the union operator as provided by Aurora.

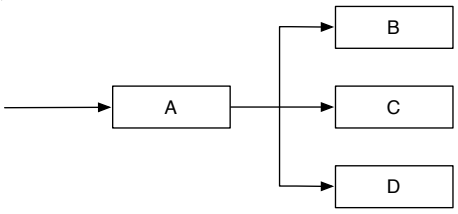
(2.13)
```
Select IStream(*)
From F1 [Rows 1000], F2 [Rows 1000]
Where F1.A = F2.A
Union
Select IStream(*)
From F3 [Rows 1000], F4 [Rows 1000]
Where F3.B = F4.B
```

(2.14)



- The *duplicate* operator allows a single flow to be duplicated in order to use it as an input for different processing chains. Example 2.15 shows an Aurora rule that takes a single flow of items, processes it through the operator `A`, then duplicates the result to have it processed by three operators `B`, `C`, and `D`, in parallel.

  (2.15)



- The *group-by* operator is used to split an information flow into partitions in order to apply the same operator (usually an aggregate) to the different partitions. Example 2.16 uses the group-by operator to split the last 1000 rows of flow $F1$ based on the value of field B, to count how may items exist for each value of B.

  (2.16)

```
Select IStream(Count(*))
From F1 [Rows 1000]
Group By F1.B
```

- The *order-by* operator is used to impose an ordering to the items of an input flow. It is a blocking operator so it is usually applied to well defined portions of the input flows.

Parameterization. In many IFP applications it is necessary to filter some flows of information based on information that are part of other flows. As an example, the fire protection system of a building could be interested in being notified when the temperature of a room exceeds 40^oC but only if some smoke has been detected *in the same room*. Declarative and imperative languages address this kind of situations by joining the two information flows, i.e., the one about room temperature and that about smoke, and imposing the room identifier to be the same. On the other hand, pattern-based languages do not provide the join operator. They have to capture the same situation using a rule that combines, through the conjunction operator, detection of high temperature and detection of smoke in the same room. This latter condition can be expressed only if the language allows filtering to be *parametric*.

Given the importance of this feature we introduce it here even if it cannot be properly considered an operator by itself. More specifically,

we say that a pattern-based language provides parameterization if it allows the parameters of an operator (usually a selection, but sometimes other operators as well) to be constrained using values taken from other operators in the same rule.

(2.17)
```
(Smoke(Room=$X)) & (Temp(Value>40 AND Room=$X))
```

Example 2.17 shows how the rule described above can be expressed using the language of Padres; the operator $ allows parameters definition inside the filters that select single information items.

Flow creation. Some languages define explicit operators to create new information flows from a set of items. In particular, flow creation operators are used in declarative languages to address two issues:

- some of them have been designed to deal indifferently with information flows and with relational tables a-la DBMS. In this case, flow creation operators can be used to create new flows from existing tables, e.g., when new elements are added to the table.
- other languages use windows as a way to transform (part of) an input flow into a table, which can be further processed by using the classical relational operators. Such languages use flow creation operators to transform tables back into flows. As an example, CQL provides three operators called *relation-to-stream* that allow to create a new flow from a relational table T: at each evaluation cycle, the first one (IStream) streams all new elements added to T; the second one (DStream) streams all the elements removed from T; while the last one (Rstream) streams all the elements of T at once. Consider Example 2.18: at each processing cycle the rule populates a relational table (T) with the last ten items of flow F1, and then, it streams all elements that are added to T at the current processing cycle (but were not part of T in the previous cycle). In Example 2.19, instead, the system streams all elements removed from table T during the current processing cycle. Finally, in Example 2.20, all items in T are put in the output stream, independently from previous processing cycles.

  (2.18)
  ```
  Select IStream(*)
  From F1 [Rows 10]
  ```

(2.19)
```
Select DStream(*)
From F1 [Rows 10]
```

(2.20)
```
Select RStream(*)
From F1 [Rows 10]
```

Aggregates. Many IFP applications need to aggregate the content of multiple, incoming information items to produce new information, for example by calculating the average of some value or its maximum. We can distinguish two kinds of aggregates:

- *Detection aggregates* are those used during the evaluation of the condition part of a rule. In our functional model, they are computed and used by the *Decider*. As an example, they can be used in detecting all items whose value exceeds the average one (i.e., the aggregate), computed over the last 10 received items.
- *Production aggregates* are those used to compute the values of information items in the output flow. In our functional model, they are computed by the *Producer*. As an example, they can be used to output the average value among those part of the input flow.

Almost all existing languages have predefined aggregates, which include minimum, maximum, and average. Some pattern-based languages offer only production aggregates, while others include also detection aggregates to capture patterns that involve computations over the values stored in the *History*. In declarative and imperative languages aggregates are usually combined with the use of windows to limit their scope (indeed, aggregates are usually blocking operators and windows allow to process them on-line). Finally, some languages also offer facilities to create *user-defined aggregates (UDAs)*. As an example of the latter, we have already shown the ESL rule 2.10, which defines an UDA to compute the smallest positive value among the received ones. It has been proved that adding complete support to UDAs makes a language Turing-complete [54].

2.4 IFP Systems: a Classification

In the following we use the concepts introduced in Section 2.3 to present and classify existing IFP systems. We provide a brief description of each system and summarize its characteristics by compiling four tables.

Table 2.1 focuses on the functional and processing models of each system: it shows if a system includes the *Clock* (i.e., it supports periodic processing) and the *Knowledge Base*, and if the maximum size of the sequences (*Seq*) that the *Decider* sends to the *Producer* is bounded or unbounded, given the set of deployed rules. Then, it analyzes if information items produced by fired rules may re-enter the system (*recursion*), and if the rule set can be changed by fired actions at run-time. Finally, it presents the processing model of each system, by showing its selection, consumption, and load shedding policies.

Table 2.2 focuses on the deployment and interaction models, by showing the type of deployment supported by each system, and the interaction styles allowed in the observation, notification, and forwarding models.

Table 2.3 focuses on the data, time, and rule models, presenting the nature of the items processed by each systems (generic data or event notifications), the format of data, the support for data uncertainty, and the nature of flows (homogeneous or heterogeneous). It also introduces how the notion of time is captured and represented by each system, the type of rules adopted, and the possibility to define probabilistic rules.

Finally, Table IV focuses on the language adopted by each system, by listing its type and the set of operators available.

Given the large number of systems reviewed, we organize them in four groups: active databases, DSMSs, CEP systems, and commercial systems (while systems belonging to previous groups are research prototypes). The distinction is sometimes blurred but it helps us better organize our presentation.

Within each class we do not follow a strict chronological order: systems that share similar characteristics are listed one after the other, to better emphasize common aspects.

2.4.1 Active databases

HiPac

Hipac [55, 16] has been the first project proposing Event Condition Action (ECA) rules as a general formalism for active databases. In particular, the authors introduce a model for rules in which three kinds of primitive events are taken into account: database operations, temporal events, and external notifications. Primitive events can be combined using disjunction and sequence operators to specify composite events. Since these operators force users to explicitly write all the events that have to be captured, we can say that, given a set of rule, the size of the sequence of events that can be selected (i.e. the size of the *Seq*

	Functional Model				Processing Model			
Name	**Clock**	**K. Base**	**Seq**	**Recursion**	**Dynamic Rule Change**	**Select. Policy**	**Consum. Policy**	**Load Shedding**
HiPac	Present	Present	Bounded	Yes	Yes	Multiple	Zero	No
Ode	Absent	Present	Unbounded	Yes	Yes	Multiple	Zero	No
Samos	Present	Present	Unbounded	Yes	Yes	?	?	No
Snoop	Present	Present	Unbounded	Yes	Yes	Program.	Program.	No
TelegraphCQ	Present	Present	Unbounded	No	No	Multiple	Zero	Yes
NiagaraCQ	Present	Present	Unbounded	No	No	Multiple	Selected	Yes
OpenCQ	Present	Present	Unbounded	No	No	Multiple	Selected	No
Tribeca	Absent	Absent	Unbounded	No	Yes	Multiple	Zero	No
CQL / Stream	Absent	Present	Unbounded	No	No	Multiple	Zero	Yes
Aurora / Borealis	Absent	Present	Unbounded	No	No	Program.	Zero	Yes
Gigascope	Absent	Absent	Unbounded	No	No	Multiple	Zero	No
Stream Mill	Absent	Present	Unbounded	No	No	Program.	Program.	Yes
Traditional Pub-Sub	Absent	Absent	Single	No	No	Single	Single	No
Rapide	Absent	Present	Unbounded	Yes	No	Multiple	Zero	No
GEM	Present	Absent	Bounded	Yes	Yes	Multiple	Zero	No
Padres	Absent	Absent	Unbounded	No	No	Multiple	Zero	No
DistCED	Absent	Absent	Unbounded	No	No	Multiple	Zero	No
CEDR	Absent	Absent	Unbounded	Yes	No	Program.	Program.	No
Cayuga	Absent	Absent	Unbounded	Yes	No	Multiple	Zero	No
NextCEP	Absent	Absent	Unbounded	Yes	No	Multiple	Zero	No
PB-CED	Absent	Absent	Unbounded	No	No	Multiple	Zero	No
Amit	Present	Absent	Unbounded	Yes	Yes	Program.	Program.	Yes
Etalis	Present	Present	Unbounded	Yes	No	Program.	Program.	No
Sase	Absent	Absent	Bounded	No	No	Multiple	Zero	No
Sase+	Absent	Absent	Unbounded	No	No	Progtam.	Zero	No
Peex	Absent	Absent	Unbounded	Yes	No	Multiple	Zero	No
Aleri SP	Present	Present	Unbounded	No	No	Multiple	Zero	?
Coral8 CEP	Present	Present	Unbounded	No	No	Program.	Program.	Yes
StreamBase	Present	Present	Unbounded	No	No	Multiple	Zero	?
Oracle CEP	Absent	Present	Unbounded	No	No	Program.	Program.	?
Esper	Present	Present	Unbounded	No	No	Program.	Program.	No
Tibco BE	Absent	Absent	Unbounded	Yes	No	Multiple	Zero	?
IBM System S	Present	Present	Unbounded	Yes	No	Program.	Program.	Yes
Microsoft StreamInsight	Present	Present	Unbounded	No	No	Multiple	Zero	No

Table 2.1: Functional and Processing Models

	Deployment Model	Interaction Model		
Name	**Deployment Type**	**Observation**	**Notification**	**Forwarding**
HiPac	Centralized	Push	Push	n.a.
Ode	Centralized	Push	Push	n.a.
Samos	Centralized	Push	Push	n.a.
Snoop	Centralized	Push	Push	n.a.
TelegraphCQ	Clustered	Push	Push	Push
NiagaraCQ	Centralized	Push/Pull	Push/Pull	n.a.
OpenCQ	Centralized	Push/Pull	Push/Pull	n.a.
Tribeca	Centralized	Push	Push	n.a.
CQL / Stream	Centralized	Push	Push	n.a.
Aurora / Borealis	Clustered	Push	Push	Push
Gigascope	Centralized	Push	Push	n.a.
Stream Mill	Centralized	Push	Push	n.a.
Traditional Pub-Sub	System dependent	Push	Push	System dependent
Rapide	Centralized	Push	Push	n.a.
GEM	Networked	Push	Push	Push
Padres	Networked	Push	Push	Push
DistCED	Networked	Push	Push	Push
CEDR	Centralized	Push	Push	n.a.
Cayuga	Centralized	Push	Push	n.a.
NextCEP	Clustered	Push	Push	Push
PB-CED	Centralized	Push/Pull	Push	n.a.
Amit	Clustered	Push	Push	Push
Etalis	Centralized	Push	Push	n.a.
Sase	Centralized	Push	Push	n.a.
Sase+	Centralized	Push	Push	n.a.
Peex	Centralized	Push	Push	n.a.
Aleri SP	Clustered	Push	Push/Pull	Push
Coral8 CEP	Clustered	Push	Push/Pull	Push
StreamBase	Clustered	Push	Push/Pull	Push
Oracle CEP	Clustered	Push	Push/Pull	Push
Esper	Clustered	Push	Push/Pull	Push
Tibco BE	Networked	Push	Push	Push
IBM System S	Clustered	Push	Push	Push
Microsoft StreamInsight	Networked	Push/Pull	Push/Pull	Push

Table 2.2: Deployment and Interaction Models

	Data Model				Time Model	Rule Model	
Name	**Nature of Items**	**Format**	**Support for Uncert.**	**Nature of Flows**	**Time**	**Rule Type**	**Probab. Rules**
HiPac	Events	Database and External Events	No	Heterogeneous	Absolute	Detecting	No
Ode	Events	Method Invocations	No	Heterogeneous	Absolute	Detecting	No
Samos	Events	Method Invocation and External Events	No	Heterogeneous	Absolute	Detecting	No
Snoop	Events	Database and External Events	No	Heterogeneous	Absolute	Detecting	No
TelegraphCQ	Data	Tuples	No	Homogeneous	Absolute	Transforming	No
NiagaraCQ	Data	XML	No	Homogeneous	Stream-only	Transforming	No
OpenCQ	Data	Tuples	No	Heterogeneous	Stream-only	Transforming	No
Tribeca	Data	Tuples	No	Homogeneous	Stream-only	Transforming	No
CQL / Stream	Data	Tuples	No	Homogeneous	Stream-only	Transforming	No
Aurora / Borealis	Data	Tuples	No	Homogeneous	Stream-only	Transforming	No
Gigascope	Data	Tuples	No	Homogeneous	Causal	Transforming	No
Stream Mill	Data	Tuples	No	Homogeneous	Stream-only	Transforming	No
Traditional Pub-Sub	Events	System dependent	System dependent	Heterogeneous	System dependent	Detecting	System dependent
Rapide	Events	Records	No	Heterogeneous	Causal	Detecting	No
GEM	Events	Records	No	Heterogeneous	Interval	Detecting	No
Padres	Events	Records	No	Heterogeneous	Absolute	Detecting	No
DistCED	Events	Records	No	Heterogeneous	Interval	Detecting	No
CEDR	Events	Tuples	No	Homogeneous	Interval	Detecting	No
Cayuga	Events	Tuples	No	Homogeneous	Interval	Detecting	No
NextCEP	Events	Tuples	No	Homogeneous	Interval	Detecting	No
PB-CED	Events	Tuples	No	Heterogeneous	Interval	Detecting	No
Amit	Events	Records	Yes	Heterogeneous	Absolute	Detecting	Yes
Etalis	Events	Records	No	Heterogeneous	Interval	Detecting	No
Sase	Events	Records	No	Heterogeneous	Absolute	Detecting	No
Sase+	Events	Records	No	Heterogeneous	Absolute	Detecting	No
Peex	Events	Records	Yes	Heterogeneous	Absolute	Detecting	Yes
Aleri SP	Data	Tuples	No	Homogeneous	Stream-only	Transforming	No
Coral8 CEP	Data	Tuples	No	Homogeneous	Stream-only	Transforming	No
StreamBase	Data	Tuples	No	Homogeneous	Stream-only	Transforming	No
Oracle CEP	Data	Tuples	No	Homogeneous	Stream-only	Transforming	No
Esper	Data	Tuples	No	Homogeneous	Stream-only	Transforming	No
Tibco BE	Events	Records	No	Heterogeneous	Interval	Detecting	No
IBM System S	Data	Various	No	Homogeneous	Stream-only	Transforming	No
Microsoft StreamInsight	Data	Tuples	No	Homogeneous	Interval	Transforming	No

Table 2.3: Data, Time, and Rule Models

		Single-item			Logic					Windows					Flow Management												
Name	Type	Selection	Projection	Renaming	Conjunction	Disjunction	Repetition	Negation	Sequence	Iteration	Fixed	Landmark	Sliding	Pane	Tumble	User Defined	Join	Union	Except	Intersect	Remove Dup	Duplicate	Group By	Order By	Param.	Flow Creation	Detect. Aggr	Product. Aggr
HiPac	Det	X				X			X																X			X
Ode	Det	X			X	X	X	X	X	X															X		X	X
Samos	Det	X			X	X	X	X	X	X	X														X			X
Snoop	Det	X			X	X	X		X	X															X			X
TelegrahCQ	Decl	X	X	X							X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
NiagaraCQ	Decl	X	X	X													X						X	X	X		X	X
OpenCQ	Decl	X	X	X	X	X		X									X	X	X	X	X	X	X	X	X		X	X
Tribeca	Imp	X	X										X		X			X							X		X	X
CQL/Stream	Decl	X	X	X							X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Aurora/Borealis	Imp	X	X									X	X		X		X	X	X		X	X	X		X		X	X
Gigascope	Decl	X									X	X	X				X	X	X			X	X		X	X	X	X
Stream Mill	Decl	X	X	X							X	X	X				X	X	X			X	X		X	X	X	X
Traditional Pub-Sub	Det	X																					X	X	X	X	X	X
Rapide	Det	X			X	X	X	X	X	X	X	X	X				X	X							X			X
GEM	Det	X			X	X		X	X	X	X	X	X				X	X							X			X
Padres	Det	X			X	X	X		X	X															X			X
DistCED	Det	X			X	X	X	X	X	X															X			X
CEDR	Det	X	X	X	X	X		X	X	X							X	X	X	X	X	X	X	X	X			X
Cayuga	Det	X	X	X	X	X			X	X							X								X		X	X
NextCEP	Det	X	X	X	X	X			X	X			X												X		X	X
PB-CED	Det	X			X	X			X	X															X			X
Amit	Det	X			X	X	X	X	X	X			X		X	X	X	X	X	X	X	X	X	X	X		X	X
Etalis	Det	X			X	X		X	X	X			X		X	X	X	X	X	X	X	X	X	X	X		X	X
Sase	Det	X			X	X			X	X			X		X		X	X	X	X	X	X	X	X	X		X	X
Sase+	Det	X			X	X			X	X			X		X		X	X	X	X	X	X	X	X	X		X	X
Peex	Det	X			X	X			X	X			X		X		X	X	X	X	X	X	X	X	X		X	X
Aleri SP	Imp+Det	X	X	X	X	X	X	X	X	X							X	X	X	X	X	X	X	X	X	X	X	X
Coral8 CEP	Decl+Det+Imp	X	X	X	X	X	X	X	X	X							X	X	X	X	X	X	X	X	X	X	X	X
StreamBase	Decl+Det+Imp	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Oracle CEP	Decl+Det+Imp	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Esper	Decl+Det	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Tibco BE	Det	X			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
IBM System S	Decl+Imp	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Microsoft StreamInsight	Decl+Imp	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Table 2.4: Language Model

component) is bounded.

The condition part of each rule is seen as a collection of queries that may refer to database state, as well as to data embedded in the considered event. We represent the usage of the database state during the condition evaluation through the presence of the *knowledge base* component. Actions may perform operations on the database as well as execute additional commands: it is also possible to specify activation or deactivation of rules within the action part of a rule, thus enabling dynamic modification of the rule base. Since the event composition language of HiPac includes a sequence operator, the system can reason about time: in particular it uses an absolute time, with a total order between events.

A key contribution of the HiPac project is the definition of *coupling modes*. The coupling mode between an event and a condition specifies when the condition is evaluated w.r.t. the transaction in which the triggering event is signalled. The coupling mode between a condition and an action specifies when the action is executed w.r.t. the transaction in which the condition is evaluated. HiPac introduces three types of coupling modes: *immediate*, *deferred* (i.e., at the end of the transaction) or *separate* (i.e., in a different transaction). Since HiPac captures all the set of events that match the pattern in the event part of a rule, we can say that it uses a fixed multiple selection policy, associated with a zero consumption one.

Ode

Ode [56, 57] is a general-purpose object-oriented active database system. The database is defined, queried, and manipulated using the O++ language, which is an extension of the C++ programming language adding support for persistent objects.

Reactive rules are used to implement *constraints* and *triggers*. Constraints provide a flexible mechanism to adapt the database system to different, application-specific, consistency models: they limit access to data by blocking forbidden operations. Conversely, triggers are associated to allowed operations and define a set of actions to be automatically executed when specific methods are called on database objects.

Constraints are associated with a class definition and consist of a condition and an optional handler: all the objects of a class must satisfy its constraints. In case of violation the handler is called: it is composed by a set of operations to bring the database back to a consistent state. In the case it is not possible to repair the violation of a constraint (undefined handler) Ode aborts the transaction; according to the type of constraint (hard or soft) a constraint violation may be repaired immediately af-

ter the violation is reported (*immediate* semantics) or just before the commit of the transaction (*deferred* semantics). Consistently with the object oriented nature of the system, constraints conditions are checked at the boundaries of public methods calls, which represent the only way to interact with the database.

Triggers, like integrity constraints, are used to detect user-defined conditions on the database objects. Such conditions, however, do not represent consistency violations but capture generic situations of interest. Trigger, like a constraint, are specified in the class definition and consist of two parts: a condition and an action. The condition is evaluated on each object of the class defining it. When it becomes true the corresponding action is executed. Unlike a constraint handler, which is executed as part of the transaction violating the constraint, a trigger action is executed as a separate transaction (*separate* semantics). Actions may include dynamic activation and deactivation of rules.

Both constraints and triggers can be classified as detecting rules expressed using a pattern-based language, where the condition part of the rule includes constraints on inner database values, while the action part is written in C++.

Ode does not offer a complete support for periodic evaluation of rules; however, users can force actions to be executed when an active trigger has not fired within a given amount of time.

We can map the behavior of Ode to our functional and data model by considering public method calls to database objects as the information items that enter the system. Evaluation is performed by taking into account the database state (i.e., the *Knowledge Base* in our model).

So far we only mentioned processing involving single information items: support for multiple items processing has been introduced in Ode [58] allowing users to capture complex interactions with the database, involving multiple methods call. In particular, Ode offers logic operators and sequences to combine single calls: the centralized nature of the system makes it possible to define a complete ordering relation between calls (absolute time). The language of Ode derives from regular expressions, with the addition of parameterization. As a consequence negation, although present, can only refer to immediately subsequent events. This way all supported operators become non-blocking. The presence of an iteration operator makes it possible to detect sequences of events whose length is unknown a priori (i.e. *Seq* is unbounded). Thanks to their similarities with regular expressions, rules are translated into finite state automata to be efficiently triggered. In [58], the authors also address the issue of defining precise processing policies; in particular, they propose a detection algorithm in which all valid set of events are captured

(multiple selection policy), while the possibility to consume events is not mentioned.

Samos

Samos [59, 60] is another example of a general-purpose object-oriented active database system. Like Ode, it uses a detecting language derived from regular expressions (including iterations); rules can be dynamically activated, deactivated, created and called during processing.

The most significant difference between Samos and Ode is the possibility offered by Samos to consider *external events* as part of the information flows managed by rules. External events include all those events not directly involving the database structure or content. Among external events, the most important are temporal events, which allow Samos programmers to use time inside rules. As a consequence, time assumes a central role in rule definition: first, it can be used to define periodic evaluation of rules; second, each rule comes with a validity interval, which can be either explicitly defined using the `IN` operator, or implicitly imposed by the system. The `IN` operator defines time-based windows whose bounds can be either fixed time points or relative ones, depending on previous detections. To do so, it relies on an absolute time. The `IN` operator is also useful to deal with blocking operators (like for example the negation); by providing an explicit window construct through the `IN` operator, Samos offers more flexibility than Ode does.

Great attention has been paid to the specification of precise semantics for rule processing; in particular it is possible to specify when a condition has to be evaluated: immediately, as an event occurs, at the end of the active transaction, or in a separate transaction. Like in HiPac, similar choices are also associated to the execution of actions. Evaluation of rules is performed using Petri Nets.

Snoop

Snoop [61] is an event specification language for active databases developed as a part of the Sentinel project [62]. Unlike Ode and Samos, it is designed to be independent from the database model, so it can be used, for example, on relational databases as well as on object-oriented ones.

The set of operators offered by Snoop is very similar to the ones we described for Ode and Samos, the only remarkable difference being the absence of a negation operator. Like Samos, Snoop considers both internal and external events as input information items. Unlike Samos it does not offer explicit windows constructs: instead, it embeds windows

features directly inside the operators that require them. For example iterations can only be defined within specific bounds, represented as points in time or information items.

Snoop enables parameter definition as well as dynamic rule activation. The detection algorithm is based on the construction and visiting of a tree.

The most interesting feature of Snoop is the possibility to specify a *context* for each rule. Contexts are used to control the semantics of processing. More specifically, four types of contexts are allowed: recent, chronicle, continuous, and cumulative. In the presence of multiple sets of information items firing the same rule, contexts specify exactly which sets will be considered and consumed by the system (i.e., they specify the selection and consumption policies). For example, the "recent" context only selects the most recent information items, consuming all of them, while the cumulative context selects all the relevant information items, resulting in a multiple policy.

Snoop, like HiPac, Ode, Samos, and like many other proposed languages for active DBMSs (e.g., [63, 64, 65] considers events as points in an time. Recently the authors of Snoop have investigated the possibility to extend the semantics of the language to capture events that have a duration. This led to the definition of a new language, called SnoopIB [38], which offers all the operators and language construct of Snoop, including contexts, but uses interval based timestamps for events.

2.4.2 Data Stream Management Systems

To better organize the presentation, we decided to start our description with all those systems, namely TelegraphCQ, NiagaraCQ, and OpenCQ, that belong to the class of DSMSs, but present a strong emphasis on periodic, as opposed to purely reactive, data gathering and execution; usually, these systems are designed for application domains in which the timeliness requirement is less critical, like for example Internet updates monitoring. On the contrary, all remaining systems are heavily tailored for the processing of high rate streams.

TelegraphCQ

TelegraphCQ [66] is a general-purpose continuous queries system based on a declarative, SQL-based language called StreaQuel.

StreaQuel derives all relational operators from SQL, including aggregates. To deal with unbounded flows, StreaQuel introduces the `WindowIs` operator, which enables the definition of various types of windows. Mul-

tiple `WindowIs`, one for each input flow, can be expressed inside a `for` loop, which is added at the end of each rule and defines a time variable indicating when the rule has to be processed. This mechanism is based on the assumption of an absolute time model. By adopting an explicit time variable, TelegraphCQ enables users to define their own policy for moving windows. As a consequence, the number of items selected at each processing cycle cannot be bounded a priori, since it is not possible to determine how many elements the time window contains.

Since the *WindowIs* operator can be used to capture arbitrary portions of time, TelegraphCQ naturally supports the analysis of historical data. In this case, disk becomes the bottleneck resource. To keep processing of disk data up with processing of live data, a shedding technique called OSCAR (Overload-sensitive Stream Capture and Archive Reduction) is proposed. OSCAR organizes data on disk into multiple resolutions of reduced summaries (such as samples of different sizes). Depending on how fast the live data is arriving, the system picks the right resolution level to use in query processing [67].

TelegraphCQ has been implemented in C++ as an extension of PostgreSQL. Rules are compiled into a query plan that is extended using adaptive modules [68, 69, 70], which dynamically decide how to route data to operators and how to order commutative operators. Such modules are also designed to distribute the TelegraphCQ engine over multiple machines, as explained in [71]. The approach is that of clustered distribution, as described in Section 2.3: operators are placed and, if needed, replicated in order to increase performance, by only taking into account processing constraints and not transmission costs, which are assumed negligible.

NiagaraCQ

NiagaraCQ [72] is an IFP system for Internet databases. The goal of the system is to provide a high-level abstraction to retrieve information, in the form of XML data sets, from a frequently changing environment like Internet sites. To do so, NiagaraCQ specifies transforming rules using a declarative, SQL-like language called XML-QL [73]. Each rule has an associated time interval that defines its period of validity. Rules can be either *timer-based* or *change-based*; the former are evaluated periodically during their validity interval, while processing of the latter is driven by notifications of changes received from information sources. In both cases, the evaluation of a rule does not directly produce an output stream, but updates a table with the new results; if a rule is evaluated twice on the same data, it does not produce further updates. We capture this behavior

in our processing model, by saying that NiagaraCQ presents a selected consumption policy (information items are used only once). Users can either ask for results on-demand, or can be actively notified (e.g., with an e-mail) when new information is available (i.e., the notification model allows both a push and a pull interaction). For each rule, it is possible to specify a set of actions to be performed just after the evaluation. Since information items processed by NiagaraCQ comes directly from Internet databases, they do not have an explicit timestamp associated.

NiagaraCQ can be seen as a sort of active database system where timer-based rules behave like traditional queries, the only exception being periodic evaluation, while change-based rules implement the reactive behavior. NiagaraCQ offers a unified format for both types of rules.

The main difference between NiagaraCQ and a traditional active database is the distribution of information sources over a possibly wide geographical area. It is worth noting that only information sources are actually distributed, while NiagaraCQ engine is designed to run in a totally centralized way. To increase scalability NiagaraCQ uses an efficient caching algorithm, which reduces access time to distributed resources and an *incremental group optimization*, which splits operators into groups; members of the same group share the same query plan, thus increasing performance.

OpenCQ

Like NiagaraCQ, OpenCQ [74] is an IFP system developed to deal with Internet-scale update monitoring. OpenCQ rules are divided into three parts: a SQL *query* that defines operations on data, a *trigger* that specifies when the rule has to be evaluated, and a *stop* condition that defines the period of validity for the rule. OpenCQ presents the same processing and interaction models as NiagaraCQ.

OpenCQ has been implemented on top of the DIOM framework [75], which uses a client-server communication paradigm to collect rules and information and to distribute results; processing of rules is therefore completely centralized. Wrappers are used to perform input transformations, thus allowing heterogeneous sources to be taken into account for information retrieval. Depending on the type of the source, information may be pushed into the system, or it may require the wrapper to periodically ask sources for updates.

Tribeca

Tribeca [76] has been developed with a very specific domain in mind: that of network traffic monitoring. It defines transforming rules taking a single information flow as input and producing one or more output flows. Rules are expressed using an imperative language that defines the sequence of operators an information flow has to pass through.

Tribeca rules are expressed using three operators: selection (called qualification), projection, and aggregate. Multiplexing and demultiplexing operators are also available, which allow programmers to split and merge flows. Tribeca supports both count based and time based windows, which can be sliding or tumble. They are used to specify the portions of input flows to take into account when performing aggregates. Since items do not have an explicit timestamp, the processing is performed in arrival order. The presence of a timing window makes it impossible to know the number of information items captured at each detection-production cycle (i.e., the *Seq* is unbounded). Moreover, a rule may be satisfied by more than one set of elements (multiple selection policy), while an element may participate in more than one processing cycle, as it is never explicitly consumed (zero consumption policy).

Each rule is translated into a direct acyclic graph to be executed; during translation the plan can be optimized in order to improve performance.

CQL/Stream

CQL [42] is an expressive SQL based declarative language that creates transforming rules with a unified syntax for processing both information flows and stored relations.

The goals of CQL are those of providing a clear semantics for rules while defining a simple language to express them. To do so, CQL specifies three kinds of operators: relation-to-relation, stream-to-relation, and relation-to-stream. Relation-to-relation operators directly derive from SQL: they are the core of the CQL language, which actually defines processing and transformation. The main advantage of this approach is that a large part of the rule definition is realized through the standard notation of a widely used language. In order to add support for flow processing, CQL introduces the notion of windows, and in particular sliding, pane, and tumble windows, which are intended as a way to store a portion of each input flow inside the relational tables where processing takes place; for this reason CQL denotes windows as stream-to-relation operators. Windows can be based both on time and on the number of

contained elements. The last kind of operators that CQL provides is that of relation-to-stream operators, which define how processed tuples can become part of a new information flow. As already mentioned in Section 2.3.8 three relation-to-stream operators exist: `IStream`, `DStream`, and `RStream`. Notice that CQL is based on a time model that explicitly associates a timestamp to input items, but timestamps cannot be addressed from the language. As a consequence, since output is based on the sequence of updates performed on a relational table, its order is separate from input order.

Items selection is performed using traditional SQL operators and never consumed (until they remain in the evaluation window), so we can say that CQL uses a multiple selection policy, associated with a zero consumption.

CQL has been implemented as part of the Stream project [43]. Stream computes a *query plan* starting from CQL rules; then, it defines a schedule for operators, by taking into account predefined performance policies.

The Stream project has proposed two major shedding techniques to deal with resource overload problem on data streams: the first addresses the problem of limited computational resources by applying load shedding on a collection of sliding window aggregation queries [20]; the second addresses the problem of limited memory, by discarding operator state for a collection of windowed joins [77].

Aurora/Borealis

Aurora [45] is a general-purpose DSMS; it defines transforming rules created with an imperative language called SQuAl. SQuAl defines rules in a graphical way, by adopting the *boxes and arrows* paradigm, which makes connections between different operators explicit.

SQuAl defines two types of operators: *windowed* operators apply a single input (user-defined) function to a window and then advance the window to include new elements before repeating the processing cycle; *single-tuple* operators, instead, operate on a single information item at a time; they include single-item operators like selection and flow operators like join, union, and group by. This allows information items to be used more than once (multiple selection policy), while they are never consumed.

SQuAl allows users to define plans having multiple input and multiple output flows and, interestingly, allows users to associate a QoS specification to each output. As output flows may be directly connected to higher-level applications, this makes it possible to customize system behavior according to application requirements. QoS constraints are used

by Aurora to automatically define shedding policies: for example some application domain may need to reduce answer precision in order to obtain faster response times. Input and output flows do not have an associated timestamp, but are processed in their arrival order.

An interesting feature of Aurora is the possibility to include intermediate storage points inside the rule plan: such points can be used to keep historical information and to recover after operators failure.

Processing is performed by a scheduler, which takes an optimized version of the user defined plan and chooses how to allocate computational resources to different operators according to their load and to the specified QoS constraints.

The project has been extended to investigate distributed processing both inside a single administrative domain and over multiple domains [78]. In both cases the goal is that of efficiently distributing load between available resources; in these implementations, called Aurora* and Medusa, communication between processors takes place using an overlay network, with dynamic bindings between operators and flows. The two projects were recently merged and all their feature have been included into the Borealis stream processor [79].

Gigascope

Gigascope [80, 81] is a DSMS specifically designed for network applications, including traffic analysis, intrusion detection, performance monitoring, etc. The main concern of Gigascope is to provide high performance for the specific application field it has been designed for.

Gigascope defines a declarative, SQL-like language, called GSQL, which includes only filters, joins, group by, and aggregates. Interestingly, it uses processing techniques that are very different from those of other data stream systems. In fact, to deal with the blocking nature of some of its operators, it does not introduce the concept of windows. Instead, it assumes that each information item (tuple) of a flow contains at least an *ordered attribute* i.e. an attribute that monotonically increases or decreases as items are produced by the source of the flow, for example a timestamp defined w.r.t. an absolute time but also a sequence number assigned at source. Users can specify which attributes are ordered, as part of the data definition, and this information is used during processing. For example, the join operator, by definition, must have a constraint on an ordered attribute for each involved stream.

These mechanisms make the semantics of processing easier to understand, and more similar to that of traditional SQL queries. However, they can be applied only on a limited set of application domains, in

which strong assumptions on the nature of data and on arrival order can be done.

Gigascope translates GSQL rules into basic operators, and composes them into a plan for processing. It also uses optimization techniques to re-arrange the plan according to the nature and the cost of each operator.

Stream Mill

Stream Mill [52] is a general-purpose DSMS based on ESL, a SQL-like language with ad-hoc constructs designed to easily support flows of information. ESL allows its users to define highly expressive transforming rules by mixing the declarative syntax of SQL with the possibility to create custom aggregates in an imperative way directly within the language. To do so, the language offers the possibility to create and manage custom tables, which can be used as variables of a programming language during information flow processing. Processing itself is modeled as a loop on information items, which users control by associating behaviors to the beginning of processing, to internal iterations, and to the end of processing. Behaviors are expressed using the SQL language.

This approach is somehow similar to the one we described for CQL: flows are temporarily seen as relational tables and queried with traditional SQL rules. ESL extends this idea by allowing users to express exactly how tables have to be managed during processing. Accordingly, the selection and consumption policies can be programmed rule by rule. Since processing is performed on relational tables, the ordering of produced items may be separate from the input order.

Besides this, Stream Mill keeps the traditional architecture of a database system; it compiles rules into a query plan whose operators are then dynamically scheduled. The implementation of Stream Mill as described in [52] shows a centralized processing, where the engine exchanges data with applications using a client-server communication paradigm.

2.4.3 Complex Event Processing Systems

Traditional content-based publish-subscribe systems

As stated in Section 2.2 traditional content-based publish-subscribe middleware [10, 82] represent one of the historical basis for complex event processing systems. In the publish-subscribe model, information items flow into the system as messages coming from multiple *publishers* while simple detecting rules define interests of *subscribers*. The detection part of each rule can only take into account single information items and select them using constraints on their content; the action part simply perform

information delivery. These kinds of rules are usually called *subscriptions*. The exact syntax used to represent messages and subscriptions varies from system to system, but the expressive power of the system remains similar, hence we group and describe them together.

Many publish-subscribe systems [83, 84, 85, 86, 87, 88, 33, 89] have been designed to work in large scale scenarios. To maximize message throughput and to reduce as much as possible the cost of transmission, such systems adopt a distributed core, in which a set of brokers, connected in a dispatching network, cooperate to realize efficient routing of messages from publishers to subscribers.

Rapide

Rapide [90, 91] is considered one of the first steps toward the definition of a complex event processing system. It consists of a set of languages and a simulator that allows users to define and execute models of system architectures. Rapide is the first system that enables users to capture the timing and causal relationships between events: in fact, the execution of a simulation produces a *causal history*, where relationships between events are made explicit.

Rapide models an architecture using a set of components, and the communication between components using events. It embeds a complex event detection system, which is used both to describe how the detection of a certain pattern of events by a component brings to the generation of other events, and to specify properties of interest for the overall architecture. The pattern language of Rapide includes most of the operators presented in Section 2.3: in particular it defines conjunctions, disjunctions, negations, sequences, and iterations, with timing constraints. Notice that Rapide does not assume the existence of an absolute time (however, event can be timestamped with the values of one or more clocks, if available): as a consequence sequences only take into account a causal order between events. During pattern evaluation, Rapide allows rules to access the state of components; we capture this interaction with the presence of the *Knowledge Base*, in our functional model.

The processing model of Rapide captures all possible set of events matching a given pattern, which means that it applies a multiple selection and a zero consumption policies. Notice that complex events can be reused in the definition of other events.

Since Rapide also uses its pattern language to define general properties of the system, it embeds operators that are not found in other proposals, like logical implications and equivalences between events. To capture these relations, Rapide explicitly stores the history of all events that led

to its occurrence. This is made easy by the fact that simulations are executed in a centralized environment.

GEM

GEM [92, 93] is a generalized monitoring language for distributed systems. It has been designed for distributed deployment on monitors running side-by-side to event generators. Each monitor is configured with a script containing detecting rules that detect patterns of events occurring in the monitored environment and perform actions when a detection occurs. Among the possible actions that a monitor can execute, the most significant is notification delivery: using notifications each monitor becomes itself the generator of an information flow, thus enabling detection of events that involve multiple monitors in a distributed fashion. It is worth noting, however, that distribution is not completely automatic, but needs the static configuration of each involved monitor. To make distributed detection possible, the language assumes the existence of a synchronized global clock. An *event-specific delaying technique* is used to deal with communication delays; this technique allows detection to deal with out-of-order arrival of information items.

GEM supports filtering of single information items, parameterization, and composition using conjunction, disjunction, negation (between two events), and sequence. Iterations are not supported. The time model adopted by GEM considers events as having a duration, and allows users to explicitly refer to the starting and ending time attributes of each event inside the rules. This allows users to define fixed and landmark windows for pattern evaluation. Moreover, an implicit sliding window is defined for each rule having blocking operators; the size of the window is not programmable by the users but depends from system constraints and load (i.e., main memory and input rates of events). Since the definition of a composite event has to explicitly mention all the components that need to be selected, the maximum number of events captured at each processing cycle is determined by deployed rules (i.e., *Seq* is bounded).

Detection is performed using a tree based structure; the action part of a rule includes, beside notification delivery, also the possibility to execute external routines and to dynamically activate or deactivate rules. The GEM monitor has been implemented in C++ using the REGIS/DARWIN [94, 95] environment that provides its communication and configuration platform.

Padres

Padres [46] is an expressive content-based publish-subscribe middleware providing a form of complex event processing. As in traditional publish-subscribe it offers primitives to publish messages and to subscribe to specified information items, using detecting rules expressed in a pattern-based language. Unlike traditional publish-subscribe, expressed rules can involve more than one information item, allowing logic operators, sequences, and iterations. Padres uses a time model with an absolute time, which can be addressed in rules to write complex timing constrains. This way it provides fixed, landmark, and sliding windows. All possible set of events satisfying a rule are captured; so we can say that the system adopts a multiple selection and zero consumption policies. The presence of an operator for iterations together with a timing window, makes it impossible to determine the maximum number of items selected by each rule a priori (*Seq* is unbounded).

Processing of rules, including those involving multiple information items, is performed in a fully distributed way, exploiting a hierarchical overlay network. To do so, the authors propose an algorithm to decompose rules in order to find a convenient placement for each operator composing them; its goal is that of partially evaluating rules as soon as possible during propagation of information elements from sources to recipients. Assuming that applying an operator never increases the amount of information to be transmitted, this approach reduces network usage.

Information processing performed inside single nodes is realized using Rete trees [96]. After detection, information items are simply delivered to requiring destinations: only juxtaposition of information inside a single message is allowed; more complex actions, like aggregates, cannot be specified.

DistCED

DistCED [97] is another expressive content-based publish-subscribe system; it is designed as an extension of a traditional publish-subscribe middleware and consequently it can be implemented (at least in principle) on top of different existing systems.

DistCED rules are defined using a detecting language. Like in Padres it is possible to combine single information items using logic operators, sequences, and iterations. DistCED also allows users to specify the time of validity for detection: this is done using a construct that builds up a time-based sliding window. DistCED uses a time model that takes

into account the duration of events and uses two different operators to express sequences of events that are allowed to overlap (*weak sequence*), and sequences of events with no overlapping (*strong sequence*). The authors consider duration of events as a simple way to model the timing uncertainty due to transmission; however no details are provided about this.

Processing is performed in a distributed way: rules are compiled into finite state automata and deployed as detectors; each detector is considered as a mobile agent, that can also be split into its components to be better distributed. When a new rule is added, the system checks whether it can be interpreted by exploiting already defined detectors; if not, it creates new automata and pushes them inside the dispatching network. To better adapt dispatching behavior to application requirements, DistCED provides different distribution policies: for example, it can maximize automata reuse or minimize the overall network usage.

CEDR

In [53] the authors present the foundations of CEDR, defined as a general purpose event streaming system. Different contributions are introduced. First of all, the authors discuss a new temporal model for information flows, based on three different timings, which specify system time, validity time, and occurrence/modification time. This approach enables the formal specifications of various consistency models among which applications can choose. Consistency models deal with errors in flows, such as latency, or out-of-order delivery. In CEDR flows are considered as notification of state updates; accordingly the processing engine keeps an history of received information and sends a notification when the results of a rule changes, thus updating the information provided to connected clients. From this point of view, CEDR strongly resembles DSMSs for Internet update monitoring, like NiagaraCQ and OpenCQ; however, we decided to put CEDR in the class of event processing systems since it strongly emphasizes detection of event occurrences, using a powerful language based on patters, including temporal operators, like sequences.

The presence of a precise temporal model enables authors to formally specify the semantics of each operator of the language. It also greatly influences the design of the language, which presents some interesting and singular aspects: windows are not explicitly provided, while validity constraints are embedded inside most of the operators provided by the system (not only the blocking ones). In the CEDR language, instances of events play a central role: rules are not only specified in term of an event expression, which defines how individual events have to be filtered,

combined and transformed; they also embed explicit instance selection and consumption policies (through a special `cancel-when` operator), as well as instance tranformations, which enable users to specify: (i) which specific information items have to be considered during the creation of composite events, (ii) which ones have to be consumed, and (iii) how existing instances have to be transformed after an event detection.

Notice that the presence of a time window embedded into operators, together with the ability to use a multiple selection policy, makes the number of items sent by the *Decider* to the *Producer* unbounded a priori. Consider for example a simple conjunction of items ($A \wedge B$) in a time window W with a multiple selection policy: it is not possible to know a priori how many couples of items A and B will be detected in W.

Cayuga

Cayuga [98] is a general purpose event monitoring system. It is based on a language called CEL (Cayuga Event Language). The structure of the language strongly resembles that of traditional declarative languages for database: it is structured in a *SELECT* clause that filters input stream, a *FROM* clause that specifies a *streaming expression*, and a *PUBLISH* clause that produces the output. It includes typical SQL operators and constructs, like selection, projection, renaming, union, and aggregates. Despite its structure, we can classify CEL as a detection language: indeed, the streaming expression contained in the *FROM* clause enables users to specify detection patterns, including sequences (using the *NEXT* binary operator) as well as iterations (using the *FOLD* operator). Notice that CEL does not introduce any windowing operator. Complex rules can be defined by combining (nesting) simpler ones. Interestingly, all events are considered as having a duration, and much attention is paid in giving a precise semantics for operator composability, so that all defined expressions are left-associated, or can be broken up into a set of left-associated ones. To do so, the semantics of all operators is formally defined using a query algebra [4]; the authors also show how rules can be translated into non deterministic automata for event evaluation. Different instances of automata work in parallel for the same rule, detecting all possible set of events satisfying the contraints in the rule. This implicitly defines a multiple selection policy; Cayuga uses a zero consumption policy, as events can be used many times for the detection of different complex event occurrences. Since Cayuga is explicitly designed to work on large scale scenarios, the authors put much effort in defining efficient data structures: in particular, they exploit custom heap management, indexing of operator predicates and reuse of shared automata instances.

Since detecting automata of different rules are strictly connected with each other, Cayuga does not allow distributed processing.

NextCEP

NextCEP [11] is a distributed complex event processing system. Similarly to Cayuga, it uses a language that includes traditional SQL operators, like filtering and renaming, together with pattern detection operators, including sequences and iterations. Detection is performed by translating rules into non deterministic automata, that strongly resemble those defined in Cayuga in structure and semantics. In NextCEP, however, detection can be performed in a distributed way, by a set of strictly connected node (*clustered* environment). The main focus of the NextCEP project is on rule optimization: in particular, the authors provide a cost model for operators that defines the output rate of each operator according to the rate of its input data. NextCEP exploits this model for *query rewriting*, a process that changes the order in which operators are evaluated without changing the results of rules. The objective of query rewriting is that of obtaining the best possible evaluation plan, i.e. the one that minimizes the usage of CPU resources and the processing delay.

PB-CED

In [99], the authors present a system for complex event detection using data received from distributed sources. They call this approach Plan-Based Complex Event Detection (PB-CED). The emphasis of the work is on defining an efficient plan for the evaluation of rules, and on limiting as much as possible transmissions of useless data from sources.

PB-CED uses a simple detecting language, including conjunctions, disjunctions, negations, and sequences, but not iterations, or reuse of complex events in pattern. The language does not offer explicit windowing constraints, but embeds time limits inside operators, like. PB-CED offers a timing model that takes into account the duration of events.

PB-CED compiles rules into non deterministic automata for detection and combines different automata to form a complex detection plan. Although PB-CED uses a centralized detection, in which the plan is deployed on a single node, simple architectural components are deployed near sources; these components just store information received from sources (without processing them) and are directly connected with the detecting node. Components near sources may operate both in push and in pull-based mode. The authors introduce a cost model for operators and use it to dynamically generate the plan with minimum cost. Since

each step in the plan involves acquisition and processing of a subset of the events, the plan is created with the basic goal of postponing the monitoring of high frequency events to later steps in the plan. As such, processing the higher frequency events conditional upon the occurrence of lower frequency ones eliminates the need to communicate the former (requested in pull-based mode) in many cases. Interestingly, the plan optimization procedure can take into account different parameters, besides the cost function; for example it can consider local storage available at sources, or the degree of timeliness required by the application scenario.

Amit

Amit is an application development and runtime control tool intended to enable fast and reliable implementation of reactive and proactive applications [100]. Amit embeds a component, called *situation manager*, specifically designed to process notifications received from different sources in order to detect patterns of interests, called *situations*, and to forward them to demanding *subscribers*. The situation manager is based on a strongly expressive and flexible detecting language, which includes conjunctions, negations, parameters, sequences, and repetitions (in the form of *counting* operators). Timing operators are introduced as well, enabling periodic evaluation of rules.

Amit introduces the concept of *lifespan* as a valid time window for the detection of a situation. A lifespan is defined as an interval bounded by two events called *initiator* and *terminator*. Amit's language enables user-defined policies for lifespans management: these policies specify whether multiple lifespans may be open concurrently (in presence of different valid initiators) and which lifespans are closed by a terminator (e.g., all open ones, only the most recently opened, etc.).

At the same time Amit introduces programmable event selection policies by applying a quantifier to each operator. Quantifiers specify which events an operator should refer to (e.g., the first, the last, all valid ones). Similarly, Amit allows programmable consumption policies by associating a consumption condition to each operator.

It is worth mentioning that detected situations can be used as part of a rule, as event notifications. This enables the definition of *nested situations*. Recursive processing is allowed as well. Despite being focused on the detection of patterns, Amit supports both detection and production aggregates.

Amit has been implemented in Java and is being used as the core technology behind the E-business Management Service of IBM Global Services [101]. It uses a centralized detection strategy: all events are

stored at a single node, partitioned according to the lifespans they may be valid for. When a terminator is received, Amit evaluates whether all the conditions for the detection of a situation have been met and, if needed, notifies subscribers.

An interesting aspect of Amit is that it implements some of the capabilities described in [35], thus supporting uncertain input and enabling uncertain rules.

Etalis

Etalis [102, 103] is a CEP system explicitly designed to simplify the definition of composite events using temporal patterns of primitive events. Its language is based on an interval time model, and offers a large number of operators to define different kinds of sequences (with partial overlapping of events, without overlapping, etc.).

The language of Etalis includes explicit time-based windows, but also embeds them inside the negation operator to make it unblocking. Interestingly, it does not include an explicit operator for iterations; on the contrary, it makes wide use of recursive processing, defining composite events starting from other composite events, thus allowing the detinition of possibly unbounded sequences of events.

To detect composite events, Etalis recursively partitions rules until they become binary, i.e., involving two events only. Binary rules are then translated into a set of Prolog rules.

Another interesting aspect in Etalis is its capability to integrate information derived from events with static information stored in some form of data base or knowledge base.

Sase

Sase [104] is a monitoring system designed to perform complex queries over real-time flows of RFID readings.

Sase defines detecting rules language based on patterns; each rule is composed of three parts: *event*, *where* and *within*. The event clause specifies which information items have to be detected and which are the relations between them; relations are expressed using logic operators and sequences. The where clause defines constraints on the inner structure of information items included into the event clause: referring to the list of operators of Section 2.3, the where clause is used to define selections for single information items. Finally, the within clause expresses the time of validity for the rule; this way it is possible to define time-based, sliding windows. The language adopted by Sase allows only detection

of given patterns of information items; it does not include any notion of aggregation.

Sase compiles rules into a query plan having a fixed structure: it is composed of six blocks, which sequentially process incoming information elements realizing a sort of pipeline: the first two blocks detect information matching the logic pattern of the event clause by using finite state automata. Successive blocks check selections constraints, windows, negations, and build the desired output. Since all these operators explicitly specify the set of events to be selected, it is not possible to capture unbounded sequences of information items (*Seq* is bounded).

Sase+

Sase+ [47, 48] is an expressive event processing language from the authors of Sase. Sase+ extends the expressiveness of Sase, by including iterations and aggregates as possible parts of detecting patterns.

Non deterministic automata are used for pattern detection, as well as for providing a precise semantics of the language. An interesting aspect of Sase+ is the possibility for users to customize selection policies using *strategies*. Selection strategies define which events are valid for an automaton transition: only the next one (if satisfying the rule's constraints), or the next satisfying one, or all satisfying ones that satisfy. Consumption of events, instead, is not taken into account.

An important contribution of [48] is the formal analysis of the expressive power of the language, and of the complexity of its detecting algorithm. On one side this analysis enables a direct comparison with other languages (e.g., traditional regular expressions, Cayuga language). On the other side, the analysis applies only to a limited set of pattern-based language and cannot yet capture the multitude of languages defined in the IFP domain.

Peex

Peex (Probabilistic Event Extractor) [105] is a system designed for extracting complex events from RFID data. Peex presents a pattern-based rule language with four clauses: *FORALL*, which defines the set of readings addressed in the rule, *WHERE*, which specifies pattern constraints, *CREATE EVENT* and *SET*, which define the type of the event to be generated and set the content of its attributes. Patterns may include conjunctions, negations, and sequences, but not iterations.

The main contribution of Peex is its support for data uncertainty: in particular, it addresses data errors and ambiguity, which can be frequent

in the specific application domain of RFID data. To do so, it changes the data model, by assigning a probability to all information items. More in details, when defining rules, system administrators can associate a confidence to the occurrence and attribute values of the defined composite events, as functions of composing ones. For example, they can state that, if three sequential readings are detected in a given time window, then an event "John enters room 50" occurs with a probability of 70%, while an event "John enters room 51" occurs with probability of 20% (probabilities need not sum to 100%). Peex enable users to re-use event definitions in the specification of others: to compute the probability of composite events, it fully takes into account the possible correlations between components. Another interesting aspect of Peex is the possibility to produce *partial events*, i.e. to produce an event even if some of its composing parts are missing. Obviously, the confidence on the occurrence of a partial event is lower than if all composing events were detected. Partial events are significant in the domain of RFID readings since sometimes source may fail in detecting or transmitting an event.

From an implementation point of view, Peex uses a relation DBMSs, where it stores all information received from sources and information about confidence. Rules are then translated into SQL queries and run periodically. For this reason, the detection time of events may be different from real occurrence time.

2.4.4 Commercial Systems

Aleri Streaming Platform

The Aleri Streaming Platform [106] is an IFP system that offers a simple imperative, graphical language to define processing rules, by combining a set of predefined operators. To increase system's expressiveness custom operators can be defined using a scripting language called Splash, which includes the capability of defining variables to store past information items, so that they can be referenced for further processing. Pattern detection operators are provided as well, including sequences. Notice, however, that pattern matching can appear in the middle of a complex computation, and that sequences may use different attributes for ordering, not only timestamps. As a consequence, the semantics of output ordering does not necessarily reflect timing relationships between input items.

The platform is designed to scale by exploiting multiple cores on a single machine or multiple machines in a clustered environment. However, no information is provided on the protocols used to distribute operators.

Interestingly, the Aleri Streaming Platform is designed to easily work together with other business instruments: probably the most significant example is the Aleri Live OLAP system, which extends traditional OLAP solutions [107] to provide near real time updates of information. Aleri also offers a development environment that simplifies the definition of rules and their debugging. Additionally, Aleri provides adapters to enable different data formats to be translated into flows of items compatible with the Aleri Streaming Platform, together with an API to write custom programs that may interact with the platform using either standing or one-time queries.

Coral8 CEP Engine

The Coral8 CEP Engine [108, 109], despite its name, can be classified as a data stream system. Indeed, it uses a processing paradigm in which flows of information are transformed through one or more processing steps, using a declarative, SQL-like language, called CCL (Continuous Computation Language). CCL includes all SQL statements; in addition it offers clauses for creating time-based or count-based windows, for reading and writing data in a defined window, and for delivering items as part of the output stream. CCL also provides simple constructs for pattern matching, including conjunctions, disjunctions, negations, and sequences; instead, it does not offer support for repetitions. Like in Aleri, the Coral8 engine does not rely upon the existence of an absolute time model: users may specify, stream by stream, the processing order (e.g., increasing timestamp value, if a timestamp is provided, or arrival order). The results of processing can be obtained in two ways: by subscribing to an output flow (push), or by reading the content of a *public* window (pull).

Together with its CEP Engine, Coral8 also offers a graphical environment for developing and deploying, called Coral8 Studio. This tool can be used to specify data sources and to graphically combine different processing rules, by explicitly drawing a plan in which the output of a component becomes the input for others. Using this tool, all CCL rules become the primitive building blocks for the definition of more complex rules, specified using a graphical, plan-based, language.

Like Aleri, the Coral8 CEP engine may execute in a centralized or clustered environment. The support for clustered deployment is used to increase the availability of the system, even in presence of failures. It is not clear, however, which policies are used to distribute processing on different machines. Load shedding is implemented by allowing administrator to specify a maximum allowed rate for each input stream.

During early 2009, Coral8 merged with Aleri. The company now plans a combined platform and tool set under the name of Aleri CEP.

StreamBase

StreamBase [110] is a software platform that includes a data stream processing system, a set of adapters to gather information from heterogeneous sources, and a developer tool based on Eclipse. It shares many similarities with the Coral8 CEP Engine: in particular, it uses a declarative, SQL-like language for rule specification, called StreamSQL [111]. Beside traditional SQL operators, StreamSQL offers customizable time-based and count-based windows. Plus, it includes a simple pattern based language that captures conjunctions, disjunctions, negations, and sequences of items.

Operators defined in StreamSQL can be combined using a graphical plan-based rule specification language, called EventFlow. User-defined functions, written in Java or C++, can be easily added as custom aggregates. Another interesting feature is the possibility to explicitly instruct the system to permanently store a portion of processed data for historical analysis.

StreamBase supports both centralized and clustered deployments; networked deployments can be used as well, but also to provide high availability in case of failures. Users can specify the maximum load for each used server, but the documentation does not specify how the load is actually distributed to meet these constraints.

Oracle CEP

Oracle [44] launched its event-driven architecture suite in 2006 and added BEA's WebLogic Event Server to it in 2008, building what is now called "Oracle CEP", a system that provides real time information flow processing. Oracle CEP uses CQL as its rule definition language, but, similarly to Coral8 and StreamBase, it adds a set of relation-to-relation operators designed to provide pattern detection, including conjunctions, disjunctions, and sequences. An interesting aspect of this pattern language is the possibility for users to program the selection and consumption policies of rules.

Like in Coral8 and StreamBase, a visual, plan-based language is also available inside a development environment based on Eclipse. This tool enables users to connect simple rules into a complex execution plan. Oracle CEP is integrated with existing Oracle solutions, which includes technology for distributed processing in clustered environment, as well

as tools for analysis of historical data.

Esper

Esper [112] is considered the leading open-source CEP provider. Esper defines a rich declarative language for rule specification, called EPL (Event Processing Language). EPL includes all the operators of SQL, adding ad-hoc construct for windows definition and interaction, and for output generation. The Esper language and processing algorithm are integrated into the Java and .Net (NEsper) as libraries. Users can install new rules from their programs and then receive output data either in a push-based mode (using listeners) or in a pull-based one (using iterators).

EPL embeds two different ways to express patterns: the first one exploits so called EPL Patterns, that are defined as nested constraints including conjunctions, disjunctions, negations, sequences, and iterations. The second one uses flat regular expressions. The two syntax offer the same expressiveness. An interesting aspect of Esper pattern is the possibility to explicitly program event selection policies, exploiting the *every* and *every-distinct* modifiers.

Esper supports both centralized and clustered deployments; in fact, using the EsperHa (Esper High Availability) mechanisms it is also possible to take advantage of the processing power of different, well-connected, nodes, to increase the system's availability and to share the system's load according to customizable QoS policies.

Tibco Business Events

Tibco Business Events [113] is another widespread complex event processing system. It is mainly designed to support enterprise processes and to integrate existing Tibco products for business process management. To do so, Tibco Business Events exploits the pattern-based language of Rapide, which enables the specification of complex patterns to detect occurrences of events and the definition of actions to automatically react after detection. Interestingly, the architecture of Tibco Business Events is capable of decentralized processing, by defining a network of event processing agents: each agent is responsible for processing and filtering events coming from its own local scope. This allows, for example, correlating multiple RFID readings before their value is sent to other agents.

IBM System S

In May 2009, IBM announced a *Stream Computing Platform*, called System S [114, 115, 116]. The main idea of System S is that of providing a computing infrastructure for processing large volumes of possibly high rate data streams to extract new information. The processing is split into basic operators, called *Processing Elements* (PEs): PEs are connected to each other in a graph, thus forming complex computations. System S accept rules specified using a declarative, SQL-like, language called SPADE [117], which embeds all traditional SQL operator for filtering, joining, and aggregating data. Rules written in SPADE are compiled into one or more PEs. However, differently from most of the presented systems, System S allows users to write their own PEs using a full featured programming language. This way users can write virtually every kind of function, explicitly deciding when to read an information from an input stream, what to store, how to combine information, and when to produce new information. This allows the definition of component performing pattern-detection, which is not natively supported by SPADE.

System S is designed to support large scale scenarios by deploying the PEs in a clustered environment, in which different machine cooperate to produce the desired results. Interestingly System S embeds a monitoring tool that uses past information about processing load to compute a better processing plan and to move operators from site to site to provide desired QoS.

Other commercial systems

Other widely adopted commercial systems exist, for which, unfortunately, documentation or evaluation copies are not available. We mention here some of them.

IBM WebSphere Business Events

IBM acquired CEP pioneer system AptSoft during 2008 and renamed it WebSphere Business Events [118]. Today, it is a system fully integrated inside the WebSphere platform, which can be deployed on clustered environment for faster processing. IBM WebSphere Business Events provides a graphical front-end, which helps users writing rules in a pattern-based language. Such a language allows detection of logical, causal, and temporal relationships between events, using an approach similar to the one described for Rapide and Tibco Business Events.

Event Zero

A similar architecture, based on connected event processing agents, is used inside Event Zero [119], a suite of products for capturing and processing information items as they come. A key feature of Event Zero is its ability of supporting real-time presentations and analysis for its users.

Progress Apama Event Processing Platform

The Progress Apama Event Processing Platform [120] has been recognized as a market leader for its solutions and for its strong market presence [121]. It offers a development tool for rule definition, testing and deployment, and a high-performance engine for detection.

Microsoft StreamInsight

Microsoft StreamInsight [122, 123, 124] is part of Microsoft SQL Server 2008 and it is based on the LINQ language [125]. LINQ was first designed to bring general purpose query facilities to the .NET framework and to the programming languages working on top of it. It is a declarative language, very similar to SQL.

In StreamInsight LINQ has been extended with time and count windows, to enable processing of streaming information in the form of event notifications. StreamInsight adopts an interval time model to support events with a duration. Moreover, LINQ supports the specification of ad-hoc aggregates and operators using traditional programming language. Finally, Microsoft provides a visual tool to specify and debug queries.

StreamInsight supports distributed deployments; however, this requires system administrator to manually configure the communication among the different processing nodes.

2.5 Discussion

The first consideration that emerges from the analysis and classification of existing IFP systems done in the previous section is a distinction between systems that mainly focus on data processing and systems that focus on event detection. This distinction does not necessarily map to the grouping of systems we adopted in Section 2.4. There are active databases which focus on event detection, while several commercial systems, which classify themselves as CEP, actually focus more on data processing than on event notification.

This distinction is clearly captured in the data model (Table 2.3), by the nature of items processed by the different systems. Moreover, by

looking at the nature of flows, we observe that systems focusing on data processing usually manage homogeneous flows, while systems focusing on event detection allow different information items to be part of the same flow.

If we look at the rule and language models (Tables 2.3 and 2.4), we may also notice how the systems that focus on data processing usually adopt transforming rules, defined through declarative or imperative languages, including powerful windowing mechanisms together with a join operator to allow complex processing of incoming information items. Conversely, systems focusing on event processing usually adopt detecting rules, defined using pattern-based languages that provide logic, sequence, and iteration operators as means to capture the occurrence of relevant events.

The data and rule models summarized in Table 2.3 also show that data uncertainty and probabilistic rules have been rarely explored in existing IFP systems. Since these aspects are critical in many IFP applications, i.e., when data coming from sources may be imprecise or even incorrect, we think that support for uncertainty deserves more investigation. It could increase the expressiveness and flexibility, and hence the diffusion, of IPF systems.

Coming back to the distinction between DSP and CEP, the time model (Table 2.3) emphasizes the central role of time in event processing. All systems designed to detect composite events introduce an order among information items, which can be a partial order (*causal*), or a total order (using an *absolute* or an *interval* semantics). Conversely, stream processing systems often rely on a *stream-only* time model: timestamps, when present, are mainly used to partition input streams using windows and then processing is performed using relational operators inside windows.

The importance of time in event detection becomes even more evident by looking at Table 2.4. Almost all systems working with events include the sequence operator, and some of them also provide the iteration operator. These operators are instead not provided by stream processing systems, at least those coming from the research community. Table 2.4, in fact, shows that commercial systems adopt a peculiar approach. While they usually adopt the same stream processing paradigm as DSP research prototypes, they also embed pattern-based operators, including sequence and iteration. While at first this approach could appear to be the most effective way to combine processing abstractions coming from the two worlds, a closer look to the languages proposed so far reveals many open issues. In particular, since all processing is performed on relational tables defined through windows, it is often unclear how the semantics of operators for pattern detection maps to the partitioning mechanisms introduced by windows. One gets the impression

that several mechanisms were put together without investing much effort in their integration and without carefully studying the best and minimal set of operators to include in the language to offer both DSP and CEP processing support. In general, we can observe that this research area is new and neither academia nor industry have found how to combine the two processing paradigms in a unifying proposal [126].

If we focus on the interaction style, we notice (Table 2.2) that the push-based style is the most used both for observation and notification of items. Only a few research systems adopt pull-based approaches: some of them (NiagaraCQ and OpenCQ) motivate this choice through the specific application domain in which they work, i.e., monitoring updates of web pages, where the constraint on timeliness is less strict. Among the systems focusing on such domain, only PB-CED is able to dynamically decide the interaction style to adopt for data gathering according to the monitored load.

As for the notification model, different commercial systems use an hybrid push/pull interaction style: this is mainly due to the integration with tools for the analysis of historical data, which are usually accessed on-demand, using pull-based queries. Finally, we observe that almost all systems that allow distributed processing adopt a push-based forwarding approach to send information from processor to processor. In Chapter 7 we will introduce and evaluate several protocols for distributed processing, including one that allow a dynamic switch between push and pull, to adapt the forwarding model to the actual load, trying to minimize the bandwidth usage and increase throughput [127].

Closely related with the interaction style is the deployment model. By looking at Table 2.2 we may notice a peculiarity that was anticipated in Section 2.2: the few systems that provide a networked implementation to perform filtering, correlation, and aggregation of data directly in network, also focus on event detection. It is an open issue if this situation is an outcome of the history that brought to the development of IFP systems from different communities, having different priorities, or if it has to do with some technical aspect, like the difficulty of using a declarative, SQL-like language in a networked scenario, in which distributed processing and minimization of communication costs are the main concern. Clustered deployments, instead, have been investigated both in the DSP and CEP domains.

Along the same line, we may notice (Table 2.1) how the systems that support a networked deployment do not provide a knowledge base and vice versa. In our functional model, the knowledge base is an element that has the potential to increase the expressiveness of the system but could be hard to implement in a fully networked scenario. This may

explain the absence of such a component in those systems that focus on efficient event detection in a strongly distributed and heterogeneous network.

Another aspect related with the functional model of an IFP system has to do with the presence or absence of the *Clock*. Table 2.1 shows how half of the systems include such a (logical) component and consequently allows rules to fire periodically, while the remaining systems provide a purely reactive behavior. This feature seems to be independent from the class of the system.

We cannot say the same if we focus on load shedding. Indeed, Table 2.1 shows that all systems providing such a mechanism belong to the DSP world. Load shedding, in fact, is used to provide guaranteed performance to users, and this is strictly related with the issue of agreeing QoS levels between sources, sinks, and the IFP system. While all CEP systems aim at maximizing performance through efficient processing algorithms and distribution strategies, they never explicitly take into account the possibility of negotiating specific QoS levels with their clients. It is not clear to us whether the adoption of these two different approaches (i.e., negotiated QoS vs. fixed, usually best-effort, QoS) really depends on the intrinsically different nature of data and event processing, or if it is a consequence of the different attitudes and backgrounds of the communities working on the two kinds of systems.

Another aspect differentiating CEP from DSP systems has to do with the possibility of performing recursive processing. As shown in Table 2.1, indeed, such mechanism is often provided by systems focusing on events, while it is rarely offered by systems focusing on data transformation. This can be explained by observing how recursion is a fundamental mechanism to define and manage hierarchies of events, with simple events coming from sources used to define and detect first-level composite events, which in turn may become part of higher-level composite events.

Another aspect that impacts the expressiveness of an IFP system is its ability of adapting the processing model to better suit the need of its users. According to Table 2.1 this is a mechanism provided by a few systems, while we feel that it should be offered by all of them. Indeed, while the multiple selection policy (with zero consumption) is the most common, in some cases it is not the most natural. As an example, in the fire alarm scenario described in Section 2.3.1, a single selection policy would be better suited: in presence of a smoke event followed by three high temperature events a single fire alarm, not multiple, should be generated.

Similarly, in Section 2.3.1 we observed that the maximum length of

the sequence *Seq* of information items that exits the *Decider* and enters the *Producer* has an impact on the expressiveness of the system. Here we notice how this feature can help us drawing a sharp distinction between traditional publish-subscribe systems and all other systems, which allow multiple items to be detected and combined. On the other hand, it is not clear if there is a real difference in expressiveness between those rule languages that result in a "bound" length for sequence *Seq* and those that have it "unbound". As an example, a system that allows to combine (e.g., join) different flows of information in a time-based window requires a potentially infinite *Seq*. The same is true for systems that provide an unbounded iteration operator. Conversely, a system that would provide count-based windows forcing to use them each time the number of detectable items grows, would have a bounded *Seq*. While it seems that the first class of languages is more expressive than the second, how to formally define, measure, and compare the expressiveness of IFP rule languages is still an open issue, which requires further investigation.

As a final consideration, we may notice (Table 2.1) that the ability to dynamically change the set of active rules was available in most active databases developed years ago, while it disappeared in more recent systems (with the remarkable exception of GEM and Tribeca). This is a functionality that could be added to existing systems to increase their flexibility.

2.6 Related Work

In this section we briefly discuss the results of on-going or past research which aims at providing a more complete understanding of the IFP domain. In particular we cover four different aspects: (i) we present work that studies general mechanisms for IFP; (ii) we review specific models used to describe various classes of systems, or to address single issues; (iii) we provide an overview of systems presenting similarities with IFP systems; (iv) we discuss existing attempts to create a standard for the IFP domain.

2.6.1 General Mechanisms for IFP

Many researchers focused on developing general mechanisms for IFP by studying (i) rule processing strategies, (ii) operator placement and load balancing algorithms, (iii) communication protocols for distributed rule processing, (iv) techniques to provide adequate levels of QoS, and (v) mechanisms for high availability and fault tolerance.

Query optimization has been widely studied by the database community [128, 129, 130]. Since in IFP systems different rules co-exist, and may be evaluated simultaneously when new input is available, it becomes important to look at the set of all deployed rules to minimize the overall processing time and resource consumption. This issue is sometimes called the *multi-query optimization* problem [131]. Different proposals have emerged to address this problem: they include shared plans [72], indexing [132, 133, 98], and query rewriting [11, 134] techniques. In the area of publish-subscribe middleware, the throughput of the system has always been one of the main concerns: for this reason different techniques have been proposed to increase message filtering performance [26, 23, 135, 136, 137].

When the processing is distributed among different nodes, beside the definition of an optimal execution plan, a new problem emerges: it is the *operator placement* problem, that aims at finding the optimal placement of entire rules or single operators (i.e., rule fragments) at the different processing nodes, to distribute load and to provide the best possible performance according to a system-defined or user-defined cost function. Some techniques have been proposed to address this issue [138, 139, 46, 79, 140, 31, 141, 30, 142, 143, 144, 127], each one adopting different assumptions on the underlying environment (e.g., cluster of well connected nodes or large scale scenarios with geographical distribution of processors) and system properties (e.g., the possibility of replicating operators). Most importantly, they consider different cost metrics; most of them consider (directly or indirectly) the load at different nodes, thus realizing load balancing, while others take into account network parameters, like latency and/or bandwidth. A classification of existing works can be found in [145].

In distributed scenarios it is of primary importance to define efficient communication strategies between the different actors (sources, processors, and sinks). While most existing IFP systems adopt a push-based style of interaction among components, some works have investigated different solutions. A comparison of push and pull approaches for Web-based, real-time event notifications is presented in [146]; some works focused on efficient data gathering using a pull-based approach [147], or a hybrid interaction, where the data to received is partitioned into push parts and pull ones according to a given cost function [148, 99]. Chapter 7 will introduce a hybrid push/pull approach also for the interaction between different processing nodes [127].

Different contributions, primarily from people working on stream processing, focused on the problem of bursty arrival of data (which may cause processors' overload) [149], and on providing required levels of QoS.

Many works propose load shedding techniques [19, 20, 77, 67, 150, 151] to deal with processors' overload. As already mentioned in Section 2.4, the Aurora/Borealis project allows users to express QoS constraints as part of a rule definition [152]: such constraints are used by the system to define the deployment of processing operators and to determine the best shedding policies.

Recently, a few works coming from the event-based community have addressed the problem of QoS. In particular, some of them have studied the quality of event delivery with respect to different dimensions [153, 154, 155]. Often such dimensions are merged together to obtain a single variable; the objective is that of properly allocating system resources in order to maximize such a variable. In [156] the authors study the quality of event delivery using a multi-objective approach in which the two competing dimensions of completeness of delivery and delay are considered together. The authors present an offline algorithm capable of finding the set of optimal solution to the multi-obective function, as well as a greedy approach that works online.

In [40], the authors address the problem of defining a proper semantics for event detection when communication and synchromization errors may occur. Their proposal extends the rule definition language with policies, so that the behavior of the system in presence of errors can be customized by the users, according to their needs. Several policies are presented, for example a *No-False-Positive* policy ensures the sinks that all received events have actually occurred, while a *Max-Delay* policy does not offer guarantees about the quality of results, but instead ensures that the delay between event detection and event delivery does not overcome a given threshold.

Finally, a few works have focused on high-availability and fault tolerance for distributed stream processing. Some of them focused on fail-stop failures of processing nodes [157, 158], defining different models and semantics for high-availability according to specific application needs (e.g., the fact that results have to be delivered to sinks at least once, at most once, or exactly once). They proposed different algorithms based on replication. Interestingly, [159] also takes into account network failures and partitioning, proposing a solution in which, in case of a communication failure, a processor does not stop but continues to produce results based on the (incomplete) information it is still able to receive; such information will be updated as soon as the communication can be reestablished. This approach defines a trade-off between availability and consistency of data, which can be configured by sinks.

2.6.2 Other Models

As we have seen, IFP is a large domain, including many systems. Although the literature misses a unifying model, which is able to capture and classify all existing works, various contributions are worth mentioning. First, there are the community specific system models, which helped us understanding the different visions of IFP and the specific needs of the various communities involved. Second, there are models that focus on single aspects of IFP, like timing models, event representation models, and language related models.

System Models

As described in Section 2.2, two main models for IFP have emerged, one coming from the database community and one coming from the event-processing community.

In the database community active database systems represent the first attempt to define rules to react to information in real-time, as it becomes available. The research field of active database systems is now highly consolidated and several works exist offering exhaustive descriptions and classifications [16, 17, 160].

To overcome the limitations of active database systems in dealing with flows of external events, the database community gave birth to the Data Stream Managements Systems. Even if research on DSMSs is relatively young, it appears to have reached a good degree of uniformity in its vocabulary and in its design choices. There exist a few works that describe and classify such systems, emphasizing common features and open issues [9, 50]. Probably the most complete model to describe DSMSs comes from the authors of Stream [21]; it greatly influenced the design of our own model for a generic IFP system. Stream is also important for its rule definition language, CQL [42], which captures most of the common aspects of declarative (SQL-like) languages for DSMS clearly separating them through its stream-to-relation, relation-to-relation, and relation-to-stream operators. Not surprisingly many systems, even commercial ones [44, 110] directly adopt CQL or a dialect for rule definition. However, no single standard has emerged so far for DSMSs languages and competing models have been proposed, either adopting different declarative approaches [54] or using a graphical, imperative approach [45, 106].

Our model has also been developed by looking at currently available event-based systems, and particularly publish-subscribe middleware. An in depth description of such systems can be found in [10, 82]. Recently, several research efforts have been put in place to cope with rules involving

multiple events: general models of CEP can be found in [1, 27, 161].

Models for Single Aspects of IFP

A relevant part of our classification focuses on the data representations and rule definition languages adopted by IFP systems, and on the underlying time model assumed for processing. Some works have extensively studied these aspects and they deserve to be mentioned here. As an example, some papers provide an in depth analysis of specific aspects of languages, like blocking operators, windows, and aggregates [162, 54, 163, 164]. Similarly, even if a precise definition of the selection and consumption policies that ultimately determine the exact semantics of rule definition languages is rarely provided in existing systems, there are some papers that focus on this issue, both in the area of active databases [32] and in the area of CEP systems [165, 100].

In the areas of active databases and event processing systems, much has been said about event representation: in particular a long debate exists on timing models [37, 38], distinguishing between a *detection*, and an *occurrence* semantics for events. The former associates events with a single timestamp, representing the time in which they were detected; it has been shown that this model provides a limited expressiveness in defining temporal constraints and sequence operators. On the other hand, the *occurrence semantics* associates a duration to events. In [39], the authors present a list of desired properties for time operators (i.e., sequence and repetition); they also demonstrate that it is not possible to define a time model based on the occurrence semantics that could satisfy all those properties, unless time is represented using timestamps of unbounded size. It has been also demonstrated that, when the occurrence semantics is used, an infinite history of events should be used by the processing system to guarantee a set of desired properties for the sequences operators [39].

Finally, as we observed in Sections 2.4 and 2.5, DSMSs, and particularly commercial ones, have started embedding (usually simple) operators for capturing sequences into a traditional declarative language: this topic has been extensively studied in a proposal for adding sequences to standard SQL [49].

All these works complement and complete our effort.

2.6.3 Related Systems

Besides the systems described in Section 2.4, other tools exist, which share many similarities with IFP systems.

Runtime Verification Tools

Runtime verification [166, 167, 168, 169] defines techniques to check whether the run of a system under scrutiny satisfies or violates some given rules, which usually model correctness properties [170].

While at first sight runtime verification may resemble IFP we decided not to include the currently available runtime verification tools into our list of IFP systems for two reasons. First, they usually do not provide general purpose operators for information analysis and transformation; on the contrary they are specifically designed only for checking whether a program execution trace satisfies or violates a given specification of the system. Second, the specific goal they focus on has often led to the design of ad-hoc processing algorithms, which cannot be easily generalized for other purposes.

In particular, as these tools are heavily inspired by model checking [171], rules are usually expressed in some kind of linear temporal logic [172]. These rules are translated into *monitors*, usually in the form of finite state or Büchi automata, which read input events coming from the observed system and continuously check for rules satisfaction or violation. According to the type of monitored system, events may regard state changes, communication or timing information. Guarded systems may be instrumented to provide such pieces of information, or they can be derived through external observation. It is worth noting that while the monitored systems are often distributed, the currently available runtime verification systems are centralized.

Focusing on the language, we may observe that temporal logics enable the creation of rules that resemble those that can be defined using pattern-based languages, as described in Section 2.3. In particular, they allow composition of information through conjunctions, disjunctions, negations, and sequences of events. Additionally, much work is based on logics that also include timing constraints [173, 174]; this introduces a flexible way to manage time, which can be compared to user defined windows adopted by some IFP systems. Another aspect explored by some researchers is the possibility to express parameterized rules through variables [175], which are usually associated with universal and existential quantifiers.

A notable difference between IFP systems and runtime verification tools is that the former only deal with the history of past events to produce their output, while the latter may express rules that require future information to be entirely evaluated. As the output of runtime verification systems is a boolean expression (indicating whether a certain property is satisfied or violated) different semantics have been proposed

to include all the cases in which past information is not sufficient to evaluate the truth of a rule. Some work uses a three values logic, where the output of the verification can be *true*, *false*, or *inconclusive* [176, 177]. Other work considers a given property as satisfied until it has not been violated by occurred events [166]. Another approach is that of adopting a four values logic, including the *presumably true* and *presumably false* truth values [178]. When a violation is detected some of the exisiting tools may also be programmed to execute a user defined procedure [179], for example to bring back the system in a consistent state.

Runtime Monitoring Tools

Similarly to IFP systems, runtime monitoring tools consider not only the satisfaction or violation of properties, as it happens in runtime verification, but also manipulation of data collected from the monitored system. Accordingly, some monitoring tools, even when developed with the specific domain of monitoring in mind, present design choices that can be easily adapted to the more general domain of IFP. For this reason, we included them in the list of IFP systems analyzed in Section 2.4. Others are more strongly bound to the domain they are studied for: we describe them here.

A lot of effort has been recently put into runtime monitoring of service oriented architectures (SOAs) and, more specifically web services. The focus is on the analysis of the quality of service compositions, to detect bottlenecks and to eliminate them whenever possible. Different rule languages have been proposed for SOA-oriented monitoring systems. Some of them are similar to the pattern-based languages described in Section 2.3 and express content and timing constraints on the events generated by services [180, 181]. Others adopt imperative or declarative languages [182] to express transforming rules. Events usually involve information about service invocations, including the content of exchanged messages. Moreover, runtime monitoring languages often provide constructs to define variables and to manipulate them, making it possible to store and aggregate information coming from multiple invocations.

From an implementation point of view, some systems integrate monitoring rules within the process definition language (e.g., BPEL) that describes the monitored process, thus enabling users to express conditions that must hold at the bounds of service invocations [183, 184, 185]. Other systems, instead, adopt an autonomous monitor.

Scalable Distributed Information Management Systems

Like many IFP systems, distributed information management systems [186, 187] are designed to collect information coming from distributed sources to monitor the state and the state's changes in large scale scenarios. Such systems usually split the network into non overlapping zones, organized hierarchically. When information moves from layer to layer it is aggregated through user-defined functions, which progressively reduce the amount of data to forward. This way, such systems provide a detailed view of nearby information and a summary of global information. Getting the real data may sometimes require visiting the hierarchy in multiple steps.

If compared with the IFP systems presented in Section 2.4, distributed information management systems present some differences. First, they are not usually designed to meet the requirements of applications that make strong assumption on timeliness. This also reflects in the interaction style they offer, which is usually pull-based (or hybrid). Second, they are generally less flexible. In fact, they usually focus on data dissemination and not on data processing or pattern detection. Aggregation functions are only used as a summarizing mechanisms, and are not considered as generic processing functions [186]. As a consequence they usually offer a very simple API to applications, which simply allows to install new aggregation functions, and to get or put information into the system.

2.6.4 IFP Standardization

Recently, much effort has been put in trying to define a common background for IFP systems. Proposals came mainly from the event processing community, where a large discussion on the topic is undergoing, starting from the book that introduced the term complex event processing [1] and continuing on web sites and blogs [188, 189]. An *Event Processing Technical Society* [190] has been founded as well, to promote understanding and advancement in the field of event processing and to develop standards.

All these efforts and discussions put a great emphasis on possible applications and uses of CEP systems, as well as on the integration with existing enterprise solutions. For these reasons, they received a great attention from industry, which is rapidly adopting the term "complex event processing". On the other hand this work is still in its infancy and no real unifying model has been proposed so far to describe and classify complex event processing systems. Our work goes exactly in this

direction.

2.7 Conclusions

The need for processing large flows of information in a timely manner is becoming more and more common in several domains, from environmental monitoring to finance. This need was answered by different communities, each bringing its own expertise and vocabulary and each working mostly in isolation in developing a number of systems we collectively called IFP systems. In this first chapter we surveyed the various IFP systems developed so far, which include active databases, DSMSs, CEP systems, plus several systems developed for specific domains.

Our analysis shows that the different systems fit different domains, from data processing to event detection, and focus on different aspects, from the expressiveness of the rule language, to performance in large scale scenarios, adopting different approaches and mechanisms.

From the discussion in Section 2.5 we identify several research challenges for the IFP domain. First of all, it would be usefull to characterize the differences in term of expressiveness among the existing rule languages. In fact, while our analysis allows to draw a line between languages oriented toward data processing and languages oriented toward event detection, putting in evidence the set of operators offered by each language, it is still an open issue to capture how each operator contributes to the actual expressiveness of the language.

As we have seen, the current trend in commercial products is to offer languages that merge together a processing style that is typical of data processing languages, with advanced operators for pattern detection. This approach often makes languages complex and counterintuitive. Currently, we believe that none of the existing proposals has been able to organically combine full data processing and full event detection capabilities into a single solution. Moreover, we believe that further investigations are still needed to understand whether such a solution is first of all feasible, and than useful for existing application scenarios.

Starting from these premises, in Chapter 3 we present TESLA, a rule definition language explicitly designed for event-based scenarios. By focusing on a specific domain, TESLA remains compact, providing all the features required for event detection using a small set of expressive operators.

Related with the issue of expressiveness is the ability to support uncertainty in data and rule processing. Only a few work have addressed this issue, but we feel this is an area that requires more investigation,

since it is something useful in several domains. Along the same line, we noticed how the ability for a rule to programmatically manipulate the set of deployed rules, by adding or removing them, is common in active databases but is rarely offered by more recent IFP systems. Again, we think that this is potentially very useful in several cases, and should be offered to increase the expressiveness of the system. We are currently investigating both issues, and we plan to introduce them as part of TESLA in the near future.

Despite several systems have been now developed, proposing efficient processing algorithms, we believe that further advancements in the field are still possible. We discuss this issue in detail in Chapters 5, 4, and 6, providing new solutions that can easily take advantage of the increasing power of modern parallel hardware.

A final issue has to do with the topology of the system and the interaction style. In our survey we observed that IFP systems either focus on throughput, looking at performance of the engine with an efficient centralized or clustered implementation, or they focus on minimizing communication costs, by performing filtering, correlation, and aggregation of data directly in network. Both aspects are relevant for a system to maximize its throughput in a widely distributed scenario and they are strongly related with the forwarding model adopted (either push or pull).

We will discuss these issues in details in Chapter 7, where we compare different protocols for distributed processing.

To conclude this chapter, we observe that although the IFP domain is now mature, and many systems were developed by academia and industry, there is still space for new innovative approaches, possibly integrating some of the ideas already present in some of the systems we analyzed. The model presented in Section 2.3 may ease this job, helping researchers coming from different communities and background in comparing the different approaches developed so far and in understanding how they can be combined together.

3 The TESLA Language

3.1 Introduction

As we have seen in Chapter 2, the increasing number of applications requiring large amount of information to be timely processed as it flows from the peripheral to the center of the system, led to the developement of different IFP systems and languages.

In this chapter we focus on the languages used to define processing rules, i.e., to express how incoming information has to be processed. As already mentioned in Chapter 2 we believe that none of the languages proposed so far is entirely adequate to fully support the needs that come from application scenarios. On one side, the data transformation approach adopted by the DSP model is not suited to recognize patterns of incoming items tied together by complex temporal relationships. On the other side, CEP languages are often oversimplified, providing only a small set of operators, insufficient to express a number of desirable patterns and the rules to combine incoming information to produce new knowledge. Even worse, the semantics of such languages is usually given only informally, which leads to ambiguities and makes it difficult compare the different proposals.

To overcome these limitations, we propose a new language called *TESLA* (*Trio-based Event Specification LAnguage*). Each TESLA rule considers incoming data items as *notifications of events* and defines how (*composite*) *events* are defined from simpler ones. Despite an easy to use, clean syntax, with a limited number of different operators, TESLA is highly expressive and flexible, as it provides content and temporal constraints, parameterization, negations, sequences, aggregates, timers, and fully customizable policies for event selection and consumption. At the same time, not to incur in the semantic ambiguities affecting many existing languages, the TESLA semantics is formally specified by using *TRIO* [191, 192], a first order, metric temporal logic.

The rest of the chapter is organized as follows: in Section 3.2 we better motivate the limitations we found in existing works and clarify our design goals; in Section 3.3 we present the TRIO logic. In Section 3.4 we introduce the TESLA event model and system architecture, and describe our language in details, providing the semantics for all valid operators,

drawing some conclusions in in Section 3.5.

3.2 Why a new language

To better justify the need for a new event specification language we use an example, which illustrates the main limitations of existing approaches and shows the kind of expressiveness and flexibility we need.

3.2.1 A motivating example

Consider an environmental monitoring application that processes information coming from a sensor network. Sensors notify their position, the temperature they measure, and the presence of smoke. Now, suppose a user has to be notified in case of fire. She has to teach the system to recognize such a critical situation starting from the raw data measured by sensors. Depending from the environment, the application requirements, and the user preferences, the notion of fire can be defined in many different ways. Here we present four possible defining rules and we use them to illustrate some of the features an event processing language should provide.

i. Fire occurs when temperature higher than 45 degrees and some smoke are detected in the same area within 3 min. The fire notification has to embed the temperature actually measured.

ii. Fire occurs when temperature higher than 45 degrees is detected and it did not rain in the last hour.

iii. Fire occurs when there is smoke and the average temperature in the last 3 min. is higher than 45 degrees.

iv. Fire occurs when at least 10 temperature readings with increasing values and some smoke are detected within 3 min. The fire notification has to embed the average temperature of the increasing sequence.

First of all, they *select* relevant notifications from the history of all received ones according to a set of constraints. Two kinds of constraints are used: the first one selects elements on the basis of the values they carry, either to choose single notifications (e.g. those about temperature higher than 45 degrees) or to choose a set of related notifications (e.g. those coming from the same area). The latter case is usually called *parameterization*. The second kind of selection constraints operates on

the timing relationships among notifications (e.g. selecting only those generated within 3 minutes) and allows to capture *sequences* of events (e.g., high temperature followed by smoke or vice-versa).

Beside selection, rule (*ii*) introduces *negation*, by requiring an event not to occur in a given interval. Similarly, rule (*iii*) introduces *aggregates*. In particular, it defines a function (average) to be applied to a specified set of values (temperature readings in the last 3 minutes) to calculate the value to be carried by the complex event. Rule (*iv*) is interesting as it combines the two kinds of selection constraints (those on values and those on timing) to define an *iteration* that selects those elements that bring growing temperature readings. Such kind of rules is common in various domains, like in financial applications for stock monitoring, where they are used to promptly detect relevant market trends.

Finally, when the desired combination of notifications has been detected, rules have to specify which notification to create (e.g. fire) and they have to define its inner structure (e.g. the notification has to embed a temperature reading).

As discussed in Chapter 2 existing systems significanlty differ in the set of operators they provide, which, in turn, determine their expressiveness. As an example, not all languages provide parameterization or iterations [7]. We think that a language for CEP should be able to express all the constructs above: selection, parameterization, negations, aggregates, sequences, and iterations. Additionally, it should be simple and unambiguous, i.e., it should be easy to write rules having a clear and precise semantics.

3.2.2 Limitations of existing languages

As discussed in Chapter 2, most of the languages recently proposed in the literature to express rules like those above can be classified in two groups: *Data Stream Processing* (DSP) and *Complex Event Processing* (CEP) languages.

As observed during our analysis of existing systems, the most representative language of the first class is CQL [42], created within the Stream project [43].

We report here the main features of CQL, to better motivate why we believe that DSP language are not suitable to naturally express some of the rules presented above. CQL is designed to deal with homogeneous *streams* of timestamped tuples, all sharing the same schema. Each CQL rule (*query* in CQL jargon) takes one or more streams as inputs and produces one output stream. Queries are defined using three types of operators. *Stream-to-Relation* (S2R) operators select a portion of a stream

to implicitly create a traditional database table. They operate either on time (e.g. selecting tuples received in the last 5 minutes every time a new tuple arrives) or on the number of elements (e.g. selecting the last 10 tuples every time a new one arrives). *Relation-to-Relation* (R2R) operators are mainly standard SQL operators. *Relation-to-Stream* (R2S) operators generate new streams from tables, after data manipulation. Each CQL query is composed by a S2R operator, one or more R2R operators, and one R2S operator.

As already observed in Chapter 2, a key aspect of languages like CQL is forgetting the ordering between elements when moving from streams to relations. No explicit sequencing operators are provided and timestamps, if not artificially introduced as part of the tuples' schema, cannot be referenced during R2R processing. As a result, rules like the fourth of our example, which involve a single sequence of information items, are very hard to write in CQL, or even impossible if tuples do not include explicit references to time. More in general, all queries that select elements from the history of received items using timing constraints do not find a natural support in DSP languages.

As we have seen in Section 2.4 more complex DSP languages exist, which extend the expressive power of CQL: a remarkable example is represented by ESL [52], a Touring complete DSP language. With these languages detecting complex patterns becomes possible but it remains difficult to support and far from natural, since the general schema of the language remains that of CQL. Moreover, such languages, with their more complex and less common syntax, suffer some limitations typical of CEP languages (see below), mainly in terms of lack of a rigorous semantics.

As we said, CEP languages present a different processing model w.r.t. DSP ones. If we consider again our modeling framework, the main differences can be found in the data model, time model, and language model. First of all they consider incoming information as event notifications. Accordingly, they usually adopt an absolute, or interval time model, thus specifying an ordering relationship among items. This allows them to include explicit operators that rely on time, including sequences. This model is better suited to naturally express the rules in our example; however, existing languages present two problems.

First, most of them are extremely simple and present serious limitations in terms of expressiveness; for example, some languages force sequences to capture only adjacent events [98], making it impossible to express rules like (*i*) and (*iii*) above. Often negations are not allowed [46, 98], or cannot be expressed through timing constraints [193], like in rule (*ii.*) above. Other widespread limitations are the lack of a

full-fledged iteration operator (Kleene+ [194]), that could allow rules to capture a priori unbounded repetitions of events, like in rule (*iv.*), and the lack of processing capabilities for computing aggregates.

Second, a formal definition of operators is hardly ever provided, leading to semantic ambiguities. A typical problem is related with the specification of selection and consumption policies (see Section 2.3.2). Only a few language precisely express their choice in terms of selection and consumption policies, leaving to the system implementation the task of fixing them.

Without a precise selection and consumption policies, even the most basic rules becomes difficult to understand. As an example of this problem, consider rule (i) above. It defines a simple conjunction of two items, which is supported by almost all existing CEP languages.

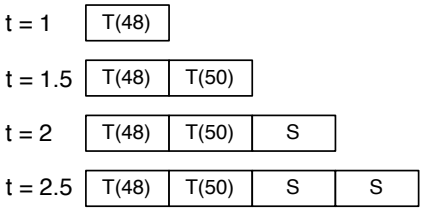


Figure 3.1: Monitored events history

However, if you consider the sequence of events depicted in Figure 3.1 the ambiguities related with the lack of precise policies emerge. Suppose that all events come from the same area and `T(x)` represents the event notification *temperature* $= x$, while `S` represents the presence of smoke. Initially, an event `T(48)` occurs, followed by another one, `T(50)`. When the event `S` is received the first issue arises: how many fire notification should the system generate? Two (one for each pair `<T(48),S>` and `<T(50),S>`) or just one? And, if we choose to deliver only one fire notification which value of temperature should it embed? These questions cannot be answered without a precise selection policy. Now suppose that the system reacts by producing two notifications, what happens when another event `S` occurs, like at $t = 2.5$? In some sense the two events `T` have already been "used": should they be considered again or not? In this case the ambiguity rise from a lack of precise consumption policy.

It is worth noting that apart from a problem of lack of precise semantics, the case above also evidence a problem of expressiveness, since different applications may require different event selection and consumption policies. Not only, sometimes a single application may need different policies for different rules. Alert notifications, for example, are usually

required only once, even when they can be generated by multiple combinations of events; financial analysts, on the contrary, may be interested in all possible combinations of stock events. For this reason we think that selection and consumption policies should be accessible and customizable within the rule specification language, thus allowing the rule manager to choose the most appropriate one for each application and each rule within the same application. On the contrary, as we have seen in our analysis of systems in Section 2.4, existing CEP languages (e.g. [46, 195, 98]), define, often implicitly and only through an actual reference implementation, a unique selection and consumption policy and do not allow users to change them. Some remarkable exceptions exist: in particular some languages designed for Active DBMSs allow users to choose selection and consumption policies as part of the definition of a rule [55]. However, also in these cases, only a few predefined choices are possible and users cannot tailor them to their needs.

3.2.3 TESLA design goals

Moving from these considerations we designed TESLA to overcome the limitations found in other languages, providing a high degree of expressiveness to users while keeping a simple syntax with a rigorously defined semantics. In particular, TESLA provides selection operators, parameterization, negations, aggregates, sequences, iterations, and fully customizable event selection and consumption policies, while also supporting reactive and periodic rule evaluation within a common syntax. At the same time, we provide a formal semantics for TESLA using a first order temporal logic. The remainder of this paper discusses these issues in details.

3.3 TRIO: A brief overview

TRIO [191, 192] is a first order logical language augmented with temporal operators, which enable to express properties whose value change over time. TRIO temporal operators, unlike those of conventional temporal logic, provide a metric on time: they express the length of time intervals quantitatively. The meaning of a TRIO formula is not absolute: it is given with respect to a current time instant that is left implicit in the formula. These two properties make TRIO well suited to naturally specify events and their occurrence.

The alphabet of TRIO includes sets of names for variables, functions, and predicates, plus a fixed set of operators, including propositional symbols ($\wedge$, $\neg$), quantifiers ($\forall$), and the temporal operators *Futr* and *Past*.

TRIO is a typed language: variables, functions, and predicates have their own type, which determines the set of values they can assume, return, or take as arguments. Among the allowed types there is a distinguished one, required to be numeric in nature: the *temporal domain*. TRIO distinguishes between *time-dependent* variables (resp. functions and predicates), whose value may change with time, and *time-independent* ones, whose value is independent from time.

The syntax of TRIO is recursively defined as follows:

- Every variable is a term
- Every n-ary function applied to n terms is a term

If a term is a variable, then its type is the type of the variable; if the term results from the application of a function, then its type is the range of the function.

- Every n-ary predicate applied to n terms of the appropriate types is a formula
- If A and B are formulas, $\neg A$ and $A \wedge B$ are formulas
- If A is a formula and x is a time-independent variable, $\forall x\, A$ is a formula
- If A is a formula and t is a term of the temporal type, then *Futr(A, t)* and *Past(A, t)* are formulas

Abbreviations for the propositional operators $\vee$, $\rightarrow$, $true$, $false$, $\leftrightarrow$ and for the derived existential quantifier $\exists$ are defined as usual.

The semantics of TRIO is formally defined in [191, 192], here we focus on the two temporal operators *Futr* and *Past*. In particular, formula *Past(A, t)* (resp. *Futr(A, t)*) is true if A holds t time units in the past (resp. future) w.r.t. the current time, which is left implicit in the formula.

A lot of temporal operators have been derived from *Futr* and *Past*. In the following we will make use of two of them: always (*Alw(A)*) and within the past ($WithinP(A, t_1, t_2)$), where A is a formula and t_1, t_2 terms of temporal domain; they are defined as follows:

$$\begin{aligned} Alw(A) = A \wedge\ & \forall t(t > 0 \rightarrow Futr(A, t)) \\ \wedge\ & \forall t(t > 0 \rightarrow Past(A, t)) \end{aligned}$$

$$WithinP(A, t_1, t_2) = \exists x(t_1 \leq x \leq t_1 + t_2 \wedge Past(A, x))$$

Before concluding this brief overview of TRIO, it is worth noting how such logic only requires the temporal domains to be numeric, and does not dictate further constraints: it can be, for example, either discrete or continuous. TESLA keeps the same property, enabling designers to choose the more suitable temporal domain according to their needs.

3.4 Language Definition

In this Section we present the TESLA language in details; in particular we describe the TESLA event and rule models, then we define the general structure of rules and we show how they can be translated into TRIO formulas that precisely define their semantics. Finally, we present all the possible patterns of events that can be captured through TESLA rules, describing their use through various examples.

3.4.1 TESLA event and rule model

In TESLA we assume events, i.e. things of interest, to occur instantaneously at some points in time. In order to be understood and processed, events have to be observed by *sources* (see Figure 3.2), which encode them in *event notifications* (or simply *events*). We assume that each event notification has an associated *type*, which defines the number, order, names, and types of the *attributes* that build the notification. Notifications have also a timestamp, which represents the occurrence time of the event they encode. The issue of who timestamps events, e.g., the sources or the CEP system, and the need of ad-hoc mechanisms to cope with out-of-order arrivals, have been discussed in the past [196] and are out of the scope of this chapter. These are system issues that do not impact the TESLA language, which is meant to process events in timestamp order, whoever sets it. By referring to the time model introduced by our modelling framework in Section 2.3.6, we can say that TESLA adopts an *absolute* time.

As an example, an event can be the temperature reading in a room at a specific time. A sensor may observe this event and generate the following notification:

$$Temp@10(Room = "Room1",\ Value = 24.5)$$

Where *Temp* represents the type of the notification and 10 is the timestamp. The *Temp* type defines two attributes: a string that identifies the room in which the temperature was measured, and the actual measure (a float). As attributes are ordered, notifications may also omit their names.

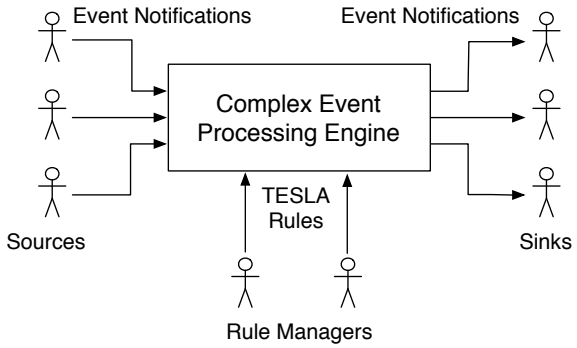


Figure 3.2: TESLA Reference Architecture

TESLA *rules* define composite events from simpler ones. The latter can be observed directly by sources or they can be composite events defined by other rules. This means that TESLA allows for *recursive processing*, as defined in our functional model in Section 2.3.1. In the following we will refer to this mechanism also using the term "hierarchies of events".

Sinks subscribe to events and receive notifications as soon as their requests are met. Subscriptions are as simple as in traditional publish-subscribe languages and include the type of relevant events together with a filter over the content of events attributes, like in the following example:

$$Subscribe(Temp,\ Room = "Room1" and\ Value > 20)$$

Subscriptions may refer to primitive events, i.e., those directly observed by sources, or to composite ones, i.e., those derived through TESLA rules. The resulting architecture distinguishes between *Rule managers* who define TESLA rules, and sinks who subscribe to events. As we will argument later, we fell that keeping the two roles separate helps building reusable rules.

3.4.2 Structure of the rules

Each TESLA rule has the following general structure:

$define$	$CE(Att_1 : Type_1, ..., Att_n : Type_n)$
$from$	$Pattern$
$where$	$Att_1 = f_1, .., Att_n = f_n$
$consuming$	$e_1, .., e_n$

Intuitively the first two lines define a (composite) event from its constituents, specifying its structure — $CE(Att_1$: $Type_1$, ..., Att_n: $Type_n)$ — and the pattern of simpler events that lead to the composite one. The *where* clause defines the actual values for the attributes Att_1, .., Att_n of the new event using a set of functions f_1, .., f_n, which may depend on the arguments defined in *Pattern*. Finally, the optional *consuming* clause defines the set of events that have to be invalidated for further firing of the same rule.

3.4.3 Semantics of rules

Each TESLA rule associates a patter of events p with the composite event e it represents. Accordingly, to provide a precise semantics for rules we need to express the logical equality between the validity of p at a given instant and the occurrence of e at the same instant.

In TRIO we can express the occurrence of an event using a time-dependent predicate, which becomes true when the event occurs. On the other hand, as we will prove through various examples, in TESLA several events of the same type, possibly with the same attribute values, may occur at the same time. To differentiate them we have to introduce the concept of *label*: it is a unique global identifier for event notifications. Notice that labels are only required to translate TESLA rules into TRIO formulas, while they do not appear inside the TESLA language, which operates at a higher level of abstraction. While we can safely assume that events coming from external sources already have their own unique label, we have to define the label of those composite events defined through TESLA rules. To do so, we observe that a given set of events s can satisfy a rule r at most once (we will prove this later through the *uniqueness of selection* theorem). We leverage this property assuming that a time-independent label generation function *lab* is defined, which returns new labels taking two arguments: a unique rule identifier, and a set of labels (those of the set of notifications s that satisfied the rule leading to the new event). For labels to uniquely identify composite events, *lab* has to be injective:

$$\forall\ r_1, s_1, r_2, s_2$$
$$((lab(r_1, s_1) = lab(r_2, s_2)) \ \leftrightarrow\ (r_1 = r_2 \wedge s_1 = s_2))$$

Once labels have been introduced we can use them to formally define the occurence of events through the predicate $Occurs(Type, Label)$, which is true at the time when the event of type *Type* having label *Label* occurs, and false in every other instant. The fact that labels are unique and that a given event notification occurs only once, is formally captured through the following formulas:

$$Alw\ \forall e_1, e_2 \in E, \forall l \in L$$
$$((Occurs(e_1, l) \wedge Occurs(e_2, l)) \rightarrow e_1 = e_2)$$

$$Alw\ \forall e_1, e_2 \in E, \forall l \in L, \forall t > 0\ (Occurs(e_1, l) \rightarrow$$
$$(\neg Past(Occurs(e_2, l), t) \wedge \neg Futr(Occurs(e_2, l), t)))$$

Where L is the set of all valid labels and E is the set of all event types. The first formula states that, in a given instant of time, there cannot be two notifications having the same label and different types. The second formula guarantees that, if an event with label l occurs at time t, no other events having the same label can occur at different times.

To reason about the content of event notifications we introduce the domain N of all valid names for event attributes. Since attributes can have different types (e.g. string, int, float) for each type X we define a time-independent function $attVal_X : L \times N \rightarrow X$: given a label l and the name of an attribute n it returns the value of the attribute in the event notification having label l. For simplicity, in the following examples we assume all attributes share a common domain V, so that we can use a single function generically called $attVal$ to associate attribute names to their values. For the same reason, from now on we will omit types from the *define* clause of our rules.

Using the elements defined above, a generic TESLA rule, in the form shown in Section 3.4.2, is translated into the following TRIO formula (we omit the translation of the *consuming* clause, as it will be addressed later):

$$
\begin{aligned}
&Alw\ \forall l_1, .., l_m \in L, \forall n_1, .., n_n \in N \\
&((Occurs(CE, lab(r, \{l_1, .., l_m\}))\ \leftrightarrow\ Pattern) \wedge \\
&(Pattern\ \rightarrow\ attVal(lab(r, \{l_1, .., l_m\}), n_1) = f_1) \wedge \\
&(Pattern\ \rightarrow\ attVal(lab(r, \{l_1, .., l_m\}), n_n) = f_n))
\end{aligned}
$$

Where $\mathbb{N}$ is the set of all natural numbers and $r \in \mathbb{N}$ represents a unique identifier for the translated TESLA rule, while $l_1, .., l_m \in L$ are the labels of all event notifications captured by *Pattern* and $n_1, .., n_n$ are the attribute names for the event type CE. The TRIO formula asserts that, in every instant of time in which *Pattern* becomes true, an event notification of type CE occurs, whose label is defined by the *lab* function applied to the number of the rule r and the set of labels of all event notifications captured by the pattern (and viceversa). The formula also specifies the values for the new event's attributes using the functions defined in the corresponding TESLA rule.

3.4.4 Valid patterns

In the discussion so far we left unspecified the inner structure of a pattern; we now introduce all operators used in TESLA to define valid patterns.

Event occurrence

The simplest type of event pattern represents the occurrence of a single event, possibly satisfying a set of constraints. As an example consider the following requirement: *generate an overflow notification when the level of water in a river overcomes 20; the notification must include the name of the river.* This can be translated into the following TESLA rule:

$$
\begin{aligned}
&define && Overflow(Name) \\
&from && WaterLevel(Level > 20)\ as\ WL \\
&where && Name = WL.Name
\end{aligned}
$$

As the example shows, TESLA puts the constraints over the content of an event into parentheses after the type of the event. In this case a single constraint is needed but TESLA accepts conjunctions of constraints as well. To access the field of an event TESLA uses a *dot* notation *event-name.attribute-name*. A name is associated to an event using the *as*

keyword. When only an event of a given type is present in the pattern, like in our example, it is also possible to omit the *as* clause and use the event type as the name of the event. We can also split the type of the selected event from the constraints on its attributes; for example the *from* clause of the previous formula could be written as *WaterLevel() as WL and WL.Level>20*.

Translating in TRIO rules involving single events is easy. Notice that to keep formulas more compact, here and in the following we assume all free variables to be universally quantified at the outermost level and all formulas to start with the *Alw* operator, which we omit. Using these conventions we provide the translation of a general rule selecting a single event:

$$
\begin{array}{ll}
define & CE(Att_1, .., Att_n) \\
from & SE(Att_x \ op \ Val_x) \\
where & Att_1 = f_1, .., Att_n = f_n \quad \triangleq
\end{array}
$$

$$
\begin{array}{l}
(Occurs(CE, lab(r, \{l_1\})) \ \leftrightarrow \\
(Occurs(SE, l_1) \ \wedge attVal(l_1, Att_x) \ op \ Val_x)) \wedge \\
(Occurs(SE, l_1) \ \wedge attVal(l_1, Att_x) \ op \ Val_x) \ \rightarrow \\
(attVal(lab(r, \{l_1\}), Att_1) = f_1 \ \wedge .. \wedge \\
attVal(lab(r, \{l_n\}), Att_n) = f_n)
\end{array}
$$

To capture the meaning of this formula, consider the situation in which two different events of type *SE*, *A* and *B*, occur at the same time. As the two events necessarily have different labels, l_A and l_B, to satisfy the formula above two different *CE* notifications must be generated, having labels $lab(r, \{l_A\})$ and $lab(r, \{l_B\})$. The two notifications are guaranteed to be distinct (i.e. to represent different event occurrences), as the *lab* function is injective.

Event composition

To capture the occurrence of several, related events, TESLA provides three *event composition* operators: *each-within*, *first-within*, and *last-within*. All event composition operators bind the occurrence of an event to the occurrence of another one, introducing a detection window. Using the terminology introduced in Section 3.2 we can say that they differ from each other according to the *selection policy* they define. As an example, consider the following rules:

$$
\begin{array}{ll}
define & Fire(Val) \\
from & Smoke()\ and \\
 & each\ Temp(Val > 45)\ within\ 5min\ from\ Smoke \\
where & Val = Temp.Val
\end{array}
$$

$$
\begin{array}{ll}
define & Fire(Val) \\
from & Smoke()\ and \\
 & last\ Temp(Val > 45)\ within\ 5min\ from\ Smoke \\
where & Val = Temp.Val
\end{array}
$$

Both rules define the *Fire* event from *Smoke* and *Temp*. The first rule leads to notify a *Fire* event for each *Temp* event higher than 45 occurred within 5 minutes from the smoke detection (if any). We say that the *each-within* operator defines a *multiple selection* policy as it uses any available *Temp* notification in a given time window. The second rule, instead, creates a single *Fire* notification by selecting only the latest *Temp* event higher than 45 occurred within 5 minutes from the smoke detection. The *first-within* operator exhibits a similar behavior, by selecting the first event in the specified time interval. We say that the *last-within* and *first-within* operators define a *single selection* policy as they force the selection of at most one element. The formal definitions of all event composition operators are shown below (for space reasons we omit the attribute constraints and assignments):

$$
\begin{array}{l}
define\ CE\ from\ A\ and\ each\ B\ within\ x\ from\ A \ \triangleq \\
Occurs(CE, lab(r, \{l_0, l_1\})) \leftrightarrow \\
(Occurs(A, l_0) \ \wedge \ WithinP(Occurs(B, l_1), Time(l_0), x))
\end{array}
$$

$$
\begin{array}{l}
define\ CE\ from\ A\ and\ last\ B\ within\ x\ from\ A \ \triangleq \\
Occurs(CE, lab(r, \{l_0, l_1\})) \leftrightarrow \\
(Occurs(A, l_0) \ \wedge \ WithinP(Occurs(B, l_1), Time(l_0), x) \\
\wedge \ \neg \exists t \in (Time(l_1), Time(l_0)]\ Past(Occurs(B, l_2), t) \\
\wedge \ (\neg Past(Occurs(B, l_3), Time(l_1)) \ \wedge \ l_3 > l_1))
\end{array}
$$

$$
\begin{aligned}
&define\ CE\ from\ A\ and\ first\ B\ within\ x\ from\ A \triangleq \\
&Occurs(CE, lab(r, \{l_0, l_1\})) \leftrightarrow \\
&(Occurs(A, l_0) \wedge WithinP(Occurs(B, l_1), Time(l_0), x) \\
&\wedge \neg\exists t \in [x, Time(l_1))\ Past(Occurs(B, l_2), t) \\
&\wedge (\neg Past(Occurs(B, l_3), Time(l_1)) \wedge l_3 < l_1))
\end{aligned}
$$

Here we used the time dependent function $Time$, which takes a label as argument and returns the occurrence time of the event having that label with respect to the current time, left implicit. Using such function, the definition of the *each-within* operator is straightforward: it only adopts the *WithinP* temporal operator, binding *CE* notifications to each B event found in the valid time interval. The definition of *last-within* and *first-within* operators introduce an additional constraint imposing no B events to occur after (resp. before) the selected one in the defined time interval. Notice that to provide a unique order between events in presence of simultaneous occurrences, we assume an ordering between labels.

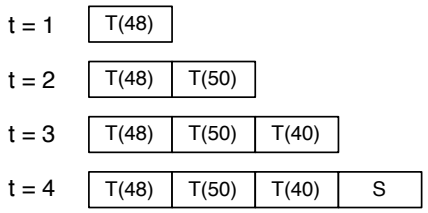


Figure 3.3: A possible history of event occurrences

To better understand the difference between single and multiple selection operators, consider again the two TESLA rules described above. We show an example of event history in Figure 3.3. `S` represents a *Smoke* event, while `T(n)` represents an event of type *Temp* (n being the measured temperature). When at `t=4` a *Smoke* event is detected, the rule that uses the *each-within* operator combines it with every *Temp* event greater than 45, which result in two different *Fire* notifications. On the contrary, the rule that uses the *last-within* operator results in a single *Fire* notification, that combining the *Smoke* event with the latest *Temp* event greater than 45 (i.e. the one received at time `t=2`).

TESLA also offers a generalized version of the *first-within* and *last-within* operators, called *k-first-within* and *k-last-within*. They can be used to capture the second, third, etc. event occurrence from the beginning (resp. end) of a specified interval. The formal definition of the

semantics of these operators is omitted as it can be easily derived from the definition of the basic *single selection* operators.

As a final remark, notice that TESLA allows the definition of rules that combine multiple composition operators; they can be connected in series, defining chains of event occurrences; or in parallel, allowing more event occurrences to be bound to a single one. The rule below shows both options:

$$
\begin{array}{ll}
define & D() \\
from & A() \; and \; each \; B() \; within \; 5min \; from \; A \; and \\
 & last \; C() \; within \; 3min \; from \; A \; and \\
 & last \; D() \; within \; 6min \; from \; B \; and \\
 & first \; E() \; within \; 2min \; from \; D \; and \\
 & E \; within \; 8min \; from \; A
\end{array}
$$

Notice, in particular, the use of the *within* operator in the last line, which introduces an additional timing constraint for an already defined event E.

Parameterization

In Section 3.2 we introduced parameterization as one of the required features in an event specification language. As an example, consider again the rule about *Fire* notifications as defined through the *each-within* operator in the previous section. Knowing that a *Smoke* and a *Temp* events occurred within 5 minutes may be meaningless if we don't know whether the two events come from the same area. To express similar relationships, TESLA introduces parameters through the *$* operator. Suppose that both *Temp* and *Smoke* events have an attribute called *Area*: the following rule exemplifies the use of parameters to force the two events of interest to come from the same area:

$$
\begin{array}{ll}
define & Fire(Val) \\
from & Smoke(Area = \$x) \; and \\
 & each \; Temp(Val > 45 \; and \; Area = \$x) \\
 & within \; 5min \; from \; Smoke \\
where & Val = Temp.Val
\end{array}
$$

Defining the semantics of parameters in TRIO is straightforward, we simply add one or more constraints on the values of attributes. For

example, the rule above requires $attVal(l_0,$
$Area) = attVal(l_1, Area)$ where l_0, l_1 are the labels of the selected *Smoke* and *Temp* events.

Theorem: Uniqueness of selection

As mentioned at the very beginning of this section, all TRIO formulas used so far are correct under the assumption that a set of events can be selected by a given rule only once. We call this assumption: *uniqueness of selection*. Together with the adoption of an injective function to define labels, it makes it impossible to generate different events sharing the same label. As the operators described so far define all the event selection strategies allowed in TESLA, we are now ready to prove that the uniqueness of selection assumption is satisfied. In the remainder of the paper we present new operators that extend the expressiveness of TESLA. None of them, however, introduces new selection mechanisms: at most they add new constraints, reducing the set of composite events that may occur. Accordingly, none of them influences the uniqueness of selection.

Proof. All TESLA rules joins the occurrence of a (composite) event to the occurrence of a pattern of (simpler) events, one of which must occur at the same time of the composite one, while the others occur in the past. This guarantees that a given rule r is satisfied by a set of events E only once, at time t. In fact, future evaluations of the same rule r at time $t_1 > t$ would require at least a new event e to occur at evaluation time t_1. On the other hand, since rules only refer to current and past events, e cannot be part of E, so the pattern of events satisfying r at t_1 must differ from E.

Timers

Several application domains require rules to be evaluated periodically. TESLA supports periodic rules using special events called *timers*. As an example, we may require a rule to be evaluated only at 9.00 of Friday by using $Timer(H = 9, M = 00, D = Friday)$ in its *from* clause. This approach keeps the syntax of the language simple, using a uniform approach for both reactive and periodic rules, and it does not change the semantics of rules (without impacting the uniqueness of selection property).

If we look back at our functional model in Section 2.3.1, we can map timers on the elements produced by the *clock* logical component to enable periodic processing.

Negation

Applications often need to reason not only about the events occurred, but also about those that did not occur. As an example, we could detect a fire when *Temp* and *Smoke* events are detected in the same area in absence of *Rain*. To deal with similar cases, TESLA introduces the *not* operator, which defines an interval of time in which a given event must not occur. Such interval can be defined in two ways: using two events as the interval bounds or using a single event together with the duration of the interval. The following rules introduce the two cases:

$$
\begin{array}{ll}
define & Fire(Val) \\
from & Smoke(Area = \$x)\ and \\
 & each\ Temp(Val > 45\ and\ Area = \$x) \\
 & within\ 5min\ from\ Smoke\ and \\
 & not\ Rain(Area = \$x)\ between\ Temp\ and\ Smoke \\
where & Val = Temp.Val
\end{array}
$$

$$
\begin{array}{ll}
define & Fire(Val) \\
from & Smoke(Area = \$x)\ and \\
 & each\ Temp(Val > 45\ and\ Area = \$x) \\
 & within\ 5min\ from\ Smoke\ and \\
 & not\ Rain(Area = \$x)\ within\ 5min\ from\ Smoke \\
where & Val = Temp.Val
\end{array}
$$

The semantics of these two forms that the *not* operator may assume is defined below. Notice that the first syntax is allowed only when the relative order between the two events defining the interval of time is known. This happens when both events belong to a common chain of event occurrences.

$$
\begin{array}{l}
define\ D\ from\ A\ and\ each\ B\ within\ x\ from\ A \\
and\ not\ C\ between\ B\ and\ A \ \triangleq \\
Occurs(D, lab(r, l_0, l_1)) \leftrightarrow \\
(Occurs(A, l_0) \wedge WithinP(Occurs(B, l_1), x) \wedge \\
\neg \exists t \in [Time(l_0), Time(l_1))\ (Past(Occurs(C, l_2)), t))
\end{array}
$$

$$
\begin{aligned}
&C\ when\ A\ and\ not\ B\ within\ x\ from\ A \triangleq \\
&Occurs(c, lab(r, l_0)) \leftrightarrow (Occurs(A, l_0) \wedge \\
&\neg\exists t \in [Time(l_0), Time(l_0) + x)\ (Past(Occurs(B, l_1)), t))
\end{aligned}
$$

Event consumption

As discussed in Section 3.2, one of the main limitations of existing CEP languages is the lack of customizable event selection and consumption policies. While the *xxx-within* operators introduced so far allow users to adopt the preferred event selection policy, TESLA uses the *consuming* clause to deal with event consumption, allowing users to specify the selected events that have to become invalid for future detections (by the same rule). As an example, consider the following rule:

$$
\begin{aligned}
&define && Fire(Val) \\
&from && Smoke()\ and\ each\ Temp(Val > 45) \\
& && within\ 5min\ from\ Smoke \\
&where && Val = Temp.Value \\
&consuming && Temp
\end{aligned}
$$

It consumes all selected *Temp* events, so a new *Smoke* would not fire the rule until a new *Temp* (followed by a further *Smoke*) happens.

To define the semantics of rules that include the *consuming* clause, we introduce a new time dependent TRIO predicate called *Consumed*. Given a rule identifier r and a label l, *Consumed(r,l)* remains false until the event with label l is consumed by the rule r, and it holds true after event consumption. Formally, we ask the *Consumed* operator to satisfy the following properties:

$$
\begin{aligned}
&Alw\ \forall l \in L, \forall r \in \mathbb{N} \\
&(Consumed(r, l) \rightarrow \forall\ t > 0, Futr(Consumed(r, l), t))
\end{aligned}
$$

$$
\begin{aligned}
&Alw\ \forall l \in L, \forall e, r \in \mathbb{N}, \forall S \\
&((\neg\exists t > 0\ (Past(Occurs(e, lab(r, S), t))) \wedge\ l \in S) \\
&\rightarrow\ \neg Consumed(r, l))
\end{aligned}
$$

The former guarantees that once an event with label l has been consumed by a rule r, it always remains consumed in the future (w.r.t. r). The latter guarantees that if an event e has not (yet) been captured by a rule r (i.e. it is not part of a set of labels S selected by the rule in the past), it cannot be considered as consumed w.r.t. r. To show how the *Consumed* predicate can be used to formalize the semantics of rules including a *consuming* clause, we provide the translation of the TESLA rule above in TRIO:

$$Occurs(Fire, lab(r, \{l_1, l_2\})) \leftrightarrow (Occurs(Smoke, l_1) \\ \wedge\ WithinP(Occurs(Temp, l_2), Time(l_1), 5) \wedge \\ attVal(l_2, Value) > 45\ \wedge\ \neg Consumed(r, l_2))$$

$$(Occurs(Smoke, l_1) \wedge \\ WithinP(Occurs(Temp, l_2), Time(l_1), 5) \wedge \\ attVal(l_2, Value) > 45\ \wedge\ \neg Consumed(r, l_2))\ \rightarrow \\ \forall t > 0\ Futr(Consumed(r, l_2), t)$$

The first formula differs from the standard translation used so far as it requires all events appearing in the *consuming* clause (i.e. *Temp*) not to be consumed at evaluation time. The second specifies that, if at time t the pattern is satisfied selecting the *Temp* event with label l_2, such event has to be considered consumed in the future. Using the *consuming* clause together with single selection operators it is possible to define rules that always capture a specific event (e.g. the first, or the last) among non consumed ones only. To the best of our knowledge no other languages based on patterns are expressive enough to define similar rules. Also notice that event consumption is valid only within a rule. We think that this semantics is the most natural for a CEP system, in which multiple rules use the same event notifications independently, possibly with different aims. Defining a *global consume* operator would be straightforward, however we preferred not to introduce it as we think that it would have made rule definition and management a harder tasks.

Aggregates

The importance of aggregates has been discussed in Section 3.2. Aggregates apply a function to a set of values S to generate a new value v. TESLA allows v to be used wherever a value is allowed; in particular v

can be assigned to an attribute of the composite event being defined or it can be used inside the constraints that select the relevant events. TESLA aggregates capture values from events in a specified time interval. As for negations, time intervals can be specified through the occurrence of two events or through a single occurrence plus the duration of the interval. The following examples show the two cases:

$$
\begin{array}{ll}
define & AvgTemp(Val) \\
from & Timer(M\%5 == 0) \\
where & Val = Avg(Temp().Value) \\
 & within\ 5min\ from\ Timer
\end{array}
$$

$$
\begin{array}{ll}
define & HighVal(Name, Val) \\
from & Stock(Name = \$y, Val = \$x)\ and \\
 & last\ Opening()\ within\ 1day\ from\ Stock\ and \\
 & \$x > Avg(Stock(Name = \$y).Val) \\
 & between\ Opening\ and\ Stock \\
where & Val = S.Val,\ Name = S.Name
\end{array}
$$

The first rule is evaluated periodically (every 5 minutes) and generates an *AvgTemp* notification embedding the average value of temperature readings in the last 5 minutes. The second rule generates a new *HighVal* notification when the value of a *Stock* overcomes the average value computed from the last *Opening*.

The following TRIO formula provides the semantics for a generic aggregate function *Fun* applied to the attribute *Val* of events X occurred between A and B. The formula defines a *Set* including all values of attribute *Val* in events of type X occurred between A and B. Formally *Set* is defined as a set of label-value couples, in such a way that the same value coming from n different events is considered n times. The function *Fun* uses values in *Set* to produce the result.

$$
\begin{array}{l}
Fun(X.Val)\ between\ A\ and\ B\ =\ Y\ \triangleq \\
\forall\ Set\ (\forall x(x \in Set\ \leftrightarrow \exists l \in L(x =< l, attVal(l, Val) > \\
\wedge\ withinP(Occurs(X, l), Time(B), Time(A)))) \\
\rightarrow\ Fun(Set) = Y)
\end{array}
$$

Recursive processing

Most languages for CEP do not provide a separation between event definition rules and users subscriptions; subscribers deploy so called *composite subscriptions* that embed the pattern of events they are interested in. While this approach has no impact on the expressiveness of the language, in our opinion it makes definition of rules more difficult, as it does not allow composite events to be reused.

On the contrary, TESLA enables composite events defined through a rule to be used inside other rules thus realizing *recursive processing*; this way users may easily create very expressive hierarchies of events.

We think that this approach better fits the nature of many applications in which sources provide a high volume of low level information items which need to be filtered and combined at different logical levels to produce results for the final users. As an example, consider a weather forecast application: sensors provide information about their locations and the temperature they read. As a first step, a rule manager may define an average temperature event that is generated every 5 minutes and include all the readings coming from a given area. Then, these events can be combined into patterns defining temperature trends. Finally, trends can be used together with other data (e.g. about wind) to provide weather forecast.

Notice that a correct definition of hierarchies require no circular dependencies to exist between rules. This constraint cannot be verified directly inside the TESLA language, as it requires knowledge that is outside the scope of single rules, however it can be easily checked by a CEP engine at rule deploy time.

Iterations

We have already seen in Chapter 2 that several existing languages for CEP define ad-hoc operators to express bounded or unbounded iterations of patterns (Kleen closures) [194, 98, 46].

Such operators may be useful to detect trends: as an example they enable patterns involving continuously increasing values for a stock. A precise definition of iteration operators has to take into account a number of aspects: selection and consumption policies, minimum and maximum number of iterations, relations between attribute values, termination criteria, etc., which complicate the language, both syntactically and semantically.

On the other hand TESLA, with its ability to define hierarchies of events and to adopt different selection and consumption policies for dif-

ferent rules, is expressive enough to not require special operators to capture iterations. A great advantage in term of elegance and simplicity. This is in line with our analysis in Section 2.3.8, where we show that a language that provides both a sequence operator and recursive processing can mimic iterations through recursive rules.

As an example, suppose we want to capture every iteration of events of type A, where the attribute Val never decreases, to notify an event B that contains the number of A that are part of the iteration. This is captured by the following TESLA rules:

$$
\begin{array}{ll}
define & RepA(Times, Val) \\
from & A() \\
where & Times = 1 \; and \; Val = A.Val
\end{array}
$$

$$
\begin{array}{ll}
define & RepA(Times, Val) \\
from & A(\$x) \; and \; last \; RepA(Val \leq \$x) \; within \; 3min \\
 & from \; A \\
where & Times = RepA.Times + 1 \; and \; Val = \$x \\
consuming & RepA
\end{array}
$$

$$
\begin{array}{ll}
define & B(Times) \\
from & RepA() \\
where & Times = RepA.Times
\end{array}
$$

Notice how TESLA provides a high degree of flexibility. For example, it is easy to modify the event selection and consumption policies or content and timing constraints in the above rules to change the events actually captured, e.g., to capture only the longest iteration occurred.

3.5 Conclusions

In this chapter we presented TESLA, our event specification language for CEP. We explicitly designed TESLA to capture the needs of event based applications, providing a simple and compact syntax while offering high expressiveness and floxibility.

TESLA supports content-based event filtering and allows to easily capture complex relations among temporally related patterns of events. It supports parameterization, negations, and aggregates, offering standard

and periodic rules within a single framework. It also clearly separates the role of rules, used to define composite events, from the role of subscriptions, used to express the interests of sinks, allowing recursive definition of rules to easily specify expressive hierarchies of events. All these features, together with the ability of specifying fully customizable policies for event selection and consumption, allows TESLA to easily define event iterations without requiring an explicit Kleene operator, i.e., keeping the language syntax simple and elegant.

Moreover, TESLA is among the first languages for CEP to offer a formal semantics, expressed using a temporal logic. This eliminates the ambiguities typical of most other languages and allows system designers to formally check the correctness of their implementation.

Since an event detection language is pointless without an engine interpreting it, in the next chapters we present the design of different algorithm for efficient event detection using TESLA rules and their implementation on a running system.

4 Content-Based Matching on Parallel Hardware

4.1 Introduction

As highlighted in Chapter 2, selection is the fundamental operator for CEP. Indeed, selecting which primitive events are relevant for a composite one is the first step that every CEP system has to perform. This is done by *matching* the *content* of primitive events against some kind of filter present in rules.

Moreover, the same kind of content-based matching has also to be performed at the end of the CEP process, to decide the sinks interested in the composite events just identified. As already shown in Chapter 2, this is the same action at the base of publish-subscribe systems. For this reason, publish-subscribe is the area in which the content-based matching problem has been studied in details. Accordingly, in this chapter we will compare with the results achieved in this field, using the terminology developed for the research on publish-subscribe systems.

The content-based matching process is not a trivial activity [10], and in many situations the matching component can easily become the bottleneck of a publish-subscribe system. On the other hand, in several scenarios the performance of the publish-subscribe infrastructure may be a key factor. As an example, in financial applications for high-frequency trading [197], a faster delivery event notifications may produce a significant advantage over competitors. Similarly, in intrusion detection systems [198], the ability to forward the huge number of event notifications that results from observing the operation of a large network is fundamental to detect possible attacks before they could compromise the system.

This aspect has been clearly identified in the past and several algorithms have been proposed for efficient content-based matching [199, 200, 26, 201]. While these algorithms differ in most aspects, they have one in common: they were all designed to run on conventional, sequential hardware. If this was reasonable years ago, when parallel hardware was the exception, today this is no more the case. Modern CPUs integrate multiple cores and more will be available in the immediate future. If

this was not enough, modern Graphical Processing Units (GPUs) integrate hundreds of cores, general enough to be used not only for graphic processing but for processing in general.

Unfortunately, parallel programming is still a complex task and traditional algorithms have often to be fully redesigned to leverage parallel hardware. This is definitely true in the case of GPUs, whose cores can be used simultaneously only to perform data parallel computations; but multicore CPUs have their peculiarities, too. As an example, the different cores of a multicore CPU share a single cache and bus toward the main memory, making memory management critical. This task is also critical in GPUs, which usually have no direct access to the main memory and do not offer hardware managed caches, leaving all the duties in the hand of programmers.

Moving from these premises, in this chapter we present *Parallel Content Matching* (*PCM*), a data parallel algorithm explicitly designed to leverage off-the-shelf parallel hardware (i.e., multicore CPUs and GPUs) to perform content-based matching. We present two incarnations of this algorithm: one for multicore CPUs using OpenMP [202], the other for GPUs that implement the CUDA architecture [203]. We study their performance comparing them against SFF [200], the (sequential) matching component of Siena, often taken as a benchmark for its efficiency. This analysis demonstrates how the use of parallel hardware can bring impressive speedups in content-based matching. At the same time, by carefully analyzing how PCM performs under different workloads, we also identify the peculiar aspects of parallel CPU and GPU programming that mostly impact performance.

The remainder of the chapter is organized as follow: Section 4.2 introduces the data model and the terminology we use in the rest of the chapter. Section 4.3 offers an overview of the OpenMP and CUDA programming models, focusing on the aspects more relevant for our purpose. Section 4.4 describes PCM and how we implemented it on OpenMP and CUDA. The performance of these two implementations in comparison with SFF is discussed in Section 4.5, while Section 4.6 presents related work. Finally, Section 4.7 provides some conclusive remarks and describes future work.

4.2 Events and Predicates

For ease of exposition we report here the models of events and subscriptions already introduced in Chapter 3. Here, we introduce a more precise terminology (derived from the publish-subscribe research area)

to identify the different components that build events and subscriptions, to simplify the following description and analysis of PCM.

To be as general as possible we are adopting a data model and a terminology that are very common among event-based systems [200]. We represent an *event notification*, or simply *event*, as a set of *attributes*, i.e., $\langle name, value \rangle$ pairs. Values are typed and we consider both numerical values and strings. As an example, $e_1 = [\langle area, \text{“}area1\text{”} \rangle, \langle temp, 25 \rangle, \langle wind, 15 \rangle]$ is an event that an environmental monitoring component could publish to notify the rest of the system about the current temperature and wind speed in the area it monitors.

The interests of components are modeled through *predicates*, each being a disjunction of *filters*, which, in turn, are conjunctions of elementary *constraints* on the values of single attributes. As an example, $f_1 = (area = \text{“}area1\text{”} \wedge temp > 30)$ is a filter composed of two constraints, while $p_1 = [(area = \text{“}area1\text{”} \wedge temp > 30) \vee (area = \text{“}area2\text{”} \wedge wind > 20)]$ is a predicate composed of two filters. A filter f *matches* an event notification e if all constraints in f are satisfied by the attributes of e. Similarly, a predicate matches e if at least one of its filters matches e. In the examples above, predicate p_1 matches event e_1.

Given these definitions, the problem of content-based matching can be stated as follow: given an event e and a set of *interfaces*, each one exposing a predicate, find those interfaces e should be delivered through, i.e., those exposing a predicate matching e.

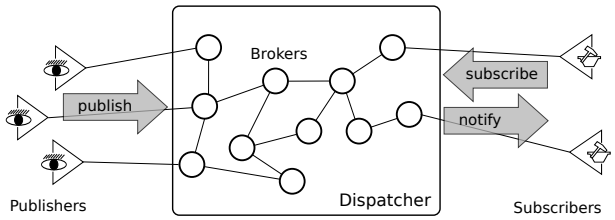


Figure 4.1: A typical publish-subscribe infrastructure

In a centralized publish-subscribe infrastructure, it is the *dispatcher* that implements this function, by collecting predicates that express the interests of subscribers (each connected to a different “interface”) and forwarding incoming event notifications on the basis of such interests. In a distributed publish-subscribe infrastructure the dispatcher is composed of several *brokers*, each one implementing the content-based matching function above to forward events to its neighbors (other brokers or the subscribers directly connected to it).

In the case of traditional publish-subscribe systems, the dispatcher simply forwards the events received from the publishers to interested subscribers. In the case of a CEP system, as we have seen, there may be one or more brokers in charge of using received events to detect patterns and generate notification of composite events accordingly. In this chapter we focus on forwarding only, to isolate its contribution to the overall performance of a system.

4.3 Parallel Programming Models

Attaining good performance with parallel programming is a complex task. A naive paralleling of a sequential algorithm is usually not sufficient to efficiently exploit the presence of multiple processing elements, and a complete re-design of the algorithm may be necessary, taking into account the peculiarity of the underlying architecture and its programming model. In this section we present the architectures and programming models considered in this chapter, focusing on the abstractions offered and on the aspects that mostly affect performance.

4.3.1 Multicore CPU Programming with OpenMP

OpenMP (Open Multi-Processing) is an API for shared memory multiprocessing programming in C/C++ and Fortran, consisting of a set of compiler directives and library routines. OpenMP provides thread programming at a high level: the programmer specifies which portions of code should execute in parallel, while the compiler decides low-level details, including the creation of threads and the assignment of tasks to threads. In particular, to develop data parallel algorithms like PCM, OpenMP supports the SPMD (Single Program Multiple Data) implementation strategy [204], through the following primitives:

Parallel regions. The programmer defines regions of code that have to be executed in parallel by different threads, and explicitly specifies the number of threads to be used. Inside a parallel region, each thread is uniquely identified by its *ThreadNum*, which the programmer may access and use to differentiate the execution flows of threads.

Shared memory. Threads in a parallel region can declare private, thread-local variables, but they can also access variables from a common shared memory. Limiting data dependencies among threads is a key aspect to obtain good performance.

Synchronization. To control the access to shared memory, OpenMP

provides two different kinds of synchronization primitives: *critical sections* and *barriers*. Critical sections specify portions of parallel regions that must be executed in mutual exclusion by the different threads. Barriers are synchronization points at which all threads must wait before any is allowed to proceed [205].

On top of them, OpenMP also offers some higher level primitives. The most remarkable example (also used in the remainder of this chapter) is the *parallel for*, which parallels for loops. Although OpenMP 3.0 introduces new directives to simplify task parallelism, we did not make use of them while implementing our algorithm.

4.3.2 GPU Programming with CUDA

Introduced by Nvidia in Nov. 2006, the CUDA architecture offers a new programming model and instruction set for general purpose programming on GPUs. Different languages can be used to interact with a CUDA compliant device: we adopted CUDA C, a dialect of C explicitly devoted to program GPUs. The CUDA programming model is founded on five key abstractions:

Hierarchical organization of thread groups. The programmer is guided in partitioning a problem into coarse sub-problems to be solved *independently* in parallel by *blocks* of threads, while each sub-problem must be decomposed into finer pieces to be solved *cooperatively* in parallel by all threads within a block. This decomposition allows the algorithm to easily scale with the number of available processor cores, since each block of threads can be scheduled on any of them, in any order, concurrently or sequentially.

Shared memories. As in OpenMP, CUDA threads may access data from multiple memory spaces during their execution: each thread has a *private local memory* for automatic variables; each block has a *shared memory* visible to all threads in the same block; finally, all threads have access to the same *global memory*.

Barrier synchronization. Since thread blocks are required to execute independently from each other, no primitive is offered to synchronize threads of different blocks. On the other hand, threads within a single block work in cooperation, and thus need to synchronize their execution to coordinate memory access. In CUDA this is achieved exclusively through *barriers*.

Separation of host and device. The CUDA programming model assumes that CUDA threads execute on a physically separate *device* (the

GPU), which operates as a coprocessor of a *host* (the CPU) running a C/C++ program. The host and the device maintain their own separate memory spaces. Therefore, before starting a computation, it is necessary to explicitly allocate memory on the device and to copy there the information needed during execution. Similarly, at the end results have to be copied back to the host memory and the device memory have to be deallocated.

Kernels. Like parallel regions in OpenMP, *kernels* are special functions that define a single flow of execution for multiple threads. When calling a kernel k, the programmer specifies the number of threads per block and the number of blocks that must execute it. Inside the kernel it is possible to access two special variables provided by the CUDA runtime: the *threadId* and the *blockId*, which together allow to uniquely identify each thread among those executing the kernel. As for the *ThreadNum* in OpenMP, conditional statement involving these variables can be used to differentiate the execution flows of different threads.

Architectural Issues

If compared to OpenMP, the CUDA model provides thread programming at a lower level. There are details about the hardware architecture that a programmer cannot ignore while designing an algorithm for CUDA.

The CUDA architecture is built around a scalable array of multithreaded *Streaming Multiprocessors* (*SMs*). When a CUDA program on the host CPU invokes a kernel k, the blocks executing k are enumerated and distributed to the available SMs. All threads belonging to the same block execute on the same SM, thus exploiting fast SRAM to implement the shared memory. Multiple blocks may execute concurrently on the same SM as well. As blocks terminate new blocks are launched on freed SMs.

Each SM creates, manages, schedules, and executes threads in groups of parallel threads called *warps*. Individual threads composing a warp start together but they have their own instruction pointer and local state and are therefore free to branch and execute independently. On the other hand, full efficiency is realized only when all threads in a warp agree on their execution path, since CUDA parallels them executing one common instruction at a time. If threads in the same warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

An additional issue is represented by memory accesses. If the layout of data structures allows threads with contiguous ids to access contiguous

memory locations, the hardware can organize the interaction with memory into several memory-wide operations, thus maximizing throughput. This aspect significantly influenced the design of PCM's data structures, as we discuss in the next section.

In summary, we can say that, similarly to OpenMP, the CUDA programming model ease the implementation of data parallel algorithms using a SPMD implementation strategy. However, while OpenMP runs on hardware architectures where different threads are free to execute different instructions without incurring in additional overhead, the CUDA architecture is designed to execute efficiently only data parallel code that operates on contiguous memory regions.

Inside a single SM, instructions are pipelined but, differently from modern CPU cores, they are executed in order, without branch prediction or speculative execution. To maximize the utilization of its computational units, each SM is able to maintain the execution context of several warps on-chip, so that switching from one execution context to another has no cost. At each instruction issue time, a warp scheduler selects a warp that has threads ready to execute (not waiting on a synchronization barrier or for data from the global memory) and issues the next instruction to them.

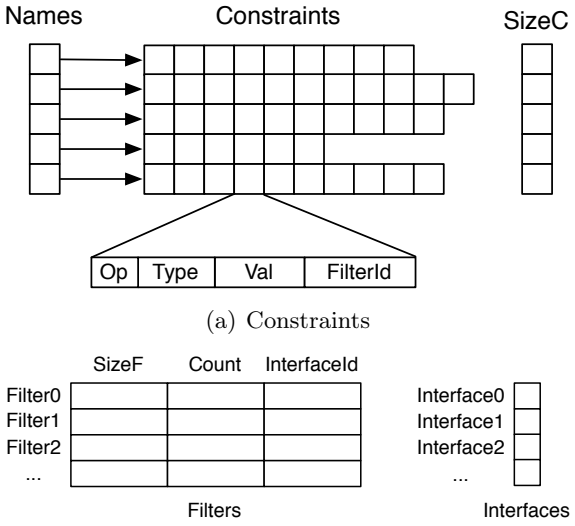
Finally, to give an idea of the capabilities of a modern GPU supporting CUDA, we provide some details of the Nvidia GTX 460 card we used for our tests. It includes 7 SMs, which can handle up to 48 warps of 32 threads each (for a maximum of 1536 threads). Each block may access a maximum amount of 48KB of shared, on-chip memory within each SM. Furthermore, it includes 1GB of GDDR5 memory as global memory. This information must be carefully taken into account when programming: shared memory must be exploited as much as possible, to hide the latency of global memory accesses, but its limited size significantly impacts the design of algorithms.

4.4 The PCM Algorithm

This section describes our PCM algorithm and its implementations in OpenMP (*OCM*) and CUDA (*CCM*). As mentioned in Section 4.1, PCM is designed to maximize the amount of data parallel computations, minimizing the interactions among threads. Moreover, it adopts a memory layout in which information is stored in contiguous areas. This leverages the CUDA ability to combine memory accesses to maximize throughput, but it also proves (see Section 4.5) to provide maximum performance in OpenMP.

PCM is composed of two phases: a *constraint selection* phase and a *constraint evaluation and counting* phase. When an event enters the engine, the first phase selects, for each attribute a of the event, the set of constraints having the same name as a. These constraints are evaluated in the second phase, using the value of a. In particular, when a constraint c is satisfied, we increase the counter associated to the filter c belongs to. A filter f matches an event when all its constraints are satisfied and so does the predicate p it belongs to. When this happens, the event can be forwarded to the interface exposing p.

4.4.1 Data Structures



(a) Constraints

(b) Filters and Interfaces

Figure 4.2: Data structures

Figure 4.2 shows the data structures we create and use during processing. Figure 4.2a shows the data structures containing information about constraints. In particular, Table `Constraints` groups existing constraints into multiple rows, one for each name. Each element of the table stores information about a single constraint: its operator (`Op`), its type (`Type`), its value (`Val`), and an identifier of the filter it belongs to (`FilterId`). If different filters share a common constraint c (which we expect to be common in real applications), we duplicate c. This simplifies memory layout and minimizes the size of each element of table `Constraints`, which, in turn, maximizes the number of elements that can

be accessed in parallel through a memory-wide operation by CUDA. Section 4.5 measures the benefits of this solution. Moreover, since different rows of table `Constraints` may include a different number of elements, the actual size of each row is stored in vector `SizeC`. Finally, map `Names` ($\langle name, rowId \rangle$) associates each attribute name with the corresponding row in `Constraints`.

Figure 4.2b shows the data structures containing information about filters and interfaces. Each row of table `Filters` represents a different filter and stores its size (`SizeF`), i.e., the number of constraints it is composed of, the number of currently satisfied constraints (`Count`), and the interface it belongs to (`InterfaceId`). Vector `Interfaces` represents the output of our algorithm. It is a vector of bytes, one for each interface. A value of one in position x means that the event under processing must be forwarded through interface x. Both `Count` and `Interfaces` are set to zero before processing an event and updated (in parallel) during processing.

In the implementation adopting OpenMP, all data structures are stored in the main (CPU) memory. In the CUDA version almost all of them are permanently stored into the GPU memory; a choice that minimizes the need for CPU-to-GPU communication during event processing. The only exception is represented by map `Names`, stored on the CPU.

4.4.2 Implementing PCM in OpenMP (OCM)

Evaluating constraints in parallel using OpenMP is rather easy. When an event e enters the engine, for each attribute a in e, we use map `Names` to select the row of table `Constraints` associated to the name of a, then we use a *parallel for* to evaluate the constraints part of this row in parallel.

When a constraint c is satisfied, we have to update field `Count` of the filter that includes c. Since different threads may try to update such a field concurrently, we define this update as an *atomic operation*. Atomic operations in OpenMP are a special, low overhead, type of critical sections, which can be applied to variable assignments only. After updating field `Count`, we check whether it has become equal or greater than `SizeF`. In that case we set to one the corresponding element in vector `Interfaces`.

As a final remark, we notice that our algorithm uses the parallel for loop also to reset field `Count` and vector `Interfaces` before processing each event. Indeed, in our preliminary tests this solution proved to significantly outperform not only the sequential for loop but also a call to memset.

4.4.3 Implementing PCM in Cuda (CCM)

The implementation of PCM on CUDA deserves more discussion. As for OpenMP, when an event e enters the engine, it is the CPU that uses map `Names` to select the relevant lines in table `Constraints`. In principle, we could avoid this phase, transferring the whole content of e to the GPU and letting it find the relevant constraints using parallel threads. However, some preliminary experiments we performed show that the computational effort required to select constraints on the CPU using a STL map is negligible (less than 1% of the overall processing time); on the other hand, this selection can potentially filter out a huge number of constraints, thus making the subsequent evaluation on the GPU much faster.

Figure 4.3: Input data

As a result of this first phase, the CPU builds table `Input`, as in Figure 4.3. It stores, for each attribute a in e, the id of the row in `Constraints` associated to the name of a (`RowId`), the type of a (`Type`), and its value (`Val`). This information is transferred to the GPU memory to match e against relevant constraints. This evaluation step is performed entirely on the GPU, launching a kernel that uses thousands of threads working in parallel, each one evaluating a single attribute a of e against a single constraint among those relevant for a in table `Constraints`.

As we mentioned in Section 4.3, at kernel launch time the developer must specify the number of blocks executing the kernel and the number of threads composing each block. Both numbers can be in one, two, or three dimensions. Figure 4.4 shows our organization of threads and blocks. It shows an example in which each block is composed of only 4 threads, but in real cases 256 or 512 threads per block are common choices. We organize all threads inside a block over a single dimension (x axis), whereas blocks are organized in two dimensions. The y axis is mapped to event attributes, i.e., to rows of table `Input` in Figure 4.3. The x axis, instead, is mapped to set of constraints. Since the number of constraints with the same name of a given attribute may exceed the

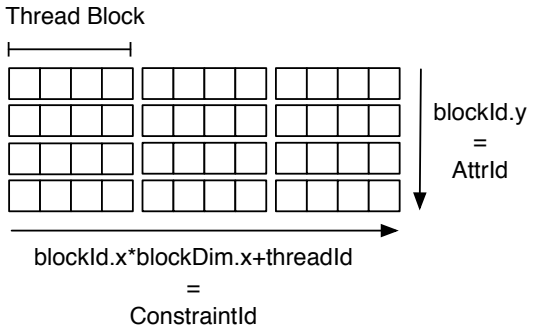


Figure 4.4: Organization of blocks and threads

maximum number of threads per block, we allocate multiple blocks along the x axis.

The kernel function is presented in Algorithm 1. At the first two lines, each thread determines its x and y coordinates in the bi-dimensional space presented in Figure 4.4 using the values of the blockId and threadId variables, initialized by the CUDA runtime. More specifically, the value of y is directly given by the y value of blockId, while the value of x is computed as blockId.x·blockDim.x+threadId, where blockDim.x is the x size of each block.

At this point each thread reads the data it requires from tables `Input` and `Constraints`. Since all threads in the same block share the same attribute, i.e., the same element in table `Input`, we copy such element from the global memory to the block shared memory once for all. More specifically, the command at line 3 defines a variable shInput in shared memory, which the first thread of each block (the one having threadId equal to 0) sets to the appropriate value taken from table `Input`. All other threads wait until the copy is finished by invoking the __syncthreads() command in line 7. Our experiments show that this optimization increases performance by 2-3% w.r.t. the straightforward approach of letting each thread separately access table `Input` from global memory

Afterward, each thread uses the `RowId` information copied into the shInput structure to determine the row of table `Constraints` it has to process (each thread will process a different element of such row). Since different rows may have different lengths, we instantiate the number of blocks (and consequently the number of threads) to cover the longest among them. Accordingly, in most cases we have too many threads. We check this possibility at line 9, immediately stopping unrequired threads.

Algorithm 1 Constraint Evaluation Kernel

```
1:  x = blockId.x·blockDim.x+threadId
2:  y = blockId.y
3:  __shared__ shInput
4:  if threadId==0 then
5:     shInput = input[y]
6:  end if
7:  __syncthreads()
8:  rowId = shInput.RowId
9:  if x ≥ SizeC[rowId] then
10:    return
11: end if
12: constraint = Constraints[x][rowId]
13: type = shInput.Type
14: val = shInput.Val
15: if ! sat(constraint, val, type) then
16:    return
17: end if
18: filterId = constraint.FilterId
19: count = atomicInc(Filters[filterId].Count)
20: if count+1==Filters[filterId].SizeF then
21:    interfaceId = Filters[filterId].InterfaceId
22:    Interfaces[interfaceId] = 1
23: end if
```

This is a common practice in GPU programming, e.g., see [206]. We will analyze its implication on performance in Section 4.5.

In line 12 each thread reads the constraint it has to process from table `Constraints` in global memory to the thread's local memory (i.e., hardware registers, if they are large enough), thus making future accesses faster. Also notice that our organization of memory allows threads having contiguous identifiers to access contiguous regions of the global memory. This is particularly important when designing an algorithm for CUDA, since it allows the hardware to combine different read/write operations into a reduced number of memory-wide accesses, thus increasing performance.

In lines 13 and 14 each thread reads the type and value of the attribute it has to process and uses them to evaluate the constraint it is responsible for (in line 15). We omit for simplicity the pseudo code of the sat

function that checks whether a constraint is satisfied by an attribute. If the constraint is not satisfied the thread immediately returns, otherwise it extracts the identifier of the filter the constraint belongs to and updates the value of field `Count` in table `Filters`. As already observed for OpenMP, different threads may try to update the same counter concurrently. To avoid clashes, we exploit a special atomicInc operation offered by CUDA, which atomically reads the value of a 32bit integer from the global memory, increases it, and returns the old value. In line 20 each thread checks whether the filter is satisfied, i.e., if the current number of satisfied constraints (old count plus one) equals the number of constraints in the filter. If this happens, the thread extracts the identifier of the interface the filter belongs to and sets the corresponding position in vector `Interfaces` to 1.

As we mentioned in Section 4.3, CUDA provides best performance when threads belonging to the same warp follow the same execution path. After table `Input` is read (line 5) and all unrequired threads have been stopped (line 10), there are two conditional branches where the execution path of different threads may potentially diverge. The first one, in line 15, evaluates a single attribute against a constraint, while the second one, in line 20, checks whether all the constraints in a filter have been satisfied before setting the relevant interface to 1. The threads that follow the positive branch in line 20 are those that process the last matching constraint of a filter. Unfortunately, we cannot control the warps these threads belong to, since this depends from the content of the event under evaluation and from the scheduling of threads. Accordingly, there is nothing we can do to force threads on the same warp to follow the same branch. On the contrary, we can increase the probability of following the same execution path within function sat in line 15 by grouping constraints in table `Constraints` according to their type, operator, and value. This way we increase the chance that threads in the same warp, having contiguous identifiers, process constraints with the same type and operator, thus following the same execution path into sat. Our experiments, however, show that such type of grouping of constraints provides a very marginal performance improvement and only under specific conditions. On the other hand, it makes creation of data structures (i.e., table `Constraints` that needs be ordered) much slower. We will come back to this issue in Section 4.5.

Reducing Memory Transfers

To correctly process an event, we first need to reset fields `Count` and vector `Interfaces`. In a preliminary implementation, we let the CPU

perform these operations through the cudaMemset command, which allows to set all bytes of a memory region on the GPU to a common value (0 in our case). Although optimized, the cudaMemset command introduces a relevant overhead, since it involves a communication between the CPU and the GPU over the (slow) PCI-E bus. On the other hand, since the CUDA architecture executes blocks in non deterministic order and does not allow inter-blocks communication, we could not straightforwardly reset data structures within our kernel. Indeed, assuming a subset of threads is used to reset data immediately after launch, we cannot know in advance whether they are executed immediately, or if other threads (in other blocks) get executed before.

To overcome this problem, we decided to duplicate data structures `Count` and `Interfaces`. When processing an event we use one copy of these data structures and we reset the other using different threads in parallel, while we reverse the role of the two copies at the next event. This way we avoid a call to cudaMemset, and we reset data structures within the same kernel that executes the matching algorithm, reducing the interactions between CPU and GPU memory to the minimum. We only copy the event content to the GPU (filling table `Input`) and copy back vector `Interfaces` to the CPU when computation is finished. Next section analyzes the benefits of this approach in details.

4.5 Evaluation

Our evaluation had several goals. First, we wanted to compare PCM with a state of the art sequential algorithm, to understand the real benefits in parallelizing the matching process, both on CPUs and on GPUs. Second, we wanted to analyze our code, to better understand the implementation choices that mostly impact on performance. Third, since the performance of matching are influenced by a large number of parameters, we wanted to explore the parameter space as broadly as possible, to isolate the aspects that make the use of a particular algorithm or hardware architecture more profitable. Finally, we wanted to test the behavior of our solutions when deployed on a complete system.

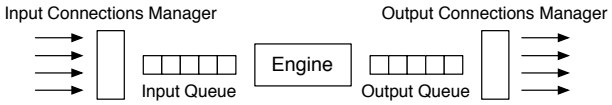


Figure 4.5: Architecture of a matching system

To this extent, we considered (and implemented) a generic system ar-

chitecture, as shown in Figure 4.5. It includes three main components: the `Input Connections Manager` handles the connections with sources. It receives events from the network, as streams of bytes, decoding and storing them into the `Input Queue`. The `Engine` only executes the matching algorithm: it picks up events from the `Input Queue`, computes the set of destinations they have to be delivered to, and stores the results in the `Output Queue`. Finally, the `Output Connections Manager` handles the connections with the sinks, by reading events from the `Output Queue`, serializing them and delivering them to interested sinks.

The rest of the section is organized into three main parts. We first concentrate on the `Engine` and analyze its latency in processing a single event, under various workloads. During these tests we also monitor the time required for installing subscriptions, and the memory usage required by the different implementations. Second, we consider the case in which several events are stored in the `Input Queue` ready to be analyzed, and we measure the benefits of processing them in parallel, if any. Finally, we study the behavior of the system as a whole, using a remote client as an event generator[1].

As a reference sequential algorithm, we used SFF [200] (v. 1.9.4), the matching algorithm used inside the Siena event notification middleware, which is known in the community for its performance. Similarly to PCM, SFF is a counting algorithm, which runs over the attributes of the event under consideration, counting the constraints they satisfy until one or more filters have been entirely matched. When a filter f is matched, the algorithm marks the related interface, purges all the constraints and filters exposed by that interface, and continues until all interfaces are marked or all attributes have been processed. The set of marked interfaces represents the output of SFF. To maximize performance under a sequential hardware, SFF builds a complex, strongly indexed data structure, which puts together the predicates (decomposed into their constituent constraints) received by subscribers. A smart use of hashing functions and pruning techniques, allows SFF to obtain state-of-the-art performance under a very broad range of workloads.

All the tests of this section were executed on a AMD Phenom II machine, with 6 cores running at 2.8GHz, and 8GB of DDR3 Ram. To compile OpenMP we used the GCC compiler, version 4.4.5. The GPU was a Nvidia GTX 460 with 1GB of GDDR5 Ram. We used the CUDA runtime 4.0 for 64 bit Linux. Nowadays both the CPU and the GPU we adopted are considered mid-low level hardware. On the other hand,

[1] All the code used during our evaluation is available for download from http://cudamatcher.sf.net

Number of events	1000
Attr. per event, min-max	3-5
Number of interf.	10
Constr. per filt., min-max	3-5
Filt. per interf., min-max	22500-27500
Number of names	100
Distribution of names	Uniform
Numerical/string constr.	50% / 50%
Numerical operators	$=$(25%),$\neq$(25%),$>$(25%),$<$(25%)
String operators	$=$(25%),$\neq$(25%),$subStr$(25%),$prefix$(25%)
Number of values	100

Table 4.1: Parameters in the Default Scenario

they were top level one year ago, and they have a similar price, so the comparison is fair.

4.5.1 Latency of Matching

To evaluate the latency of the matching process, we defined a default scenario, whose parameters are listed in Table 4.1, and used it as a starting point to build a number of different experiments, by changing such parameters one by one and measuring how this impacts the performance of CCM, OCM, and SFF. In our tests we let each algorithm process 1000 events, one by one, and we calculate the average processing time. To avoid any bias, we repeated all tests (including those reported in Section 4.5.2 and 4.5.3) 10 times, using different seeds to randomly generate subscriptions and events, and we plot the average value measured. The 95% confidence interval of this average was always below 1% of the measured value, so we omitted it from all the plots.

Default Scenario

Table 4.2 (first row) shows the processing times measured by the algorithms under analysis in the default scenario. This is a relatively easy-to-manage scenario. It includes one million constraints on the average, which is not a huge number for large scale applications. Under this load, SFF requires slightly more than 1ms to process a single event, while OCM requires 0.21ms, and CCM 0.139ms, providing a speedup of respectively 4.92x and 7.45x. This shows the benefit of a parallel algorithm w.r.t. a sequential one, when multiple processing units are available.

To see how the same algorithm performs when only one processing unit is available, we run OCM allocating a single processing thread to

	CCM	OCM	SFF
Processing time - Default scenario)	0.139ms	0.21ms	1.035ms
Subscriptions deployment time - Default scenario	527.5ms	483.22ms	992.28ms
Data structure size - Default scenario	33.9MB (GPU)	33.9MB (CPU)	42.4MB (CPU)
Processing time - Zipf distribution of names	2.06ms	5.15ms	14.68ms

Table 4.2: Analysis of Matching Algorithms

OpenMP. Although OCM does not make use of advanced indexing techniques, as SFF does, it shows similar sequential performance, with an average processing time of 1.21ms. This also indicates that OCM scales very well when increasing the number of cores, with almost 6x speedup with 6 cores.

Deployment of Subscriptions

Besides measuring processing time, we were also interested in studying the time required to create the data structures used during event evaluation. Indeed, they need to be generated at run-time, when new subscriptions are deployed on the engine. Even if it is common to assume that the number of publishing largely exceeds the number of subscribing/unsubscribing in any event-based application, this time may become relevant in some scenarios, and consequently deserves to be analyzed. Table 4.2 (second row) shows the average time required by SFF, OCM, and CCM to create their data structures. To attain its performance under sequential hardware, SFF needs complex structures, which require about 1s to be built in our default scenario. On the contrary, CCM and OCM adopt simpler data structures: this allows OCM to be twice as fast in creating them, with an overall deployment time of less than 500ms. When using CCM, most data structures have to be installed on the GPU memory. Moving data from the CPU to the GPU memory may introduce non negligible delays, caused by the (relatively) limited PCI-Ex performance. In particular, we observed that latency is the most limiting factor w.r.t. bandwidth, accordingly CCM builds all data structures on the CPU memory, and transfer them to the GPU using a single copy. With this solution CCM is less than 50ms slower than OCM.

While Table 4.2 reports the results for the default scenario only, the considerations above are true in all scenarios we tested. OCM is always

at least twice as fast as SFF, while the overhead to copy data on the GPU in a single chunk is marginal.

The complexity of the data structures used by SFF also reflects on their size. As shown in Table 4.2 (third row), OCM and CCM require less memory: the default scenario occupies 33.9MB vs 42.4MB. It is worth mentioning that the maximum occupancy of GPU memory we measured in our tests was below 200MB. Since GPUs nowadays have at least 1GB of Ram, contrary to what happens in other domains, GPU memory occupancy is not a problem for content-based matching.

An Analysis of CCM Processing Times

Figure 4.6 analyzes the cost of the different operations performed by CCM during the matching process in the default scenario. In particular, it splits processing time into three parts: the time required to copy data from the CPU to the GPU memory, the time required to execute the kernel, and the time used to copy results back to the main memory. We compare two different versions of our algorithm: as described in Section 4.4.3, `Memset` resets field `Count` and vector `Interfaces` invoking the cudaMemset command twice, before kernel starts (this time is counted into the "copy data" step); on the contrary, `Kernel` resets data structures directly inside the kernel code.

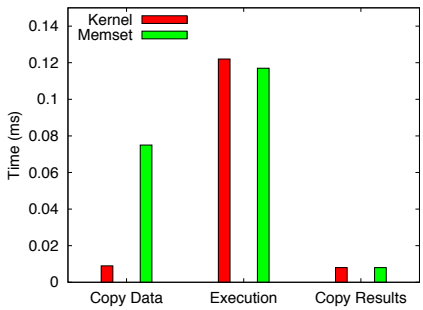


Figure 4.6: An Analysis of CCM Processing Times

First of all, we observe that in both cases the cost of executing the kernel dominates the others. If we compare the two versions of CCM, we notice how the kernel's execution time is higher when data structures are reset inside the kernel but by a very low margin (moving from 0.117ms to 0.122ms). On the other hand, the cudaMemset operation is quite inefficient and avoiding it strongly reduces the execution time of the first step of the algorithm (from 0.075ms to 0.009ms), thus making the `Kernel` version about 30% faster. In the `Kernel` version, which we will use in

the remainder of this section, the cost for moving data back and forth over the (slow) PCI-E bus represents about 12% of the overall matching time, meaning that we are exploiting the computational power of the GPU while introducing a relatively small overhead.

Distribution of Names

The distribution of the names adopted inside constraints is an important aspect for CCM. Indeed CCM creates and launches a number of threads that depends from the size of the longest row in `Constraints` among those selected by the names appearing in the incoming event. In presence of rows with very different sizes, the number of unrequired threads may be relevant. To investigate the impact of this aspect, we changed the distribution of names for both constraints and attributes, moving from the uniform distribution adopted in the default scenario to a Zipf distribution. Table 4.2 (fourth row) shows the results we obtained. First of all we notice how all algorithms significantly increase their matching time w.r.t. the default scenario. Indeed, they all use names to reduce the number of constraints to process, and this pre-filtering becomes less effective with a Zipf distribution. In this scenario, the number of interfaces selected by each single event becomes much higher with respect to the reference scenario. In many cases an event selects all available interfaces. This advantages SFF, which prunes already selected interfaces and stops as soon as all interfaces are selected. This explains why the speedup of OCM over SFF moves from 4.92x in the default scenarios to 2.85x. On the other hand, the speedup of CCM against SFF remains unchanged, meaning that launching a higher number of unrequired threads has a minimal impact on CCM.

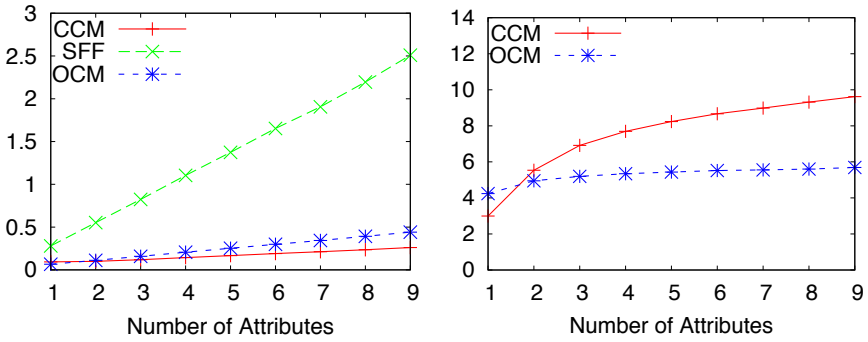
During our evaluation of CCM we also investigated the benefits of ordering constraints according to their type, operator, and value, thus increasing the probability that threads of the same warp follow a common execution path inside the function sat. In the general case, this approach does not significantly decrease the processing time of events, while it has a very negative impact on the cost of deploying subscriptions. Accordingly, we decided not using it. However, when using a Zipf distribution, a large number of constraints share a very limited set of names. In this particular case ordering constraints introduces a measurable speedup moving the overall processing time from 2.06ms to 1.92ms. We can conclude that the idea of ordering constraints is worth in all those cases when the number of constraints with the same name grows above a few hundred thousands (in the Zipf case we have 500.000 constraints with the most common name, on average).

Removing Duplicate Constraints

As mentioned in Section 4.4, SFF stores and processes each constraint only once, even when it belongs to different filters, while PCM duplicates them, thus keeping its data structures simple and effective to access, allowing CUDA to easily perform memory-wide operations. To study the impact of this design choice, we also implemented an ad-hoc version of PCM following the same idea as SFF. The times we measured in the default scenario were 0.31ms for CCM and 0.27ms for OCM, which are greater than those measured for the standard algorithms (see Table 4.2). Since we expect the presence of identical constraints into different filters to be common in real applications, we modeled this aspect in our default scenario, which includes one million constraints, but only 80000 unique ones. This means that by removing duplicates we reduce the number of stored constraints by an order of magnitude but this is not enough to balance the penalty of accessing more complex data structures. On the other hand, we cannot exclude that, especially in the OCM case where the differences are more limited, specific workloads, with even larger number of equal constraints, may advantage this solution.

Number of Attributes

Figure 4.7 shows how performance changes with the average number of attributes inside events. In particular, Figure 4.7a shows the processing time of the three algorithms. All of them exhibit higher matching times with a higher number of attributes. Indeed, increasing the number of attributes in the incoming events also increases the work that need to be performed, since each of them has to be compared with the stored constraints. However, SFF performs all these evaluations sequentially, while OCM and CCM perform them in parallel. This explains why the processing time increases faster in SFF, making the advantage of OCM and CCM larger with a higher number of attributes. It is interesting to note that the copies between CPU and GPU memory represent a fixed overhead for CCM. This explains why CCM performs worse than OCM when the processing cost is very reduced (i.e., with only one attribute). However, when the complexity of processing increases, CCM easily outperforms both SFF and OCM. This is evident if we consider the speedup over SFF, plotted in Figure 4.7b: OCM moves from 4x with one attribute to about 6x with 9 attributes, while CCM becomes ten times faster than SFF with 9 attributes.



(a) Event Processing Time (ms) (b) Speedup w.r.t. SFF

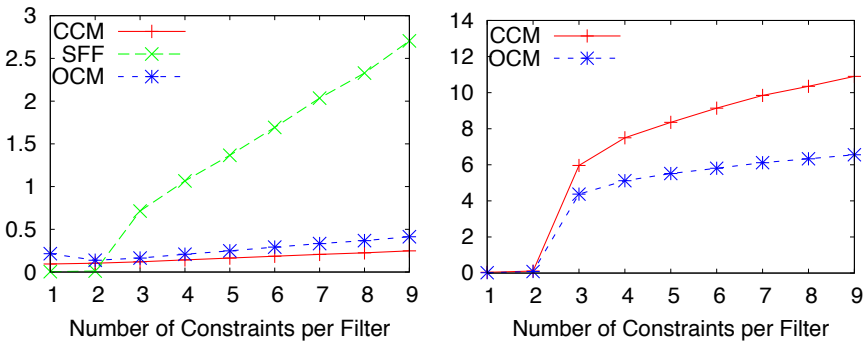
Figure 4.7: Number of Attributes

Number of Constraints per Filter

Figure 4.8 shows how performance changes with the average number of constraints in each filter. Increasing such number, while keeping a fixed number of filters, increases the overall number of constraints deployed in the engine, and thus the complexity of matching. This is confirmed by Figure 4.8a. As in the previous case, the possibility to process constraints in parallel advantages CCM and OCM: all algorithms show a linear trend in processing time, but SFF times grow much faster. The speedups w.r.t. SFF with 9 constraints per filter (see Figure 4.8b) overcome 6x for OCM and 10x for CCM. With one or two constraints per filter SFF performs better than OCM and CCM. This is a very special (and quite unrealistic) case in which the chance to find a matching filter for a given interface is very high, such that at the end all events are relevant for all interfaces. The pruning techniques of SFF work at their best in this case, while OCM and CCM always process all constraints, albeit in parallel.

Number of Filters per Interface

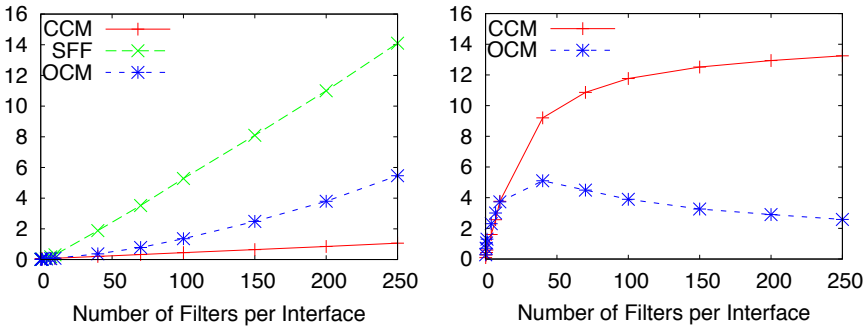
Figure 4.9 shows how performance changes with the number of filters per interface. As in the scenario above, increasing such number also increases the overall number of constraints, and thus the complexity of matching. Accordingly, all the algorithms show growing processing times (see Figure 4.9a). Interestingly, the matching time of OCM seems to grow more than linearly: indeed, with a large number of filters, the cost of setting their counter back to zero after processing becomes more and



(a) Event Processing Time (ms) (b) Speedup w.r.t. SFF

Figure 4.8: Number of Constraints per Filter

more relevant w.r.t. the overall processing time. Moreover, using a large number of filters rapidly increases the number of matching interfaces, allowing SFF to frequently exploit the pruning optimizations described above. This is evident if we look at the speedup w.r.t. SFF (Figure 4.9b): the curve of OCM decreases after 50000 filters per interfaces, when the optimizations of the sequential algorithm start to balance the benefits of parallelization. On the other hand, CCM exploits an architecture with a much higher number of cores: as a consequence the speedup of CCM continues to grow, but slowing as the number of filters per interface increases. With 250000 filters per interfaces, it registers a speedup of 13x w.r.t. SFF.



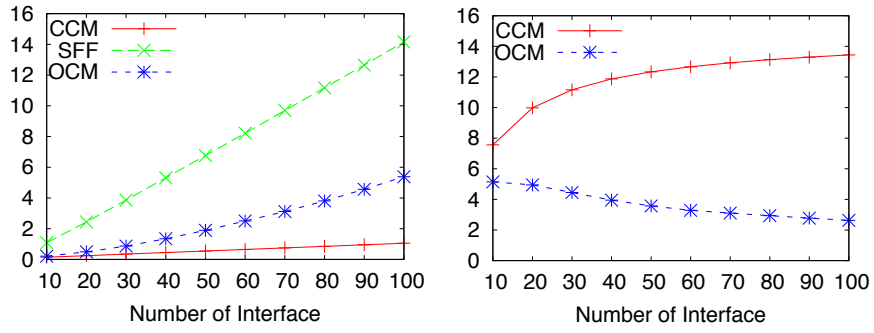
(a) Event Processing Time (ms) (b) Speedup w.r.t. SFF

Figure 4.9: Number of Filters per Interface

Also observe how a very small number of filters favors SFF: it performs better than OCM with less than 700 filters, and better than CCM with up to 1000 filters. Under such circumstances the matching is very fast, with all algorithms registering an average processing time below 0.05ms, and the (almost fixed) overhead of the parallel architectures becomes relevant. Additionally, the fact that this overhead is greater for CUDA than for OpenMP explains why OCM performs better than CCM with up to 10000 filters.

Number of Interfaces

Another important aspect that significantly influences the behavior of a content-based matching algorithm is the number of interfaces. In Figure 4.10 we analyze its impact on SFF, OCM, and CCM, moving from 10 to 100 interfaces. Notice that 100 interfaces may represent a realistic scenario, in which several clients are served by a common event dispatcher that performs the matching process for all of them. As in the previous experiments, increasing the number of interfaces also increases the number of constraints, and thus the complexity of matching. Accordingly, all algorithms show growing processing times as the number of interfaces grows (Figure 4.10a). Also in this case the matching time of OCM grows more than linearly, so that its speedup w.r.t. SFF decreases with the number of interfaces (Figure 4.10b). On the contrary, the speedup of CCM increases with the complexity of processing, moving from 7x to more than 13x.



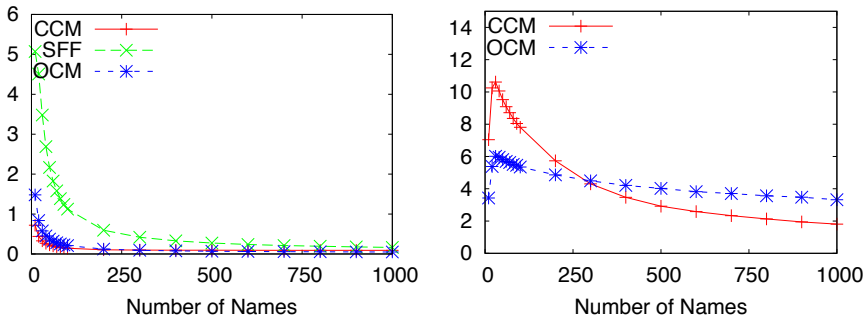
(a) Event Processing Time (ms)

(b) Speedup w.r.t. SFF

Figure 4.10: Number of Interfaces

Number of Names

All the algorithms under analysis make use of the attribute names in the incoming event to select which constraints have to be evaluated. Accordingly, the total number of names used inside constraints and attributes represent a key performance indicator. Figure 4.11a shows how the processing times change with the number of names for attributes and constraints. Increasing this number allows the "constraint selection" phase (common to all algorithm, and always performed on the CPU), to discard a higher number of constraints. Accordingly, the cost of the "constraint evaluation and counting" phase (the more expensive in terms of computation and also the one that CCM performs on the GPU) decreases. This is confirmed by Figure 4.11a: all algorithms perform better when the number of names increases, especially when moving from 10 to 100 names. After this threshold times tend to stabilize. This fact can be explained by observing that over a certain number of names the "constraint evaluation and counting" phase becomes so simple that the processing time cannot decrease anymore.



(a) Event Processing Time (ms)

(b) Speedup w.r.t. SFF

Figure 4.11: Number of Different Names

If we look at the speedup (Figure 4.11b), we observe two different regions. When moving from 10 to 30 names the speedups of CCM and OCM over SFF increase quickly. Indeed, with 10 names, the probability for a filter to match an event is very high allowing SFF to leverage its pruning techniques. This benefit vanishes when moving to 30 names. After this threshold the speedups decrease. Indeed, as we already noticed, increasing the number of names drops the complexity of the "constraint evaluation and counting" phase, which CCM and OCM perform in parallel. With 1000 names, both CCM and OCM still outperform SFF, but by a lower margin. Moreover, as the complexity of the "constraint evaluation and counting" decreases, the fixed overhead of memory copies

to/from the GPU becomes relevant for CCM, which performs worse than OCM with more than 300 names.

Finally, we observe that these results are obtained with a uniform distribution of names. A Zipf distribution, considered more representative of several real-case scenarios [207], would strongly favor CCM, as shown in Table 4.2 (fourth row). Indeed, in such scenario, most constraints would refer to a few attribute names, those also present in most events. In other terms, the results obtained with a Zipf distribution of a large number of names are similar to those obtained with a uniform distribution of much less names.

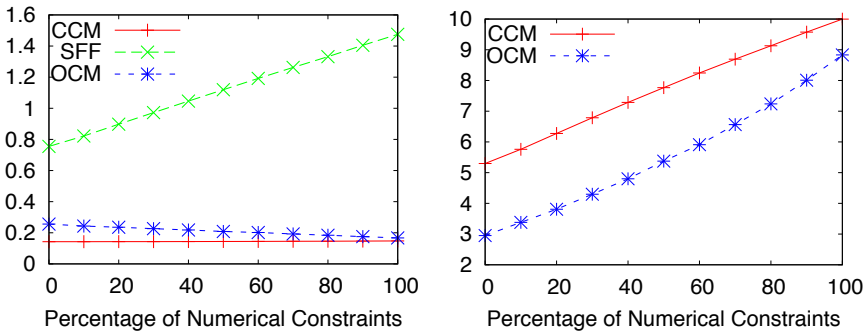
Type of Constraints

Figure 4.12 shows how performance changes when changing the type of constraints. In particular, we measured the processing time when changing the percentage of constraints involving numerical values (the remaining ones involve strings). When considering the impact of constraint types, different aspects cooperate in determining the overall processing times. On the one hand, matching numerical values is less expensive than matching strings. On the other hand, the chances of matching a numerical constraint are higher than those of a string constraint, which results in a greater matching effort to increase counters and check if all constraints of a filter have been matched. The second aspect is less relevant for CCM and OCM, where all operations are performed in parallel, while it has a greater impact on SFF. This explains why the matching time of SFF increases with the number of numerical constraints, while CCM and OCM show constant (and even decreasing, in the case of OCM) processing times. As shown in Figure 4.12b, this results in a constantly increasing speedup of CCM and OCM w.r.t. SFF.

4.5.2 Processing Events in Parallel

Consider the reference system architecture shown in Figure 4.5. With a low input rate, events do not accumulate in the `Input Queue`: as soon as the `Input Connections Manager` enqueues an event, it is immediately dequeued and processed by the `Engine`. However, in case of bursts, the rate at which events are put in the `Input Queue` may temporarily become higher than the processing rate of the `Engine`. In this case many events may be waiting in the `Input Queue`, ready to be processed. In this section we investigate whether processing such events in parallel may introduce some benefits for the algorithms we are considering.

To do so we created 1000 events and deployed them in the `Input`



(a) Event Processing Time (ms) (b) Speedup w.r.t. SFF

Figure 4.12: Type of Constraints

`Queue`. Then we let all algorithms pick up and process all of them k at a time (in parallel), while varying k from 1 to 100, measuring the total time spent. Subscriptions and events were generated using the parameters of our default scenario. In the case of algorithms running on the CPU, we used an OpenMP parallel for to iterate over events. This implementation is possible also with CUDA, since starting from the CUDA Toolkit 4.0 different CPU threads can safely invoke different kernels on the same GPU. However, this would waste CPU resources that may be useful for other purposes, as we will show in the next section. Accordingly, we followed a different route, modifying CCM to use a different copy of the `Count` and `Interfaces` structures for each event to be processed in parallel. This way we could use a single kernel for multiple events, thus avoiding the overhead of launching multiple kernels. Figure 4.5.2 shows the results we measured. When increasing the number of events processed in parallel, SFF significantly increases its performance, moving from more than one second to about 240ms to process 1000 events. We also observe how this time quickly drops moving from 1 to 6 events processed in parallel, where 6 is exactly the number of available cores. The processing time then increases again with 7 events and slowly decreases, becoming constant.

OCM shows a different behavior. Its processing time shows a minimum when 3 events are processed in parallel and increases after this point, to stabilize at about 500ms. On the other hand, this minimum is still greater than the time measured when events are picked up sequentially. Indeed, the minimum average processing time per event measured here is

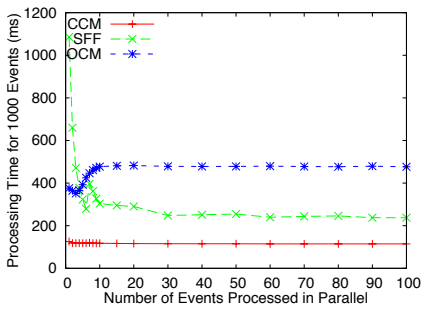


Figure 4.13: Processing Events in Parallel

0.36ms vs 0.21ms of the previous section (see Table 4.2). We can conclude that OCM does not benefit from processing different events in parallel. Finally, CCM shows a small benefit from considering multiple events in parallel: its processing time decreases from 141ms, when considering a single event, to 114ms, when considering 10 events and above. This result can be easily explained: most of the processing resources available on the GPU are already exploited to process a single event, so that increasing the number of events provides a small advantage.

4.5.3 Overall System Performance

As a final experiment, we were interested in studying the benefits of the parallel algorithms when used inside a complete system. Accordingly, we implemented the system architecture shown in Figure 4.5, and adopted a remote client as a source and sink of events. In this scenario, one thread was used inside the `Input Connections Manager`, and one inside the `Output Connections Manager`. The `Input Queue` had a finite size of 100 events. The remote client was used to generate events at increasing input rate and we measured the processing rate, i.e., the rate at which events were processed by the `Engine`. This two rates initially coincide. However, when the input rate begins overcoming the capabilities of the `Engine`, incoming events accumulate on the `Input Queue`. When it is completely filled, the `Input Connections Manager` have to drop incoming events, and the two rates start diverging. Notice that, since we are adopting an unbounded `Output Queue`, the processing rate we measure is proportional to the throughput of the system.

Figure 4.5.3 shows the results we measured. For SFF and CCM we considered two version, one processing events sequentially, and one pro-

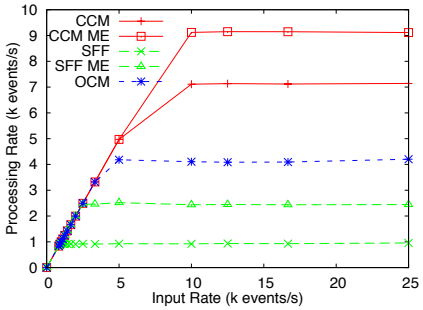


Figure 4.14: Overall System Performance

cessing multiple events in parallel (denoted as `ME` in Figure 4.5.3). Ideally, the processing rate should be the inverse of the matching latency of the `Engine`, as measured in the previous sections. However, when considering the overall system, some CPU resources are used by the two `Connections Managers` to perform marshalling/unmarshalling of events and to handle the communication with sources and sinks, thus potentially reducing the performance of the matching process. This is confirmed by the results we measured for the parallel version of SFF and for OCM. Indeed, both of them require all 6 CPU cores of our test system to reach the peak performance measured in the previous sections. When less cores are available (some of them being allocated to other tasks) the overall performance drops. Take OCM as an example. It exhibits a processing time of 0.21ms per event in our reference scenario, meaning that it should be able to process more than 4700 events/s. However, when used within a complete system its processing rate hardly reaches 4000 events/s. Similarly, SFF exhibits an average matching time of 0.237ms when processing multiple events in parallel. This should translate into a processing rate of more than 4200 events/s, while the performance we measured for the complete system are below 2600 events/s. On the other hand, both versions of CCM do not suffer this problem. This highlights a key, very positive aspect of CCM: since it is entirely executed on the GPU it only requires one CPU core to start memory copies and to launch kernels inside the `Engine`, while the other cores are free for other system tasks, in particular marshalling/unmarshalling and communication. The same holds for SFF but this cannot be considered as a positive aspect. Indeed, SFF, with its purely sequential approach, is under-utilizing available resources (i.e., CPU cores).

4.5.4 Final Considerations

Our experience and the results presented so far allow us to draw some general conclusions about content-based matching on parallel hardware. First of all we may observe that the matching problem is relatively easy to parallelize. Indeed, only a few operations (updates of filter counters) need to be performed atomically. This allows our OCM algorithm to scale linearly with the number of available cores. On the other hand, a sequential algorithm like SFF can introduce a number of optimizations that reduce the gap with OCM under particular scenarios. In general, we measured a speedup between 2x and 6x depending from the scenarios we tested. Moreover, when multiple events are available for processing, SFF may process them in parallel with some advantages. We have seen that the same is not true for OCM, since it already uses all available resources to process a single event. This reduces the speedup of OCM over SFF when considering the maximum processing rate (or throughput) of the overall system.

Our experience in developing CCM let us draw some conclusions about CUDA. First of all, programming CUDA is (relatively) easy, while attaining good performance is (very) hard. Memory accesses and transfers tend to dominate over processing (especially in our case) and must be carefully managed (see the case of the initial cudaMemset operation), while having thousands of threads, even if they are created to be immediately destroyed, has a minimal impact. Also, we observed a fixed cost to pay to launch a kernel, which makes (relatively) simple problems not worth being demanded to the GPU (see the case of simple filters with 1-2 constraints). Fortunately, at least for our problem, it is easy to determine whether the set of subscriptions installed in the system is large enough to take advantage of CCM. In practical terms, we implemented a translator that allows us to switch dynamically between the CPU (running SFF or OCM) and the GPU (running CCM) to always get the best performance. Focusing on the specific problem we addressed, we notice how using a GPU may provide impressive speedup w.r.t. using a CPU. More importantly, this speedup grows with the scale of the problem to solve. This is true in our test system and becomes even more true if we consider that Nvidia currently offers much better graphic cards, and also cards (TESLA GPUs) explicitly conceived for high performance computing. Moreover, using the GPU has the additional, fundamental advantage of leaving the CPU free to focus on those jobs (like I/O) that do not fit GPU programming, as demonstrated by our analysis of the processing rate.

4.6 Related Work

Last decade saw the development of a large number of content-based publish-subscribe systems [82, 208, 10, 209, 210] first exploiting a centralized dispatcher, then moving to distributed solutions for improved scalability. Despite their differences, they all share the need of matching event notifications against subscriptions. Two main categories of matching algorithms can be found in the literature: *counting* algorithms [199, 200] and *tree-based* algorithms [26, 201]. Both SFF and PCM are counting algorithms: they maintain a counter for each filter that records the number of constraints satisfied by the current event. On the contrary, a tree-based algorithm organizes subscriptions into a rooted search tree. Inner nodes represent an evaluation test; nodes at the same level evaluates constraints with the same name; leaves represent the received predicates. Given an event, the search tree is traversed from the root to the leaves. At every node, the value of an attribute is tested, and the satisfied branches are followed until the fully satisfied predicates (and corresponding interfaces) are reached at the leaves. To the best of our knowledge, no existing work has demonstrated the superiority of one of the two approaches. SFF, used for comparison in Section 4.5, is usually cited among the most efficient matching algorithms.

Despite the efforts described above, content-based matching is still considered to be a complex and time consuming task [211]. To overcome this limitation, researchers have explored two directions: on the one hand they proposed to distribute matching among multiple brokers, exploiting covering relationships between subscriptions to reduce the amount of work performed at each node [212]. On the other hand, they moved to probabilistic matching algorithms, trying to increase the performance of the matching process, while possibly introducing evaluation errors in the form of false positives [213, 214]. The use of a distributed dispatcher is orthogonal w.r.t. our work. Indeed, the brokers that build a distributed dispatcher have to perform the same kind of matching analyzed in this chapter. Accordingly, PCM can be used in distributed scenarios, contributing to further improve performance. At the same time, some of the ideas behind PCM can be leveraged to exploit parallel hardware to speedup probabilistic algorithms. Indeed, probabilistic matching usually involves encoding events and subscriptions as Bloom filters reducing the matching process to a comparison of bit vectors. This is a strongly data parallel computation, which would perfectly fit OpenMP and CUDA. We plan to explore this topic in the future.

The idea of parallel matching has been recently addressed in a few works. In [215], the authors exploit multi-core CPUs both to speedup

the processing of a single event and to parallel the processing of different events. The results they obtain seem to be worse than those obtained by PCM. Indeed, they measure a speedup of about 3x w.r.t. a sequential version of their algorithm, with 8 available cores, while PCM shows a speedup of about 6x with 6 cores. Other works investigated how to parallel matching using ad-hoc (FPGA) hardware [216], while we focus on off-the-shelf hardware. To the best of our knowledge, PCM is the first matching algorithm to be implemented on GPUs. A preliminary version of the algorithm, with an analysis of its performance (in terms of latency, only) when implemented on GPUs, has been published in [136].

The adoption of GPUs for general purpose programming is relatively recent and was first enabled in late 2006 when Nvidia released CUDA [203]. Since then, commodity graphics hardware has become a cost-effective parallel platform to solve many general problems, including image processing, computer vision, signal processing, and graphs algorithms. An extensive survey on the application of GPU for general purpose computing can be found in [217].

4.7 Conclusions

In this chapter we presented a parallel content-based matching algorithm and its implementation both on a multi-core CPU, using OpenMP, and on CUDA GPUs. We compared it with SFF, the matching algorithm of Siena, well known for its efficiency. Results demonstrate the benefits of parallelism in a wide spectrum of scenarios: on the CPU we have a speedup between 2x and 6x on our reference 6 core hardware; on the GPU we have even higher speedups, reaching 13x in the most challenging scenarios. This reflects the difference in processing power of the two platforms, as acknowledged by Intel itself [218].

Moreover, delegating to the GPU the effort required for the matching process brings additional advantages when considering the whole system, by leaving the main CPU free to perform other tasks.

Notice that, although our presentation focuses on the case of content-based publish-subscribe systems, the problem of matching is more general. Indeed, as observed by others [200, 10], several applications can directly benefit from a content-based matching service. They include intrusion detection systems and firewalls, which need to classify packets as they flow on the network; intentional naming systems [219], which realize a form of content-based routing; distributed data sharing systems, which need to forward queries to the appropriate servers; and service discovery systems, which need to match service descriptions against service

queries.

As observed in Chapter 2, matching is the main building block for a CEP system. First, it realizes the selection of the primitive events relevant for a composite one. Second, it is used to decide the sinks interested in the composite events identified. In the following of this thesis we continue our analysis of the processing algorithms for CEP. In particular, in Chapter 5 we introduce T-Rex, our complex event processing system, focusing on the algorithm it implements to efficiently evaluate TESLA rules. In Chapter 6 we explore the possibility to adopt CUDA GPUs inside T-Rex to increase its performance when it comes to deal with complex rules, involving a large number of events.

5 The T-Rex Engine

5.1 Introduction

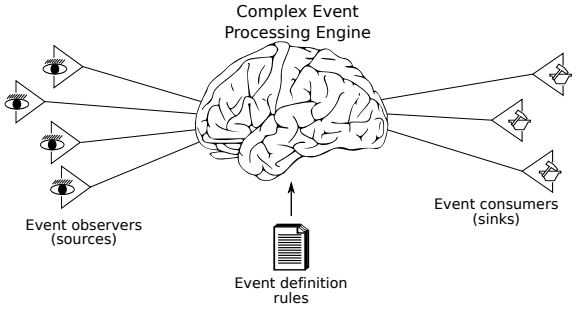


Figure 5.1: The high-level view of an CEP application

Figure 5.1 shows again our reference architecture for a CEP application. In Chapter 3 we already introduced TESLA, our event definition language, while in Chapter 4 we presented the design and implementation of a parallel matching algorithm to select the primitive events relevant for a composite one and to determine the sinks interested in a specific event notification, based on its content. In this chapter we address the design and implementation of T-Rex, a *CEP engine* that interprets TESLA rules and efficiently evaluates incoming primitive events to detect composite ones.

While this chapter mainly focuses on the description of the processing algorithms used inside the engine to efficiently evaluate events, we also developed client-side libraries for different platforms with the aim of building a complete *CEP middleware*.

As we discussed in Section 2.4, almost all existing CEP (and, more in general, IFP) systems make use of processing algorithms that evaluate rules building results incrementally, as new events enter the engine. Typically these processing algorithms are based on automata, and several implementations, both from the academia (e.g. Cayuga [98]) and from the industry (e.g. Esper [112]) proved their efficiency.

However, TESLA presents some significant difference w.r.t. existing languages. First of all, it offers time-bounded sequences of non-

contiguous events. Second, it allows users to combine different sequences of events inside a single rule, as shown in Section 3.4. Patterns built using these features are non-trivial to represent using structures based on automata. This, in turn, may potentially impact also the efficiency of a processing algorithm.

Following this intuition, we implemented two different versions of T-Rex, one based on automata, and one based on a radically different approach, in which processing is not performed incrementally, but delayed as much as possible. A detailed analysis of the two approaches show how the second perform better in the vast majority of scenarios we tested. Moreover, as we will see in Chapter 6, this second approach is also much easier to parallelize, taking advantage of parallel hardware, if available.

The rest of the chapter describes T-Rex in details and analyzes its performance. In particular Section 5.2 introduces the reationale behind the design of the two processing algorithms we implemented. Section 5.3 and Section 5.4 describe the two processing algorithms in details. Section 5.5 presents the general structure of the T-Rex engine, while Section 5.6 and Section 5.7 show how the presented algoriths have been implemented into T-Rex to efficiently process event notifications. Section 5.8 evaluates the performance of T-Rex, providing an detailed analysis of the two algorithms proposed and comparing them with a state of the art commercial product. Finally, Section 5.9 provides some conclusive remarks.

5.2 Event Processing Algorithms for TESLA

As we have seen in Chapter 3, at the heart of the TESLA language is the capability of specifying sequences of events: a composite event occurs when the last event of all the sequences in a rule is detected. Let us call this event *Terminator*. The goal of a processing algorithm is to analyze the history of primitive events received, looking for the relevant sequences to create a composite event from each of them. Consider for example Rule R1 below.

```
Rule R1
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          each Temp(area=$a and value>45)
          within 5 min. from Smoke
where     area=Smoke.area and measuredTemp=Temp.value
```

Rule R1 defines a composite event `Fire` starting from a single sequence of two events, `Temp` and `Smoke`. `Smoke` is the terminator of Rule R1; accordingly, Rule R1 can fire only when a `Smoke` event is detected.

When implementing a CEP algorithm, two opposite approaches can be followed. On the one hand, we can store incoming primitive events, postponing all the processing to detect sequences until a terminator enters the engine. On the other hand, we can process events incrementally as they arrive, storing the results of intermediate computation. The second approach potentially decreases the amount of processing performed when a terminator is detected; however, it may demand for more memory to store partial results. In our analysis we consider both approaches, designing an algorithm for each of them. We call the algorithm following the first approach *Column-based Delayed Processing* (*CDP*), since it organizes the history of received events into columns and delays processing them to the time when a terminator arrives. Similarly, we denote the algorithm that follows the second approach *Automata-based Incremental Processing* (*AIP*), since it processes events incrementally as they arrive and stores partial results as automata.

5.3 The AIP Algorithm

This section provides an overview of the AIP processing algorithm. First of all, when adopting the AIP algorithm, T-Rex translates each rule into what we define an *automaton model*, which is composed of one or more *sequence models*. At the beginning, each sequence model is instantiated in a *sequence instance* (or simply *sequence*). New sequences are created at run-time, while event notifications enter the engine. More specifically, when a new event notification enters T-Rex several things may happen:

- new sequences can be created by duplicating existing ones;
- existing sequences can be moved from a state to the following one;
- existing sequences can be deleted, either because they arrive to an accepting state, representing the detection of a new composite event, or because they become invalid, i.e., unable to proceed any further.

5.3.1 Creation of Automata

As described in Chapter 3, each TESLA rule filters event notifications according to their type and content, and uses the `*-within` operators to define one or more sequences of events. Additional constraints between

events can be introduced using parameters and negations. Consider for example the following Rule R2, which includes rule R1 introducing the `Wind` event to build a complex enough rule to describe our algorithm in all its aspects:

```
Rule R2
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          each Temp(area=$a and value>45)
          within 5 min. from Smoke and
          each Wind(area=$a and speed>20)
          within 5 min. from Smoke and
          not Rain(area=$a) between Smoke and Wind
where     area=Smoke.area and measuredTemp=Temp.value
```

Rule R2 captures two sequences of events that concur in building the `Fire` event. One is composed by a `Temp` event followed by `Smoke`, the other is composed by a `Wind` event followed by `Smoke`. These two sequences share a common event (i.e., `Smoke`). This is typical with TESLA, as it forces all sequences defined by a rule to share at least their last event: the one that implicitly determines the time at which the rule fires and the composite event is detected, i.e., the terminator. Besides this relationship, Rule R2 defines additional relationships between the two sequences of events that build the `Fire` event:

1. the `Smoke`, `Wind`, and `Temp` events have to refer to the same "area". This is captured by using the shared parameter `$a`;
2. the interval between the `Wind` and `Smoke` events cannot include any `Rain` event. This is captured by using the `not-between` TESLA operator.

These considerations can be generalized in an algorithm that translates a generic TESLA rule R into an automaton model for event detection. It operates as follows: first, the sequences of events captured by R are identified and a *sequence model* is built for each sequence in R. A sequence model is a linear, deterministic, finite state automaton. Each event in a sequence of events captured by R is mapped to a state in the sequence model, and a transition between two states s_1 and s_2 is labeled with the *content and timing constraints* that an incoming event has to satisfy to trigger the transition. Figure 5.2 shows the two sequence models, $M1$ and $M2$, derived from Rule R2.

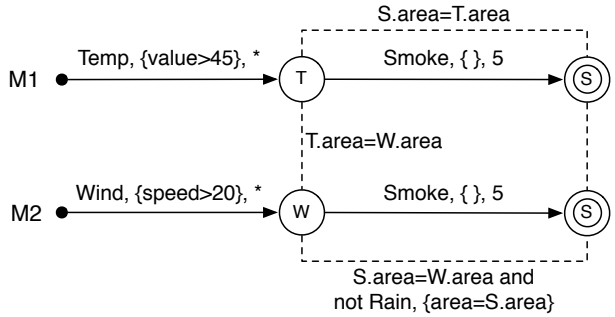


Figure 5.2: Event detection automata for Rule R2

After the sequence models originating from rule R are built, they are annotated and combined in a single automaton model by introducing the relationships among the states captured by the rule (modeled through dashed lines in Figure 5.2). In particular, the states shared by two or more sequence models are connected together, while the relationships determined by parameters and negations are introduced. Figure 5.2 shows the automaton model that results from Rule R2, which includes the relationships indicating the fact that S is a shared state, while states T, W, and S are connected by a parameter, and states W and S are connected by the negated event `Rain` that should not occur between them.

5.3.2 Processing Algorithm

To describe how automaton models created from TESLA rules are used to process incoming events we first describe the processing algorithm for single sequence models, then we show how sequences can interact to capture generic TESLA rules, like the Rule R2 above.

Processing of sequences

Consider a rule that captures a single sequence of events like rule R1 (reported below for clarity), which represents the first half of Rule R2. As we mentioned above, this rule is translated into the sequence model $M1$ of Figure 5.2.

```
Rule R1
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
```

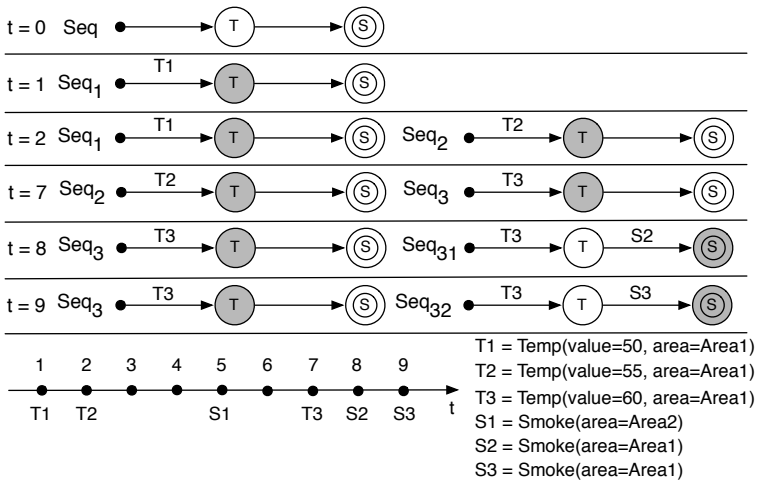


Figure 5.3: An example of sequence processing

```
          each Temp(area=$a and value>45)
          within 5 min. from Smoke
where     area=Smoke.area and measuredTemp=Temp.value
```

When the rule is deployed, the corresponding model is created and a single sequence is instantiated from the model and installed in the system, waiting in its initial state for the arrival of appropriate events. When a new event e arrives, the algorithm reacts as follows: (i) it checks whether the type, content, and arrival time of e satisfy a transition for one of the existing sequences; if not, the event is immediately discarded. If a sequence Seq in a state s_1 can use e to move to its next state s_2, the algorithm (ii) creates a copy S' of Seq, and (iii) it uses e to move S' to state s_2. Notice that the original sequence Seq remains in state s_1, waiting for further events. Sequences are deleted when it becomes impossible for them to proceed to the next state since the time limits for future transitions have already expired. Notice that a sequence in its initial state is never deleted as it cannot expire.

As an example of how processing of sequences works, consider Rule R1 and the corresponding model $M1$. Figure 5.3 shows, step by step, how the set of incoming events drawn at the bottom is processed. At time $t = 0$ a single sequence Seq of model $M1$ is present in the system, waiting in its initial state. Since Seq does not change with the arrival of new events, we omit it in the figure for all time instants greater than 0. At

time $t = 1$ an event $T1$ of type `Temp` enters the system. Since it matches type, content, and timing constraints for the transition to state T, we duplicate Seq, creating Seq_1, which advances to T. Similarly, at $t = 2$, the arrival of a new event $T2$ of type `Temp` creates a new sequence Seq_2 from Seq and moves it to state T. At time $t = 5$ a `Smoke` event arrives but from the wrong area, so it is immediately discarded. At time $t = 7$, Seq_1 is deleted, since it has no possibility to proceed further without violating the timing constraint of its outgoing transition. At the same time, the arrival of event $T3$ generates a new sequence Seq_3. At time $t = 8$ Seq_2 is deleted, while the arrival of an event $S2$ of type `Smoke` from the correct area duplicates Seq_3, generating and advancing Seq_{31} to its accepting state S. This means that a valid sequence, composed by events $T3$ and $S2$ has been recognized. After detection, the sequence Seq_{31} is deleted. Similarly, at $t = 9$, the arrival of $S3$ causes the creation of sequence Seq_{32} and the detection of the valid sequence composed by $T3$ and $S3$.

Notice that all active sequences store the information about all the events they used to reach their current state, i.e., their content and the time when they occurred. As we said, the occurrence time of events is used to check when a sequence can be deleted, while the content of events is used to check the value of relevant attributes, like `area` in our example, and to build the content of the generated events.

As a final remark, notice that sequences may include negations, which constrain an event e not to occur in a given interval i. In particular, the interval i can be expressed using two events (say, e_1 and e_2) belonging to the same sequence, or using an event and a maximum time-span (say, e_3 and t). The two cases are processed using different techniques: in the first case, when e arrives all sequences that have already reached the state associated with the event e_1, but not that associated with e_2, are immediately deleted; in the second case, instead, all e events occurred within t are stored. When a sequence arrives at the state associated with event e_3, we check if there are instances of e arrived within t: if so, the sequence is immediately deleted, otherwise it may proceed.

Processing of complete rules

As we said, generic TESLA rules usually capture multiple sequences of events, always sharing at least one event with each other. Consider for example Rule R2: it is translated into the automaton of Figure 5.2, composed of models $M1$ and $M2$, which share the state S entered when an event of type `Smoke` is detected. Now suppose that the arrival of a `Smoke` event causes three sequences, instances of model $M1$, and two

sequences, instances of model $M2$, to move to their final accepting state. Each possible couple of sequences $(S1, S2)$ (with $S1$ instance of model $M1$ and $S2$ instance of model $M2$) captures a valid set of events for the entire rule, which means that six different composite events `Fire` are detected, all sharing the same `Smoke` event. On the contrary, if a `Smoke` event is only used to accept sequences of type $M1$, but none of type $M2$ (or vice-versa), we can safely discard all sequences using it, since they have no chance to be combined with sequences of model $M2$ (resp. $M1$) to satisfy all rule's constraints.

More in general, when a sequence S, instance of a sequence model M, can move to state X shared with models $M_1, .., M_n$ using an event e, our algorithm first checks if e can cause a transition to X in at least one sequence for each model $M_1, .., M_n$. If this is not the case, then S is deleted, otherwise it advances to state X. Similar checks are performed to verify other relationships among different sequences, like those resulting from the use of parameters. As an example, Rule R2 includes a parameter constraint on the `area` attribute of `Temp` and `Wind` events. This constraint has to be checked when two sequences (one instance of $M1$, the other instance of $M2$) that may potentially result in detecting a `Fire` event, both reach their final state.

Selection and consumption policies

Another feature of TESLA that we need to capture in our processing algorithm is the definition of programmable event selection and consumption policies. Capturing event consumption is straightforward: simply, when a sequence S resulting from a Rule R arrives to its accepting state using and consuming an event e, we can delete all other sequences resulting from the same Rule R that made use of e and have not yet arrived to their accepting state. Indeed, they are all invalidated by the consumption of e.

With respect to event selection, the described algorithm, by duplicating sequence instances at each state transition, captures all possible sequences of events that satisfy the content and timing constraints of a rule, i.e., this algorithm always captures the multiple selection policy typical of the `each-within` operator. However, TESLA includes operators, like `first-within` and `last-within`, which define different selection policies in which only a subset of all possible sequences need to be selected. The most simple way to implement these operators is to select or discard sequences only when they arrive to their final accepting state. In the most general case this approach is also the only possible one. Consider for example the following Rule R3, which adds a consuming clause

to Rule R2:

Rule R3

```
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          last Temp(area=$a and value>45)
          within 5 min. from Smoke
where     area=Smoke.area and measuredTemp=Temp.value
consuming Temp
```

Now suppose we receive two `Temp` events, say $T1$ and $T2$, in this order. When $T2$ arrives we cannot discard $T1$, as we could initially be tempted to do; indeed, the arrival of a `Smoke` event would cause the generation of a new composite event, but also the consumption of $T2$, making $T1$ become the last event received, and so the right candidate for further rule evaluations. On the other hand, rule specific optimizations are possible: take for example Rule R2, which does not consume events. In this case, when $T2$ occurs, $T1$ cannot become the last occurred event for next evaluations, so it can be safely deleted. More in general, in the presence of a `last-within` operator associated with an event X, without a `consuming` clause for X, we can safely discard sequences composed by previous events of type X when a new one is detected. Similar rule-specific optimizations are possible also when the `first-within` operator is involved. Consider for example the following Rule R4:

Rule R4

```
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          first Temp(area=$a and value>45)
          within 5 min. from Smoke
where     area=Smoke.area and measuredTemp=Temp.value
```

and suppose we receive two events of type `Temp`, $T1$, at time $t = 1$, and $T2$, at time $t = 4$. If we receive an event S of type `Smoke` at time $t = 5$ we need to duplicate only the sequence including $T1$ (i.e., the first event of type `Temp` within 5 minutes from S) not the one including $T2$. Notice that in this case, in which the `first-within` operator is used, we cannot delete the sequence including $T2$, as this event can become the first `Temp` event valid when time proceeds and $T1$ becomes too old to be considered again. However, by duplicating only the sequences that include the first detected event we greatly reduce the number of automata stored in the system.

Computing Aggregates

As we have seen, TESLA rules may include the definition of aggregates. As an example they may be used to extract the averave temperature over a set of events received from sensors deployed in a wide area. If needed, AIP computes aggregates at the end of processing, after an automaton has arrived to its acceptng state. More in particular, each automaton stores all received events that may be potentially useful for the computation of aggregates, according to the content, timing, and parameter constraints expressed in its corresponding rule. When it arrives to its accepting states, it analyzes them and uses their values to compute the desired function.

5.4 The CDP Algorithm

While the AIP algorithm processes rules incrementally, as new events enter the engine, the CDP algorithm takes the opposite approach: it stores all events received until a terminator is found. To simplify the analysis, instead of keeping a flat history of all received events, each rule R organizes them into columns, one for each primitive event appearing in the sequence defined by R.

5.4.1 Creation of Columns

As an example of how columns are created by the CDP algorithm, consider the following Rule R5.

```
Rule R5
define    ComplexEvent()
from      C(p=$x) and each B(p=$x and v>10)
          within 8 min. from C and
          last A(p=$x) within 3 min. from B
```

Rule R5 defines a sequence including three primitive events of type `A`, `B`, and `C`, respectively. The algorithm creates three columns (see Figure 5.4), each labeled with the type of the primitive events it stores and with the set of constraints on their content. The maximum time interval allowed between the events of a column and those of the previous one (i.e., the window expressed through the `*-within` operator) are modeled using a double arrow. Similarly, additional constraints coming from parameters are represented as dashed lines. Notice that the last column reserve space for a single event.

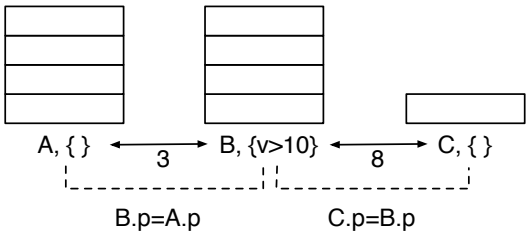


Figure 5.4: Columns for Rule R5

5.4.2 Processing Algorithm

For ease of exposition, we first discuss how the CDP algorithm works when rules involve a single sequence of events, like in Rule R5. Then we extend it to consider complete rules, involving multiple sequences.

Processing of sequences

When a new event e enters the engine, it is processed as follows. First, we check whether it matches (i.e. satisfies type and content constraints of) one or more columns. If this is the case, e is added on top of the matching columns, otherwise it is immediately discarded. If among the matched columns there is the last one (c_ℓ), the processing of the events stored so far starts. Processing is performed column by column, from the last one to the first one, creating *partial sequences* of increasing size at each step. More precisely:

- the timestamp of e is used to find the index i of the first valid element in column $c_{\ell-1}$, by looking at the time window;
- all events in column $c_{\ell-1}$ having an index $i' < i$ are deleted, since they have no chance to enter the window in the future;
- the operation is repeated for each column, considering the timestamp of the first event left in column c_k to delete old events from column c_{k-1};
- e is combined with all the events stored in column $c_{\ell-1}$ that satisfy timing constraints, parameters, and selection policies, creating partial sequences of two events;

- each partial sequence is used to select elements from the previous column $c_{\ell-2}$. The algorithm is repeated recursively until the first column is reached, generating zero, one, or more sequences including one selected event from each column;
- one composite event is generated for each sequence.

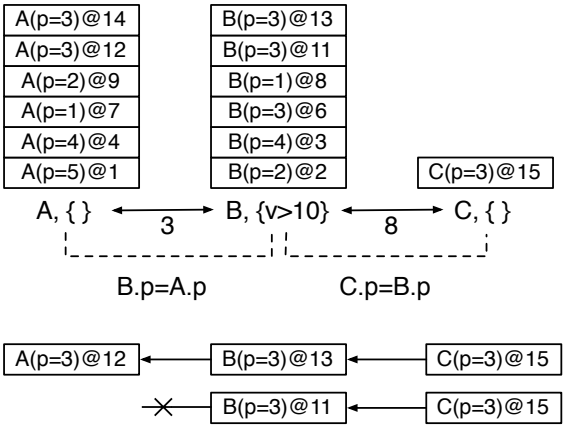


Figure 5.5: An example of processing using columns

To better understand how the algorithm works, consider again Rule R5 and the situation in Figure 5.5, where the events stored in each column are represented with their type, their value for the attribute `p`, and their timestamp (which we assume integer, for simplicity). The event `C(p=3)@15` was the last entering the engine. Since it is a terminator for Rule R5, it starts the processing algorithm. Its timestamp (i.e., `15`) is used to compute the index of the first valid element in Column B, i.e., `B(p=1)@8`, as it results by noticing that the window between Columns B and C is 8. Previous events are removed from Column B, while the timestamp of `B(p=1)@8` is used to remove elements from Column A whose timestamp is lower than 5. The remaining events are evaluated to detect valid sequences. First, Column B is analyzed: events `B(p=3)@13` and `B(p=3)@11` are both valid. Both are selected, since a multiple selection policy is defined between events `B` and `C`. This generates two partial sequences (`<C(p=3)@15, B(p=3)@13>` and `<C(p=3)@15, B(p=3)@11>`), as shown at the bottom of Figure 5.5. Event `B(p=1)@8` is not selected, since it violates the constraint on attribute `p`. The two partial sequences above are used to select events from Column A. Sequence `<C(p=3)@15, B(p=3)@13>` selects event `A(p=3)@12`, which is the only one satisfying both its timing and parameter constraints. Indeed,

the event A(p=3)@14 is not valid since its timestamp is greater than the timestamp of B(p=3)@13. On the contrary, sequence <C(p=3)@15, B(p=3)@11> does not select any event as none of those in Column A satisfies its timing and parameter constrains. At the end of the processing, the only valid sequence detected is <C(p=3)@15, B(p=3)@13, A(p=3)@12>, so a single composite event is generated.

Processing of complete rules

The example above shows how the CDP algorithm detect a single sequence of events. As we have seen in Section 5.3 detecting complete rules, including more than one sequence, is non trivial for the AIP algorithm, which requires a post-processing phase to merge all the instances detected for each sequence.

On the contrary, the CDP algorithm can combine different sequences directly during processing, and does not need an ad-hoc merging phase. To better understand how CDP manages the presence of multiple sequences, consider the following Rule R6.

```
Rule R6
define    ComplexEvent()
from      E() and each D() within 3 min. from E
          and each C() within 3 min. from E
          and each B() within 3 min. from D
          and each A() within 3 min. from C
```

Rule R6 is composed of two sequences, one including events A, C, and E, and one including events B, D, and E. While it is theoretically possible to evaluate the two sequences separately and merge them only at the end, as in the AIP algorithm, our tests have shown that this is less efficient that processing multiple sequences concurrently.

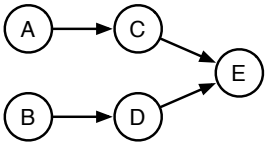


Figure 5.6: Column Dependency Graph for Rule R6

To do so, we create a dependency graph among columns. As an example, Figure 5.6 shows the column dependency graph for Rule R6. Events of kind A can be extracted only when we know the timestamps of all

valid events of type `C`. Accordingly, there is a dependency between `A` and `C`; on the contrary there is no dependency between `A` and `B`.

Starting from the column dependency graph, we extract an order for column evaluation in which each column is evaluated after the column it depends from. In our example, a valid order could be `E`, `D`, `C`, `A`, and `B`. Notice that dependencies can be determined when a rule is deployed. Accordingly, the evaluation order can be computed at deploy time and used during processing.

Processing is then performed moving from column to column, as in the case of a single sequence. The only difference is that partial results store events coming from different (partial) sequences. When all columns have been visited the partial results produced already contain all valid pattern of events detected, and do not need further processing to combine sequences together.

Processing negations

In CDP, negations are considered as a special kind of columns: as all other columns, they become part of the column dependency graph of a rule, which determines when they can be evaluated. The only difference w.r.t. other columns is the result of processing: if a partial sequence captures events from a negation column, this means that it violates a negation constraint expressed in the original rule, and accordingly it is immediately discarded.

Computing aggregates

As for negations, also aggregates are considered as a special kind of columns. Events matching the content constraints of an aggregate are stored in its corresponding column c. After all partial sequences have been identified, CDP selects, for each partial sequence, the relevant items from c according to the timing and parameter constraints expressed in the rule under processing. The value of selected items is then used to compute the desired aggregate value (e.g., sum, average).

Event consumption

In CDP, handling event consumption is a simple operation. Indeed, when a Rule R fires and a composite event is generated, CDP removes all consumed events that are still stored in the columns for R. Differently from AIP, CDP does not store intermediate processing results, which could be potentially affected by the consumption of one or more events.

Accordingly, as we will see in Section 5.8, consumption clauses introduce less overhead in CDP than in AIP.

5.5 The Architecture of T-Rex

In this section we present in details the general architecture of the T-Rex engine. T-Rex has been implemented in C++: the choice of this language allows a fine-grained control of the memory layout, thus making it possible for processing algorithms to implement more efficient data structures for event processing.

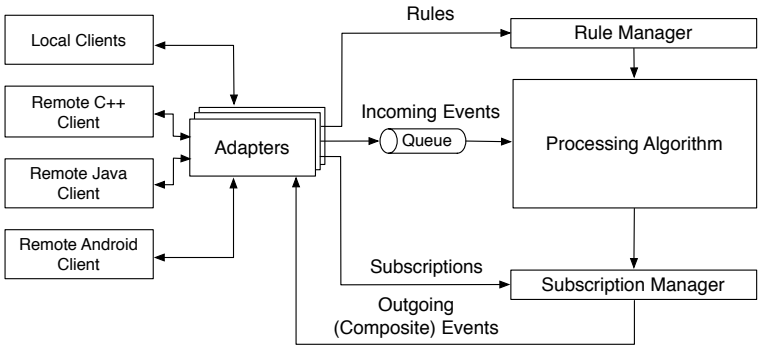


Figure 5.7: The Architecture of T-Rex

The general architecture of the T-Rex is presented in Figure 5.8. External clients communicate with the T-Rex engine through a set of `Adapters`, which allow them to easily deploy new rules, and send and receive events. Up to now we have implemented four different adapters: the first one allows the communication with local clients, running in the same process of the T-Rex system, by mean of direct method calls. It has been used during the testing and evaluation phase of the system, as discussed in Section 5.8. The second and the third adapters allow the interaction with remote clients written in the C++ or Java languages. The fourth adapter allows the communication with mobile devices based on Android [220]. The last three adapters implement the marshalling and unmarshalling of events and TESLA rules, as well as the communication through sockets.

When a client deploys a new rule, the `Rule Manager` processes it, generating the data structures required by the `Processing Algorithm`. For example, rules are translated into automaton models when the AIP algorithm is adopted, and into columns when the CDP algorithm is adopted.

When a new event enters the system, it is first put in a FIFO `Queue`.

This component is used to avoid the loss of input events due to temporary bursts. Administrators can choose the maximum size of the queue according to their needs: having a long queue decreases the probability of losing events, but potentially introduce latency. The `Processing Algorithm` picks up events from the `Queue`, processes them, and produces composite events.

All composite events are sent to the `Subscription Manager`, which keeps track of the interests (subscriptions) of clients and uses this information to deliver them the relevant events through the adapters. It performs the matching process, as described in Chapter 4.

5.6 Implementing the AIP Algorithm

In this section we describe how the AIP algorithm has been implemented in T-Rex, presenting the features that contribute to an efficient processing of events.

As we mentioned, since AIP incrementally computes results as new events enter the system, it may generate a large number of sequence instances. Accordingly, in implementing this algorithm in C++, we adopted advanced memory management techniques to avoid duplicating data (like events) shared by multiple sequences, and we used ad-hoc indexing techniques to minimize the number of sequences to consider for each incoming event.

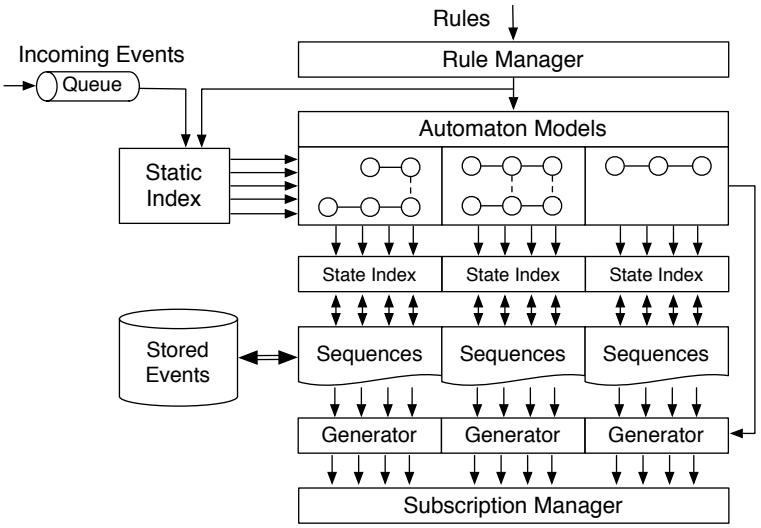


Figure 5.8: The T-Rex Engine using AIP

Figure 5.8 shows the inner logical components used by T-Rex to execute the AIP algorithm. When a client deploys a new rule, the `Rule Manager` generates its corresponding automaton model M, and passes M to the `Automaton Models` component, which stores all the information about sequence models (i.e., states, constraints on transitions, parameters, etc.) as well as the information about inter-sequence relationships.

Notice that a rule R may include two types of constraints that affect the sequences originating from R: some of them, like the type of the event triggering a transition, can be directly associated to the sequence models defined from R as they do not depend from the events actually captured, others have to be associated to sequence instances. We call *static* the former, *dynamic* the latter. For instance, consider Rule R1, the constraint on attribute `value` of event `Temp` (i.e., `Temp.value>45`) is static and the same holds for the timing constraint (i.e., `Temp` must occur `within 5 min. from Smoke`). Conversely, the constraint on attribute `area` of event `Smoke` (i.e., `Smoke.area=$a`) is dynamic, as it depends on the value of the attribute `area` of event `Temp`, which is specific to the sequence being considered and unknown at model creation time. During the processing of rules, the `Rule Manager` identifies static constraints and uses them to compile the `Static Index`, which is used to efficiently perform a preliminary type and content-based filtering of incoming events. Since this is the same filtering at the base of traditional publish-subscribe systems, we were able to re-use well known techniques developed by the community working on publish-subscribe. In particular, the `Static Index` implements a counting algorithm as described in [23].

When the next event to be processed exits the `Queue`, the `Static Index` identifies which sequence models and which states it may influence. If none is influenced, the event is immediately discarded; otherwise the event is delivered to the `Sequences` component, which stores all active sequence instances generated for each model. To speed up the access to sequence instances from the related model, the `State Index` maps sequence models to existing instances according to their current state, i.e., for each sequence model M and state S it maps the instances of M waiting in S. Once all the sequences affected by an incoming event have been selected, T-Rex performs a further selection by looking at dynamic constraints. In particular: (i) it checks timing constraints, deleting all sequences violating them, and (ii) it looks at parameters, ignoring sequences that present non-valid parameter values for the event under processing. All remaining sequences are duplicated and advanced, as described in Section 5.3. Notice that sequences generated from different models are completely independent from each other and can be safely processed in parallel. Accordingly, T-Rex performs actual processing of

sequence instances using a pool of threads, so that it can take advantage of multi-core hardware.

Since a single incoming event may be used in many different sequences, we limit memory usage by avoiding an explicit copy of its content. Instead, each sequence keeps references to all the events it used to arrive to its current state, and a single copy of each used event is saved in the `Stored Events` data structure. As soon as all sequences using an event e are deleted from the system, T-Rex removes e from the `Stored Events`.

When one or more sequences arrive to their accepting state, they are forwarded to the `Generator` component, which performs several operations: (*i*) it retrieves the set of events that compose the various sequences it receives; (*ii*) it uses the information stored into the `Automaton Models` component to combine those sequences that are instances of the same automaton model, as described in Section 5.3; (*iii*) it computes the values of the attributes of the new composite event notifications captured by the received sequences.

5.7 Implementing the CDP Algorithm

In this section we present the implementation of CDP inside the T-Rex engine, focusing on the aspects that contribute to its performance, and on the main differences w.r.t. the AIP algorithm.

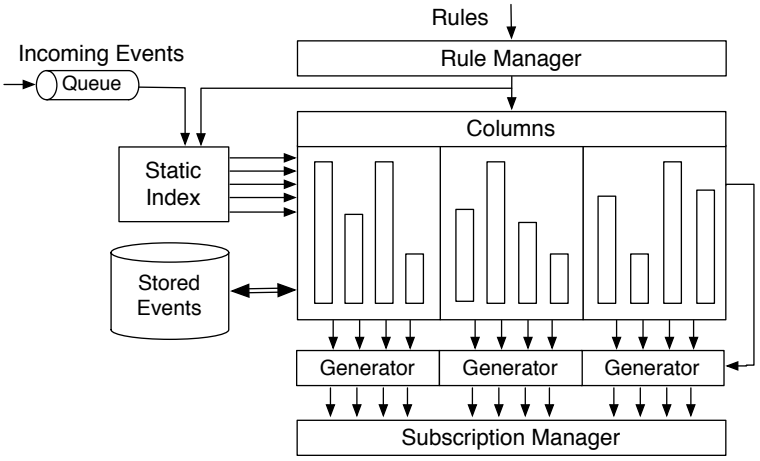


Figure 5.9: The T-Rex Engine using CDP

Figure 5.9 shows the inner structure of T-Rex when implementing the CDP algorithm. The `Rule Manager` processes incoming TESLA rules and produces `Columns`, as described in Section 5.4. As in AIP the `Rule`

`Manager` isolates static constraints to build the `Static Index`. When a new event e exits the `Queue`, the `Static Index` is used to decide which columns e has to be put in. If e also enters the last column of a rule, then the computation is started. Notice that, as for AIP, structures built from different rules are completely independent from each other. Accordingly, T-Rex exploits a pool of threads to process them concurrently on multi-core hardware.

As we already observed in Section 5.4, CDP processes different sequences belonging to the same rule together. Accordingly, it does not need additional components beside `Columns` to store single sequences before they are merged. As in AIP, a single copy of each event relevant for the processing is actually saved in the `Stored Events` data structure, while `Columns` refer to them only through pointer. This avoid duplication of events, thus reducing the memory required during processing.

As we have seen in Section 5.4 two main operations are performed during processing: *i.* old events are removed from columns; *ii.* relevant events are extracted from columns. To efficiently support both this operations columns have been implemented as circular buffers. This way, removing an event does not require to move remaining ones; on the contrary, it simply involves updating the pointers to the first and last element. Second, since events are ordered according to their timestamps, finding the range of events that satisfy specific timing constraints is efficiently implemented in logarithmic time, using a binary search.

When patterns of events satisfying a given rule are detected, they are forwared to the `Generator` components. As in the AIP algorithm, the generator components is used to compute the values of the attributes of the composite event notification as expressed in the fired rule. This may also include computing aggregates. However, differently form AIP, the `Generator` does not need to merge together different sequences, since is performed directly inside `Columns`.

5.8 Evaluation

Our evaluation had two main goals: on one hand we wanted to compare T-Rex with other processing systems that could handle some of the typical rules that can be found in CEP applications, like those described in Chapter 3; on the other hand we wanted to study the performance of our system in a wide range of scenarios, comparing the results obtained with the AIP and with the CDP algorithms.

5.8.1 Comparison with a State of the Art System

As we have seen in Chapter 3, finding a system having features comparable with those of T-Rex is not a simple task. Most exiting systems are based on generic data processing rules, not suitable to capture sequences of event notifications. On the other hand, available systems that are explicitly designed to deal with event patterns often present a limited expressiveness, if compared with TESLA. We finally decided to use Esper [112] for many reasons: it is an enterprise level product, widely used (it includes a commercial version, EsperHA) and mature (we used version 4.0.0); it adopts an expressive language with a rich syntax but also puts great emphasis on efficiency and performance; it is open source and provides extensive documentation; it is embedded in Java, which makes the task of writing and executing tests easier.

All the results discussed below have been collected using a 2.8GHz AMD Phenom II PC, with 6 cores and 8GB of DDR3 RAM, running 64 bit Linux. We use a local client to generate events at a constant rate and to collect results. Using this approach, the interaction with the CEP engine is realized entirely through local method invocations; this choice eliminates the impact of the communication layer on the results we collected and allows us to measure the raw performance of the two CEP engines. We set the maximum size of the input queue to 100 events, small enough to emphasize a difference in the processing time through a loss of events. We studied the behavior of the two systems at different input rates, executing each test 10 times: for each measure we plot the average value and the 95% confidence interval.

Event filtering

At the base of a CEP engine is the operation of selecting, or filtering, input events before using them to capture complex patterns. For this reason we decided to start our analysis by comparing the filtering capabilities of T-Rex and Esper.

To do so we deployed 1000 different rules on both systems. Each rule simply filters input events according to the value of an attribute and returns a new event containing such a value. On average, each event is selected by a single rule; this means that, ideally, the event output rate should be equal to the input rate.

Notice that the filtering capabilities are executed by a single component, independently from the processing algorithm adopted for pattern detection. Accordingly, the two versions of T-Rex, the one based on AIP and the one based on CDP, present identical results.

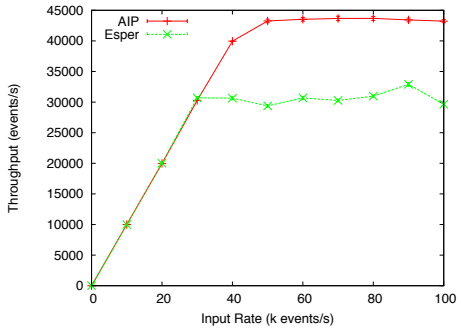


Figure 5.10: Filtering: Comparison between T-Rex and Esper

Figure 5.10 shows the performance of the two systems in terms of throughput, i.e. composite events detected per second. Since the two systems were processing identical loads with the same set of deployed rules, they should theoretically detect the same number of composite events. However, when the input rate increases, both engines start to drop input events as their input queue becomes full: the rate at which events start to be dropped as well as the number of dropped events and consequently the throughput is a measure of the processing overhead.

We can observe how both systems can handle high event input rates, in the order of tens of thousands events per second. Esper, however, starts to drop input events at the rate of 30000 events per second, with a maximum throughput that is below 35000 composite events per second. T-Rex, instead, can handle up to 40000 events per second without dropping from the input queue and registers a maximum throughput of about 45000 composite events per second. As we said, this is a measure of the fact that T-Rex processes events faster than Esper when the deployed rules simply require input event filtering and composite event generation.

Case study

As a second step we wanted to compare the two engines using a realistic case study. Accordingly, we decided to adopt one of the definitions of fire introduced in Section 3.2. More specifically, we used the following definition:

- Fire occurs when temperature higher than 45 degrees and some

smoke are detected in the same area within 3 min. The fire notification has to embed the temperature actually measured.

In translating it into TESLA rules we considered both a multiple selection policy (as modeled by TESLA Rule R1) and a single selection policy (as modeled by TESLA rule R3). We report here Rule R1 and Rule R3 for simplicity.

Rule R1

```
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          each Temp(area=$a and value>45)
          within 5 min. from Smoke
where     area=Smoke.area and measuredTemp=Temp.value
```

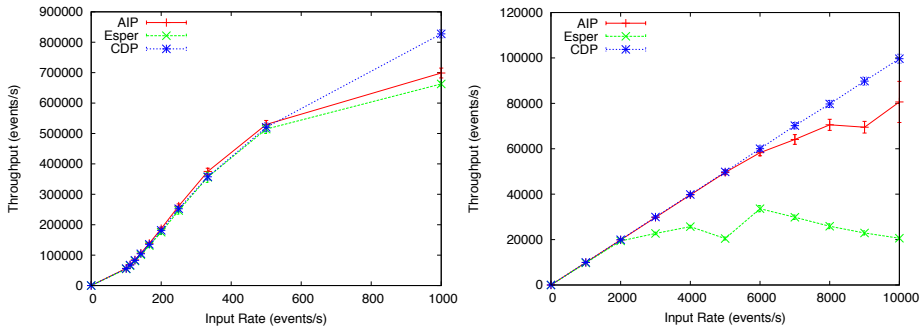
Rule R3

```
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          last Temp(area=$a and value>45)
          within 5 min. from Smoke
where     area=Smoke.area and measuredTemp=Temp.value
```

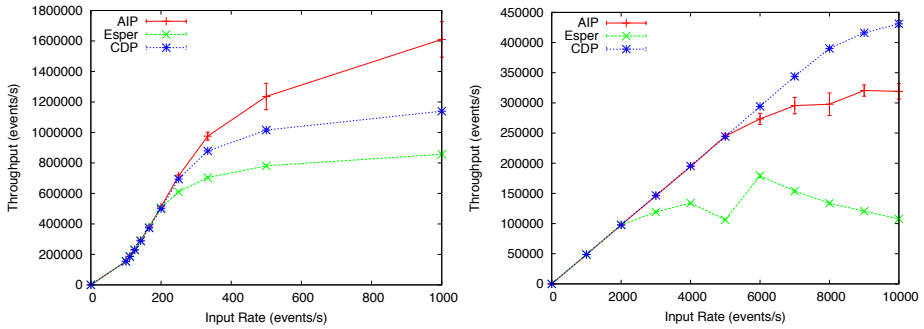
To stress the two systems, each test was performed with 1000 different rules deployed, all having the same structure (that of Rule R1 and Rule R3), but defining 10 different composite events ($\texttt{Fire}_1$, defined from $\texttt{Temp}_1$ and $\texttt{Smoke}_1$, ..., $\texttt{Fire}_{10}$, defined from $\texttt{Temp}_{10}$ and $\texttt{Smoke}_{10}$) and asking for a different minimum temperature (from 1 to 100). The value of the temperature of incoming $\texttt{Temp}_x$ events was uniformly distributed between 1 and 100, while all events shared the same `area` attribute, to maximize the probability of using events. Given these assumptions, each incoming event was selected by 50 different rules, on average. Both when using Rule R1 and Rule R3, we evaluated the engines with three different loads, generating respectively 10%, 50%, and 90% of $\texttt{Smoke}_x$ events (the remaining events are $\texttt{Temp}_x$). Both systems were configured to take advantage of the six cores available on the hosting machine.

Figure 5.11 shows the results we obtained in the different scenarios in terms of throughput. The left column includes the results regarding Rule R1, while the right column shows the results for Rule R3.

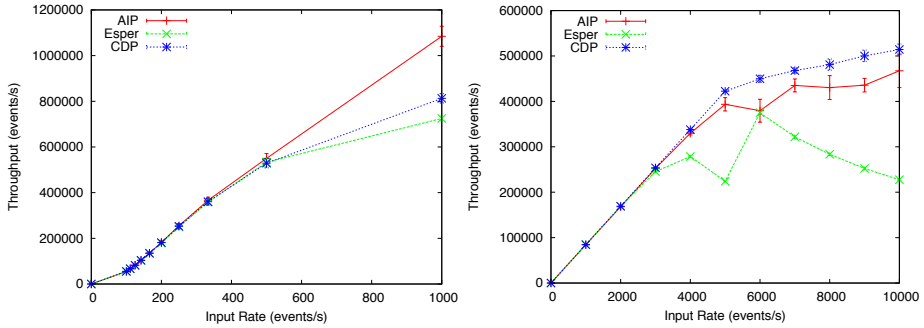
In general, we observe that both systems perform very well, with a throughput of hundreds of thousands of composite events detected per second, even if the hardware we used for the tests is entry level for a production CEP system. Even more important from our point of view



(a) 10% of $\texttt{Smoke}_x$ events



(b) 50% of $\texttt{Smoke}_x$ events



(c) 90% of $\texttt{Smoke}_x$ events

Figure 5.11: Comparison between T-Rex and Esper using Rule R1 (left) and R3 (right)

is that T-Rex outperforms Esper in all scenarios, both when using AIP and CDP.

In particular, T-Rex is capable of processing much more input events before starting to drop them from the input queue, and when it drops them it drops less than Esper. The tests also show how the different selection policies associated with Rule R1 and Rule R3 influence the behavior of both engines: Esper performs better when the multiple selection policy is adopted (its performance are closer to those of T-Rex), while the gap between the two engines increases when the single selection policy is adopted. Also notice how the throughput of T-Rex seems to be still increasing even at the highest input rates we were able to generate. The same is not true for Esper: when rule R3 is adopted (right column) the throughput starts decreasing after the input rate overcomes 6000 events/s; moreover, even at lower rates, it exhibits an irregular trend.

If we compare AIP and CDP, we notice that CDP performs better when a single selection policy is adopted, independently from the percentage of $\texttt{Smoke}_\texttt{x}$ events. On the contrary, AIP performs better with a multiple selection policy and a large number of $\texttt{Smoke}_\texttt{x}$ events. In these particular scenarios, indeed, there is a large number of terminators, and they all require a lot of processing to be combined with previously detected $\texttt{Temp}_\texttt{x}$ events. Moreover, Rule R1 defines a sequence of two events only, thus making automata easier to manage. As we will see in the following of this section, this is the only scenario in which AIP performs better than CDP.

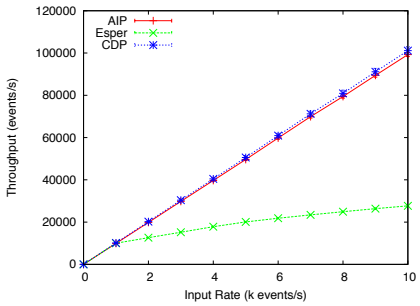
Computing aggregates

The previous experiments considered two important aspects of a CEP system: the capability of filtering input events and that of detecting sequences of events. As shown in Chapter 3, another important feature for a CEP system is the capability of computing the content of composite events by aggregating the values of attributes in primitive events. To evaluate the performance of T-Rex and Esper in this area we used the following Rule R7.

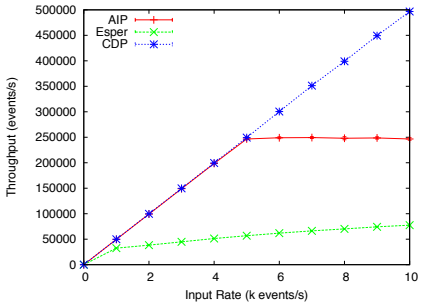
Rule R7

```
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          45 < $t=Avg(Temp(area=$a).value
          within 5 min. from Smoke)
where     area=Smoke.area and measuredTemp=$t
```

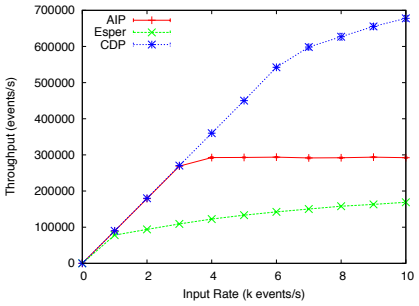
As in the previous test, we adopted 10 different definitions of `Fire` (`Fire`$_1$, ..., `Fire`$_{10}$) and 100 different thresholds for the value of `Temp`$_x$, resulting in 1000 different deployed rules. To increase the throughput of the system, the workload was generated to always satisfy the constraints on the values of `Temp`$_x$: this way every input `Smoke`$_x$ event led to the generation of 100 `Fire`$_x$ events. Also in this case we evaluated the engines with three different loads, generating respectively 10%, 50%, and 90% of `Smoke`$_x$ events.



(a) 10% of `Smoke`$_x$ events



(b) 50% of `Smoke`$_x$ events



(c) 90% of `Smoke`$_x$ events

Figure 5.12: Comparison between T-Rex and Esper using Rule R7

Figure 5.12 shows the results we obtained. If we look at the performance of AIP, we observe that it never drops input events when the number of `Smoke`$_x$ events is small (10%). When the percentage of `Smoke`$_x$ events increases, a higher throughput is generated. However, AIP starts dropping input events as the input rate increases: as a consequence, when the input rate overcomes a given threshold, the throughput becomes constant. This is not true for CDP: with a percentage of `Smoke`$_x$ events of 10% and 50% it does not drop input events, even at the highest input rate we tested. When considering 90% of `Smoke`$_x$ events CDP starts dropping events, but its curve continues to increase.

The main task in Rule R7 is the selection of relevant $\texttt{Temp}_x$ events to produce an aggregate, i.e., the average value. The difference in performance between AIP and CDP suggests that this task is simplified when events are organized in columns.

Finally, if we compare T-Rex with Esper we notice how the latter produces a much lower throughput: in all three tests, Esper starts dropping events at an input rate of about 1000 events/s.

5.8.2 In-depth Analysis with Synthetic Workloads

The direct comparison with Esper on a realistic workload demonstrated the efficiency of our system, even when compared with an advanced commercial product. As stated at the beginning of this section, we were also interested in studying the behavior of T-Rex under different loads. Accordingly we performed several tests using different synthetic workloads.

We defined a default scenario, in which 1000 rules were deployed in the system, all of them defining a sequence composed by two events. Each incoming event was relevant for exactly 1% of the deployed rules; each rule (and each state inside a rule) had the same probability to select incoming events. The windows between two consecutive events in a sequence were 15 seconds long on average, ranging uniformly between 14 and 16 seconds.

Starting from this scenario we studied the behavior of T-Rex when changing some parameters. In particular we focused on: (*i.*) the number of events defined in each sequence, (*ii.*) the number of sequences defined in each rule, (*iii.*) the number of rules deployed in the system, (*iv.*) the percentage of events selected by each rule, and (*v.*) the average size of the windows. Moreover, we evaluated the scalability of our system by measuring how performance changes when increasing the number of CPU cores used (unless otherwise stated we use all available cores). Finally, we studied the influence of some TESLA constructs, namely negation and consumption, on the performance of T-Rex. Since the selection policy used in rules greatly influences the behavior of the system, we ran all experiments twice: once using a multiple selection policy for all rules (i.e., the `each-within` operator) and once using a single selection policy (i.e., the `last-within` operator).

During our tests we measured the average time needed by T-Rex (both using AIP and CDP) to process a single input event. In particular we measured the time elapsed from the instant when the event starts to be actively processed (i.e. when it exits the queue of input events) to the instant when all sequences affected by that event have been processed, including the time needed to produce new composite events for those

sequences, if any, which arrived at their final state. This measure is extremely important: first of all it tells us how each parameter impacts the processing time of T-Rex. Second, it tells us which is the maximum input rate T-Rex can handle. For example, if the processing time is 1 millisecond, we can theoretically handle up to 1000 input events per second. Notice, however, that this is only a theoretical maximum, indeed, the processing time we measure is an average value: single events may take longer to be processed; accordingly, when using a finite input queue, we may start dropping input events before this maximum rate.

To provide more details we also plot the throughput curves for each workload we tested. By comparing the trend of the throughput we can learn more about the input rate at which events actually start to be dropped from the finite queue, but also about the kind of rules we are processing: some of them, indeed, generate significantly more output (i.e., composite) events than others.

Number of events in sequences

Figure 5.13 and Figure 5.13 study the behavior of T-Rex when changing the number of events captured by each sequence. For AIP this represents the number of states composing each sequence model, while for CDP it represents the number of columns defined for each sequence.

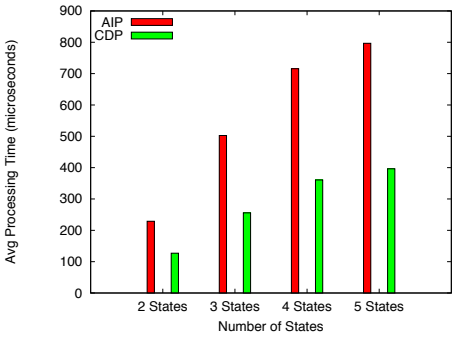
Increasing the number of states in a sequence, when using a multiple selection policy (Figure 5.13), increases the complexity of processing. When using AIP this results in a larger number of sequence instances to be duplicated during processing. When using CDP this results in more events captured at each column, creating more partial results.

Accordingly, both when using AIP and CDP, the processing time constantly increases with the number of states (Figure 5.13a). However, AIP presents higher processing times, and the difference between AIP and CDP increases with the number of sequence states. In particular, AIP moves from 200 microseconds required to process sequences of 2 states to 800 microseconds for processing sequences of 5 states, whereas CDP moves from 100 to less than 400.

Results are different when a single selection policy is adopted (Figure 5.14). AIP scales better, and its processing times move from 70 to less than 90 microseconds. This is a result of the optimizations described in Section 5.3 for rules involving a `last-within` operator, which allow AIP to minimize duplication of sequences when new events arrive.

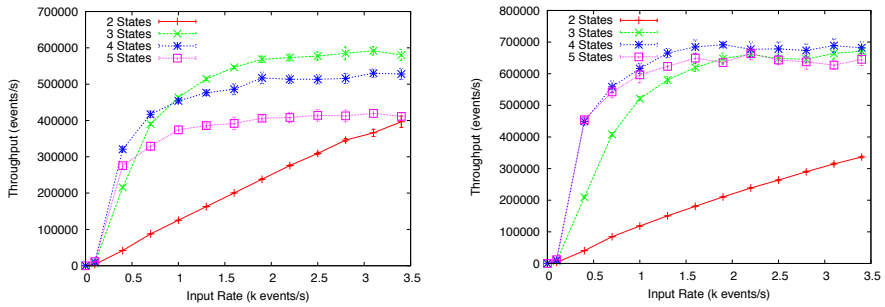
Also in this case, however, CDP performs better. Most significantly it exhibits a processing time of about 50 microseconds which seems to be independent from the number of states. When a single selection

policy is adopted, a single event has to be selected from each column. Accordingly, increasing the number of columns has a negligible impact on performance.



Average Processing Time (Microseconds)

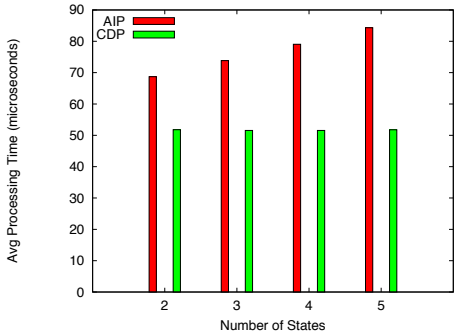
(a) Average Processing Time



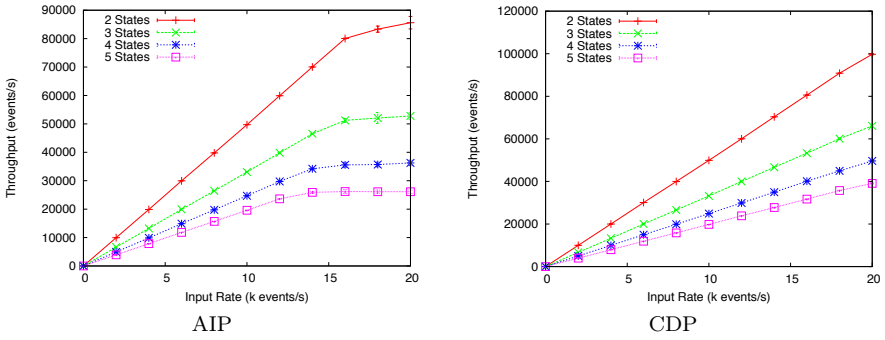
(b) Throughput

Figure 5.13: Number of states in sequences (Each-Within)

As for the throughput, in presence of a multiple selection policy we expect it to grow with the number of states per sequence, since the number of possible combinations of primitive events increases. This trend is confirmed when the number of states moves from 2 to 3, but when it grows further the throughput decreases (Figure 5.13b). This is an effect of the increased processing overhead, which results in dropping more and more input events, thus negatively affecting throughput. On the other hand, we may notice that using a multiple selection policy with long sequences represents an extreme situation. Indeed it is very hard to find real applications that asks for all possible combinations



(a) Average Processing Time

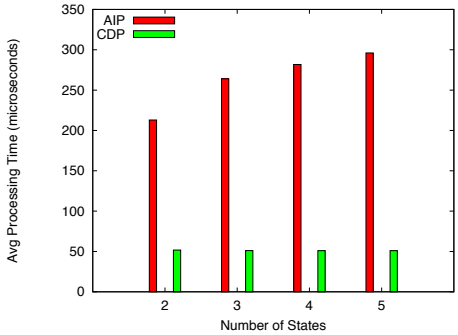


(b) Throughput

Figure 5.14: Number of states in sequences (Last-Within)

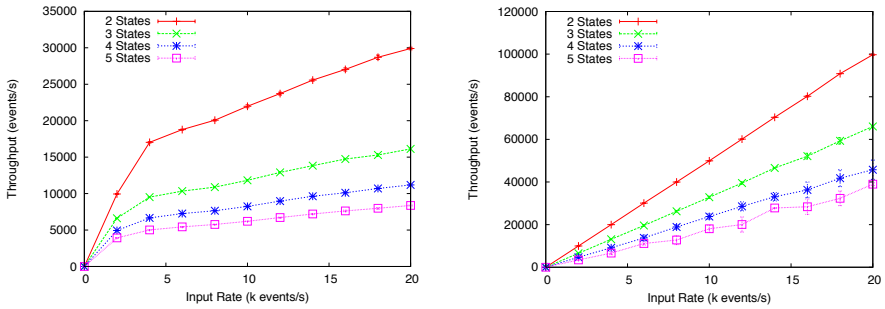
of long series of events. Such an application, in fact, would need to process a number of composite events (the output of the CEP system, as in Figure 5.13b) much higher than the number of primitive events captured. On the contrary, we expect applications to use a CEP system to filter out most events, while returning only a small number of useful information. However, even under this stressing conditions, T-Rex can handle more than 1000 input events per second while producing up to 600000 composite events with the AIP algorithm and more than 700000 with the CDP algorithm.

Different considerations hold when the `last-within` operator (i.e., a single selection policy) is used. In this case the throughput grows linearly



Average Processing Time (Microseconds)

(a) Average Processing Time



(b) Throughput

Figure 5.15: Number of states in sequences (First-Within)

until at least 12000 input events per second (Figure 5.14b). This means that our default queue of 100 elements allows T-Rex to easily handle such large input traffic. We also observe that increasing the length of sequences makes the throughput curve diverge from its linear behavior (i.e., T-Rex dropping events) sooner when using AIP: this is due to the slight increase in processing time. The same in not true for CDP, which allows T-Rex to process all events without dropping them. Finally, we notice that the longer the sequences the lower the throughput. This is easily explained by remembering that a single selection policy inhibits duplication of sequences while longer sequences require more input events to arrive to an accepting state.

In this first scenario we also tested the usage of the `first-within` operator (Figure 5.15). Ideally, this case should resemble the one using the `last-within` operator, with a low processing time and the ability of accepting a large number of input events before starting dropping them. This is true when the CDP algorithm is adopted.

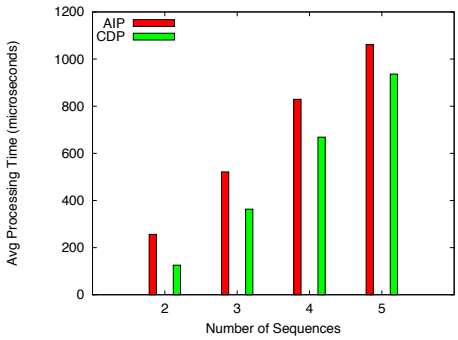
On the contrary, when considering AIP, we notice some differences: the processing overhead is less than that registered when using a multiple selection policy, but it is higher than that registered when using the `last-within` operator. This can be explained by remembering what we observed at the end of Section 5.3: the `last-within` operator, in absence of consumption clauses, allows greater optimizations than the `first-within` operator, with less sequences that need to be created and more that can be deleted each time a new event arrives. As a result, the performance of AIP when processing rules using the `first-within` operator is somehow in the middle between the performance measured when using the `each-within` and those measured when using the `last-within`. This trend appears in all the scenarios we tested. As a consequence we will omit the analysis of the `first-within` policy in the following discussions.

Number of sequences in rules

Figure 5.16 and Figure 5.17 show how the system reacts when the number of sequences composing each rule varies. When a multiple selection policy is used (Figure 5.16), we observe a linear increase in the processing time, both in AIP and in CDP: indeed, the system needs to perform additional work to combine sequences together and to produce the higher number of composite events that derives from this combination. By looking at the throughput graph, it becomes even more evident how the number of sequences influences the work to be done: more sequences means more possibility to combine events, and hence more composite events to generate.

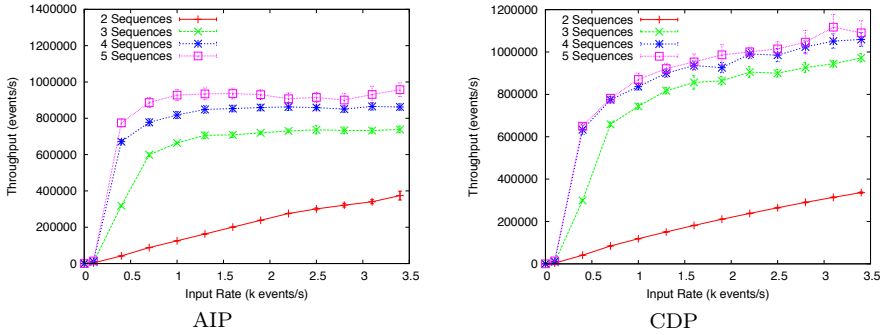
The same is not true when a single selection policy is adopted (Figure 5.17). The processing time of AIP registers a moderate increase (from about 70 microseconds to 100 microseconds), while CDP remains constant. At the same time, when the number of sequences increases, the throughput decreases. Indeed, with the same fraction of events captured by each state in a rule, the system detects about the same number of valid sequences, independently from the number of sequences composing each rule, but more input events are needed to generate a composite one when the number of sequences per rule increases.

Also in this case, it is important to observe how the use of a multiple



Average Processing Time (Microseconds)

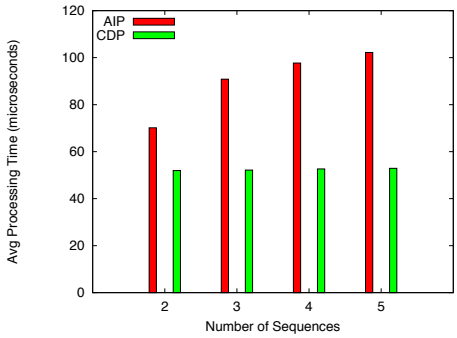
(a) Average Processing Time



(b) Throughput

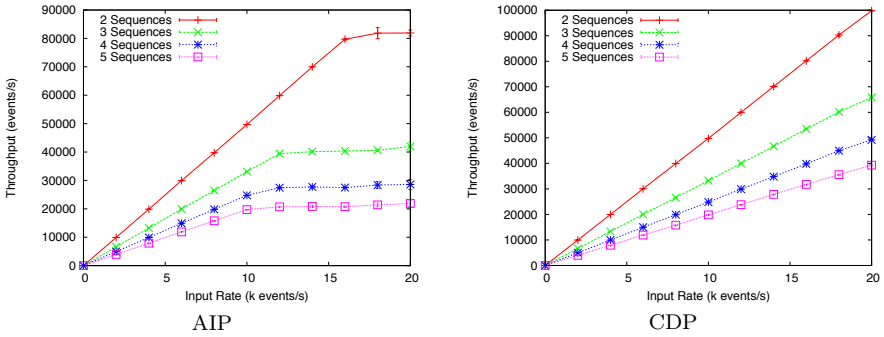
Figure 5.16: Number of sequences in rules (Each-Within)

selection policy combined with a large number of sequences represents an unrealistic scenario, which we tested only to stress the system. Even in this extreme case T-Rex can handle about 1000 input events, producing more than a million composite events per second. This is a good result, especially if we consider that not only the processing of input events, but also the generation of composite events (and their internal data structure) demands for computational resources.



Average Processing Time (Microseconds)

(a) Average Processing Time



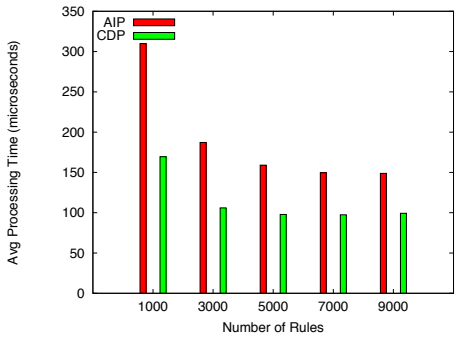
(b) Throughput

Figure 5.17: Number of sequences in rules (Last-Within)

Number of rules

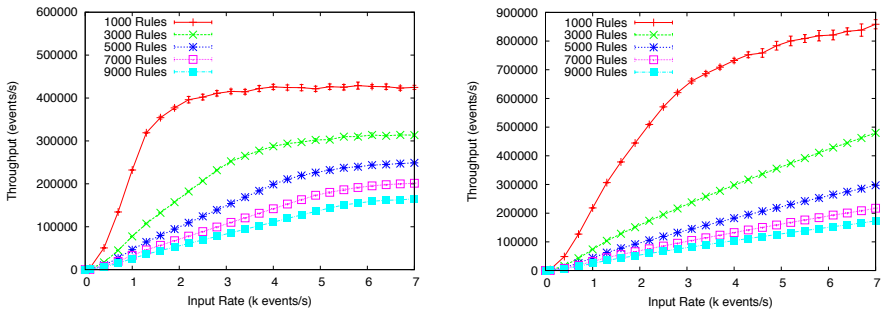
Figure 5.18 and Figure 5.19 study how T-Rex reacts when the number of deployed rules increases while keeping fixed the number of them triggered by each incoming event (10). This is an important case since we envision real deployments where a single CEP engine serves different applications, each interested in different kinds of events.

In this scenario, when a multiple selection policy is adopted (see Figure 5.18) the number of generated events decreases with the number of rules; indeed, each rule receives fewer events and thus has fewer chances to combine them into valid sequences. This effect is emphasized by the fact that the rate of events entering each rule decreases when we deploy



Average Processing Time (Microseconds)

(a) Average Processing Time

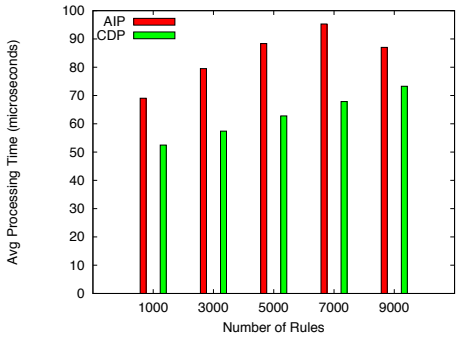


(b) Throughput

Figure 5.18: Number of deployed rules (Each-Within)

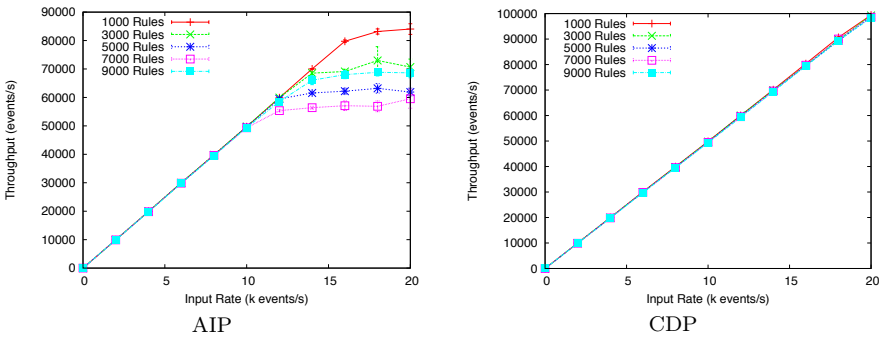
a higher number of rules, and thus the probability of violating timing constraints (and thus canceling existing sequences) becomes higher. This also affects the processing time, that slightly decreases when the number of rules increases, both for AIP and CDP.

The same is not true when a single selection policy is adopted (Figure 5.19). In this case each rule can fire at most once for each incoming event, and the effect of a growing number of rules only negatively affects the processing time, with more sequences to consider at each step. In absence of dropped events the output rate is not influenced by the number of rules deployed since we are considering the case in which each event always trigger the same number of rules: 10.



Average Processing Time (Microseconds)

(a) Average Processing Time



(b) Throughput

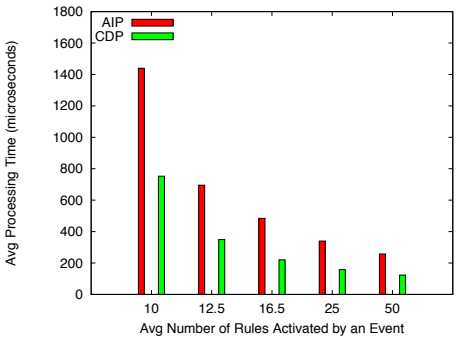
Figure 5.19: Number of deployed rules (Last-Within)

However, while CDP never drops input events, the higher processing time of AIP makes it drop events when the input rate overcomes 10000 events per second.

Number of triggered rules

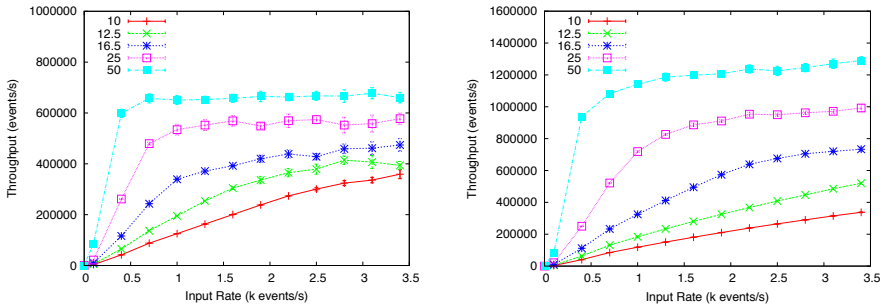
Figure 5.20 and Figure 5.21 study the behavior of T-Rex when the number of rules triggered by an incoming event grows. As expected, under these conditions, independently from the selection policy adopted, the processing time grows. Most importantly, when doubling the average number of rules involved in the processing of an input event, the processing time doubles as well: this means that T-Rex does not introduce

any additional overhead and scales linearly. Similar considerations hold for the throughput, which grows when the number of rules triggered by each event grows. Notice, once again, that this scenario is explicitly designed to stress the system: the number of output events generated per second is extremely large if compared with the input rate.



Average Processing Time (Microseconds)

(a) Average Processing Time



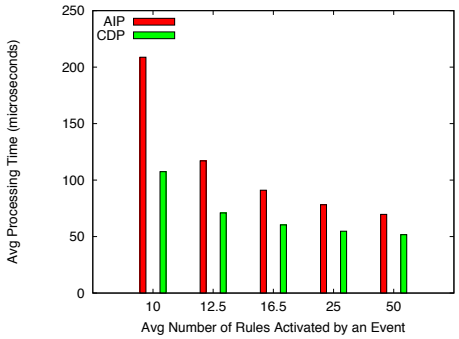
AIP CDP

(b) Throughput

Figure 5.20: Number of triggered rules (Each-Within)

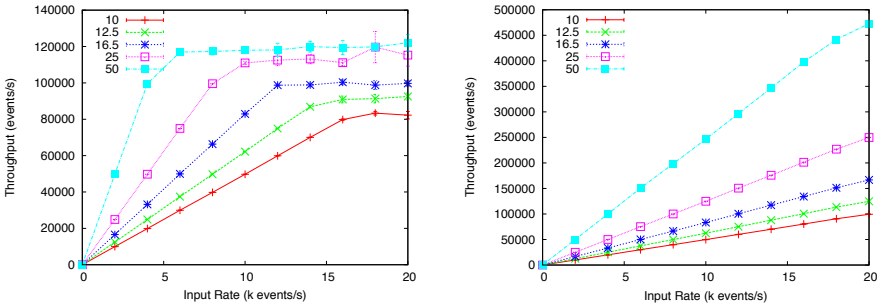
Average Windows Size

Figure 5.22 and Figure 5.23 analyze the impact of the windows size on processing. As expected, when using a multiple selection policy (Figure 5.22) both processing time and throughput grow since more events enter the window, more sequences are created and more composite events are generated. Again, scalability is good, both for AIP and CDP, since the increase in processing time is less than linear.



Average Processing Time (Microseconds)

(a) Average Processing Time



(b) Throughput

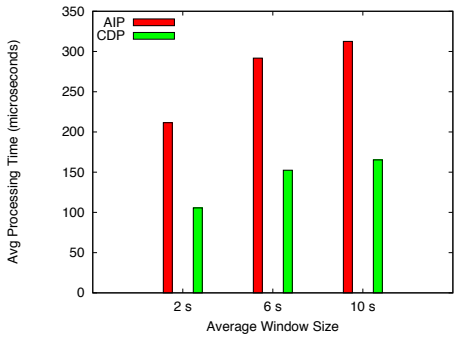
Figure 5.21: Number of triggered rules (Last-Within)

Conversely, a single selection policy is not influenced by the window size (see Figure 5.23). Indeed, to stress the system we are delivering events at a very high rate such that even with the shortest window we tested enough events arrive to complete detection.

Scalability

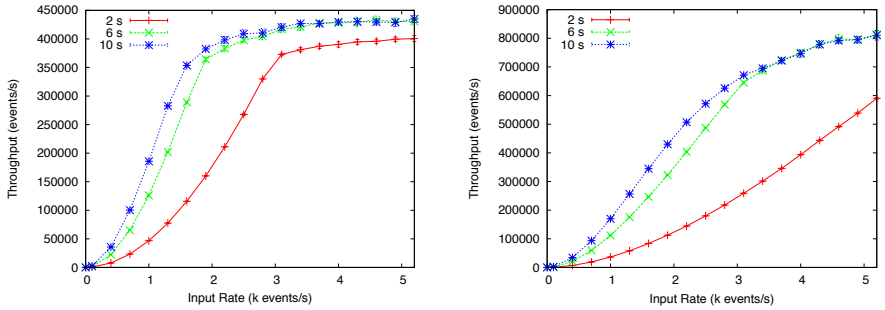
Figure 5.24 studies the impact of multi-core hardware on the performance of T-Rex. As stated in Section 5.6 and Section 5.7, both when using AIP and CDP, our system makes use of a thread pool to parallelize event processing.

Notice that T-Rex exploits multiple processing cores only to evaluate



Average Processing Time (Microseconds)

(a) Average Processing Time

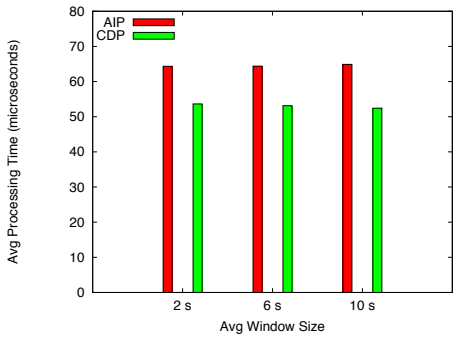


(b) Throughput

Figure 5.22: Average Windows Size (Each-Within)

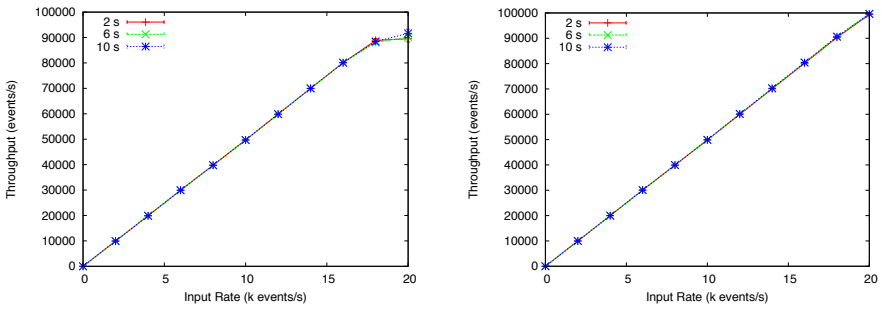
rules in parallel, while the evaluation of single rules is performed sequentially. The issue of parallelizing the processing of single rules becomes relevant in the case of extremely complex rules, involving a large number of events. We will address this problem in Chapter 6, where we propose an implementation of CDP explicitly tailored for GPUs.

To execute this test we first studied the best number of thread in the thread pool when varying the number of available cores. Then, using this value, we tested the processing time and the throughput with different cores: to limit the number of cores that T-Rex could use, we exploited the *taskset* Linux command. To increase the load of the system we adopted our default scenario with a multiple selection semantics and a



Average Processing Time (Microseconds)

(a) Average Processing Time



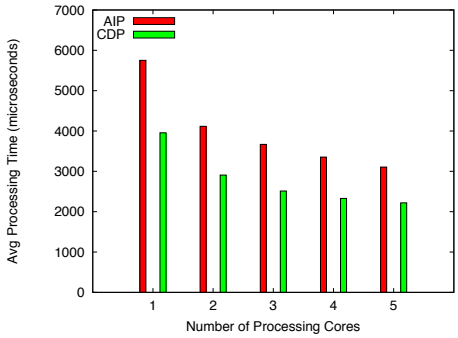
AIP CDP

(b) Throughput

Figure 5.23: Average Windows Size (Last-Within)

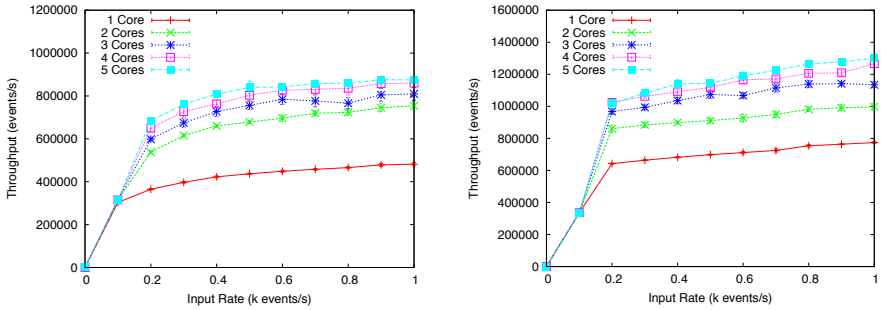
low selectivity (each event triggers 100 rules on average). Since the tests were executed on a 6 core machine, our analysis is limited to up to 5 cores (one is used by the client generating and submitting events).

What we observe is that the use of more cores brings benefits to the system. This is particularly evident when moving from 1 to 2 cores, but also moving to 3, 4, and 5 cores continues to bring benefits. Notice that each thread in the thread pool processes an equal number of rules; however, we cannot know in advance the cost for processing each rule. This implies that different threads may require different times to accomplish their tasks. Since the system waits for all threads to complete, the overall processing time is the processing time of the slowest thread. This



Average Processing Time (Microseconds)

(a) Average Processing Time



(b) Throughput

Figure 5.24: Scalability

explains why doubling the number of cores does not double the overall performance. Additionally, the filtering of events is performed using a highly efficient, but sequential counting algorithm.

Figure 5.24 shows that both using AIP and CDP, when moving from 1 to 5 cores, we reduce the processing time by a half and we almost double the maximum throughput.

Use of negation

Figure 5.25 investigates the impact using negation in rules. In particular, we use Rule R8 below.

```
Rule R8
define    Fire(area: string, measuredTemp: double)
from      Temp(area=$a and value>45) and
          not Rain(area=$a) within 5 min. from Temp
where     area=Temp.area and measuredTemp=Temp.value
```

To maximize the generation of events we operated as for Rule R1 and R3, deploying 1000 different rules with the structure of Rule R8 but defining 10 different composite events ($\texttt{Fire}_1$, defined from $\texttt{Temp}_1$ and $\texttt{Rain}_1$, ..., $\texttt{Fire}_{10}$, defined from $\texttt{Temp}_{10}$ and $\texttt{Rain}_{10}$) and asking for a different minimum temperature in $\texttt{Temp}_x$ events (from 1 to 100). Then we studied the behavior of T-Rex when changing the percentage of $\texttt{Rain}_x$ events w.r.t. all incoming events. Since the input rate of events was particularly high, we decided to adopt a small window of 1 second, meaning that $\texttt{Rain}_x$ events become invalid after 1 second.

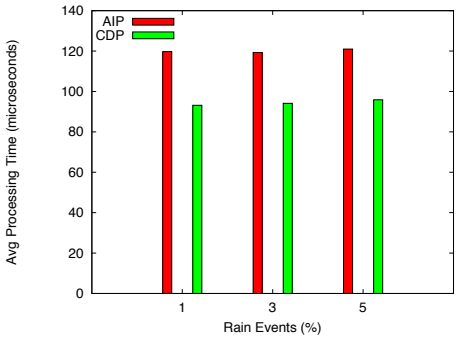
What we observe is that the processing time is not influenced by the percentage of $\texttt{Rain}_x$ events. Indeed the system stores all $\texttt{Rain}_x$ events received in the last second and checks if any has been stored when receiving a valid $\texttt{Temp}_x$. When using CDP this operation can be performed more efficently, taking advantage of the algorithms implemented for lookup in columns.

The percentage of negative events strongly influences the throughput, with fewer composite events captured when the percentage of $\texttt{Rain}_x$ events grows. As a final note we observe how the presence of negation increases the confidence interval, meaning that different runs have a high probability to produce different results. Indeed, when using a negation the order in which events are received can significantly influence the output rate.

Use of the Consuming clause

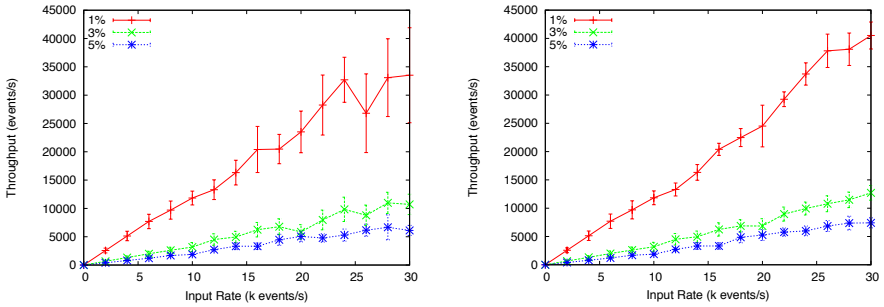
Finally, Figure 5.26 and Figure 5.27 study the impact of adding a consuming clause to our default scenario. In particular, since the default scenario defines a sequence of two states, we test the performance when asking the consumption of the events participating in the first state.

When adopting a multiple selection policy (Figure 5.26) the processing time decreases when adding the consuming clause, both for AIP and CDP. Indeed, in the multiple selection case a large number of partial results is generated during processing (automata for AIP and partial sequences for CDP). Adding a consuming clause helps the system remove some of them, i.e., those using a consumed event. Also note how the presence of a consuming clause greatly reduces the number of generated events.



Average Processing Time (Microseconds)

(a) Average Processing Time



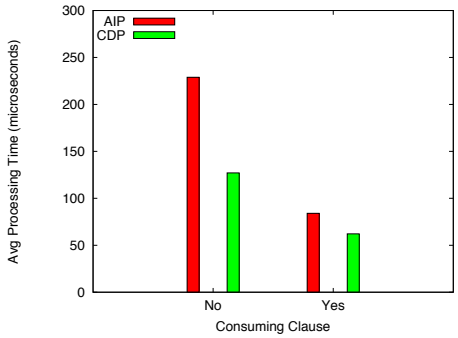
AIP CDP

(b) Throughput

Figure 5.25: Use of Negation

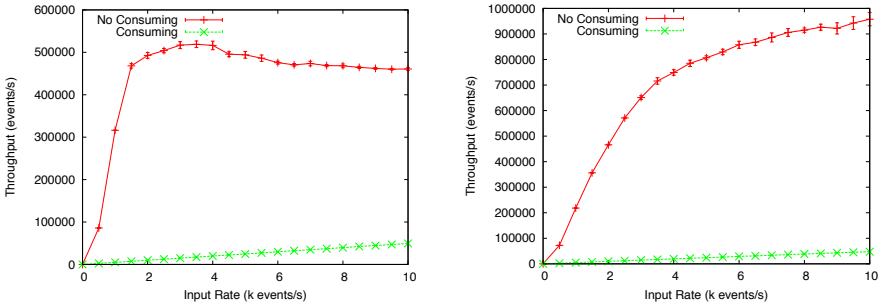
When adopting a single selection policy (Figure 5.27), instead, the number of generated events is not seriously affected by the presence of a consuming clause. This is because, even when consuming one or more events, it is highly probable to find another event of the same kind within the required time window that allows the system to produce a composite event.

In this case adding a consuming clause adds some overhead to AIP, which cannot delete sequences as new events arrive, as explained at the end of Section 5.3. This is also visible in the throughput graph: adding the consuming clause makes AIP drop input events earlier. On the contrary, CDP is not affected by this problem: indeed, to satisfy



Average Processing Time (Microseconds)

(a) Average Processing Time



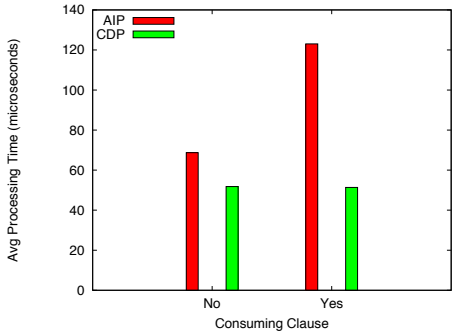
(b) Throughput

Figure 5.26: Use of the Consuming clause (Each-Within)

the constraints imposed by a consuming clause, it simply needs to remove consumed events from existing columns after a composite event has been generated. As we have already seen, columns are explicitly designed to make this operation efficient, thus reducing its impact on the performance of the system.

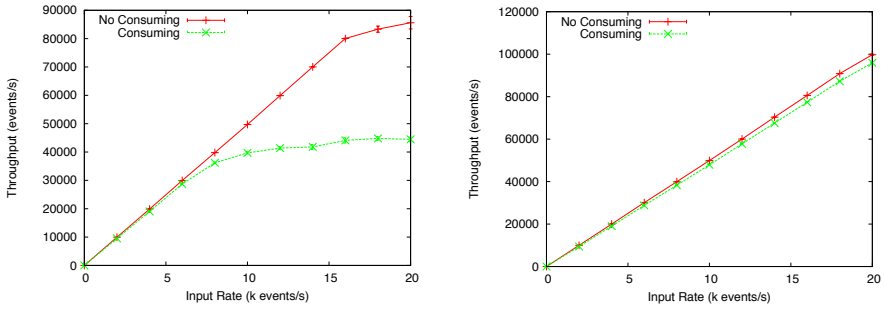
5.9 Conclusions

In this chapter we introduced T-Rex, a CEP middleware explicily designed to support TESLA rules. T-Rex proved to be capable of efficiently processing large volumes of input events even with thousands of



Average Processing Time (Microseconds)

(a) Average Processing Time



(b) Throughput

Figure 5.27: Use of the Consuming clause (Last-Within)

rules deployed.

We presented and evaluated two different processing algorithm: the first one, AIP, processes events incrementally, as they enter the system, and stores partial results in data structures based on automata. The second one, CDP, takes the opposite approach. It simply stores received events into columns and delays processing until a candidate terminator is detected.

In Section 5.8 we compared T-Rex with Esper, one of the most widely used commercial solution for CEP, known for its expressiveness and efficiency. Our analysis show that T-Rex performs better in all the different scenarios we tested. Moreover, using synthetic workloads we analyzed

the behavior of T-Rex in a large number of situations, showing how it provides good performance even with extremely challenging scenarios.

Interestingly, we measured significantly better performance with CDP w.r.t. AIP, in almost all the scenario we tested. In particular, CDP is better suited to process rules adopting a single selection policy and scales better when the complexity of rules increases, e.g., when the length of sequences increases, a negation or a consumption clause is added.

It is worth mentioning that most existing CEP systems explicitly designed to detect temporal patterns of events are based on automata, or more in general on incremental processing. This is true both for research prototypes coming from the academia (e.g., [98, 100, 11]) and for mature systems coming from the industry (e.g., [112]).

It is still an open issue to understand if approaches similar to CDP can be extended to other systems, possibly including different operators w.r.t. TESLA.

Another important aspect of CDP that its processing is strongly data parallel. Accordingly, it can be efficiently implemented in parallel hardware, to increase its performance in presence of complex rules. In Chapter 6 we will discuss its implementation on modern GPUs based on the CUDA architecture [203], with a detailed evaluation of its performance.

6 Using GPUs for Low Latency Event Processing

6.1 Introduction

In Chapter 5 we presented T-Rex in details, focusing on the processing algorithms implemented for the detection of composite events. We have seen how T-Rex leverages the presence of multiple processing cores to evaluate different TESLA rules in parallel.

On the other hand, each single rule is processed by one core only, using a strictly sequential algorithm. While this is reasonable for many application scenarios, there are situations that involve very complex rules, which require the analysis of a huge number of events. In this chapter we address this issue, by investigating how the evaluation of a TESLA rule can be implemented on modern GPUs based on the CUDA architecture.

While our analysis is based on TESLA rules, we focus on key operators offered by almost all existing rule languages, namely sequence detection, parameter evaluation, and aggregate computation, thus we believe that our analysis and the solutions presented here can be easily extended to other engines as well.

As discussed in Chapter 5, T-Rex offers two different processing algorithms, namely AIP (Automata-Based Incremental Processing) and CDP (Column-Based Delayed Processing). The first one processes events incrementally as they arrive, storing the results of intermediate computation in form of automata. The second one postpones all the processing until a terminator enters the engine.

As we have seen in Chapter 4, CUDA offers a lot of computational power but only for data parallel algorithms on pre-allocated memory areas. Unfortunately, the AIP algorithm does not fall in this category. Each automaton differs from the others and requires different processing, while new automata are continuously built and deleted at runtime. For these reasons, we implemented the AIP algorithm on the CPU only, while we implemented the CDP algorithm both on the CPU and GPU.

We compared the performance of the three resulting engines under various workloads, to understand which aspects make the choice of each

architecture more profitable. Our study shows that the use of GPUs brings impressive speedups in all scenarios involving complex rules, when the processing power of the hardware overcomes the delay introduced by the need of copying events from the main to the GPU memory and vice versa. At the same time, multi-core CPUs scale better with the number of rules deployed in the engine.

The rest of the chapter is organized as follows: Section 6.2 recalls the main features of the CDP algorithms, focusing on the main functions we analyze here, i.e., sequence detection, parameter evaluation, and aggregates computation. Section 6.3 presents in details how the CDP algorithm has been implemented on CUDA. Section Section 6.4 evaluates its performance under different workloads, comparing it with the original implementations of T-Rex, as discussed in Chapter 5. Finally, Section 6.5 provides some conclusive remarks.

6.2 The CDP Algorithm

For ease of exposition, we first report here the main computational steps performed by the CDP algorithm to analyze received primitive events, detect patterns, and compile corresponding composite event notifications.

As said in Section 6.1, in this chapter we focus on some key operations common to almost all exisiting CEP engines: sequence detection, parameter evaluation, and aggregates computation.

6.2.1 Detecting Sequences

For each Rule R, the CDP algorithm organizes received events into columns, one for each primitive event appearing in the sequence defined by R.

To simplify the analysis, instead of keeping a flat history of all received events, each rule R organizes them into columns, one for each primitive event appearing in the sequence defined by R. As an example, the following Rule R9:

```
Rule R9
define    ComplexEvent()
from      C(p=$x) and each B(p=$x and v>10)
          within 8 min. from C and
          last A(p=$x) within 3 min. from B
```

would be translated in the columns shown in Figure 6.1. The algorithm creates three columns, each labeled with the type of the primitive events

it stores and with the set of constraints on their content. The maximum time interval allowed between the events of a column and those of the previous one (i.e., the window expressed through the `*-within` operator) are modeled using a double arrows, while additional constraints coming from parameters are represented as dashed lines.

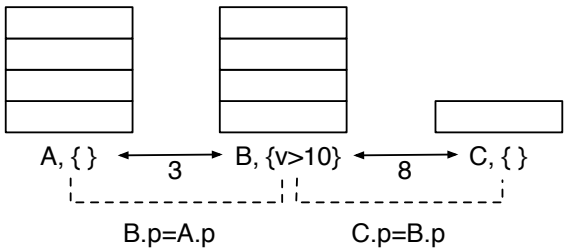


Figure 6.1: Columns for Rule R9

Processing starts when If an incoming event e matches the content constraints of a column c it is added on top of c. Detection of valid sequences starts when an event enters the last column, c_ℓ. Processing is performed column by column, from the last one to the first one, creating *partial sequences* of increasing size at each step. In particular:

- the timestamp of e is used to find the index i of the first valid element in column $c_{\ell-1}$, by looking at the time window;
- all events in column $c_{\ell-1}$ having an index $i' < i$ are deleted, since they have no chance to enter the window in the future;
- the operation is repeated for each column, considering the timestamp of the first event left in column c_k to delete old events from column c_{k-1};
- e is combined with all the events stored in column $c_{\ell-1}$ that satisfy timing constraints, parameters, and selection policies, creating partial sequences of two events;
- each partial sequence is used to select elements from the previous column $c_{\ell-2}$. The algorithm is repeated recursively until the first column is reached, generating zero, one, or more sequences including one selected event from each column;
- one composite event is generated for each sequence.

6.2.2 Computing Aggregates

As we have seen in the previous chapters, computing aggregates is one of the main operations offered by CEP engines. Often, it may involve the computation of a given function (e.g., sum, average) considering the values of a large number of events.

As mentioned in Chapter 5, CDP stores events relevant for the computation of an aggregate into a column c of a special kind. After all partial sequences have been detected, CDP extracts, for each sequence, the events in c that satisfy timing and parameter constraints and uses them to compute the desired function.

6.3 Implementing the CDP Algorithm on CUDA

In Section 5.6 we presented in details how the CDP algorithm has been implemented on the CPU. An aspect is particularly important for the following discussion: to avoid duplication of events among different rules, we save all received events in a common storage, while columns only contain pointers to the actual content of events.

Due to the different programming model, porting the CDP algorithm to CUDA is not straightforward. First of all we had to re-think the data structures used to represent columns. Indeed, memory management is a critical aspect in CUDA: the developer is invited to leverage the fact that CUDA assembles together (and computes in a single memory wide operation) concurrent memory access to contiguous areas from threads having contiguous identifiers in the same warp. Using pointers to events inside columns, as in the implementation on CPU, would lead to memory fragmentation, making it impossible to control memory accesses from contiguous threads. Accordingly, in the CUDA implementation columns do not hold pointers to events but copies of them.

Since GPU memory has to be pre-allocated by the CPU (and allocation has a non-negligible latency), we implemented each column as a statically allocated circular buffer. Moreover, we choose to perform some operations, those that would not benefit of a parallel hardware, directly on the CPU, which keeps its own copy of columns. In particular, when an event e enters the engine and matches a state s for a rule r, it is added to the column for s in main memory. Then, a copy of the event to the GPU memory is issued asynchronously: this means that the CPU can go on without waiting for the copy to end. If e is a terminator for r, the CPU uses the information about windows to determine which events has to be considered from each column. We delegate this operation to the CPU since it requires to sequentially explore columns in order from

the last one (the result of the computation on a column is needed to start the computation on the previous one), while computation on each column can be efficiently performed using a sequential algorithm, i.e., a binary search. Once this operation has ended, the CPU invokes the GPU to process the relevant events from each column. In particular, we implemented two different kernels, optimized respectively for multiple selection and single selection policies.

6.3.1 Multiple selection policy

When using a multiple selection policy to process a column c, each partial sequence generated at the previous step may be combined with more than one event in c. Our algorithm works as follows:

- it allocates two arrays of sequences, called seq_{in} and seq_{out}, used to store the input and output results of each processing step. Sequences are represented as fixed-size arrays of events, one for each state defined in the rule;
- it allocates an integer *index* and sets it to 0;
- at the first step seq_{in} contains a single sequence with only the last position occupied (by the received terminator);
- when processing a column c, a different thread t is executed for each event e in c and for each sequence seq in seq_{in};
- t checks if e can be combined with seq, i.e., if it matches timing and parameter constraints of seq;
- if all constraints are satisfied, t uses a special CUDA operation to atomically read and increase the value of *index*. The read value k identifies the first free position in the seq_{out} array: the thread adds e to seq in position c and stores the result in position k of seq_{out};
- when all threads have finished, the CPU copies the value of *index* into the main memory and reads it;
- if the value of *index* is greater than 0 it proceeds to the next column by resetting the value of *index* to 0 and swapping the pointers of seq_{in} and seq_{out} into the GPU memory;
- the algorithm continues until *index* becomes 0, or all the columns have been processed. In the first case no valid sequence has been detected, while in the second case all valid sequences are stored in seq_{out} and can be copied back to the CPU memory.

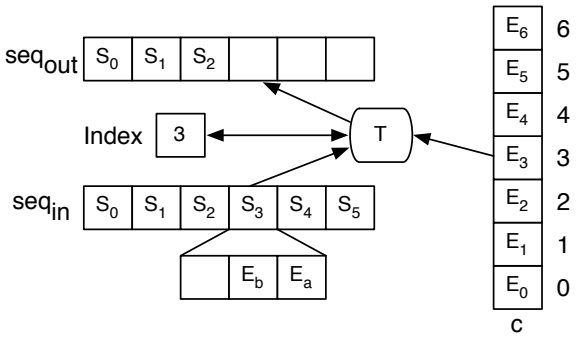


Figure 6.2: CDP Algorithm on CUDA (Multiple Selection Policy)

To better understand how the CDP algorithm works, consider the example in Figure 6.3.1. It shows the processing of a rule R defining a sequence of 3 primitive events. Two columns have already been processed resulting in six partial sequences of two events each, while the last column c to be processed is shown in figure. Since there are 6 sequences stored in seq_{in} and 7 events in c, our computation requires 42 threads. Figure 6.3.1 shows one of them, thread T, which is in charge of processing the event E_3 and the partial sequence S_3. Now suppose that E_3 satisfies all the constraints of Rule R, and thus can be combined with S_3. T copies E_3 into the first position of S_3; then, it reads the value of *index* (i.e., 3) and increases it. Since this operation is atomic, T is the only thread that can read 3 from *index*, thus avoiding memory clashes when it writes a copy of S_3 into the position of index 3 in seq_{out}.

In our implementation, we use different thread blocks to process different elements in seq_{in}: this way all thread in a block share the same element in seq_{in}. Similarly, threads with contiguous identifiers are used to process contiguous positions in the column: this increases the performance of memory access, since the hardware can combine operations issued by different threads.

6.3.2 Single selection policy

When using a single selection policy to process a column c, each partial sequence generated at the previous step can be combined with at most one event in c. Accordingly, the processing algorithm is changed as follows: instead of a single *index*, we define an array of *indexes*, one for each sequence in seq_{in}. As in the case of a multiple selection policy, each

thread checks whether it is possible to combine an event e from column c with one partial sequence in seq_{in}. Consider the example in Figure 6.3.2.

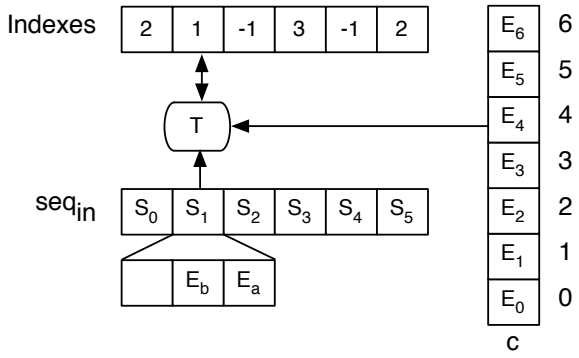


Figure 6.3: CDP Algorithm on CUDA (Single Selection Policy)

Thread T is in charge of processing event E_4 and sequence S_1. Now assume that E_4 satisfies all constraints for Rule R. Assume the `last-within` operator is adopted. In this case thread T calls an atomic function that stores in the index associated to the sequence S_1 the maximum between the currently stored value (i.e., 1) and the index of E_4 in the column (i.e., 4). All positions in *indexes* are initially filled with the value -1. When all threads have completed their processing, each position of *indexes* contains the index of the last event in column c that can be combined with the corresponding sequence, or -1 if no valid events have been found. At this point, seq_{in} is updated by adding the selected events inside partial sequences, and by removing the sequences with a negative value in the *index* array. Notice that in this case we directly update seq_{in} with no need to define an additional array for output. The same algorithm also applies to the case a `first-within` operator is used: it is sufficient to modify the initial value in *indexes* and to compute the minimum index instead of the maximum.

6.3.3 Computing Aggregates on CPU and CUDA

To compute aggregates on the CPU we navigate through all events stored for it, selecting the relevant ones (i.e., those that match constraints on parameters, if any) and calculate the aggregate function on their value. With CUDA, the process is performed in parallel by using a different thread to combine couples of stored events. All threads in a block cooperate to produce a single value, using shared memory to store the partial results of the computation.

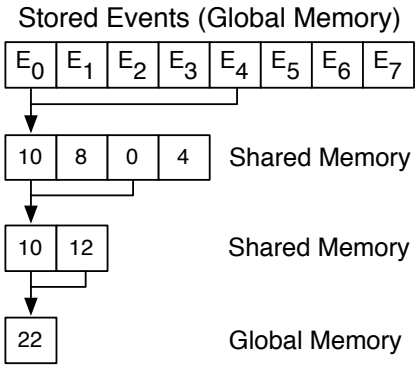


Figure 6.4: Computing aggregates on CUDA

To understand how the algorithm works, consider Figure 6.4. Assume for simplicity that 8 events (E_0, .., E_7) have been stored for the aggregate, and that the function to compute is the sum of a given attribute *att* for all events satisfying some constraints. We use a single block composed of 4 threads (t_0, .., t_3). Each thread t_x reads two events stored in the global memory, in position x and $x + 4$: for example, thread t_0 reads events E_0 and E_4, as shown in Figure 6.4. This way, contiguous threads read contiguous positions in global memory. Each thread t_x checks whether the event e it is considering satisfies the constraints defined by the rule. If e does not satisfy them, it participate in the aggregate with a value of 0 (neutral for the sum), otherwise the value of *att* is used. Then, each thread t_x sums the values computed for its events and stores the result into the shared memory, in position x. The rest of the computation is performed in the shared memory: at each step the size of the array is halved, with one thread computing the sum of two values. This way, at each step, also the number of active threads is halved. At the last step, only one thread (t_0) is active, which stores the final results into the global memory, where the CPU retrieves it.

Since CUDA limits the number of threads per block to 512, each block can analyze up to 1024 events, producing a single value. If the number of events stored for the aggregate is greater we use more blocks and thus more than one result is produced. In this case, we apply the algorithm again, by considering the partial values produced at the previous step as input values for the next step, until a single value is produced.

6.3.4 Managing Multiple Rules with CUDA

As we said, in the current CUDA implementation, columns include copies of events. While this choice is fundamental to obtain good performance, it consumes a lot of GPU memory. The amount of memory actually required for each rule depends from the number of states it defines, from the number of aggregates it includes, from the maximum size of each column, and from the size of events (at least the part relevant for the rule). In our tests, complex rules with a relevant history of one million events consume up to 100MB of GPU memory. This means that current GPU models, which usually include from 1 to 2GB of memory, can manage 10 to 20 of such complex rules. Notice that this represents a worst case scenario, since it is unusual for a rule to involve so many events and since we do not consider the (common) case of rules sharing events of the same type (i.e., sharing the same columns). Moreover, our code statically allocates columns as circular buffers, which we had to size big enough for our worst scenario. A better but more complex solution would be to allocate GPU memory in pages.

In any case, our CUDA implementation checks whether the GPU memory is sufficient to store all deployed rules. If it is not, information about rules is kept in the main memory and copied to the GPU only when a terminator is detected, before starting sequence detection and aggregates computation. While this solution removes the limitation on the maximum number of rules that the engine can manage, it introduces some extra overhead due to the additional copies to the GPU memory. In Section 6.4 we analyze how this impacts performance.

6.3.5 Use of Multi-Core CPUs

As already specified in Chapter 5, T-Rex leverages multi-core CPUs to process different rules in parallel, both when using the AIP algorithm and the CDP algorithm. As said, since creating new threads may be an expensive operation, we use a thread pool of fixed size. When a new event e enters the engine, a copy of e is sent to each thread, then we wait until all threads finish their processing to collect results (i.e., generated composite events).

During our analysis, we also tried to combine the use of multiple threads and the use of CUDA, by launching different CUDA kernels in parallel. Unfortunately, we did not achieve any relevant improvement in performance. Indeed, each kernel uses a number of blocks that is large enough to exploit all GPU hardware resources. Accordingly, different kernels, even if launched by different CPU threads in parallel, actually

execute sequentially on the GPU. A multi-threaded solution, however, may become useful in presence of multi-GPU configurations, to enable concurrent processing of complex rules on different GPUs, under control of a different CPU thread.

Finally, we decided not to use different CPU cores to parallelize the processing of a single rule. Indeed, real applications often deal with a large number of deployed rules; moreover complete systems implement several other task beside event processing, including marshalling/unmarshalling of notifications and network communication. Therefore, they easily exploit all available cores by processing different rules in parallel. Our approach simply demands most complex computations (if any) to the GPUs, using them as co-processors.

6.4 Evaluation

As we have already seen in Chapter 5, evaluating the performance of a CEP engine is not easy: even when considering a limited number of operators the processing time strongly depends from the workload. Unfortunately, to the best of our knowledge, there are no publicly available workloads coming from real deployments. The Event Processing Technical Society [190] is currently promoting a survey to understand how event processing technologies are used, but there are no results available, yet.

Accordingly, we decided to follow the same path as in Chapter 5 and use synthetic workloads, generating a large number of them to explore the parameter space as broadly as possible and we present here the most notable results, emphasizing the aspects that influence the behavior of our implementations more significantly.

We defined a default scenario in which only three types of primitive events: `A`, `B`, and `C`, exist, each one including three integer attributes. We assume that: (*i.*) primitive events are uniformly distributed among these three types; (*ii.*) their attributes are uniformly distributed in the interval $[1, 50000]$; and (*iii.*) they enter the engine one at each clock tick and they are timestamped by the engine with such clock tick. We also assume that the following Rule R10 has to be processed:

```
Rule R10
define    CE(att1: int, att2: int)
from      C(att=$x) and last B(att=$x)
          within 100000 from C and
          last A(att=$x) within 100000 from B
where     att1=$x and
```

```
att2=Sum(A(att=$x).value within 100000 from B)
```

It defines a sequence of three states, one for each event type, joined by a constraint on the value of attribute `att`, which requires an aggregate to be computed. This is in line with several real world cases. As an example, stock monitoring applications usually: (i) detect trends (by looking at sequences of events); (ii) filter events that refer to the same company (using parameter constraints); (iii) combine values together (using aggregates). Similarly, a monitoring application may use rules very close to Rule R10 to detect fire.

Notice that Rule R10 has a default window size of 100000 timestamps (i.e., clock ticks). Again, this is in line with several real world cases that require a lot of events to be processed before detecting a relevant situation. As an example, in stock monitoring applications a lot of events enter the engine for each second, each referring to a different company. All of them have to be considered to detect which stock (i.e., company) satisfies the trend captured by each rule. Similarly, to detect fire the engine has to process events coming from the entire network (i.e., a lot of events) to find the area where smoke and high temperature occur.

During our tests, we initialize the system by submitting a number of events equal to the window size. At this point the number of events entering the engine balances the number of events discarded by timing constraints and we start our measures, which involve submitting other 100000 primitive events and calculating the average time required by the different algorithms to process each of them. Since in our defaut scenario all primitive events entering the engine are captured by the deployed rule, what we are measuring is the average time needed by a rule r to process an event e that is relevant for r. We are ignoring the time needed to select the set of rules interested in e since we found it to be negligible, even with a large number of deployed rules.

Notice how this default scenario significanlty differs from the one presented in Chapter 5. Here we are indeed focusing on a single rule that demands large processing resources. We believe the two analyzed workloads to be complementary. In real world systems, we expect both a large number of simple rules and a few number of very complex, resource demanding rules. As we will see in the following of this section, the former are not suited to be processed on a GPU, while the second ones can receive greate advantages from the use of parallel hardware. This suggests that GPUs and CPUs can work side by side, each one on a sub-set of deployed rules, to provide the best architecture and processing algorithm for each of them.

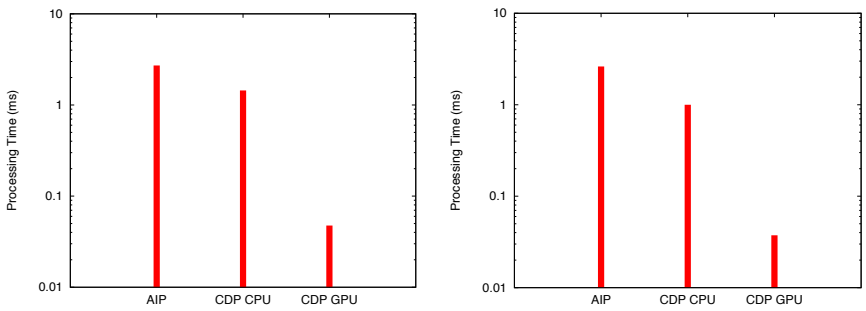
Starting from the default scenario above we measured the performance

of our algorithms when changing the following parameters: (*i.*) the number of primitive events in rules (i.e., the number of states in relevant sequences), (*ii.*) the size of windows, (*iii.*) the number of values allowed for each attribute, (*iv.*) the number of aggregates defined in each rule, and (*v.*) the number of rules deployed in the engine. Notice also that our default scenario uses the aggregate to define a value inside the generated complex event `CE`, but it does not use it to filter out sequences. We also explored the other case by using a slightly different rule and varying the percentage of sequences filtered out by the aggregate after detection. Finally, since the selection policy may significantly influence processing time, all tests have been repeated twice, once using the `last-within` operator and once using the `each-within` operator.

Tests were executed on the same reference hardware adopted in Chapter 5 and Chapter 4: AMD Phenom II machine, with 6 cores running at 2.8GHz, and 8GB of DDR3 RAM. The GPU was a Nvidia GTX 460 with 1GB of GDDR5 RAM. We used the CUDA runtime 4.0 for 64 bit Linux platforms.

Default scenario

Figure 6.5 shows the processing time measured in the default scenario. If we consider the two algorithms running on the CPU, we observe that CDP performs slightly better than AIP, independently from the selection policy. More interesting is the comparison of the CPU vs. the GPU running the same CDP algorithm. In such scenario the use of the GPU provides impressive speedups: more than 30x with a multiple selection policy and more than 25x with a single selection one.



(a) Multiple Selection (b) Single Selection

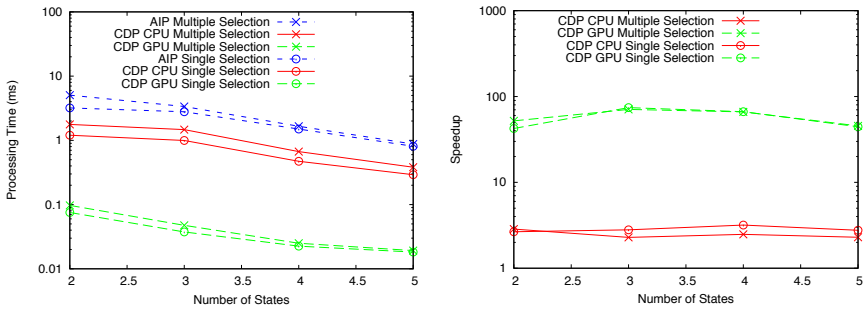
Figure 6.5: Default Scenario

In all cases, we measure a relatively small difference between the results obtained with a multiple selection policy and with a single selection

policy. Indeed, the default scenario makes use of a large number of values for each attribute, making the constraints on parameters difficult to be satisfied, and thus limiting the number of valid sequences detected even in presence of a multiple selection policy.

Length of sequences

Figure 6.6 shows how the performance changes with the number of states in each sequence. In particular, Figure 6.6(a) shows the processing time of our algorithms taken separately, while Figure 6.6(b) shows the speedup of each algorithm w.r.t. AIP (the slowest one) under the same selection policy.



(a) Processing Time

(b) Speedup w.r.t. AIP

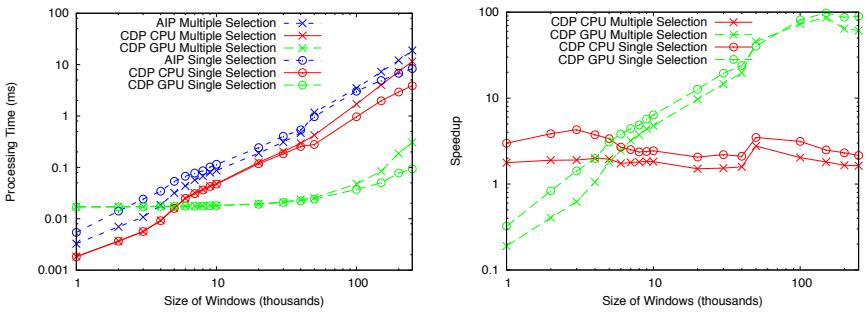
Figure 6.6: Length of Sequences

As in our default scenario, each event entering the engine is captured by one and only one state of the rule (we increase the number of event types with the length of sequences). Since we do not change the size of windows, this results in lowering the number of events to process for each state when the sequences grow. This explains why the average time to process an event decreases when the length of sequences grows (see Figure 6.6(a)). Looking at Figure 6.6(a) and comparing lines with the same pattern (i.e., same algorithm), we notice that, as in the default scenario, the difference when moving from the single to the multiple selection policy is limited.

Figure 6.6(b) confirms the results of the default scenario: CDP performs better than AIP but it is the usage of the GPU which provides the biggest advantages. The figure also shows that the speedup of the fastest algorithms (CDP on the CPU and GPU) w.r.t. to the slowest one (AIP) does not change significantly with the length of sequences.

Size of Windows

The size of windows is probably the most significant parameter when considering the use of GPUs. Indeed, increasing the size of windows increases the average number of events to be considered at each state. While the CPU processes those events sequentially, the GPU uses different threads running in parallel. On the other hand, there is a fixed cost to pay in using the GPU, i.e., to transfer data from the main to the GPU memory and to activate a CUDA kernel. As a result, using the GPU is convenient only when there is a significant number of events to process at each state. Figure 6.7(a) summarizes this behavior: on one hand, the cost of the algorithms running on the CPU grows with the size of windows, as expected. On the other hand, the cost of the CDP algorithm running on the GPU is initially constant at 0.017ms (it is dominated by the fixed cost associated with the use of CUDA) and it starts growing only when the number of available cores is not enough to compute events entirely in parallel. This growing is faster under a multiple selection policy, which uses more threads and produces more composite events to be transferred back to the main memory. With our default scenario, the smallest size of windows that determines an advantage in using the GPU is 4000. With a sequence of 3 states this results in considering 1333 events in each state, on average. This is an important result, since it isolates one dimension to consider when deciding the hardware architecture to adopt. If the CEP engine is used for applications whose rules need to store and process a small number of events for each state then it is better to use a CPU, otherwise a GPU is the best choice.



(a) Processing Time

(b) Speedup w.r.t. AIP

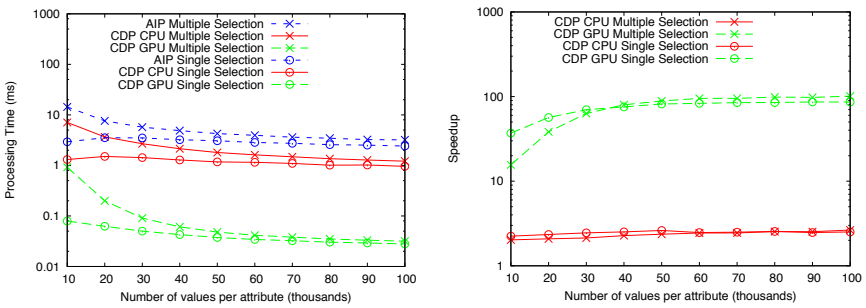
Figure 6.7: Size of Windows

Looking at Figure 6.7(b) we observe that the speedup of CDP running on the CPU w.r.t. AIP remains constant with the size of windows, while,

as already noticed, the GPU performs better and better as the size of windows grows. With a window of 250000 events and a multiple selection policy the speedup offered by the GPU is close to 100x against AIP and close to 35x against CDP. Similar values hold for the single selection policy.

Number of values

Another factor that significantly influences the performance of our algorithms is the number of primitive events filtered out by constraints on parameters, which, in our workload, is determined by the number of values allowed for each attribute. Figure 6.8(a) shows that, under a multiple selection policy, a higher number of values results in lower processing times. Indeed, when the number of values grows, less primitive events satisfy the constraint on parameter `$x` of Rule R10, which results in detecting less composite events. The GPU implementation is the one that mostly benefits from this aspect, since it has to transfer less composite events back to the main memory, through the (relatively) slow PCI-e bus.



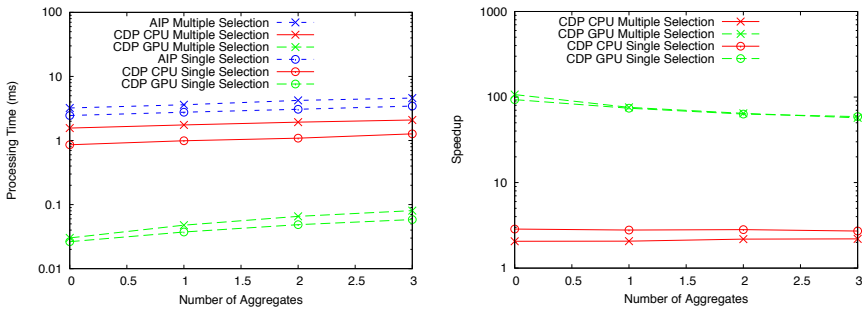
(a) Processing Time (b) Speedup w.r.t. AIP

Figure 6.8: Number of values

On the GPU, the same behavior is registered under a single selection policy. The same is not true for AIP and CDP running on the CPU, which exhibit constant processing times under a single selection policy. Indeed, when there is no need to perform (slow) memory transfers, the advantage of reducing the number of composite events detected is balanced by the greater complexity in detecting them: with a few events matching the rule constraints, more and more events have to be processed before finding the one that fires the single detection. The speedup graph (Figure 6.8(b)) confirms the considerations above. The GPU is more influenced than the CPU by a change in the number of attribute values.

Number of aggregates

Figure 6.9 shows how the number of aggregates that must be computed for a rule influences the processing time. During our analysis we kept fixed (to the value of 0.33, as in Rule R10) the probability for a primitive event of being relevant for each aggregate, independently from the number of aggregates defined in a rule. Figure 6.9(a) shows that, both under single and multiple selection policies, increasing the number of aggregates only marginally impacts performance. Indeed, in our scenario few composite events are captured at each interval, and the computation of aggregates is started only when a valid sequence is detected. The greater cost of computing 0 vs. 3 aggregates in this few cases explains why performance (marginally) degrade when the number of aggregates to calculate grows.



(a) Processing Time

(b) Speedup w.r.t. AIP

Figure 6.9: Number of aggregates

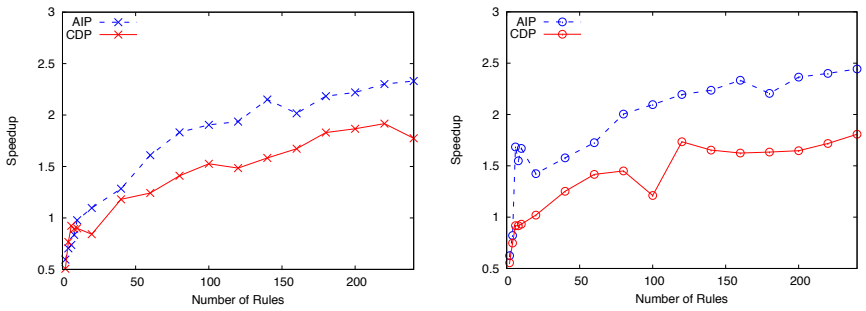
Figure 6.9(b) shows that the GPU implementation is more affected by an increased number of aggregates: indeed, even if the GPU computes the aggregates faster than the CPU, copying the primitive events to the columns storing data for aggregates increases the number of memory transfers from the main to the GPU memory, which we already noticed being a bottleneck for CUDA.

Use of multi-threading

All previous tests considered a single rule, and consequently a single thread of execution on the CPU. Here we study how performance changes when multiple rules are deployed and a pool of threads is used to process them in parallel. While some results on the use of multi-threading have already been presented in Chapter 5, in this chapter we are considering a different workload, including more complex rules. The values measured

here become relevant for a direct comparison with the results obtained by the GPU implementation.

During this analysis we are interested in capturing the (common) case when a primitive event entering the engine is relevant for a subset only of the deployed rules. Accordingly, we consider rules having the same structure of Rule R10 but using different event types, in such a way that each primitive event entering the engine is captured by 1/10 of the rules. We will see how this choice impacts the performance on the GPU in the next section, here we are interested in preliminarily studying the use of multi-threading on a multi-core CPU.



(a) Multiple Selection (b) Single Selection

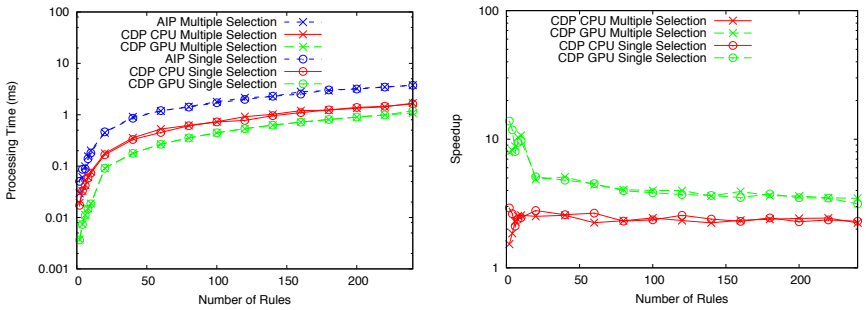
Figure 6.10: Use of multi-threading

In particular, Figure 6.10 shows the speedup of the multi-threaded CPU algorithms w.r.t. the single-threaded case, when the number of deployed rules grows. For the multi-threaded case we used a thread pool whose size was experimentally determined to best match the number of rules and the number of available cores (6 in our test system). Both AIP and CDP take advantage from the use of multi-threading when the number of rules increases: on our 6 cores CPU the maximum speedup we could achieve is slightly below 2.5x. Notice that with a small number of rules (below 10), the single-threaded implementation performs slightly better than the multi-threaded one, due to the overhead in synchronizing multiple threads.

We observe how these results are in line with what we measured in Chapter 5, where we considered an even larger number of rules, but usually involving a reduced number of processing events, and thus demanding less processing capabilities.

Number of rules

After analyzing the influence of multi-threading we are ready to test the behavior of our algorithms (running the faster, multi-threaded version of code on the CPU), when the number of rules grows. Figure 6.11(a) shows that the performance of all our algorithms increases linearly when the number of rules to process grows (a linear function is a curve in a logarithmic graph).



(a) Processing Time

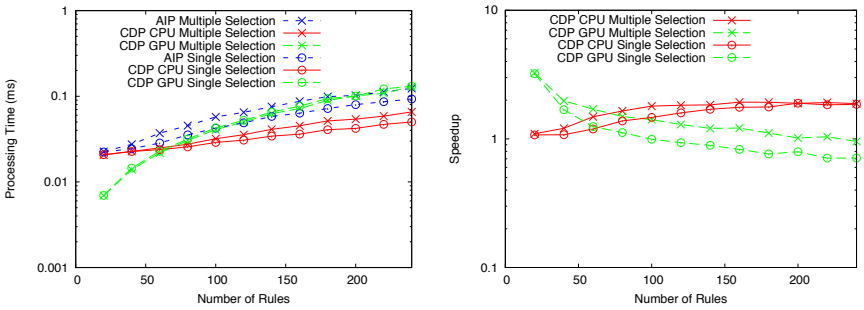
(b) Speedup w.r.t. AIP

Figure 6.11: Number of rules

More interesting is Figure 6.11(b), which compares the CDP algorithm running on CPU and GPU w.r.t. the AIP algorithm. First of all we notice that the speedup gained by using the GPU is lower than that measured in our default scenario, even with a few rules. This behavior can be explained by remembering that we moved from a scenario where each primitive event entering the engine is relevant for the only rule available, to a scenario in which the same events are relevant for only 1/10 of the rules. With a fixed size of windows and a growing number of rules, this means that each rule captures much less primitive events than in the default scenario, i.e., less events have to be stored and processed for each rule. As we observed while analyzing the influence of the size of windows on performance, this phenomenon advantages the CPU more than the GPU. Moreover, the reduced processing complexity also reduces the differences between the single and the multiple selection policy in all algorithms, as shown both by Figure 6.11(a) and 6.11(b).

A second consideration that is evident looking at Figure 6.11(b) is the quick drop of the GPU speedup when more than 10 rules are deployed into the engine. This can be explained by remembering what we said in Section 6.3.4: the giga byte of RAM available on our GPU is enough to store the events relevant for at most 10 different rules. More rules require the CDP algorithm implemented on the GPU to continuously move data

from the main to the GPU memory. The impact of this operation is evident: the speedup of the GPU significantly drops when moving from 10 to 20 rules, both under single and multiple selection policies.



(a) Processing Time

(b) Speedup w.r.t. AIP

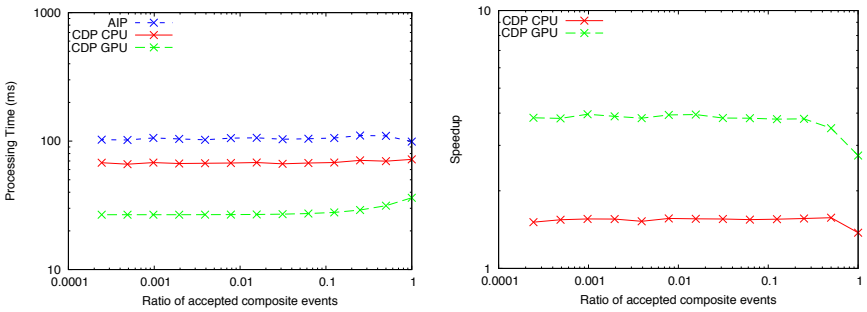
Figure 6.12: Number of rules (simple rules)

To better understand the actual limits of the CUDA architecture, we repeated the experiment above by further decreasing the number of rules influenced by each event and hence the number of events to consider when processing each rule. In particular, we considered a scenario where each primitive event entering the engine is captured by only 1/100 of the available rules. To balance the effect of this change on the number of composite events captured, we also reduced the number of possible values for each attribute to 1000. Finally, to consider the most challenging case for the GPU, we let the number of deployed rules go from 20 (twice those that may enter the GPU memory) to 250. Intuitively, this scenario is challenging for CUDA because it uses a large number of "simple rules", i.e., rules that require few events to be processed at each terminator. Figure 6.12 demonstrates that this is indeed a very tough scenario for CUDA. Independently from the selection policy adopted, the CDP algorithm running on the CPU outperforms the same algorithm running on the GPU from 50 rules and above. This is not the first time we see the CPU outperform the GPU in our tests. The same happened for a single rule when we decreased the size of windows. Even in that case the rule became "simple" as it involved few events. On the other hand, in that scenario there was a bound: the size of windows cannot become negative. Moreover, the (absolute) processing times were very small, so the (relative) advantage of the CPU was not relevant, in practice. This is not the case here. The number of rules may grow indefinitely, at least in theory, and the more rules we have the better the CPU performs w.r.t. the GPU, the longer are the (absolute) processing times. This means

that the (relative) advantage of the CPU grows and becomes relevant also in practical, absolute terms. We may conclude that handling a large number of rules represents a real issue for CUDA.

Selectivity of aggregates

In all the tests so far, we used aggregates as values for the generated complex events, but we did not use them to filter out sequences. We analyze this case here, by changing the form of rules. In particular, we relaxed the constraints on parameters and the size of windows to detect a large number of valid sequences (we adopted a multiple selection policy), while we used an aggregate to filter out part of them. This workload is challenging for the CDP algorithm running on the GPU. Indeed, detecting valid sequences and evaluating aggregates for each of them are operations performed separately and in order by the GPU. As a consequence, increasing the number of detected sequences also increases the amount of (sequential) work to be performed. Moreover, each composite event generated must be copied from the GPU to the CPU memory, introducing additional delay. To highlight this aspect, we changed the number of sequences filtered out by the aggregate, thus changing the number of events generated at the end of the process.



(a) Processing Time

(b) Speedup w.r.t. AIP

Figure 6.13: Ratio of accepted composite events

Figure 6.13 shows the results we obtained. First of all we observe how the increased complexity of the workload significantly increases the processing time of all algorithms w.r.t. the default scenario. Secondly, as we expected, this scenario is challenging for the GPU, which is still performing better than the CPU (both AIP and CDP algorithms) but with a smaller speedup w.r.t. the default scenario. On the other hand, the fact that the GPU still performs better than the CPU is a sign that the advantages it provides in reducing the processing time dominates

the cost of transferring data around. Finally, we notice that the more composite events are accepted (i.e., pass the filter of the aggregate) the lower is the advantage in using the GPU as the cost of memory transfers increases.

6.5 Conclusions

In this chapter, we studied how a CEP engine may take advantage of modern GPUs to speed up processing. In particular, we considered some of the most common operators offered by existing CEP engines, i.e., sequence detection, parameter evaluation, and aggregates computation. We described in details how our CDP algorithm, introduced in Chapter 5, can be implemented on the CUDA architecture for general purpose programming on GPUs.

We analyzed the performance of our CUDA implementation using a large number of workloads, and we compared it with our original T-Rex engine, using both the AIP and the CDP algorithm, to identify the aspects that make the use of each architecture more profitable. The large number of processing cores offered by modern GPUs makes them more suitable to process complex rules, storing and analyzing a huge number of primitive events: during our analysis we registered speedup of more than 40x on the average processing time required by a single rule. However, the programming model and hardware implementation of CUDA introduce an overhead that makes the use of GPUs inconvenient when rules are very simple. Moreover, handling a large number of rules represents a significant issue when using CUDA, since its programming model forces a large number of memory copies, thus increasing processing time.

In conclusion, considering the impressive speedups in managing complex rules together with the limitations listed above, we believe that GPUs should be used as co-processors, processing only part of the rules: the most complex ones. Simple rules can be easily handled by the CPU, making use of multi-core hardware if available.

7 Distributed Event Processing

7.1 Introduction

As already mentioned in the previous chapters, event-based applications usually involve a large number of distributed components, possibly dispersed over a wide geographical area. For this reason, beside implementing efficient processing algorithms for the detection of composite events, it becomes of primary importance for a complete CEP system to offer a good *deployment* strategy.

The deployment strategy defines *i.* how the processing load is distributed over different nodes and *ii.* how these nodes interact and communicate with each other to produce the required results and to deliver them to interested component.

The first aspect is often referred to as the *operator placement* problem: given a network of processors and a set of processing rules, it tries to find the best possible mapping of the operators defined in rules on available processing resources. Depending from the application requirements, the operator placement may pursue different goals, e.g., maximize the quality of service by reducing the latency required to deliver notifications to interested parties, or minimize the usage of network resources.

In the last few years, several works have addressed the operator placement problem, offering different solutions [145]. The problem is known to be extremely complex to solve, even for small instances with a reduced number of processors and rules. Accordingly, existing approaches are often based on approximated optimization algorithms or heuristics; moreover, they usually rely on a centralized decider, which collects all the relevant information about the network status and locally computes a solution for the problem. Only a few proposals have considered a decentralized algorithm for solving the operator placement [221]. Finally, most operator placement algorithms are studied for clustered systems (see Section 2.3.3), in which all processing nodes are colocated and well connected [143, 144]: in this setting, the operator placement problem essentially translates into a load balancing problem.

As mentioned, when considering a deployment strategy, operator placement is only half of the problem: when the processing is split among different nodes, it also becomes necessary to precisely define the proto-

cols that govern the interaction among them, specifying how rules and subscriptions are deployed, how primitive events are forwarded from the sources to the processing nodes, and how composite events notifications are finally delivered to interested sinks.

These issues are usually not considered in existing CEP systems: most of them are based on a centralized deployment, in which all the processing is performed on a single machine (e.g. [98, 48]). Even when distributed processing is allowed, the communication among nodes often requires manual configuration [123].

In this chapter, we present different processing strategies for the T-Rex CEP system. Our solutions are explicitly tailored to large scale distributed scenarios: they take into account the topology of the processing network as well as the location of event sources and their generation rates. Moreover, they do not rely on centralized deciders; on the contrary the different processing nodes autonomously decide which parts of the processing they have to execute locally and which parts can and should be delegated to other nodes. All proposed deployment strategies have been fully implemented as part of T-Rex, including the protocols needed to organize processing nodes into an overlay network, to deploy rules and subscriptions, and to deliver nodifications from sources to sinks.

We also show a preliminar evaluation and comparison of the processing strategies implemented, using the Omnet++ network simulator [222]. Our analysis shows that our strategies for distributed processing bring significant advantages in terms of network traffic w.r.t. a centralized solution, in which a single processing node performs all the work to combine primitive events and to deliver composite events to sinks. At the same time, some of the proposed strategies also reduce the latency required to deliver notifications to interested clients, often considered one of the most important metric to evaluate a CEP system.

Finally, some of the proposed strategies also include mechanisms that enable T-Rex to automatically adapt to the network traffic, and in particular to event generation rates. Despite being theoretically promising, these mechanisms provide, at least in our simulations, only limited benefits in terms of network traffic. On the other hand, we believe they deserve more investigations, on a wider range of workloads. This should allow to better isolate the cases in which they appear more suitable, thus enabling their activation only in the specific cases that can significantly benefit from their application.

The rest of the chapter is organized as follows: in Section 7.2 we present how we model our processing network. In Section 7.3 we present in details the deployment strategies we considered. In Section 7.4 we evaluate such strategies. Finally in Section 7.5 we survey related work,

providing some conclusive remarks in Section 7.6.

7.2 Network Model

Figure 7.1 shows the deployment architecture of T-Rex. As in previous chapters, we consider a set of `sources` that observe and publish primitive events and a set of `sinks` that express their interests in receiving both primitive and composite events by subscribing to them.

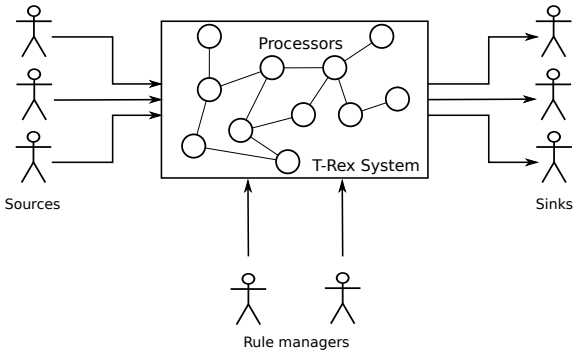


Figure 7.1: The deployment architecture of T-Rex

Differently from previous chapters, we now assume T-Rex to be internally built around different `processors`, connected together to form an *overlay network*, which cooperate to detect and route events from sources to sinks. Using the terminology introduced in Chapter 2, we are considering a *networked* deployment model. Each external client (either a source or a sink) is connected to the system through a link to a single processor.

As usual, the processing is performed according to the set of TESLA rules that `rule managers` deployed on the system.

As we will better show in Section 7.3, the deployment strategies that we studied for T-Rex rely on the knowledge of the kind of events produced at each source. Accordingly, we ask sources to *advertise* the type of events they will publish. This builds a contract between the sources and the T-Rex system: only events whose type has been advertised can be published; such contract is exploited by the T-Rex system to enable distributed detection of composite events, as explained in Section 7.3. It is worth mentioning that the use of advertisements has been first introduced (and widely adopted) by publish-subscribe systems [223].

7.3 Deployment Strategies

We now discuss in details the deployment strategies considered in this work. For ease of exposition, we separately describe the main protocols used as building blocks to define complete strategies: in particular we show *i.* how nodes organize themselves into one or more *processing trees*; *ii.* how advertisements and subscriptions are forwarded among processing nodes; *iii.* how rules are recursively partitioned and distributed among available processing node; *iv.* how event notifications are forwarded; *v.* how we can exploit traffic information to limit event transmission.

For simplicity, the following discussion assumes that nodes cannot fail. However, all the mechanisms described below can be extended to detect and react to node failures.

7.3.1 Building Processing Trees

As said in Section 7.2 we consider different processors connected in an overlay network. We do not impose any condition on the structure of such an overlay: nodes can be connected in any way, forming a generic graph.

However, to simplify the communication, our deployment strategies organize nodes into one or more *processing trees*. More precisely, they use a processing tree to collect primitive events from sources[1] (the leaves) and forward them to a processing node responsible for their analysis (the root of the tree). The same tree is then used to distribute the results of processing (i.e., composite events), following the reverse paths, from the root down to the leaves. For simplicity, in the following discussion we assume sinks to be interested only in composite events; subscriptions to primitive events are easily handled using well known protocols for content-based routing (see for example [28, 224]).

As we will better show in the following sections, our strategies define mechanisms for distributed processing: although event notifications may potentially need to be forwarded until they reach the root of a processing tree, they are already filtered and (partially) processed at intermediate nodes, thus reducing the amount of information that flows along the tree, as well as the processing load at the root node.

Since we want to reduce the latency required to collect information from sources and to deliver results to sinks, we build *Shortest Path Trees*,

[1] For simplicity we forget about sinks and sources to focus on the overlay network of processors; for this reason in the following we use the term *sink* (resp. *source*) to indicate the processor a sink (resp. source) is connected to.

using the link delay as a cost metric. In particular, the creation of a tree rooted at a processor P is always started by P itself. It sends a special message $\texttt{CreateTree}_P$ to all its neighbors. When a node N receives such a message it behaves as follows.

- If N receives the message for the first time, it marks the sender S as its father in the tree rooted in P; it sends an `ACK` message to S and forwards the message to all its neighbors except S.
- If N already received the message, it sends a `NACK` message to the sender S.

When a node N receives an `ACK`, it marks the sender as its child for the tree rooted in P. N obtains a complete knowledge about its children in the tree as soon as it receives an `ACK` or `NACK` message from all its neighbors.

This protocol allows all nodes to obtain local knowledge about a tree, i.e., their father and the set of their children.

7.3.2 Single Tree vs. Multiple Trees

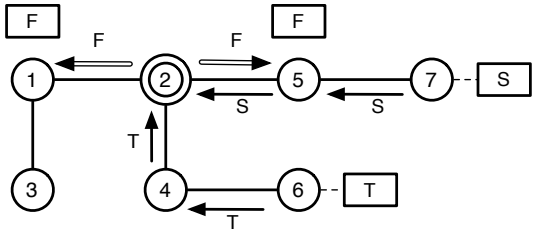
In our analysis we consider two classes of deployment strategies. The first one organizes all the nodes into a single processing tree: in this class of solutions one node C is elected as a sort of center of the network. All events move from sources to C going up along its tree. As mentioned, while moving along this path, events are filtered and (partially) processed according to the TESLA rules deployed on the system. When they arrive to C the processing is complete and the corresponding composite events are generated. They are delivered to interested sinks by flowing back down the tree.

The second class of strategies, instead, creates one tree for each sink. In this class of solutions primitive events flow from the sources along different trees. More precisely, if a sink S is interested in a composite event *ce*, all primitive events that contribute to the generation of *ce* move from their sources up along the tree of S. The same happens for all sinks interested in *ce*.

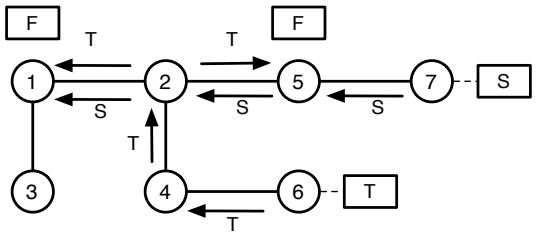
Processing is performed as in the previous case: as primitive events move toward the root of a tree, they are combined to generate composite ones. However in this case the root of the tree always coincides with an interested sink (and there is a separate tree for each interested sink). Therefore, when a composite event reaches the root of the tree, there is no need to further distribute it. On the one hand, this removes the need for spreading composite events once they have been detected. On

the other hand, this class of solutions potentially duplicates primitive events, by forwarding them over multiple trees.

To better understand the differences between the two classes of strategies consider Figure 7.2. It represents a sample overlay network with 7 processors. Now assume a single TESLA rule has been deployed in the system, which processes `Smoke` (`S`) and `Temp` (`T`) events to detect possible occurrences of `Fire`. There is a single source of `Smoke`, connected to processor 7, and a single source of `Temp`, connected to processor 6, while there are two sinks interested in `Fire`, one connected to node 1, and one connected to node 5.



(a) Single Tree



(b) Multiple Trees

Figure 7.2: Single Tree vs. Multiple Trees

An example of single tree strategy is shown in Figure 7.2(a), where node 2 is chosen as the root of the tree. `Temp` and `Smoke` events flow from their sources up to node 2. Here they are combined to detect `Fire`; finally, `Fire` notifications are delivered from node 2 to the sinks (node 1 and node 5).

The adoption of a multiple tree strategy is shown in Figure 7.2(b). One tree is built for each sink interested in `Fire` (i.e., node 1 and node 5). Sources forward events along both trees: for example node 6 sends `Temp` events to 5 using the path 6-4-2-5, and to node 1 using the path 6-4-2-1. Since the first three nodes share a common sub-path (i.e., the first three nodes), a single copy of each event notification is actually delivered along this sub-path. To do so, we adopt a special header field for

event notifications, which stores the set of trees an event has to traverse. Similarly, source 7 delivers `Smoke` notifications both to processor 1 and to processor 5. Notice how in this case there are no `Fire` events flowing the network, since they are autonomously detected at each sink.

As we said, the adoption of single or multiple processing trees defines two classes of solutions. Indeed, other aspects have to be considered as well to completely identify a solution, and in particular *i.* where the different processing steps required to generate composite events take place (e.g., only in the root, or also inside intermediate nodes), and *ii.* how the different processors interact with each other (e.g., using a push or a pull approach).

7.3.3 Forwarding of Advertisements

As we said in Section 7.2, we require external sources to send `Advertisement` messages to notify the T-Rex system about the set of event types they will publish. Depending from the class of deployment strategy adopted, advertisements are forwarded along a single tree, or along multiple trees.

For simplicity, we show how advertisements are forwarded on a single processing tree, since the same procedure is applied for each tree in the case more than one is defined.

Advertisements are forwarded from sources up to the root of the tree. Each node saves all the event types contained in the advertisements coming from its descendants in an *advertisement table* (one for each defined tree). In Figure 7.3 we show the advertisement table of processor 2 after it has been filled (we denote the set of message types advertised by processor x as $types(x)$). Processor 1 is the root of the tree.

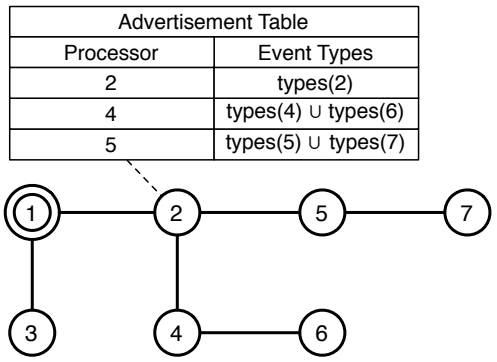


Figure 7.3: Forwarding of advertisements

7.3.4 Forwarding of Subscriptions

Subscriptions are forwarded as advertisements: they move from the sinks up the the root of each processing tree. As in the case of advertisements, they are combined at each step and stored in a local *subscription table* (one ofr each tree) at each node.

Subscriptions are then used to forward (composite) event notifications along the opposite path, from the root down to the leaves of the tree. When receiving an event notification, each node computes the set of children it has to be delivered to, by executing a content-based matching algorithm.

Notice, however, that this process is necessary only when a single tree strategy is adopted. On the contrary, multiple trees strategies do not require subscriptions to be forwarded, since each sink becomes responsible for producing all the composite events required by its local client.

7.3.5 Rule Deployment

As shown in the previous sections, processing is performed along trees: primitive events are forwarded from the sources up to the root, and they are filtered and processed at each step. To do so, rules are installed in the root, and then recursively partitioned into simpler *derived rules* while moving toward the sources. The partitioning algorithm exploits the information stored in the advertisement tables of each node, to process (and filter out, if necessary) event notifications as near as possible to sources.

In the case of a single tree strategy, the deployed rules are forwarded to the root of the single tree, which is responsible for partitioning them and distributing the resulting "derived rules".

On the contrary, in the case of a multiple trees strategies, rules are accessible by each processor in the overlay network: they are actually stored at one or more nodes, but all nodes in the network can download them on demand.

When a sink S issues a subscription for composite events of type c, the processor S is connected to becomes in charge of detecting c; accordingly it considers all the rules that generate events of type c (asking them to the nodes storing them, if needed), and partitions and distributes them along its own processing tree.

Partitioning TESLA rules

We now show how TESLA rules are partitioned. As we have seen in Chapter 3, TESLA is an expressive language, which provides operators

to express complex temporal patterns, including negations, parameters, and aggregates. These features make the partitioning algorithm relatively complex. For the sake of clarity, we prefer to present it through examples, to better show the reasons behind the different cases. The interested reader can download the code implementing our strategies from http://home.elet.polimi.it/margara.

As a first example consider the following Rule R11 and the processing tree shown in Figure 7.4.

```
Rule R11
define    CompEvent()
from      A() and last B() within 5 min. from A
          and last C() within 5 min. from B
          and last D() within 5 min. from C
          and last E() within 5 min. from D
```

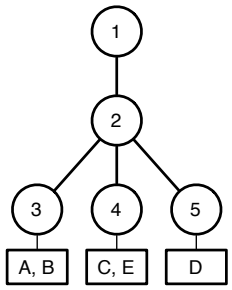


Figure 7.4: Rule Deployment: an Example

In Figure 7.4, processor 1 is the root of the processing tree, and there are three sources: node 3 produces primitive events of kind `A` and `B`; node 4 produces events of kind `C` and `E`; finally, node 5 produces events of kind `D`. This information is stored in the advertisement table of node 2; since advertisements are combined at each level of the tree, node 1 has a single entry in its advertisement table, stating that all kinds of events (`A`, `B`, `C`, `D`, and `E`) come from node 2.

In this example, the partitioning of Rule R11 is performed as follows: by looking at its advertisement table, node 1 observes that node 2 has all the information needed to correctly process the entire rule R11. Accordingly, it does not partition it, but sends the complete rule to processor 2, thus delegating the entire processing to it, including the generation of derived composite events.

Node 2 looks at the advertisements coming from its children: none of them produces all the events required to process the complete Rule

R11. Accordingly, node 2 remains the responsible for producing composite events: it splits Rule R11 into derived rules, and forwards them to processor 3, 4, and 5.

Derived rules simply contain the pattern of events to be detected (as usually specified in the `From` clause of TESLA rules), but they do not specify a composite event to be generated. They are only used to limit as much as possible the number of event notifications that are forwarded up along the processing tree. When a node N, responsible for processing a derived rule R', receives a set of primitive events PE that satisfy the pattern expressed into R', it simply forwards all event notifications in PE to its father.

Consider for example processor 3: its local clients produce all events of kind `A` and `B`. To correctly process Rule R11, node 2 does not need to receive all the events of kind `A` and `B` produced, but only notifications of `A` events preceeded by a `B` event in the previous 5 minutes; moreover, since the `last-within` operator is used, only the last `B` event before each `A` is relevant.

Accordingly, processor 2 creates the following derived rule, and delivers it to processor 3.

```
A() and last B() within 5 min. from A
```

Similarly, node 2 does not need to receive all events of kind `C`, but only those preceeded by an event of kind `E`. Accordingly, it creates and sends the following sub-rule to processor 4.

```
C() and each E() within 10 min. from C
```

Notice that `C` and `E` are not contiguous elements in the sequence defined by Rule R11, but they are separated by event `D` (which is not produced by the sources of processor 4). Accordingly, the derived rule considers a timing constraints that sums the time limits between `C` and `D` together with the time limit between `D` and `E`. Similarly, with its local knowledge, processor 4 cannot apply optimizations derived from the use of a single selection policy: therefore, the derived rule is defined using the `each-within` operator.

Finally, processor 5 receives a simple derived rule that asks the forwarding of all events of type `D`. In our example we only considered event types: however, all the constraints on the content of events expressed in rules are exploited to provide a more fine-grained partitioning of rules.

The procedure described here is applied recursively: derived rules are themselves split into other derived rules, until all sources have been reached.

Handling Events from Multiple Sources

To better understand how the partitioning of rules is performed, consider again Rule R11, now with the processing tree represented in Figure 7.5.

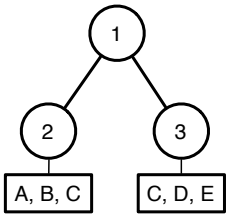


Figure 7.5: Rule Deployment: Events from Multiple Sources

At a first sight, it may be tempting to split Rule R11 into two derived rules, one involving `A`, `B`, and `C` (for processor 2), and one involving `C`, `D`, and `E` (for processor 3). However, neither processor 2, nor processor 3 receive all events of type `C` and thus may produce wrong results if they consider `C` during processing. It is processor 1 that is responsible for combining events of type `C` with the others. More in general, the detection of a certain type t of events may be delegated, from the father to a child in the processing tree, only when the child is *the only one* that advertises the event type t.

Accordingly, in the situation shown in Figure 7.5, Rule R11 is split into three derived rules.

The first one involves events `A` and `B` and is forwarded to processor 2:

```
A() and last B() within 5 min. from D
```

The second one involves events `D` and `E` and is forwarded to processor 3:

```
D() and last E() within 5 min. from D
```

The last one involves events of type `C` and is forwarded both to processor 2 and to processor 3:

```
C()
```

Handling Parameters

As we have seen in Chapter 3, TESLA rules may include parameters that bind the content of different primitive events. While partitioning a rule, if a parameter p involves only events that are captured by a

derived rule d, than p is added to d. Otherwise, if p involves events from different derived rules, it cannot be attached to any of them; in this case parameter p is checked at a higher node of the processing tree, where all involved primitive events are received.

Consider for example Rule R12 below and the two processing trees in Figure 7.6.

Rule R12

```
define    CompEvent()
from      A(v=$x) and last B(v=$x) within 5 min. from A
          and last C() within 5 min. from B
```

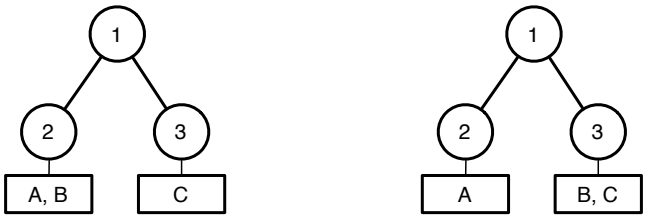


Figure 7.6: Handling Parameters

In Figure 7.6(a), both events of type `A` and `B` come from the same node 2. In this case the parameter can be added to the derived rule sent to processor 2, which becomes:

```
A(v=$x) and last B(v=$x) within 5 min. from A
```

On the contrary, in the situation shown in Figure 7.6(b), events of type `A` come from node 2, while events of type `B` comes from node 3. Accordingly, the derived rule sent to processor 2 (i.e., `A()`) does not contain any reference to the parameter, and the same applies to the derived rule sent to processor 3:

```
B() and last C() within 5 min. from B
```

Node 1 remains responsible for detecting th complete Rule R12 and for checking the values of attribute `v` in events `A` and `B`.

Handling Negations

Similarly to parameters, negations can be attached to derived rules only if they include all the primitive events used to specify their time bound.

Consider for example the following Rule R13 and the two processing trees in Figure 7.7.

```
Rule R13
define    CompEvent()
from      A() and last B() within 5 min. from A
          and last C() within 5 min. from B
          and not D() between C and B
```

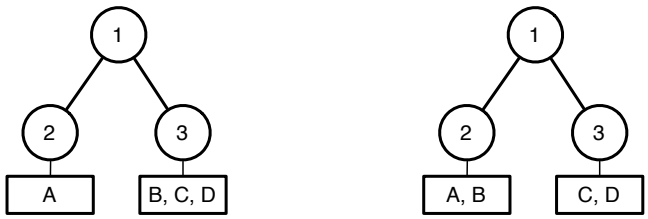


Figure 7.7: Handling Negations

In Figure 7.7(a) both events `B` and `C` come from the same node as the negated event `D`. Accordingly, we can include the negation inside the derived rule delivered to processor 3, which becomes:

```
B() and last C() within 5 min. from B
and not D() between C and B
```

On the contrary, in Figure 7.7(b), events of type `B` are detected by processor 2. In this case the negation cannot be included as part of the derived rule for processor 3. All events of type `D` have to be delivered to node 1, which is responsible for processing the negation.

Accordingly, node 2 receives the following derived rule:

```
A() and last B() within 5 min. from B
```

while node 3 receives two different derived rules, one for events of type `C` (derived rule `C()`), and one for events of type `D` (derived rule `D()`).

Computing Aggregates

In the deployment strategies described in this work, all the aggregates of a rule R are always computed by the node responsible for generating composite events for R.

In the case of rules involving complex aggregates over large volumes of data, it would be possible to modify this behavior by including special messages between processors that deliver the (partial) results of aggregates. We plan to explore this aspect in the near future. However, as we have seen in Chapter 5, processing usually does not introduce a significant delay. Furthermore, as explained in Section 6, processing of complex rules can take advantage of GPUs, if available, to further reduce such delay. In any case, the time to compute aggregates remains negligible when compared with the time required to forward information on a wide area network. Accordingly, we do not expect to receive significant advantages in terms of delay from the definition of incremental aggregate evaluation. On the other hand, performing aggregation in-network, as primitive events flow from the sources to the root of the processing tree may contribute to reduce the number of notifications to forward, and thus to limit the usage of network resources.

7.3.6 Forwarding of Events

As mentioned, primitive events are forwarded from the sources up along the processing trees. In the case of multiple trees strategy, event notifications are labeled with the set st of trees they are relevant for.

When a processor p receives a primitive event e, it reads the set st; for each tree t in st, p extracts the set of rules (and derived rules) deployed, and uses them to process e. All produced results (either composite events, in the case of complete rules, or primitive events, in the case of derived rules) are delivered to the father of p in tree t.

Given the results shown in Chapter 5 and Chapter 6, we adopt our Column-Based Delayed (CDP) algorithm to process events at each node. As we will see in Section 7.4, the adoption of CDP also limits the memory usage of the different nodes, allowing us to simulate larger networks.

As a final remark, notice that different derived rules may be deployed on a single processor p, all regarding the same tree t. In this case, it becomes possible for a primitive event to be produced as a result of the processing algorithm more than once, also at different time instants. To avoid duplicate transmissions, each processor keeps a history of already forwarded events for each tree it participates in. The size of this history is dynamically computed depending from the timing constraints expressed

in rules, which in turn determine the maximum period of time in which events are kept in columns for processing.

7.3.7 Push vs. Pull-Based Forwarding

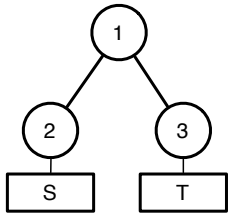


Figure 7.8: Limitations of the Push-Based Forwarding

Consider now the processing tree shown in Figure 7.8. Processor 1 is responsible for detecting `Fire` starting from `Temp` (`T`) and `Smoke` (`S`), using Rule R3, defined in Chapter 5 and reported here for simplicity.

```
Rule R3
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          each Temp(area=$a and value>45)
          within 5 min. from Smoke
where     area=Smoke.area and measuredTemp=Temp.value
```

`Smoke` events are produced by processor 2, while `Temp` event are produced by processor 3. In certain areas it may be common to receive a large number of event notifications about high temperature, while `Smoke` notifications are much less frequent. Since processor 1 can produce `Fire` only when it receives both `Smoke` and `Temp`, the vast majority of events delivered by node 3 are simply discarded; forwarding them to processor 1 only wastes network resources.

Starting from these considerations we introduced in our deployment strategies the concept of *pull-based forwarding* as opposed to the more common push-based approach. In particular, every derived rule has an associated *mode* which can be either *push* or *pull*. A push derived rule requires the processor receiving it to promptly send all matching sets of primitive events up to its parent; on the contrary a pull derived rule requires the processor to store matching sets of events until the parent explicitly asks for them. So, in the example of Figure 7.8, processor 1 could decide to send the derived rule for events of type `T` in pull mode,

and to ask for the delivery of stored events only after receiving events of type `S` from processor 2.

More specifically, the protocols used to decide the mode associated to derived rules and to ask for stored messages (in case of pull derived rules) work as follows. When a processor P receives a rule r, it processes and partitions it as explained above. Among the derived rules generated starting from r, one is selected as the *master* rule, while all the others are considered *slave*. The mode associated to the master is push, while slave derived rules have a pull mode. When processor P receives a set of events that satisfy the pattern expressed in the master, it sends an `Awakening` message to all children processing slave derived rules. When receiving this message, a child processor C, processing the slave sub rule r', starts sending all events matching the pattern expressed in r' to the parent.

Consider again our example: assume that node 1 chooses the derived rule about `Smoke` as master derived rule for Rule R3. When it receives a `Smoke` notification, it sends an `Awakening` message to node 3. Node 3 has a pull derived rule regarding the detection of `Temp` notifications: when it detects a high temperature, it stores the corresponding notification in a buffer, where it remains for the next 5 minutes (the timing constraints expressed in Rule R3; after this time limit `Temp` events have no chance to be relevant for the rule). When it receives an `Awakening` message Rule R3 sends all the event notifications stored (if any) to node 1, followed by a special `End` message, to notify that all events have been delivered. After receiving the `End` message node 1 starts processing. In this case we say that the `Awakening` opens a *past window* of 5 minutes for the slave rule.

The way slave derived rules are processed depends from the position of the primitive events they have to detect inside the pattern specified in the original rule. As another example consider again Figure 7.8, but now assume that the derived rule about `Temp` is elected as a master. In this case, node 2 does not have to store any `Smoke` notification; on the contrary, it can discard all of them. When node 2 receives a `Awakening` message from node 1 (meaning that a `Temp` notification has been detected) it starts sending all `Smoke` events it detects in the *following* 5 minutes. In this case we say that the `Awakening` opens a *future window* of 5 minutes for the slave rule.

In the most general case, with TESLA rules that define more than one sequence, an `Awakening` can open both a past window and a future window.

It is worth mentioning that not all rules can be elected as master: an example is represented by rules used only to deliver events required

to compute negations (as the rule regarding events of type `D` in Figure 7.7(b)). Indeed, they attract notifications about events that should *not* occur; as such, they cannot guide the detection of patterns.

Finally, in all those cases in which there are parameters that involve events defined in the master rule and events defined in a slave rule, the `Awakening` message asks only for events having the right value for the shared parameters (which are added to the `Awakening` message), thus further reducing the network traffic.

In our previous example, if the derived rule about `Smoke` is selected as master, and a `Smoke` event is detected from an area A, this area is stored inside the `Awakening` message sent to node 3, so that only stored `Temp` coming from area A are forwarded.

7.3.8 Adaptive Selection of Masters

The right choice for the master vs. slave derived rules may strongly influence the performance of our protocol. Indeed, if events satisfying the master rule are received sporadically, then fewer requests are sent to children holding slave rules, which may drop several events locally (i.e., those exiting the window) resulting in less network traffic. On the contrary a master rule that continuously receives notifications eliminates the benefit of the pull-based approach. To address this issue, our protocol monitors traffic flowing in the network and let each processor adapt its choice of master rules to the traffic monitored in the previous time frame.

More specifically each processor B stores, for each rule r it is responsible for, the number n of events it received in a given amount of time t from each derived rule r' originating from r. Periodically, B computes the *generation rate* of each derived rule r', $gr(r') = n/t$, and uses it to update its decision about the master derived rule, by choosing the rule with the lowest generation rate.

In presence of multiple derived rules deployed, the selection of an appropriate master becomes more complex. Indeed, it is possible for a primitive event *pe* to participate into more than one derived rule. If at least one of this rules is selected as master, notifications about *pe* will be received in push mode. Accordingly, when multiple rules are deployed, we do not only consider the generation rate of rules, but also the number of primitive events that are already received in push mode since they are covered by other rules.

Notice how also this protocol allows nodes to take decisions autonomously. Selection of master and slave rules is a local decision that each node takes solely on the base of the traffic information it collected during processing.

In summary, the mechanism combining push and pull-based forwarding, coupled with this adaptive mechanism in the choice of which part of a rule to manage as push and which to treat as pull, results in the ability for our protocol to optimize composite event detection to the actual traffic, minimizing the number of event nofifications that need to be forwarded to correctly detect

7.4 Evaluation

We implemented all the protocols described in Section 7.3 in T-Rex. While in the near future we plan to test them on a real distributed setting, possibly adopting workloads coming from existing applications, we have currently tested them using the Omnet++ network simulator [222]. This allows to obtain a preliminary comparison of their performance under very different scenarios, while also checking their correctness under a well controlled environment. Notice that, despite the network layer, responsible for the communication between nodes, was simulated, all nodes run the complete code of T-Rex, including the marshalling and unmarshalling of packets, the CDP algorithm to detect composite events starting from primitive ones, and the matching algorithm to select the sinks each event notification has to be delivered to.

The use of a simulated environment enables a high level of control and replicability. At the same time, the choice of running the complete code of T-Rex at each simulated node allows a faster check of our implementation. On the other hand, the sum of these two choices poses some limitations on the complexity of considered scenarios. We were able to simulate networks with up to 50 processing nodes; however, we had to reduce the number of deployed rules and the event generations rates. Indeed, all nodes were executed on a single machine, sharing a limited amount of available memory (8Gb). Despite T-Rex consumes a relatively small amount of memory, Ram easily becomes the bottleneck when introducing tens of nodes.

It is worth mentioning that the processing time measured during our tests are not influenced by the simulated environment. Indeed, Omnet++ is a discrete event simulator, in which only a single node is actually executed at each point in time, with access to the whole resources of the hosting machine, an AMD Phenom II, with 6 cores running at 2.8GHz.

We studied four different strategies. The first one is denoted as `Distributed (ST)` in the following graphs. It makes use of a single processing tree (thus the name `ST`), and uses the protocol described in

Section 7.3 to split deployed rules and to realize a distributed detection of composite events. The second one is denoted `Distributed (MT)`; it performs distributed detection exploiting multiple processing trees. The third and the fourth ones (`Push-Pull (ST)` and `Push-Pull (MT)` adopt the hybrid push-pull forwarding between nodes, splitting each rule into master derived rules and slave ones, and executing an adaptive selection of master, according to network traffic. The only difference among them is that the former exploits a single processing tree, whereas the latter exploits multiple processing trees. As a term of comparison we implemented a `Centralized` strategy, in which all event notifications are delivered to a single node C (using the minimum path from the source to C), which performs all the processing, extracts composite events, and delivers them to all interested sinks.

Default scenario

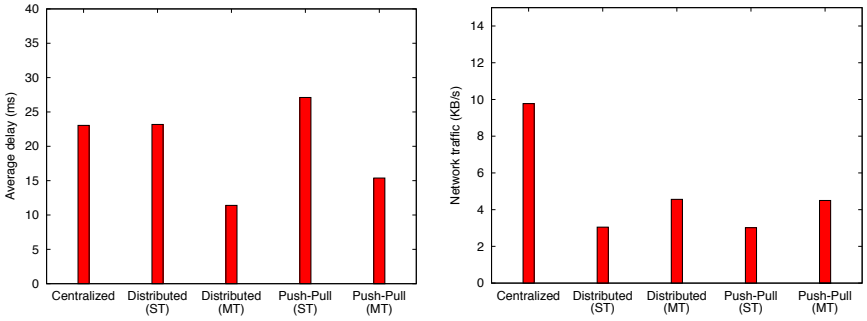
As also done in Chapter 4, Chapter 5, and Chapter 6, to perform our tests we defined a default scenario, and then we changed a number of parameters to explore their influence on the results we measured.

Our default scenario includes 20 processing nodes, each one connected with 5 other nodes, on the average. The topology has been generated using the Brite [225] topology generator, with an average link delay of 5ms. Sources produce 120 different types of primitive events. This means that the set of sources connected to each processor produces on the average 6 different types of event notifications. Each type of event has the same probability of being produced, and the generation rates vary between 1 notification every 1000 seconds and 10 notifications per second, with exponential distribution. We deploy 100 TESLA rules, each one including a sequence of 3 events with time windows of 1 minute, on the average. Each rule produces a different type of composite events, and the set of sinks connected with each processor is interested in 10 of them, on the average.

Despite its simplicity, this scenario allows us to cover almost all the aspects defined in our protocols, including the algorithms for recursive partitioning of rules and distributed detection of composite events.

To increase the traffic of events flowing in the network, we decided not to include filtering of primitive events based on their content. This means that in our default scenario rules only filter primitive events on the basis of their types. In the following, we will separately investigate how the use of content-based filtering impact on performace, showing how it strongly advantages our strategies, which are capable of pushing event filtering near to the sources, w.r.t. a centralized solution.

Before running the actual tests phase, we execute a configuration phase, in which all nodes run the protocols to build the overlay network, and all rules and subscriptions are deployed. 30 seconds after sources begin publishing primitive events, we start monitoring the behavior of the system and we keep measuring until the most rare event has been published at least 100 times. With some type of events being published every 1000 seconds, this means that our tests span more than 24 hours of (simulated) time. We repeated all measure 10 times; the 95% confidence interval we computed was always below 1% of the measured value.



(a) Average Delay (ms) (b) Average Traffic per Second (KB/s)

Figure 7.9: Default Scenario

Figure 7.9 shows the results we measured in the default scenario. In particular, Figure 7.9(a) shows the delay for obtaining results, while Figure 7.9(b) shows the overall traffic generated by the T-Rex system.

We computed the delay for obtaining results as the difference between the time in which a sink receives a composite event e, and the time in which e occurs (i.e., the time in which the last primitive event necessary for its detection occurs). Since we are working in a simulated environment, we can measure this time without incurring in synchronization errors between nodes.

If we look at Figure 7.9(a), we first observe a significant difference between the strategies that make use of a single tree (i.e., `Centralized`, `Distributed (ST)`, and `Push-Pull (ST)`), and the strategies that adopt multiple processing trees (i.e., `Distributed (MT)`, and `Push-Pull (MT)`), with the second class showing lower delays. Indeed, strategies based on multiple trees duplicate the processing required for the detection of a composite event e, exploiting the trees of all sink interested in e. This removes the need for delivering composite event notifications after processing, thus eliminating the time needed for this phase.

If we compare the strategies that adopt a single tree, we observe that

`Centralized` and `Distributed (ST)` behave almost identically, while `Push-Pull (ST)` exhibits a slightly higher delay. This was expected, since pull notifications are not delivered immediately, but only when explicitly asked. The same holds when multiple processing trees are adopted: `Push-Pull (MT)` shows higher delay w.r.t. `Distributed (MT)`.

If we look at the network traffic (Figure 7.9(b)), we immediately observe that the `Centralized` strategy generates significantly more traffic than the other strategies. This means that, in our default scenario, the in-network filtering mechanisms introduced by distributed processing of events allows nodes located near sources to discard a large number of primitive events.

As we said, in our default scenario we are not performing any kind of content-based filtering of primitive events: accordingly, the results shown in Figure 7.9(b) derive uniquely from the time-based filtering of events that do not contribute to valid sequences in the specified time windows. In the following, we will analyze how results change both when content-based filtering is added and when the size of windows is modified.

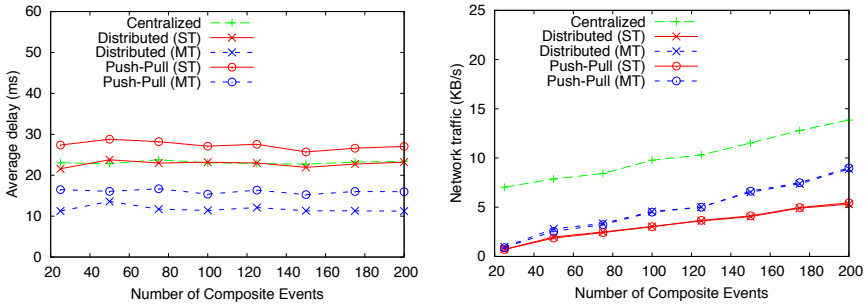
If we compare the four strategies that perform a distributed processing of events, we observe how those that make use of multiple processing trees generate more traffic. On the one hand, using multiple processing trees means that primitive events have to flow across different paths, thus producing more traffic. On the other hand, using multiple processing trees removes the need to forward composite events after their detection. The results shown in Figure 7.9(b) demonstrates that in our default scenario the cost of moving primitive events overcomes the benefits of removing the forwarding of composite events.

Finally, if we compare the `Push-Pull` and the `Distributed` strategies, we measure a very small benefit of the first ones w.r.t. the second ones. This means that in our default scenario the benefits of introducing pull notifications is marginal.

Number of composite events

Figure 7.10 shows how results change when increasing the number of composite events, and hence the number of deployed TESLA rules. Interestingly, the delay perceived by sinks (Figure 7.10(a)) does not change significantly.

When the number of rules increases, so does the complexity of processing. However, the increased processing time does not impact on the average delay. This is in line with our analysis of T-Rex in Chapter 5: processing times are in the order of tens of microseconds and are therefore dominated by communication delays.



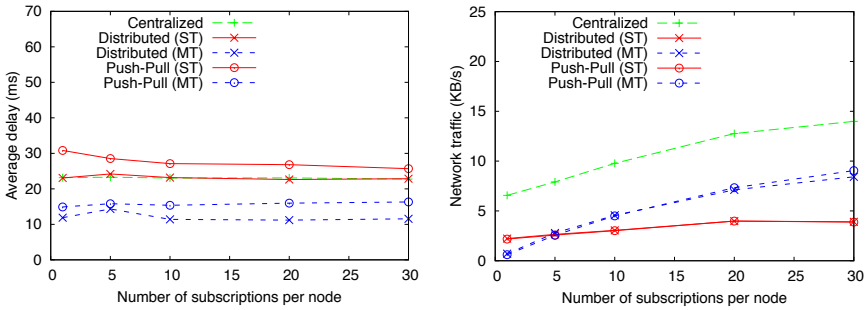
(a) Average Delay (ms) (b) Average Traffic per Second (KB/s)

Figure 7.10: Number of Composite Events

On the contrary, by looking at the overall traffic generated by T-Rex (Figure 7.10(b)), we notice that it increases with the number of deployed rules: indeed, primitive events are captured by a larger number of rules, and hence they have a higher probability of being moved from node to node for processing.

Interestingly, the number of deployed rules does not alter the differences among the strategies under analysis, with single tree strategies showing a reduced network traffic.

Number of subscriptions



(a) Average Delay (ms) (b) Average Traffic per Second (KB/s)

Figure 7.11: Number of Subscription per Node

Figure 7.11 shows how results change with the number of subscriptions issued by the sinks connected to each processing node. The number of

subscription does not influence the number of composite events generated, while it determines where composite events have to be forwarded. Moreover, it also influences the strategies involving multiple processing trees, since a higher number of sinks subscribing to a composite event e increases the number of processing tree used to detect e.

If we consider the delay perceived by sinks (Figure 7.11(a)), we observe that the results measured are not influenced by the number of subscriptions. This means that a larger number of interested sinks for each produced event (and an increased number of processing trees, for `MT` strategies) do not significantly impact on the delay, which is dominated by the latency of network links, as already observed in the previous section.

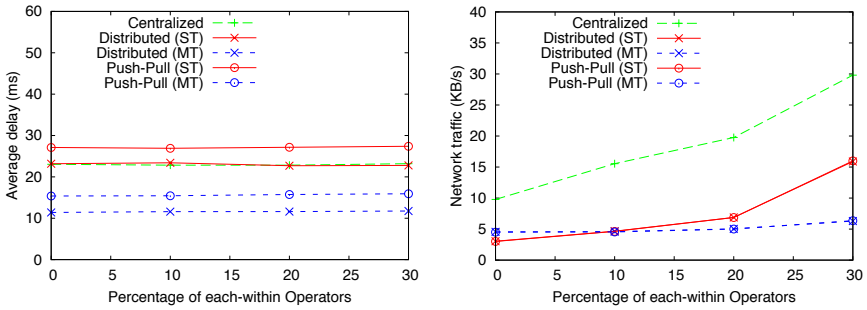
On the other hand, as expected, the network traffic increases with the number of subscriptions (Figure 7.11(b)). It is interesting to note how strategies adopting multiple processing trees perform better than strategies based on a single tree when the number of subscriptions is low (below 5 subscriptions per node). This can be easily explained by remembering that the main advantage of `MT` strategies is that they do not need to forward composite event notifications; at the same time they introduce an additional cost for moving primitive events over different trees. When the number of subscriptions is low, so is the number of processing trees adopted: accordingly the advantages of `MT` strategies come at a small cost. On the other hand, when the number of subscriptions increases, so does the number of processing nodes. Accordingly, with more than 5 subscriptions per node, the strategies based on a single tree become more convenient in terms of generated traffic.

Finally, we observe how a large number of subscriptions negatively influences the `Push-Pull (MT)` strategy. With a large number of different processing trees, the pull mechanism does not produce benefits: in the communication between two processors, primitive events asked in pull mode for a tree may indeed be necessarily forwarded in push mode for another tree, thus preventing traffic optimizations.

Selection policy

In our default scenario all sequences were defined using a single selection policy, and more precisely using the `last-within` operator. We now investigate how results change when introducing a percentage of `each-within` operators (up to 30%) inside rules.

Results are shown in Figure 7.12. As in previous sections, the average delay registered by the sinks remains constant (Figure 7.12(a)). Once again, the differences in processing times are dominated by the latencies



(a) Average Delay (ms) (b) Average Traffic per Second (KB/s)

Figure 7.12: Selection Policy

of network links.

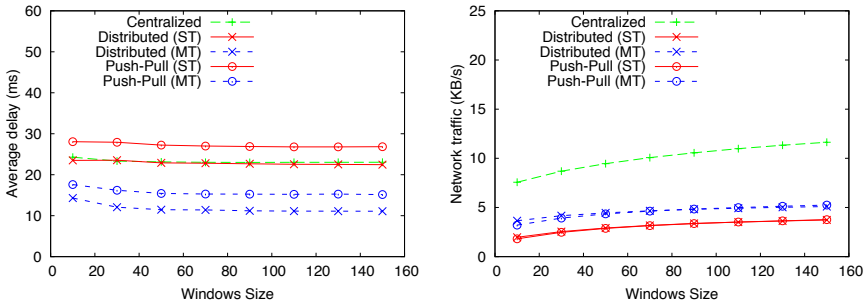
When considering the network traffic (Figure 7.12(b)), the adopted selection policy contributes in two ways: on the one hand, our recursive partitioning of rules becomes more efficient when it can exploit the presence of a single selection policy to reduce the number of primitive events to forward from processor to processor; on the other hand, a multiple selection policy produces a larger number of composite events, which need to be forwarded to interested sinks.

By analyzing the results we obtained in Figure 7.12(b), we can conclude that the first aspect has only a marginal impact. This can be deduced by observing the network traffic generated by `MT` strategies, which do not need to forward composite events after detection. Moving from 0 to 30% of `each-within` operators only produces a limited increase in network traffic.

On the other hand, strategies that make use of a single processing tree (i.e., `Centralized`, `Distributed (ST)`, and `Push-Pull (ST)`) significantly increase the network traffic when a multiple selection policy is adopted. As expected, `MT` strategies become more and more convenient when the ratio between composite events and primitive events increases, i.e., when a multiple selection policy is adopted.

Size of windows

Figure 7.13 shows the performance of the different deployment strategies when changing the size of the time windows used inside rules, moving from 30 to 120 seconds.



(a) Average Delay (ms) (b) Average Traffic per Second (KB/s)

Figure 7.13: Size of Windows

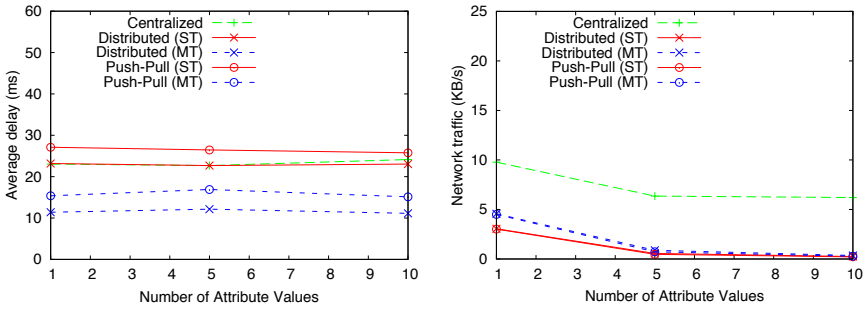
As in previous sections, the average delay observed by sinks (Figure 7.13(a)) does not change significantly. On the other hand, the size of windows has a visible impact on the generated network traffic. More in particular, as expected, all strategies exhibit higher traffic with larger windows; indeed, larger windows increase the number of primitive events participating in valid sequences, and hence the number of composite events generated.

Although it is not evident from the graph, if we compare the `Distributed` and `Push-Pull` strategies adopting a single processing tree w.r.t. the `Centralized` strategy, we observe that the differences in terms of traffic slightly decrease with the size of the windows. Indeed, larger windows reduce the possibility to filter out primitive events before they reach the root of the processing tree. On the contrary, when considering strategies using multiple processing trees w.r.t. the `Centralized` strategy, we measure larger differences with larger time windows. Indeed, the larger the windows, the higher the number of generated composite events: as already seen in previous sections, this advantages `MT` strategies, which do not need to forward composite events after they are detected.

A final observation regards the use of pull notifications. They appear to be more efficient with small windows, when the number of primitive events flowing the network is reduced.

Filtering of primitive events

As mentioned in previous sections, our default scenario does not include any kind of content-based filtering of events. In this section we investigate how results change when the value of event attributes is used to



(a) Average Delay (ms) (b) Average Traffic per Second (KB/s)

Figure 7.14: Filtering of Primitive Events

filter out primitive events.

When considering distributed processing of rules (as in the `Distributed` and `Push-Pull` deployment strategies), this operation can be performed directly in the first node that processes a primitive event, thus reducing the number of events that flow the network w.r.t. the `Centralized` approach. We consider this analysis extremely important, since we expect real applications to make wide use of the selection of primitive event based on their content while defining their rules.

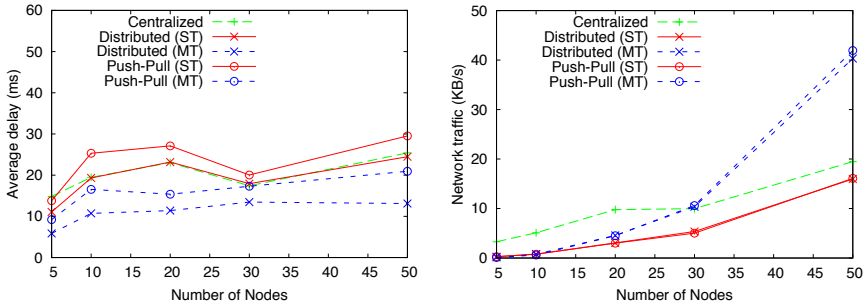
In the tests presented in Figure 7.14, each rule only selects primitive events that present a specific value for a given attribute. In our simulations we change the number of values that each attribute can assume, moving it from 1 to 10. A value of 1 means that all events are accepted for creating valid sequences (as in previous sections), while a value of 10 means that only 10% of primitive events is valid, and 90% of events can be immediately discarded.

In Figure 7.14(a) we observe the average delay measured by the sinks. As expected, it is not influenced by the presence of content-based filtering of primitive events. On the other hand, increasing the number of attribute values reduces the network traffic registered by all strategies (Figure 7.14(b)). Indeed, a higher percentage of filtered events reduces the number of generated composite events. This advantages all strategies, including the `Centralized` one.

At the same time, all the strategies that enable distributed processing obtain an additional benefit, since they can reduce the number of primitive events to be forwarded, by discarding them immediately, as they enter the processing network. This is clearly visible in Figure 7.14(a),

where all strategies that perform distributed processing register a much lower traffic w.r.t. the `Centralized` strategy, and their advantage increases with the number of values per attribute.

Number of processing nodes



(a) Average Delay (ms) (b) Average Traffic per Second (KB/s)

Figure 7.15: Number of Processing Nodes

A parameter of primary importance when considering a distribution strategy is the number of available processing nodes. Figure 7.15 shows how performance changes when moving from 5 to 50 processors. While increasing the number of nodes, we also increase the number of connections, thus limiting the differences between topologies in terms of number of hops needed to forward a packet from one node to another one. As for the default scenario, all network topologies have been generated using the Brite [225] topology generator.

When increasing the scale of the network, we also increased the number of primitive event type, and the number of deployed rules. We believe this better represents realistic scenarios, in which the heterogeneity of events observed and generated grows with the scale of the network.

Despite we limited the differences in terms of number of hops between the topologies we considered, we observe that the delay measured by sinks (Figure 7.15(a)) increases with the number of hops, when moving from 5 to 10 nodes. After 10 nodes the measured results do not exhibit a clear trend, with visible oscillations determined by the heterogeneity of the topologies adopted. This behavior is more evident for strategies adopting a single processing tree, which strongly rely on the topology when building the unique tree used to forward events. On the other hand, it is less visible when multiple trees are adopted.

The traffic generated by T-Rex (Figure 7.15(b)) increases with the size of the network. This can be explained by remembering that we increased

the number of primitive events and the number of deployed rules. The most interesting aspect, however, is the behavior of the strategies based on multiple processing trees. Since they need to forward primitive events over different paths, the traffic they generate increases with the number of possible trees, and hence with the number of nodes in the overlay network.

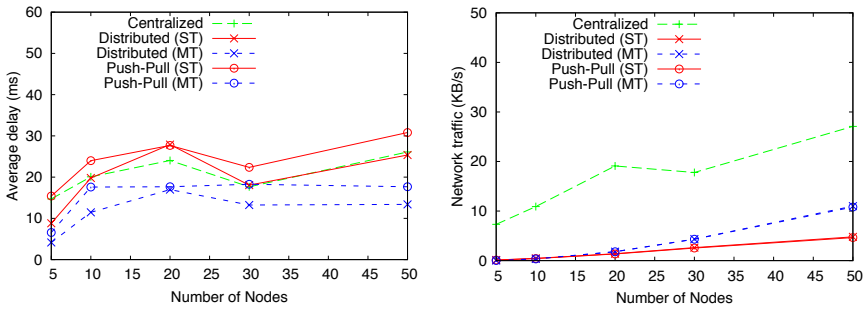
This result seems to suggest that the adoption of multiple processing trees is not convenient for large scale networks. However, it is worth mentioning that in our scenario we do not consider locality of events: primitive events may be generated at different (and distant) nodes, and they can participate in several different rules. On the other hand, we expect locality to be present in almost all real applications, in two forms.

i. Locality of primitive events in rules, meaning that rules can be divided into classes, and primitive events participate only in a limited number of classes. As an example, events of type `Temperature` could participate in rules related to environmental monitoring, but they will not probably appear in rules related to inventory management.

ii. Locality of publishers: events that participate in the same rule are often produced by nearby sources. For example all events related to inventory management are produced in a single building.

In some cases we also expect to see locality of sinks, meaning that composite events are consumed near to the place in which most of the primitive events used for their detection are published.

To better understand the benefits of locality, we defined and simulated a new scenario, in which we significantly increased the locality of primitive events (we defined classes of ten rules, all using the same primitive events), and the locality of publishers.



(a) Average Delay (ms) (b) Average Traffic per Second (KB/s)

Figure 7.16: Number of Processing Nodes (with Locality)

The results measured in this scenario are shown in Figure 7.16. Since

we did not consider the locality of sinks, the measured delay does not change significantly w.r.t. the previous scenario (see Figure 7.16(a)). On the other hand, we observe remarkable differences in the network traffic (Figure 7.16(b)). While processing strategies that adopt a single processing tree are still more efficient, the processing strategies based on multiple processing trees now show similar results, with an average network traffic that is significantly lower than those of the `Centralized` strategy.

To conclude, we can say that strategies adopting multiple processing trees suffer from the need of delivering primitive events over different paths, which can theoretically limit their applicability to large scale scenarios. However, the presence of locality significantly reduces the impact of this aspect.

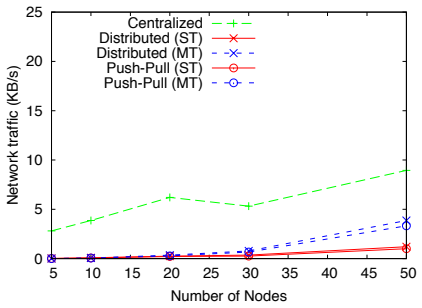


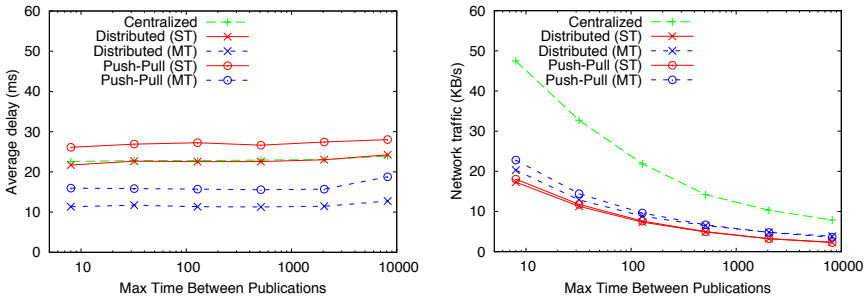
Figure 7.17: Number of Processing Nodes (with Filtering)

Finally, it is worth remembering that our scalability tests were executed without content-based filtering of events. If introduced, it can provide significant advantages to the distributed processing.

This is shown in Figure 7.17, which shows the network traffic measured when 90% of primitive events is filtered out based on the value of attributes. As in the case of locality, distributed processing becomes more efficient than centralized one, with a significantly reduced network traffic. Moreover, the reduced number of primitive events flowing the network increases the advantages of pull notifications, especially with a large number of processing nodes.

Publication rate

Finally, Figure 7.18 shows how performance changes with the publication rate. In particular, during our simulations, we fixed a maximum publication rate of 10 events per second, while we changed the minimum publication rate, moving it from 1 event every 5 seconds to 1 event every



(a) Average Delay (ms) (b) Average Traffic per Second (KB/s)

Figure 7.18: Publication Rate

10000 seconds. Since we adopt an exponential distribution, this operation has only a minimal impact on the average publication rate. On the other hand, it has a great impact on the number of composite events generated, since events of certain types may become extremely rare.

Figure 7.18(a) shows the delay measured by sinks. As expected, it is not influenced by the publication rate. Figure 7.18(b) shows the network traffic generated by the different strategies. When increasing the maximum time between publications, all strategies present a lower traffic: indeed, less composite events are generated. While this advantages all the strategies that adopt a single processing tree (and thus need to forward composite events after detection), it produces a greater advantage for strategies using distributed processing, since they can filter a higher number of primitive events that do not satisfy timing constraints expressed in rules. Accordingly, the advantage of both `Distributed (ST)` and `Push-Pull (ST)` over `Centralized` increases with the maximum time between publications.

On the other hand, strategies that adopt multiple processing trees are not influenced by the number of generated composite events, but only by the possibility of filtering out primitive events. As a result, the advantage of `Distributed (MT)` and `Push-Pull (MT)` over `Centralized` in terms of generated traffic does not change significantly with the maximum time between publications.

Finally, we observe how a small difference between publication rates makes it inconvenient to adopt mechanisms for pull notifications. Indeed, the number of primitive events forwarded in pull mode becomes small, and the cost of the mechanisms overcomes the advantages.

Final considerations

We have already seen in Chapter 5 and Chapter 6 how the choice of a particular workload can seriously affect the performance of an algorithm. When dealing with a CEP system, the number of parameters to consider is huge and it becomes even larger when distributed processing is taken into account: it becomes fundamental to consider the number of available nodes, the topology of the processing network, and the capabilities of network links. Most importantly, it becomes fundamental to know which nodes produce a certain information, and *where* they are located. For example, we have seen the importance of locality during our analysis of the scalability of strategies.

Given this premise, we believe that further investigations are needed to better analyze the different processing strategies presented here, possibly using data coming from real distributed applications. The limits derived from the usage of a network simulator currently prevent us from conducting more exhaustive studies. To overcome this limitation, in the near future we plan to move to a more powerful machine for the tests, but also to try real deployments, possibly using public testbeds like PlanetLab [226] or Emulab [227].

Keeping these considerations in mind, we can still draw some conclusions from the results measured in this section. First of all, processing times contribute only marginally to the average delay required to deliver event notifications to sinks. This suggests that strategies that reduce the number of transmissions (using multiple processing trees) result in lower delays, even if they potentially duplicate processing and increase network traffic. This becomes evident if we look at all the graphs regarding the delay measured by sinks. Even if we changed a large number of parameter, the delay remained almost constant in every test we run. The only exception is represented by the scalability test, where we changed the topology of the network. As expected, in all the test we run, the delay was lower with strategies that adopt multiple processing trees.

A final consideration regards the strategies that adopt a hybrid push-pull approach. In most of the tests we performed, they provide limited benefits in terms of network traffic, while increasing the delay measured by sinks. We still believe that this approach deserves more investigations, on a wider range of workloads. This should allow to better isolate the cases in which pull mode notifications are suitable, thus enabling an activation of the pull mechanisms only for certain rules and only between nodes that can significantly benefit from its application.

7.5 Related Work

One of the aspects addressed by the deployment strategy of a CEP system is the operator placement problem, which specifies how the operators defined inside rules are deployed on available processing nodes.

A good survey of most existing solutions for the operator placement problem can be found in [145]. A first fundamental consideration about the operator placement problem regards the kind of operators it addresses. Going back to our classification of rule specification languages presented in Chapter 2, we notice that most existing algorithms for operator placement are based on transforming languages. Indeed, these languages define a sequence of transformations that incoming data has to pass through to produce the desired output: since these transformations are applied one after the other, it is easy to deploy them on different nodes. On the contrary, detecting languages specify complex patterns, often involving timing constraints, and splitting them is not trivial. Distributed detection of pattern has been first explored in [228], with a very simple language. An important contribution comes from [11], where the authors study how patterns can be optimized for efficient distribution.

As we mentioned in Section 7.1 many solutions have been proposed for the operator placement problem [138, 139, 46, 79, 140, 31, 141, 30, 142, 143, 144, 127]. They strongly differ from each other on many aspects.

First of all, they start from different assumptions: some of them consider large scale networks of processors, while others assume clusters of colocated machines; some consider nodes with heterogeneous computational resources, while others consider homogeneous nodes; some of them assumes that single operators can be duplicated at different processing nodes, if needed, while others do not allow duplications.

Second, they are designed for different goals: for example, minize the delay required to produce results other, minimize the usage of network resources, or a combination of the two, or again minimize processing resource usage, and hence power consumption.

Since the problem is known to be NP hard, it is usually solved using approximated algorithm or heuristics. Most systems rely on a centralized decider, which collects all relevant information about the network status and locally computes a solution for the problem. Only a few proposals have considered a decentralized algorithm for solving the operator placement.

The deployment strategies proposed in this chapter solve the operator placement problem in a distributed way, by recursively splitting rules at each node; moreover, our push-pull approach exploits traffic information to alter the communication between nodes.

Beside offering a solution for the operator placement problem, a complete deployment strategy also needs to precisely define the protocols that govern the interaction among processing nodes, specifying how information is collected, processed, and finally delivered to interested sinks.

As we have seen in Chapter 2 these issues are usually not considered in existing CEP systems: most of them are based on a centralized deployment, in which all the processing is performed on a single machine (e.g. [98, 48]). Others define distributed processing, but are based on extremely simple languages if compared with TESLA, which do not capture all the needs of event-based applications [228]. Even when distributed processing is allowed, nodes often require manual configuration [123]. It is worth mentioning that some remarkable example of automated distribution can be found among DSMSs [79, 152].

Our deployment strategies include a set of protocols that define how nodes organize themselves into processing trees, handle deployed rules and subscriptions, and collect, process, and deliver event notifications.

Finally, our different solutions adopted in our protocols are inspired by work on distributed publish-subscribe systems and content-based routing [212].

7.6 Conclusions

In this chapter we presented different deployment strategies for distributed event processing in T-Rex. Given a set of connected processors, our strategies precisely define the communication among processing nodes to handle rule deployment, subscriptions, and to collect, process, and deliver event notifications.

Differently from most existing algorithms, our solutions do not rely on a centralized component that collect information and decides where the different processing operators have to be installed; on the contrary, this operation is performed in a distributed way, with each node autonomously taking decisions based on local knowledge about their neighbors.

Described deployment strategies have been fully implemented inside the T-Rex system. In this chapter, we presented a preliminar evaluation using the Omnet++ network simulator. Different aspects emerge from our analysis: first of all we observe that the processing delays introduced by T-Rex are negligible if compared with typical network delay, even when considering links with low delay and high bandwidth. This is in line with our analysis of T-Rex in Chapter 5. This suggests that duplicating the processing can be a good strategy for reducing the delay required

to deliver results to sinks. Section 7.4 confirms this intuition: strategies based on multiple processing trees provide lower delays. Moreover, as explained in Chapter 6, also in presence of complex rules that may significanlty increase the processing time, the use of GPUs as co-processors contribute in obtaining a low processing delay.

Section 7.4 also shows that the use of distributed processing can significantly reduce the network traffic required for event detection and delivery in a large range of scenarios. When considering network traffic, our analysis shows that strategies based on a single processing tree are generally more efficient in most of the scenarios we tested, at least until the number of composite events generated remains small w.r.t. the number of primitive events published by sources.

Finally, some of the deployment strategies we implemented include mechanisms that allow T-Rex to automatically adapt the interaction among processors to the network traffic, and in particular to event generation rates, by applying a hybrid push-pull approach to deliver event notifications. Although in our analysis this approach has provided only limited advantages in terms of reduction of network traffic, we believe it deserves more investigation. As a future work, we plan to test our strategies on a real network, possibly using more realistic workloads.

8 Conclusions and Future Work

Several complex information systems are designed around an event-based core that guides and controls the behavior of other sub-systems. It operates by observing primitive events that occur in the external environment, interpreting and combining them to extract higher-level composite events, and delivering them to the components in charge of reacting to them. This event-based core is realized by a Complex Event Processing (CEP) system.

The first problem one encounters when approaching CEP systems is defining them. Indeed, they are part of a wide class of systems that we call Information Flow Processing (IFP) systems, generally developed to solve the problem of processing continuous flows of information coming from heterogeneous sources to derive new knowledge. IFP systems include a wide range of heterogeneous proposals, including Active DBMSs, Data Stream Processing Systems (DSMSs), CEP systems, and many application specific products. They were historically developed by different communities, each one bringing its background, experties, and vocabulary. Comparing these systems is hard tasks and the boundaries between classes are often blurred.

In the first part of this thesis we offer a modelling framework to compare and analyze existing IFP systems. This enables to extract advantages and limitations of the different approaches, and offers a common ground to simplify the discussion and the cooperations among different research and industrial communities, reducing the effort required to merge the results produced so far.

The rest of the thesis presents in details the design and implementation of T-Rex, our CEP system explicitly studied to provide expressiveness, ease of use, efficient processing, and scalability to support large-scale distributed scenarios.

While analyzing existing systems to define our modelling framework, we soon realized that the first step for building a CEP system was the definition of a proper language for the specification of composite events starting from primitive ones. Surprisingly, we also realized that almost all existing systems are based on generic data processing language, which fail to provide the desired abstraction when it comes to implement an event-based system.

To overcome this limitation we defined TESLA, our language explicitly designed to deal with events and their temporal relations. TESLA offers a compact syntax, which allows the definition of complex temporal patterns through a small number of simple operators. It offers content and timing constraints, sequences, negations, aggregates, with customizable selection and consumption policies. Moreover TESLA is formally defined using a metric temporal logic; this makes it possible to understand its semantics, but also to check the correctness of a CEP system that processes events according to TESLA rules.

As a future work we plan to extend TESLA to deal with uncertainty: we believe this plays a central role in the future diffusion of CEP systems. Not only sources may produce events with an associated degree of uncertainty, but in many applications the uncertainty is inherently part of how we model the world. Providing support for uncertainty may raise the level of abstraction, making the definition of composite events more natural.

In the following part of the thesis we present the processing algorithm used inside T-Rex. First we present PCM, a novel algorithm for content-based matching explicitly defined to take advantage of the increasing power of parallel hardware. PCM has been implemented both on multi-core CPUs and on GPUs based on the CUDA architecture. If compared with state of the art solutions, it provides significant speedups; moreover, the use of GPUs also provides the fundamental advantage of leaving the CPU free to perform other tasks, like network communication.

Our contribution enables a new level of performance for all those systems that rely on event filtering, like publish-subscribe systems. Moreover, event filtering and forwarding is at the base of every CEP system.

Another contribution of the thesis is the design of two processing algorithms for TESLA rules. One (AIP) is based on automata and processes events incrementally, as they become available; the other one (CDP) accumulates all received events into ad-hoc data structures called columns and delays the processing as much as possible.

We implemented both algorithms in T-Rex and we tested them using a large number of workloads. First, we demonstrate the efficiency of T-Rex by comparing it with a mature commercial product. Even with thousands of TESLA rules deployed, T-Rex can handle high volumes of incoming events, with both algorithms.

The comparison of the two algorithms shows that CDP overperforms AIP in almost all the scenarios we defined. This is an important research result. Indeed, most existing systems are based on automata, or in general on incremental processing. As a future work we plan to explore this issue in details, to understand if CDP-like processing algorithms can be

applied to a wider class of languages and systems.

Although our implementations exploit the presence of multi-core CPUs to handle multiple rules in parallel, single rules are precessed sequentially, on a single core. In many application scenarios this is perfectly acceptable; however, there may exist scenarios that involve extremely complex rules. To cope with this situation, we studied how the T-Rex processing algorithms can take advantage of the presence of GPUs to adopt them as co-processors to reduce the processing time of most complex rules. Also in this case, our performance analysis on a large number of workloads shows impressive speedups w.r.t. our sequential implementation.

Finally, to deal with large scale distributed scenarios, we defined different deployment strategies that allow T-Rex to exploit the availability of different processing nodes. Our contribution includes the definition of a set of protocols that different processors can use to organize them into an overlay network, partition and distribute rules, collect events from sources, cooperatively processing them, and finally distribute them to interested parties. All defined protocols have been fully implemented in T-Rex and we provide a first evaluation using a network simulator. Our analysis shows that our strategies contribute in limiting the network traffic to detect composite events and deliver them to interested sinks in almost all the scenarios we tested. Moreover, some of the defined strategies also reduce the delay required to obtain results, which is usually considered one of the most important performance metric for CEP.

We plan to significanlty extend our evaluation in the near future, using a real distributed setting, and possibly workloads extracted from real application.

To conclude, CEP is a relatively young and continuously evolving research field. We do not believe that the contributions of this thesis represent a definitive answer to most of the problems we identified. However, we are convinced that they define a valid ground to start future research effort.

The research around CEP needs to significantly evolve to meet the requirements of application. First of all it has to provide an adequate level of abstraction to allow natural definition of composite events. We are confident that TESLA represents a step in the right direction. Second, it has to support this abstraction with efficient algorithms that exploit all the potentiality of modern hardware, as we investigated with our work on T-Rex. Finally, it has to provide automatic scalability with complex protocols that optimize the usage of available resources.

If we are able to answer all these research challenges, we will succeed in creating powerful systems that completely support event-based communication, simplifying the interaction between heterogeneous, distributed

components.

We envision a world in which mature CEP infrastructures create a strong abstraction over event detection and signalling, similar to the role DBMSs currently play for static data, hiding the algorithms required to detect complex situations and the low level details about network communication.

Bibliography

[1] 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.

[2] Krysia Broda, Keith Clark, Rob Miller 0002, and Alessandra Russo. Sage: A logical agent-based environment monitoring and control system. In *AmI*, pages 112–117, 2009.

[3] Event zero, http://www.eventzero.com/solutions/environment.aspx.

[4] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *In Proc. EDBT*, pages 627–644, 2006.

[5] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter R. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, 2009.

[6] Fusheng Wang and Peiya Liu. Temporal management of rfid data. In *VLDB*, pages 1128–1139. VLDB Endowment, 2005.

[7] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. Technical report, Politecnico di Milano, 2010. Submitted for Publication.

[8] Tim Bass. Mythbusters: Event stream processing v. complex event processing. Keynote speech at the 1st Int. Conf. on Distributed Event-Based Systems (DEBS'07), 2007.

[9] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM.

[10] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, June 2003.

[11] Nicholas Poul Schultz-Moeller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query optimisation. In *International Conference on Distributed Event-Based Systems (DEBS'09)*, Nashville, TN, USA, 07/2009 2009. ACM, ACM.

[12] Hervé Debar and Andreas Wespi. Aggregation and correlation of intrusion-detection alerts. In *Recent Advances in Intrusion Detection*, pages 85–103, 2001.

[13] E. Y.-T. Lin and Chen Zhou. Modeling and analysis of message passing in distributed manufacturing systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 29(2):250–262, 1999.

[14] Jonghun Park, Spyros A. Reveliotis, Douglas A. Bodner, and Leon F. McGinnis. A distributed, event-driven control architecture for flexibly automated manufacturing systems. *Int. J. Computer Integrated Manufacturing*, 15(2):109–126, 2002.

[15] Opher Etzion. Event processing and the babylon tower. Event Processing Thinking blog: http://epthinking.blogspot.com/2007/09/event-processing-and-babylon-tower.html, 2007.

[16] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. *SIGMOD Rec.*, 18(2):215–224, 1989.

[17] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[18] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and Zdonik S. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), 2003.

[19] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 309–320. VLDB Endowment, 2003.

[20] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 350, Washington, DC, USA, 2004. IEEE Computer Society.

[21] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, 2001.

[22] D.S. Rosenblum and A. L. Wolf. A design framework for internet-scale event observation and notification. In *Proc. of the 6th European Software Engineering Conf. (ESEC/FSE)*, LNCS 1301. Springer, September 1997.

[23] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *Proceedings of ACM SIGCOMM 2003*, pages 163–174, Karlsruhe, Germany, August 2003.

[24] Mehmet Altinel and Michael J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[25] G. Ashayer, H. K. Y. Leung, and H.-A. Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. In *Proceedings of the Workshop on Distributed Event-based Systems, co-located with the 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002. IEEE Computer Society Press.

[26] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 53–61, New York, NY, USA, 1999. ACM.

[27] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., 2010.

[28] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Comp. Syst.*, 19(3):332–383, 2001.

[29] Antonio Carzaniga and Alexander L. Wolf. Content-based networking: A new communication infrastructure. In *IMWS '01: Revised Papers from the NSF Workshop on Developing an Infrastructure for Mobile and Wireless Systems*, pages 59–68, London, UK, 2002. Springer-Verlag.

[30] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 71, Washington, DC, USA, 2006. IEEE Computer Society.

[31] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 49, Washington, DC, USA, 2006. IEEE Computer Society.

[32] D. Zimmer. On the semantics of complex events in active database management systems. In *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*, page 392, Washington, DC, USA, 1999. IEEE Computer Society.

[33] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001.

[34] Haifeng Liu and Hans-Arno Jacobsen. Modeling uncertainties in publish/subscribe systems. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 510, Washington, DC, USA, 2004. IEEE Computer Society.

[35] Segev Wasserkrug, Avigdor Gal, Opher Etzion, and Yulia Turchin. Complex event processing over uncertain data. In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 253–264, New York, NY, USA, 2008. ACM.

[36] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[37] Antony Galton and Juan Carlos Augusto. Two approaches to event definition. In *Database and Expert Systems Applications, 13th International Conference, DEXA 2002, Aix-en-Provence, France, September 2-6, 2002, Proceedings*, pages 547–556, 2002.

[38] Raman Adaikkalavan and Sharma Chakravarthy. Snoopib: interval-based event specification and detection for active databases. *Data Knowl. Eng.*, 59(1):139–165, 2006.

[39] Walker White, Mirek Riedewald, Johannes Gehrke, and Alan Demers. What is "next" in event processing? In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–272, New York, NY, USA, 2007. ACM.

[40] Dan O'Keeffe and Jean Bacon. Reliable complex event detection for pervasive computing. In *DEBS '10: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pages 73–84, New York, NY, USA, 2010. ACM.

[41] Andrew Eisenberg and Jim Melton. Sql: 1999, formerly known as sql3. *SIGMOD Rec.*, 28(1):131–138, 1999.

[42] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[43] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26:2003, 2003.

[44] Oracle. http://www.oracle.com/technologies/soa/complex-event-processing.html, 2010. Visited Nov. 2010.

[45] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: a data stream management system. In *SIGMOD*. ACM, 2003.

[46] Guoli Li and Hans-Arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware '05: Proceedings of the 6th ACM/IFIP/USENIX International Conference on Middleware*, pages 249–269. Springer-Verlag New York, Inc., 2005.

[47] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. On supporting kleene closure over event streams. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1391–1393, Washington, DC, USA, 2008. IEEE Computer Society.

[48] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 147–160, New York, NY, USA, 2008. ACM.

[49] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004.

[50] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[51] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *VLDB*, pages 215–226. VLDB Endowment, 2002.

[52] Yijian Bai, Hetal Thakkar, Haixun Wang, Chang Luo, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 337–346, New York, NY, USA, 2006. ACM.

[53] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. In *In Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR 2007)*, pages 363–374, 2007.

[54] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Query languages and data models for database sequences and data streams. In *VLDB*. VLDB Endowment, 2004.

[55] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The hipac project: combining active databases and timing constraints. *SIGMOD Rec.*, 17(1):51–70, 1988.

[56] Daniel F. Lieuwen, Narain H. Gehani, and Robert M. Arlein. The ode active database: Trigger semantics and implementation. In *ICDE '96: Proceedings of the Twelfth International Conference on*

Data Engineering, pages 412–420, Washington, DC, USA, 1996. IEEE Computer Society.

[57] Narain H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *VLDB*, pages 327–336, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[58] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Composite event specification in active databases: Model & implementation. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*, pages 327–338, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[59] Stella Gatziu and Klaus R. Dittrich. Events in an active object-oriented database system. In *Rules in Database Systems*, pages 23–39, 1993.

[60] Stella Gatziu, Andreas Geppert, and Klaus R. Dittrich. Integrating active concepts into an object-oriented database system. In *DBPL3: Proceedings of the third international workshop on Database programming languages : bulk types & persistent data*, pages 399–415, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[61] Sharma Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[62] S Chakravarthy, E Anwar, L Maugis, and D Mishra. Design of sentinel: an object-oriented dmbs with event-based rules. *Information and Software Technology*, 36(9):555 – 568, 1994.

[63] A. P. Buchmann, A. Deutsch, J. Zimmermann, and M. Higa. The reach active oodbms. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, page 476, New York, NY, USA, 1995. ACM.

[64] Henrik EngstrÃűm, Henrik Engstrm, Mikael Berndtsson, Mikael Berndtsson, Brian Lings, and Brian Lings. ACOOD Essentials, 1997.

[65] E. Bertino, E. Ferrari, and G. Guerrini. An approach to model and query event-based temporal data. *International Syposium on Temporal Representation and Reasoning*, 0:122, 1998.

[66] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, New York, NY, USA, 2003. ACM.

[67] Sirish Chandrasekaran and Michael Franklin. Remembrance of streams past: overload-sensitive management of archived streams. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 348–359. VLDB Endowment, 2004.

[68] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 261–272, New York, NY, USA, 2000. ACM.

[69] Vijayshankar Raman, Bhaskaran Raman, and Joseph M. Hellerstein. Online dynamic reordering for interactive data processing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 709–720, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[70] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. *Data Engineering, International Conference on*, 0:25, 2003.

[71] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838, New York, NY, USA, 2004. ACM.

[72] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: a scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, 2000.

[73] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for xml. *Comput. Netw.*, 31(11-16):1155–1169, 1999.

[74] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowl. and Data Eng.*, 11(4):610–628, 1999.

[75] Ling Liu and Calton Pu. A dynamic query scheduling framework for distributed and evolving information systems. In *ICDCS '97: Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, page 474, Washington, DC, USA, 1997. IEEE Computer Society.

[76] Mark Sullivan and Andrew Heybey. Tribeca: a system for managing large databases of network traffic. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[77] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 324–335. VLDB Endowment, 2004.

[78] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003. ACM.

[79] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan B. Zdonik. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, USA, 2005. ACM.

[80] Chuck Cranor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, and Oliver Spatscheck. Gigascope: high performance network monitoring with an sql interface. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 623–623, New York, NY, USA, 2002. ACM.

[81] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD '03: Proceedings of the 2003 ACM*

SIGMOD international conference on Management of data, pages 647–651, New York, NY, USA, 2003. ACM.

[82] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[83] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland, Oregon, July 2000. ACM.

[84] Robert E. Strom, Guruduth Banavar, Tushar Deepak Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel C. Sturman, and Michael Ward. Gryphon: An information flow based approach to message brokering. *CoRR*, cs.DC/9810019, 1998.

[85] R. Chand and P. Felber. Xnet: a reliable content-based publish/subscribe system. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004*, pages 264–273, Oct. 2004.

[86] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

[87] Ludger Fiege, Gero Mühl, and Felix C. Gärtner. Modular event-based systems. *Knowl. Eng. Rev.*, 17(4):359–388, 2002.

[88] Roland Balter. JORAM: The open source enterprise service bus. Technical report, ScalAgent Distributed Technologies SA, Echirolles Cedex, France, March 2004.

[89] Shrideep Pallickara and Geoffrey Fox. Naradabrokering: a distributed middleware framework and architecture for enabling durable peer-to-peer grids. In *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 41–61, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[90] D. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events, 1996.

[91] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21:717–734, 1995.

[92] M. Mansouri-Samani and M. Sloman. Monitoring distributed systems. *Network, IEEE*, 7(6):20–30, 1993.

[93] Massoud Mansouri-Samani and Morris Sloman. Gem: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4:96–108(13), 1997.

[94] Masoud Mansouri-Samani and Morris Sloman. A configurable event service for distributed systems. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, page 210, Washington, DC, USA, 1996. IEEE Computer Society.

[95] Jeff Magee, Naranker Dulay, and Jeff Kramer. Regis: a constructive development environment for distributed programs. *Distributed Systems Engineering*, 1(5):304–312, 1994.

[96] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, 1982.

[97] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A framework for event composition in distributed systems. In *In Proceedings of the 2003 International Middleware Conference*, pages 62–82. Springer, 2003.

[98] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: a high-performance event processing engine. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1100–1102, New York, NY, USA, 2007. ACM.

[99] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, 2008.

[100] Asaf Adi and Opher Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, 2004.

[101] IBM. http://www-935.ibm.com/services/us/index.wss, 2010. Visited Nov. 2010.

[102] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stuhmer, Nenad Stojanovic, and Rudi Studer. Etalis: Rule-based reasoning in event processing. In Sven Helmer, Alexandra Poulovassilis, and Fatos Xhafa, editors, *Reasoning in Event-Based Distributed Systems*, volume 347 of *Studies in Computational Intelligence*, pages 99–124. Springer Berlin / Heidelberg, 2011.

[103] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stuhmer, Nenad Stojanovic, and Rudi Studer. A rule-based language for complex event processing and reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *Web Reasoning and Rule Systems*, volume 6333 of *Lecture Notes in Computer Science*, pages 42–57. Springer Berlin / Heidelberg, 2010.

[104] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2006. ACM.

[105] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. Probabilistic event extraction from rfid data. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1480–1482, Washington, DC, USA, 2008. IEEE Computer Society.

[106] Aleri. http://www.aleri.com/, 2010. Visited Nov. 2010.

[107] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[108] Coral8. http://www.coral8.com/, 2010. Visited Nov. 2010.

[109] Coral8. http://www.aleri.com/WebHelp/coral8_documentation.htm, 2010. Visited Nov. 2010.

[110] Streambase. http://www.streambase.com/, 2010. Visited Nov. 2010.

[111] Streambase. http://streambase.com/developers/docs/latest/streamsql/index.html, 2010. Visited Nov. 2010.

[112] Esper. http://www.espertech.com/, 2010. Visited Nov. 2010.

[113] Tibco. http://www.tibco.com/software/complex-event-processing/businessevents/default.jsp, 2010. Visited Nov. 2010.

[114] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: a distributed, scalable platform for data mining. In *DMSSP '06: Proceedings of the 4th international workshop on Data mining standards, services and platforms*, pages 27–37, New York, NY, USA, 2006. ACM.

[115] Kun-Lung Wu, Kirsten W. Hildrum, Wei Fan, Philip S. Yu, Charu C. Aggarwal, David A. George, Buğra Gedik, Eric Bouillet, Xiaohui Gu, Gang Luo, and Haixun Wang. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on system s. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1185–1196. VLDB Endowment, 2007.

[116] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 431–442, New York, NY, USA, 2006. ACM.

[117] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134, New York, NY, USA, 2008. ACM.

[118] IBM. Business event processing white paper, websphere software, 2008.

[119] EventZero. http://www.eventzero.com/, 2010. Visited Nov. 2010.

[120] Progress-Apama. http://web.progress.com/it-need/complex-event-processing.html, 2010. Visited Nov. 2010.

[121] Mike Gualtieri and John Rymer. The Forrester Wave^{TM}: Complex Event Processing (CEP) Platforms, Q3 2009, 2009.

[122] Microsoft StreamInsight. http://msdn.microsoft.com/en-us/library/ee362541.aspx, 2011. Visited July 2011.

[123] Mohamed Ali. An introduction to microsoft sql server streaminsight. In *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research and Application*, COM.Geo '10, pages 66:1–66:1, New York, NY, USA, 2010. ACM.

[124] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Ying Li, V. Di Nicola, X. Wang, David Maier, S. Grell, O. Nano, and I. Santos. Microsoft cep server and online behavioral targeting. *Proc. VLDB Endow.*, 2:1558–1561, August 2009.

[125] LINQ Project Documentation. http://msdn.microsoft.com/en-us/netframework/aa904594, 2011. Visited Jan. 2011.

[126] Sharma Chakravarthy and Raman Adaikkalavan. Events and streams: harnessing and unleashing their synergy! In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 1–12, New York, NY, USA, 2008. ACM.

[127] Gianpaolo Cugola and Alessandro Margara. Raced: an adaptive middleware for complex event detection. In *ARM*, pages 1–6, New York, NY, USA, 2009. ACM.

[128] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[129] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[130] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43, New York, NY, USA, 1998. ACM.

[131] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[132] Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 203–214. VLDB Endowment, 2002.

[133] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 49–60, New York, NY, USA, 2002. ACM.

[134] Ella Rabinovich, Opher Etzion, and Avigdor Gal. Pattern rewriting framework for event processing optimization. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, pages 101–112, New York, NY, USA, 2011. ACM.

[135] Alexis Campailla, Sagar Chaki, Edmund Clarke, Somesh Jha, and Helmut Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.

[136] Alessandro Margara and Gianpaolo Cugola. High performance content-based matching using gpus. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, pages 183–194, New York, NY, USA, 2011. ACM.

[137] Alessandro Margara and Gianpaolo Cugola. High performance content-based matching using off-the-shelf parallel hardware. Technical report, Politecnico di Milano, 2011.

[138] Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. Contract-based load management in federated distributed systems. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2004. USENIX Association.

[139] Yanif Ahmad and Uğur Çetintemel. Network-aware query processing for stream-based applications. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 456–467. VLDB Endowment, 2004.

[140] Vibhore Kumar, Brian F. Cooper, Zhongtang Cai, Greg Eisenhauer, and Karsten Schwan. Resource-aware distributed stream management using dynamic overlays. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 783–792, Washington, DC, USA, 2005. IEEE Computer Society.

[141] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *OTM Conferences (1)*, pages 54–71, 2006.

[142] Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. Synergy: sharing-aware component composition for distributed stream

processing systems. In *Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, pages 322–341, New York, NY, USA, 2006. Springer-Verlag New York, Inc.

[143] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. Soda: an optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 306–325, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[144] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Buğra Gedik. Cola: optimizing stream processing applications via graph partitioning. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 1–20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[145] Geetika T. Lakshmanan, Ying Li, and Rob Strom. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, 12(6):50–60, 2008.

[146] E. Bozdag, A. Mesbah, and A. van Deursen. A comparison of push and pull techniques for AJAX. Technical report, Report TUD-SERG-2007-016a, 2007.

[147] Haggai Roitman, Avigdor Gal, and Louiqa Raschid. Satisfying complex data needs using pull-based online monitoring of volatile data sources. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1465–1467, Washington, DC, USA, 2008. IEEE Computer Society.

[148] A. Bagchi, A. Chaudhary, M.T. Goodrich, C. Li, and M. Shmueli-Scheuer. Achieving communication efficiency through push-pull partitioning of semantic spaces to disseminate dynamic information. *IEEE Transactions on Knowledge and Data Engineering*, 18(10):1352–1367, 2006.

[149] Nesime Tatbul and Stan Zdonik. Dealing with overload in distributed stream processing systems. In *ICDEW '06: Proceedings*

of the 22nd International Conference on Data Engineering Workshops, page 24, Washington, DC, USA, 2006. IEEE Computer Society.

[150] Nesime Tatbul and Stan Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 799–810. VLDB Endowment, 2006.

[151] Yun Chi, Haixun Wang, and Philip S. Yu. Loadstar: load shedding in data stream mining. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1302–1305. VLDB Endowment, 2005.

[152] Yanif Ahmad, Bradley Berg, Uğur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. Distributed operation in the borealis stream processing engine. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 882–884, New York, NY, USA, 2005. ACM.

[153] Sandeep Pandey, Kedar Dhamdhere, and Christopher Olston. Wic: a general-purpose algorithm for monitoring web information sources. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 360–371. VLDB Endowment, 2004.

[154] Jianliang Xu, Xueyan Tang, and Wang-Chien Lee. Time-critical on-demand data broadcast: Algorithms, analysis, and performance evaluation. *IEEE Trans. Parallel Distrib. Syst.*, 17:3–14, January 2006.

[155] Haggai Roitman, Avigdor Gal, and Louiqa Raschid. A dual framework and algorithms for targeted online data delivery. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints), 2010.

[156] Haggai Roitman, Avigdor Gal, and Louiqa Raschid. On tradeoffs in event delivery systems. In *DEBS '10: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pages 116–127, New York, NY, USA, 2010. ACM.

[157] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on*

Management of data, pages 827–838, New York, NY, USA, 2004. ACM.

[158] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 779–790, Washington, DC, USA, 2005. IEEE Computer Society.

[159] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1):1–44, 2008.

[160] Jennifer Widom and Stefano Ceri. Introduction to active database systems. In *Active Database Systems: Triggers and Rules For Advanced Database Processing*, pages 1–41. Morgan Kaufmann, 1996.

[161] Matthew Wright, James Chodzko, and Danny Luk. *Principles and Applications of Distributed Event-Based Systems*, chapter 1, pages 1–18. IGI Global, 2010.

[162] Arvind Arasu, Shivnath Babu, and Jennifer Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical Report 2002-57, Stanford InfoLab, 2002.

[163] Lukasz Golab and M. Tamer Özsu. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 658–669, New York, NY, USA, 2005. ACM.

[164] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming sql standard. *Proc. VLDB Endow.*, 1(2):1379–1390, 2008.

[165] Prabhudev Konana, Guangtian Liu, Chan-Gun Lee, and Honguk Woo. Specifying timing constraints and composite events: An application in the design of electronic brokerages. *IEEE Trans. Softw. Eng.*, 30(12):841–858, 2004. Member-Mok, Aloysius K.

[166] Dimitra Giannakopoulou and Klaus Havelund. Runtime analysis of linear temporal logic specifications. Technical report, RIACS, 2001.

[167] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356, London, UK, 2002. Springer-Verlag.

[168] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI 2004)*, pages 44–57, 2004.

[169] Luciano Baresi, Domenico Bianculli, Carlo Ghezzi, Sam Guinea, and Paola Spoletini. Validation of web service compositions. *IET Software*, 1(6):219–232, 2007.

[170] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[171] Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. *Handbook of automated reasoning*, pages 1635–1790, 2001.

[172] A. Pnueli. The temporal logic of programs. Technical report, Weizmann Science Press of Israel, Jerusalem, Israel, Israel, 1997.

[173] Doron Drusinsky. The temporal rover and the atg rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, 2000. Springer-Verlag.

[174] Oded Maler, Dejan Nickovic, and Amir Pnueli. From mitl to timed automata. In *FORMATS*, pages 274–289, 2006.

[175] Volker Stolz. Temporal assertions with parametrised propositions. In *RV*, pages 176–187, 2007.

[176] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and N. Garg, editors, *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*,

volume 4337 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, December 2006. Springer-Verlag.

[177] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL, 2010. ACM Transactions on Software Engineering and Methodology. Accepted for publication.

[178] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Runtime Verification, 7th International Workshop (RV 2007)*, pages 126–138, 2007.

[179] Andreas Bauer, Martin Leucker, and Christian Schallhart. Model-based runtime analysis of distributed reactive systems. In *17th Australian Software Engineering Conference (ASWEC 2006)*, pages 243–252, 2006.

[180] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 63–71, Washington, DC, USA, 2006. IEEE Computer Society.

[181] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of the execution of plans for web service composition. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*, pages 346–349. AAAI, 2006.

[182] Catriel Beeri, Anat Eyal, Tova Milo, and Alon Pilberg. Monitoring business processes with queries. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 603–614. VLDB Endowment, 2007.

[183] Luciano Baresi and Sam Guinea. Dynamo: Dynamic monitoring of ws-bpel processes. In *In Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC 2005)*, pages 478–483, 2005.

[184] Luciano Baresi and Sam Guinea. Towards dynamic monitoring of ws-bpel processes. In *In Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC 2005)*, pages 269–282, 2005.

[185] Luciano Baresi, Sam Guinea, Raman Kazhamiakin, and Marco Pistore. An integrated approach for the run-time monitoring of bpel orchestrations. In *ServiceWave '08: Proceedings of the 1st European Conference on Towards a Service-Based Internet*, pages 1–12, Berlin, Heidelberg, 2008. Springer-Verlag.

[186] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[187] Praveen Yalagandula and Mike Dahlin. A scalable distributed information management system. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 379–390, New York, NY, USA, 2004. ACM.

[188] David Luckham. http://complexevents.com/, 2010. Visited Nov. 2010.

[189] Opher Etzion. Event processing thinking. http://epthinking.blogspot.com/, 2010. Visited Nov. 2010.

[190] EPTS. http://www.ep-ts.com/, 2010. Visited Nov. 2010.

[191] C. Ghezzi, D. Mandrioli, and A. Morzenti. Trio: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, 1990.

[192] Angelo Morzenti, Dino Mandrioli, and Carlo Ghezzi. A model parametric real-time logic. *ACM Trans. Program. Lang. Syst.*, 14(4):521–573, 1992.

[193] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, New York, NY, USA, 2008. ACM.

[194] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. On supporting kleene closure over event streams. In *ICDE*, pages 1391–1393, 2008.

[195] Peter R. Pietzuch, Brian Shand, and Jean Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 18:44–55, 2004.

[196] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *PODS '04*, pages 263–274, New York, NY, USA, 2004. ACM.

[197] Ludger Fiege, Gero Mühl, and Alejandro P. Buchmann. An architectural framework for electronic commerce applications. In *GI Jahrestagung (2)*, pages 928–938, 2001.

[198] Christopher KrÃijgel, Thomas Toth, and Clemens Kerer. Decentralized event correlation for intrusion detection. In Kwangjo Kim, editor, *Information Security and Cryptology, ICISC*, volume 2288, pages 59–95. Springer Berlin / Heidelberg, 2002.

[199] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. of the 2001 SIGMOD Intl. Conf. on Management of data*, SIGMOD '01, pages 115–126, New York, NY, USA, 2001. ACM.

[200] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *Proc. of ACM SIGCOMM 2003*, pages 163–174, Karlsruhe, Germany, August 2003.

[201] Alexis Campailla, Sagar Chaki, Edmund Clarke, Somesh Jha, and Helmut Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proc. of the 23rd Intl. Conf. on Software Engineering*, ICSE '01, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.

[202] Amer Farroukh, Elias Ferzli, Naweed Tajuddin, and Hans-Arno Jacobsen. Parallel event processing for content-based publish/subscribe systems. In *Proc. of the 3rd Intl. Conf. on Distributed Event-Based Systems*, DEBS '09, pages 8:1–8:4, New York, NY, USA, 2009. ACM.

[203] CUDA. http://www.nvidia.com/object/cuda_home_new.html, 2011. Visited Jan. 2011.

[204] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A design pattern language for engineering (parallel) software: merging the plpp and opl projects. In *Proc. of the Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 9:1–9:8, New York, NY, USA, 2010. ACM.

[205] T S Axelrod. Effects of synchronization barriers on multiprocessor performance. *Parallel Comput.*, 3:129–140, May 1986.

[206] Scott Schneidert, Henrique Andrade, Buğra Gedik, Kun-Lung Wu, and Dimitrios S. Nikolopoulos. Evaluation of streaming aggregation on parallel hardware architectures. In *Proc. of the 4th Intl. Conf. on Distributed Event-Based Systems*, DEBS '10, pages 248–257, New York, NY, USA, 2010. ACM.

[207] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29:251–262, August 1999.

[208] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. Technical report, DIS, Università di Roma "La Sapienza", 2005.

[209] G. Mühl, L. Fiege, F.C. Gartner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *Proc. of the* 10^{th} *Intl. Symp. on Modeling, Analysis, and Simulation of Comput. and Telecom. Syst. (MASCOTS02)*, 2002.

[210] G. Cugola and G.P. Picco. REDS: A Reconfigurable Dispatching System. In *Proc. of the 6th Int. Workshop on Softw. Eng. and Middleware. (SEM06)*, pages 9—16, Portland, nov 2006. ACM Press.

[211] Antonio Carzaniga and Alexander L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in Lecture Notes in Computer Science, pages 59–68, Scottsdale, Arizona, October 2001. Springer-Verlag.

[212] Antonio Carzaniga, Matthew J. Rutherford, and Alexander L. Wolf. A routing scheme for content-based networking. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, China, March 2004.

[213] Antonio Carzaniga and Cyrus P. Hall. Content-based communication: a research agenda. In *SEM '06: Proceedings of the 6th Intl. Workshop on Software engineering and middleware*, Portland, Oregon, USA, November 2006. Invited Paper.

[214] Zbigniew Jerzak and Christof Fetzer. Bloom filter based routing for content-based publish/subscribe. In *Proc. of the 2nd Intl. Conf. on Distributed Event-Based Systems*, DEBS '08, pages 71–81, New York, NY, USA, 2008. ACM.

[215] Amer Farroukh, Elias Ferzli, Naweed Tajuddin, and Hans-Arno Jacobsen. Parallel event processing for content-based publish/subscribe systems. In *Proc. of the 3rd Intl. Conf. on Distributed Event-Based Systems*, DEBS '09, pages 8:1–8:4, New York, NY, USA, 2009. ACM.

[216] Kuen Hung Tsoi, Ioannis Papagiannis, Matteo Migliavacca, Wayne Luk, and Peter Pietzuch. Accelerating publish/subscribe matching on reconfigurable supercomputing platforms. In *Many-Core and Reconfigurable Supercomputing Conference (MRSC)*, Rome, Italy, 03/2010 2010.

[217] John Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron Lefohn, and Timothy Purcell. A Survey of General–Purpose Computations on Graphics Hardware. *Computer Graphics*, Volume 26, 2007.

[218] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proc. of the 37th Intl. Symp. on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[219] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. of the 17th Symp. on Operating Syst. Principles (SOSP99)*, pages 186–201. ACM Press, Dec 1999.

[220] Ed Burnette. *Hello, Android: Introducing Google's Mobile Development Platform*. Pragmatic Bookshelf, 2008.

[221] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*. IEEE Computer Society, 2006.

[222] András Varga. The omnet++ discrete event simulation system. *ESM*, 2001.

[223] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. of the 9th Symp. on Principles of Distributed Computing*, pages 219–227, Portland, Oregon, July 2000.

[224] Gianpaolo Cugola, Alessandro Margara, and M. Migliavacca. Context-Aware Publish-Subscribe: Model, Implementation, and Evaluation. In *Proc. of the Intl. Symp. on Computer and Communication*. IEEE Computer Society Press, 2009.

[225] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. Brite: An approach to universal topology generation. In *Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '01, pages 346–, Washington, DC, USA, 2001. IEEE Computer Society.

[226] PlanetLab, an open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org/, 2011. Visited Aug. 2011.

[227] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[228] Guoli Li and Hans-Arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware*. Springer-Verlag New York, Inc., 2005.