VRIJE UNIVERSITEIT BRUSSEL

Dissertation submitted in fulfilment of the requirement for the degree of Doctor of Philosophy in Science

REENTRANCY AND SCOPING IN RULE ENGINES FOR CLOUD-BASED APPLICATIONS

KENNEDY KAMBONA

June 2018

Promotor: Prof. Dr. Wolfgang De Meuter
Faculty of Science and Bio-Engineering Sciences

Dissertation submitted in fulfilment of the requirement for
the degree of Doctor of Philosophy in Sciences

Reentrancy & Scoping in Rule Engines for Cloud-based Applications

Kennedy Kambona

Promotor:
Prof. Dr. Wolfgang De Meuter

Jury:
Prof. Dr. Viviane Jonckers, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Katrien Beuls, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. Bart Jansen, Vrije Universiteit Brussel, Belgium
Prof. Dr. Joeri De Koster, Vrije Universiteit Brussel, Belgium
Prof. Dr. Slinger Jansen, Utrecht University, The Netherlands
Prof. Dr. Stijn Vansummeren, Université Libre de Bruxelles, Belgium
Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel, Belgium (promotor)

Abstract

Modern software systems are increasingly being deployed to the Cloud, leading to the rising adoption of systems that are predominantly online. An example of these are multi-tenant systems: Cloud-based applications that are shared by and instantiated for a multitude of tenants. Characteristically, such applications are often connected to different heterogeneous clients that are reactively uploading events and data, and are thus required to share the knowledge among the various tenants they support. Rather than hardcoding all this shared knowledge and ontologies in plain code, the knowledge can be easily programmed in the form of rules that orchestrate server-side logic, e.g., as business rules. In such situations, a rule engine is well-suited to accommodate the knowledge for clients of such reactive knowledge-driven applications.

Unfortunately, existing rule engines (and the rule-based languages they implement) were not conceptually designed to support and to cope with the knowledge and rules for multiple tenants at the same time. More specifically, they are unsuitable to support such heterogeneous systems because one has to manually hardcode the modularity and ownership of the knowledge for the various applications and clients, which quickly becomes complex and fallible. They further lack suitable semantics that developers can use to exploit collective or community knowledge that may apply to the data contributed by the various heterogeneous sources.

This dissertation presents Serena, a framework that provides scope-based reasoning in rule-based systems operating in heterogeneous environments. The solution exploits the fact that much of the community knowledge significant when performing reasoning and deductions can be structured in a hierarchy of scopes (that orchestrate which rules should be applicable to which incoming data). Serena augments an event-driven server with a Rete-based rule engine. Serena is distinct from existing rule-based systems due the notion of reentrancy and scoping that are efficiently ingrained at the heart of its inference engine. Serena further eases the encoding of reactive event patterns by clients in the form of scoped rules. Rule designers can utilise scoped rules to detect patterns in real-time data and to realise grouping structures in reactive knowledge-driven applications backed by a common rule-based system.

Samenvatting

Moderne softwaresystemen worden steeds vaker beschikbaar gesteld via de Cloud, waar ze online functioneren. Een voorbeeld van zulke systemen zijn multi-tenant systemen. Dit zijn Cloud-gebaseerde toepassingen die gedeeld en ge nstantieerd worden door meerdere gebruikersgroepen, zogenaamde "tenants". Karakteristiek aan zulke toepassingen is dat ze veelal verbonden zijn met meerdere heterogene gebruikers. Deze gebruikers sturen events en data op reactieve wijze naar de multi-tenant toepassing. De toepassing moet deze informatie waar nodig delen met de verschillende tenants, of juist gescheiden houden. In plaats van al deze gedeelde kennis en ontologie n manueel vast te leggen in code, kan deze kennis eenvoudig geprogrammeerd worden in de vorm van declaratieve regels, zoals "business rules". Met een regelgebaseerd systeem kan de server de kennis van de gebruikers dan op reactieve wijze verwerken.

Bestaande regelgebaseerde systemen (en de regelgebaseerde programmeertalen die ze implementeren) zijn echter niet ontworpen voor de noden van multi-tenancy. Meer bepaald missen bestaande systemen ondersteuning voor concepten uit heterogene systemen, waardoor manueel ge ncodeerd moet worden welke data bij welke toepassing of tenant hoort. Dit wordt snel complex en vergroot de kans op fouten in de regels. Bovendien ontbreken zulke systemen ook een gepaste semantiek voor het uitbuiten van "community knowledge": kennis die ook relevant is voor andere tenants buiten de bron van de data.

Deze thesis introduceert Serena: een framework waarmee regelgebaseerde systemen kunnen redeneren over "scopes" in heterogene omgevingen. De voorgestelde oplossing is gestoeld op het feit dat de relevante kennis van de community voorgesteld kan worden als een hi rarchie van scopes (die aangeven welke regels van toepassing zijn op bepaalde invoer). Serena voegt aan een traditionele event-gedreven server een regelgebaseerd systeem toe dat het Rete-algoritme gebruikt. Serena onderscheidt zich van bestaande regelgebaseerde systemen doordat reentrancy en scoping ingebakken zijn in de kern van de inference engine. Daarenboven maakt Serena het eenvoudiger om de reactieve voorwaarden van de gebruikers uit te drukken als regels met scopes. Ontwerpers van regels kunnen regels met scopes gebruiken om patronen te detecteren in real-time data en groepen te cre ren in reactieve event-gedreven toepassingen die ge mplementeerd zijn met behulp van een gezamenlijk regelgebaseerd systeem.

Acknowledgements

The Indian tale of the *The Blind Men and the Elephant* as it applies to community knowledge likewise encompasses my dissertation; as it would not be possible without the assistance, dedication and perseverance of a community surrounding my research life.

I would first like to express my gratitude and appreciation to my *promotor* Prof. Dr. Wolfgang De Meuter for his mentorship and guidance throughout the research period. In the same breath, I appreciate the insightful and candid feedback from my thesis committee about this work. Next in line to receive my sincere thanks are the people who have participated in some capacity to this research: Dr. Lode Hoste, Prof. Dr. Elisa Gonzalez-Boix, Dr. Ellie D'Hondt, Dr. Jorge Vallejos, and my officemate for years, Florian Myter. To my research sidekicks Thierry Renaux, Simon Van de Water and Humberto Rodriguez Avila, thanks for all the in-depth discussions inundated with the word *Rete*. And to my readers Simon and Nils (and later, Isaac), thanks being my extra set of eyes in this text.

I am also very grateful to everyone in the Software Languages and Engineering Lab (both current and ex-*Softies*) for providing me with valuable contributions, criticisms and other feedback. The *SOFT* lab embodies the most intellectually-thriving environment I have yet encountered – the conference trips, game days, barbecues, and other events were pretty fun. Our halls might be silent, but *in the slack* altercations abound. To me, *SOFT* is this home away from home, I will therefore be forever honoured to be a part of the family.

To the four people that have most shaped my life, my mother Margaret and father Alexander, sisters Nancy and Maureen Kambona (in order of nascency), cousin Dr. Moturi: this document is yours as much as it is mine. The same applies to other friends and family, to a lesser extent.

Finally, I would like to thank my life partner Jacky for all the support (and distractions in equal proportions). Yes, your time to reach this point is fast approaching, and I will be by your side when it does.

If you are still searching for your name here, your contributions were more important to me than you realise. Just wait for a less serious book ;) and you will most certainly be there.

Kushukuru haikudhuru. Asanteni.

Contents

List of Figures

1 Introduction

The most advanced systems currently on the Web have emerged as a result of embracing the culture of envisioning computing as a *utility*. This led to the prominence of today's Utility Computing [Arm+10]. Utility Computing lays its basis on time-sharing methodologies in the 1970s as applied to current online systems, i.e., by sharing resources in a cost-effective manner to provide services to end-users. One of the main reasons this phenomenon is popular is that the resources required to develop a software application targeted for mass markets are greatly surpassed by the resources needed to replicate its utility [Bro87].

Consequently, traditional software systems are increasingly being installed to run on the Web platform (i.e., deployed to the *Cloud*) to take advantage of Utility Computing and support a larger number of clients. Notable examples include customer relationship management provider Salesforce [Sal15], human resource and financial management service Workday [Wor15] and expense management software Concur [Con16]. Recently, research has emerged in newer paradigms such as Fog and Dew Computing [Wan15] whose main aim is to bring such services closer to the edge of the network, as opposed to the Cloud.

The current Internet age has therefore ushered in a dynamic architecture that supports the assimilation of newer structural and programming paradigms. A defining characteristic of these paradigms is that they all aspire to expose online services to a larger number of distributed end-users. These paradigms must remain robust given the high numbers of supported clients and data, and the increasing adoption of the Web by traditional desktop software developers.

As a result of the Web's dynamicity and increasing number of connected users, data-intensive applications with real-time processing of events have recently gained prominence. Events can consist of both low-level data such as raw GPS coordinates and high-level data that other applications depend on, such as detecting a package that is late for delivery. Such event data can be continuously sent over the Web to application servers in various forms as unordered, partially unbounded and/or time-varying sequences. This fundamentally differs from traditional systems in earlier ages of the Web where the processing of data was mostly done on finite, static datasets. Accordingly, there is currently a need for techniques that ease the dynamic definition of constraints that will be used to capture and extract value from this continuous, *reactive* data to infer potential higher-level knowledge that it may

uncover before the data decays. Currently, the technologies available to meet these goals are often times complicated and limiting due to the non-determinism and sheer volume of data to discern patterns on.

Recent advancements have re-discovered the use of rule-based systems consisting of rule-based languages and rule engines, in areas that take advantage of *business rules* [H+00] using engines such as Drools [Bro09] and Jess [Fri03]. These systems can be used to support the development of dynamic software applications that have been deployed to the Cloud. Rule-based systems exhibit clear advantages over other programming tools that are used in modern software: they provide an advanced concept by representing the conceptual logic of a system in an application-independent way. This is particularly useful to Web applications running on modern platforms for a number of reasons. A rule-based system reduces the complexity of developing and supporting applications, making the whole process more uniform: one of the most important contributions of rule-based systems to software development is the separation of the *what* the program should capture rather than the nuances of the *hows* [Bro87]. Furthermore, the fundamental power of rule-based systems comes not only their complex inference mechanisms but also their ability to embrace rich knowledge bases that reflect aspects of the real world [LF91].

The modern Web is characterised by a definitive *heterogeneous* environment. Different *types* of client devices (or *things* [Dav11]) contribute diverse types of data that are expected to be processed in a uniform manner promptly. Sharing brought about by Utility Computing is significant in exploiting collective knowledge that can be discovered from the data that these devices contribute. However, sharing unearths issues that can be attributed to classically isolated rule engine design. Conceptually, rule engines were designed in the era where isolated computing was prevalent. At the time, rule engines were programmed to encode a localised set of rules and to work on homogeneous data. Traditional rule engines thus suffer from a lack of proper modularisation when installed to serve such heterogeneous settings with shared data. They are characterised by a *flat design space* where activations could be observed from all data without discriminating their sources. As we will show, current methods to mitigate these problems unnecessarily retract the gains made by utilising a rule-based system in the first place: it increases the complexity of application development making it fallible, and muddles the design of the conceptual logic of the application.

The vision of this dissertation is two-fold. First, we argue that rule languages for developing heterogeneous applications should expose specialised constructs that enable rule creators to exploit community knowledge. These constructs allow combining or distinguishing events pertaining to different event sources, while keeping this logic cleanly separated from the application logic. Second, modern rule-based systems deployed to support these multi-user platforms should provide a mechanism that manages the multiple users and the knowledge that they share using *reentrancy* techniques that modularise the rule base in a way that it reflects the corresponding user fact base. In this way, the rule language constructs can enable rule creators to specify constraints that support data consolidation and/or discrimination in heterogeneous settings.

1.1 Research Context

The inspiration for the work presented in this dissertation spans several domains in software languages and engineering. We outline the specific areas next in the context of providing programming and processing support of dynamic knowledge-intensive, data-driven Web applications.

1.1.1 Community Knowledge in Heterogeneous Environments

The concept of service provision through sharing of resources in Utility Computing gave rise to heterogeneous systems such as those seen in multi-tenancy. Multi-tenancy refers to Web-based architectures that provide a shared application instance that serves a number of clients or *tenants* [Pat+11]. A multi-tenant application therefore is installed on a single instance and serves all clients from that instance. This is analogous to traditional software modules that supported simultaneous access from multiple users or terminal environments. Research in this domain splits multi-tenancy into two major groups [Guo+07]. The first is known as *multiple-instances multi-tenancy*, where tenants have their own dedicated application instance over shared hardware or operating system. The second is *native multi-tenancy* (or true multi-tenancy), where all the tenants share a single shared application instance as opposed to multiple isolated ones. These two patterns differ in terms of engineering and operational costs, customisability and scalability. Comparably, while multi-instance multi-tenancy solutions scale to support several dozen tenants, native multi-tenancy solutions can scale better and can support several hundreds of clients using similar resources [Guo+07].

The heterogeneous multi-tenant applications running on the Web platform are usually characterised by a shared server instance that is connected to different clients raising various events and exchanging data. The server therefore has the potential to reason about contributed events (as data) collectively via the concept of community knowledge. Community knowledge enables the concise collection of useful information that can be sourced from a variety of data contributed by different types of clients. This is significant in the context of heterogeneous systems due to the vastness and diversity of the types of information that can be produced, collected and processed from connected clients. These systems are thus required to manage the shared resources reused by the various applications they support. However, we argue that current approaches for such architectures use various techniques that do not intrinsically support the flexibility and expressiveness required to distinguish or capture community knowledge. In this dissertation we aim to support such heterogeneous applications through a technique that enforces partitioning of client constraints and data using rule-based programming constructs.

1.1.2 Reactive Web Applications

Recently, there has been a shift of the view of the Web server from a host that relays static documents to one that is event-driven and processes dynamic content from connected clients. The role of such a server is to wait and listen in on a stream for events, and subsequently push responses back to clients in a *reactive* way. One programming paradigm that has emerged for modern Web applications with such timely requirements is reactive programming [Bai+13]. Reactive programming puts greater significance on the data flow of a program during development, as opposed to the common control flow. To enforce this, the paradigm abstracts time-varying events and values for their consumption by a programmer. The programmer can then define functions that will react upon incoming events and their data.

Reactive programming approaches support abstractions that model continuous and discrete time-varying values. The abstractions are first-class values and can be passed around or even composed within the program. These kinds of abstractions reduce the complexity of dealing with timely event occurrences prevalent in Web applications. With imperative approaches, whenever an event occurs programmers have to explicitly perform re-computations and ensure data dependencies from changes brought about by the event are methodologically updated. With the reactive programming approach programmers need

not explicitly trigger a re-computation of time-varying data. In our work related to this thesis [KBD13] we performed an evaluation and demonstrated how reactive programming and promises can be used to build highly interactive multi-user Web applications. In this dissertation we extend the notion to support such reactive knowledge-intensive web applications in capturing community knowledge.

1.1.3 Reactive Knowledge-driven Applications

Many applications are increasingly residing on the web often in heterogeneous Cloud-based architectures. Coming up with techniques that extract value from reactive data to infer higher-level knowledge that it may uncover promptly (e.g. before the data decays) is a challenge, given its *volume* and *velocity*. Currently, the programming of applications and processing of content to meet these goals is complicated and limiting. Programming reactive Web applications involves dealing with nondeterministic external (and at times, internal) events, which are difficult to control in traditional sequential programming languages [KBD13]. Techniques that attempt to solve these issues, such as event-based programming, can cause code that becomes hard to understand in larger systems leading to what has been termed as "asynchronous spaghetti code".

To help build reactive Web applications, various libraries and frameworks for the Web have recently emerged that support reactive programming (such as Flapjax [Mey+09] and Elm [Cza12]) and stream processing (such as Esper [BV07] and Aurora [Che+03]). Most of these approaches operate on data streams and provide constructs for **computation-oriented event processing** (COEP) such as aggregation (e.g. calculating a summation based on inbound data) [Ali+15]. In this dissertation we focus on more specific **detection-oriented event processing** (DOEP) which is vital when reasoning about higher-level knowledge or combinations of events through *event patterns*. In COEP, processing on event streams is done on ordered sets, and there is often a high throughput of events that are processed by queries. DOEP focuses more on isolated event-by-event processing on potentially out-of-order events via a number of constraints formulated as rules in classical rule-based systems. Rule-based systems that provide programming logic through production rules and that perform complex event processing through rule engines are prevalent in the DOEP domain (e.g., Drools [Bro09]).

Rule engines in rule-based systems can be split into two categories according to their reasoning process: *forward* and *backward chaining* systems. In forward-chaining, the conditions of rules determine the reasoning process while in backward-chaining the goal of a rule influences the reasoning process. During rule engine execution, control is hinged on the basic re-evaluation of the data states with the rules, and not by any form of static control structures explicitly defined in the program as in conventional instruction-driven programs. Consequently, computation in such rule-based systems is said to be *data-driven* [GR98b]. Forward chaining is well-suited to detect complex events from non-deterministic simple events and is thus commonly used in detection-oriented event processing. The most popular data-driven rule-based systems with forward-chaining are production systems [New73] because they provide efficient pattern matching through the Rete algorithm [For82]. Production systems still form the core of much larger software systems in various domains today [Fri14; Pro15].

For this work, we envision supporting low to medium-scale RKDAs, providing complex services via specialised hosting on the Cloud – ranging from having rulesets of 5 rules processing 100k events per second, to having 500 rules processing 15k events per second. However, most RKDAs tend to send events to the server non-deterministically and, as such, the server often receives events in irregular intervals. We illustrate day-to-day examples of

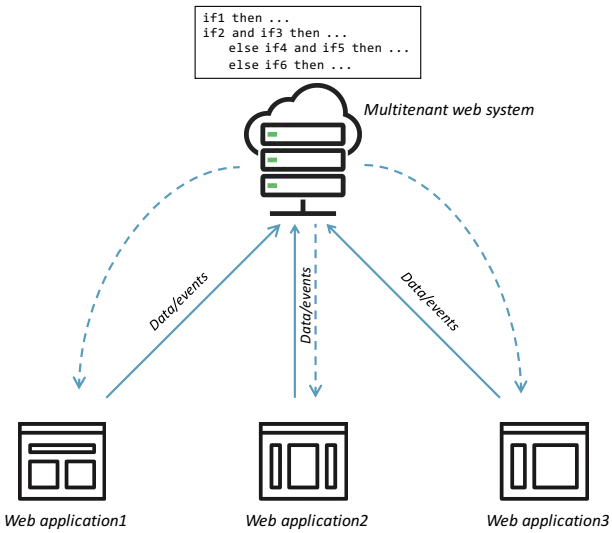


Figure 1.1: Example of a heterogeneous RKDA – These systems are required to manage the shared knowledge base reused by the various applications they support.

such applications later in Section 2.2.3.

1.2 Problem Statement

In this dissertation we focus on supporting reactive knowledge-driven Web applications. These systems are usually connected to different clients intermittently sending events and data, and are thus required to manage the shared knowledge base reused by the various applications they support. An example is a server that supports applications that require the detection of complex events from different event sources (e.g., in devices participating in a Smart Space [Wan+02]). The application server runs in the Cloud and can support applications, e.g. for security monitoring, for all clients subscribed to the service (Figure 1.1). Each client installs a security monitoring application that is connected to the server using bidirectional communication channels and the server needs to process data from clients sent as events. In order to reason about the data and extract *value*, i.e., higher-level knowledge, it is vital that the value of the sent data be extracted *efficiently* – notwithstanding the massive and intermittent nature of contributed data. Rather than hard-coding all the shared knowledge and ontologies, this knowledge can easily be programmed in the form of rules to program server-side logic (e.g., as business rules [H+00]). In such situations, a modern rule engine can be used to accommodate the knowledge for clients of these multi-tenant web systems.

The executing process of a rule-based system is known as the inference engine. It is tasked to find out which rules are relevant to the current elements in its state, the working memory, and will fire the selected rules. This is fundamentally performed in three stages forming the match-select-execute cycle [GR98b]. Unlike classic rule-based systems, in *reactive* rule-based systems computation does not halt when the working memory matches a goal or reaches an equilibrium (i.e. after the `execute` stage): the engine defaults to a `wait` stage that awaits any newly asserted facts, after which it proceeds to perform the

match-select-execute cycle (Section 2.5.3).

However, **inference engines were not conceptually designed to work in multi-user environments** such as multi-tenant architectures. Classic rule-based systems (e.g., production systems) are intrinsically **non-reentrant**: they are characterised by a flat memory and rule set where activations can be observed from all asserted facts without discriminating their sources. In other words, they are designed for single-use, in order to support isolated knowledge-driven applications.

Reentrancy is fundamental in applications that are intended to be used in multi-user environments [DD14]. The term describes computer programs that are written in such a way that multiple users can share the copies of data in memory. Reentrant code is a requirement in common multi-user systems such as operating systems; where system programmers ensure that whenever a program is executed for a particular user, there can be no other instructions that will interfere with data intended for another user. These features cannot be easily implemented in rule-based systems. In programs written using current rule-based approaches, programmers are forced to 'mimic' reentrancy by manually hard-coding distinctions between clients and their data sources within the rules (Chapter 5). Evidently, this quickly becomes complex and fallible to enforce using rule semantics when the number of users and the relationships between them increases. In the context of multi-user environments, failure to properly make these distinctions can cause unintended rule activations in other clients.

Rule engines therefore require orchestration within rules to discriminate or distinguish between data instances contributed by different entities. This dissertation proposes **scoped rules** that enforce reentrancy in multi-user rule-based systems. Users and developers specify these rules to detect patterns in real-time data and to realise grouping structures in knowledge-intensive rule-based applications.

Scoped rules enable rule creators to **distinguish between events pertaining to different clients** while keeping this logic cleanly separated from the actual semantics of the rule. As such, the basic purpose of the rule is not muddled with the logic required for distinguishing clients. This leaves the logical intent of a rule easy to understand for a rule creator. At the same time, scoping enables us exploit a number of performance optimisations in the server's inference engine during the matching process. Our approach of encoding the physical and logical organisations of multi-user applications eases the computational workload of the inference engine. This further decreases the engine's overall response time, thereby increasing its processing efficiency.

This dissertation aims to provide a reactive rule-based system for reactive knowledge-driven applications that mitigates reentrancy problems by extending its rule-based syntax with scoping extensions. The extensions enforce data discrimination during execution of the inference engine processing cycle. Concretely, the research goals are:

- To support the development of reactive Web applications for multi-user environments by exposing a reactive rule-based language for client rule definitions. This will enable detecting client-specific complex events and will absolve clients of the complexities when defining constraints in streams of events,

- To support the dynamism of today's Web environment by presenting a DOEP rule-based framework based on the Rete algorithm that allows runtime definition of rules by clients that can be *dynamically added* in the engine during execution, *incrementally evaluates* new events with newly-asserted definitions and *reactively sends feedback* to notify clients selectively,

- To allow programmers to write reentrant rules by augmenting the rule language with scope-based semantics that define *scoped rule syntax*, which allows orchestration within rules to discriminate or distinguish between instances of different client entities,
- To enforce reentrancy in multi-user rule-based systems by infusing the Rete algorithm with *scope-based reasoning* in the inference engine's execution cycle that discriminates data matches as defined in scoped rules,
- To evaluate the approaches that this dissertation proposes and their effect on the efficiency of a Rete-based inference engine when compared to classical approaches.

In short, we present our vision from the research areas of rule-based systems, reactive technologies and collective intelligence with the aim of providing a collegial approach for the advancement of the current state-of-the-art. We identify the following two general statements this thesis will claim:

1: Modern knowledge-intensive web applications have specific reactive, data-driven requirements that are solved using rule-based system consisting of an inference engine for execution and a rule-based language for programming logic.

2: That reactive knowledge-driven applications are also characterised by concurrent, multi-user features introducing reentrancy problems in traditional inference engines. This thesis presents a solution that introduces scope in rules and inference engines, and goes further to optimise the approach using encoding techniques.

Serena: Constituent Parts

$$\begin{aligned} Serena &= \text{Event-driven server} + \text{Forward-chaining rule engine} + \text{Scoping component} \\ &= Reactivity + Reasoning + Heterogeneity \end{aligned}$$

1.3 Overview

To address the challenges and to pursue the research goals mentioned in the previous section, this dissertation proceeds to employ two main program design and engineering artefacts: a rule-based programming language and a compatible rule engine presenting a unified framework that spans both the server and client Web architectures. This makes the languages both easier to use – an advantage for users – and provides avenues for engine optimisation – an advantage for the server platform.

1.3.1 Dynamic Rule Architecture

In order to fulfil the goal of easing the development of reactive knowledge-driven applications for multi-user environments, this dissertation proposes a framework that will allow remote clients to define and install logic reactive rules on the server. Rule definitions allow ways in which the system is able to monitor and adapt to user requirements more efficiently. Using the unified framework, remote clients can add reactive rules to the server. Reactive knowledge-driven applications (the focus of this thesis) require *dynamicity* not only in the data that is contributed, but also in the defined constraints or rules that clients

have uploaded to the server. Therefore, clients can dynamically upload constraints as rules and data as events to the server engine during the server engine execution, which is then appended to the rule engine. The framework also maintains push-based communication between the client and the server for instantaneous feedback in the form of notifications of activated rules. The supporting system running on the server should therefore reconfigure itself in order to dynamically adapt to the changing requirements of the remote clients - without the need for re-deployment. This goes in tandem with the *always online* nature of applications running in today's dynamic software environment.

1.3.2 Scoped Rule-based Language

In order to represent client knowledge, this work provides a rule-based syntax that clients can use to define their constraints as rules. Rule definitions allow for a declarative way to define complex event detection patterns. When defined, client rules are uploaded to the server. The rules are subsequently appended to the server's rule engine, which appends each rule to the inference engine's internal representation. Each rule defines the real-time detection constraints of interest for a particular client. In general, the inference engine will process and detect any events that the client is interested in. Once a rule is activated the framework notifies the relevant client(s) according to the client rule's notification semantics.

In a heterogeneous environment such as a multi-tenant system there is need to distinguish data from different types of clients. Instead of embedding logic for distinguishing clients in the main logic of the rule, Serena exposes scoped rule definitions by extending normal rule syntax with scope-based definitions. The scope definitions specify scope-based constraints on clients, their groups and the relationships between them. The scoped rules therefore specify which data to match, who to notify and what information is sent with the notification. Further, to ease the learning curve of rule design, the approach provides an optional intermediate means for rule definitions. Rule creators can optionally design rules for the framework using a provided graphical user interface that exposes a visual programming editor to build client rules.

1.3.3 Reentrant Rule Engine

The rule engine on the server is based on Rete, one of the most the widely-used models of knowledge representation known as the production systems model [New73]. The distinguishing feature of production systems is the use of data-sensitive rules rather than sequenced instructions as the basis of computation. To this end, this dissertation presents a reactive web server augmented with a traditional forward-chaining inference engine. This enables servers supporting Web applications to realise the computationally-intensive process of receiving and reactively processing data in order to detect complex events, together with accompanying data relevant to notify clients.

Rule-based systems consist of a number of unordered rules referencing a global working memory. Support for multiple, heterogeneous users in a single rule engine instance is needed, and can be enforced by making the Rete algorithm reentrant such that it can purely handle multiple inference states simultaneously for different sets of client rules. In order to enforce reentrancy during the inference engine's match cycles, it is imperative that an efficient representation is used to represent multiple users internally and to quickly determine the relationships between the data being processed at runtime. This dissertation proposes embracing the concepts of physical or logical *groups* of tenant clients and their relationships, common in multi-user applications [KKH11]. Examples of groups include

research groups in a university, branches in an organisation, hobby categories in forums and area zones when monitoring distributed sensor networks.

This dissertation targets reactive knowledge-driven Web applications and, as such, efficiency is immensely significant in providing a responsive service by the rule-based engine given, i) the scale of data and number of heterogeneous users and, ii) the reactive setting that these applications exhibit. A number of existing research that improves efficiency by reducing the highly combinatorial evaluations made during rule engine execution reported high performance benefits in their results [Doo95; HH93; Kim+14; MNM77; XZ10]. Given this, the proposed framework internally converts structures of clients (or client group representations) into an efficient encoding. The principal idea is that we precompute, store and maintain an internal representation efficiently as an encoding that will be used to expeditiously process constraints used to enforce reentrancy within the inference engine. Our vision is to use an encoding method that performs (near) constant-time operations to entirely determine client data relationships in the structure. This is vital because during the match-execute cycle, the Rete algorithm performs most of its processing in the join nodes as the dataset increases: therefore these operations would dramatically affect the performance per cycle.

1.4 Approach & Methodology

After presenting the overview, this section presents the approach and methodology used in this dissertation. In general, this thesis aims to invent a formalism in which rule designers in heterogeneous environments can express patterns for data discrimination concisely, in a way that reduces the accidental complexity [Bro87] encountered when expressing these patterns using normal rule logic. The drawback of using normal rule logic to express such patterns also negatively affects the execution of the underlying rule engine. We therefore observe that there exists a balance on the flexibility vs efficiency spectrum, where if the patterns we expose to rule designers are too flexible or abstract, then the implementation is no longer efficiently implemented; and if we restrict the patterns we expose, we can have a more efficient implementation. We summarise the process that will be used to address these challenges and to pursue these goals. The summary parallels the six-step design science research process that is described by Peffers et al. in [Pef+06].

Problem – The thesis focuses on reactive knowledge-driven web applications (RKDAs), introduced in Section 1.1.3, that can be supported using knowledge extraction of CEP patterns in the form of a shared rule based system; a rule-based language for programming logic to detect event patterns, and a rule engine for execution. Moreover, with a shared rule-based system developers can be able to exploit community knowledge, i.e., knowledge that has been contributed by data coming from different types of heterogeneous clients (Section 1.1.1). However, we observe that classic rule-based systems were designed for single use, and therefore run into reentrancy problems when trying to orchestrate events from a variety of heterogeneous clients (Section 1.2).

Objectives – The main objective is to provide a unified, reactive rule-based framework for RKDAs that exposes a simple formalism using scoping for expressing and enforcing data discrimination, and for capturing community knowledge efficiently in the rule engine. The research goals for this dissertation supporting this main objective were concretely discussed at the end of Section 1.2. Subsequently, this dissertation identifies the requirements that such a system should have to in order to implement the objective.

Through these requirements, we dissect the current state-of-the art, with a specific focus on the support of heterogeneity in systems that provide reasoning from a series of events in Chapters 3 and 5.

Design and development – To pursue the objectives, this work presents the design and development (in Chapters 6 and 7 respectively) of the Serena framework, a modern detection-oriented rule-based system for RKDAs that runs on a Web server. In Serena, clients can dynamically upload and install rules on the server, and can receive feedback in the form of push-based notifications. In addition, the framework presents Serena^s, which exposes a rule-based language with scoping extensions that rule designers can use to enforce data discrimination during the execution of the rule engine. This step was discussed in detail as the scoped rule-based language and its reentrant rule engine in the overview in Section 1.1.

Demonstration – The efficacy of the vision behind the Serena framework to solve the problem of reentrancy in heterogeneous rule engines is explained using a driving scenario. The scenario is representative of a typical reactive knowledge-driven application with event capture, reactive processing and rapid response of detected complex events. The scenario includes a security monitoring system requiring reactive detection and processing of independent events from different company employees, and is representative of a reactive knowledge-driven application. This dissertation presents this running example introduced in Section 2.3, that continues throughout the text, to explain how the framework solves the problems of reentrancy, and how scoping can be effectively used to capture community knowledge.

Evaluation – The benefits of the developed Serena^s framework are presented using qualitative and quantitative analyses. The qualitative analysis delves into related work and compares how current systems support the identified requirements. The analysis then proceeds to compare these current systems with the constructs provided by the Serena^s framework, showing that the proposed framework improves on the current support for heterogeneity in modern rule-based systems. For the quantitative analysis, the benefits afforded by Serena^s are evaluated using scenarios representative of an RKDA in Cloud-based, heterogeneous environment. The scenarios are modelled and tested in extensive experimental simulations, and the results are analysed, comparing them with current alternative approaches in Chapter 8.

Communication – This dissertation, and its supporting internationally peer-reviewed publications outlined in Section 1.6, are the culmination of communicating the reentrancy problems of heterogeneous rule engines in the Cloud, and how the proposed approach using scopes presents a viable solution to these problems.

1.5 Contributions

After giving a brief description of the methodology that this thesis will embrace, we summarise its main contributions as follows:

A study of the open issues, limitations and shortcomings of supporting reactive knowledge -driven Web applications in multi-user environments that require reasoning semantics. We perform a literature study on the broad domain of such knowledge-driven systems for the Web and propose a number of criteria that these systems need to fulfil.

A unified reactive, rule-based framework for heterogeneous environments that supports reactive knowledge-driven applications. The framework realises the computationally-intensive process of receiving and reactively processing data in order to detect complex events, together with sending relevant data when notifying clients.

A rule-based language augmented with extensions to define scoped rules for community knowledge. – Scope-based syntax can be applied to rules in order to support scope-based constraints in heterogeneous rule-based systems.

A reentrant inference engine that embraces scope-based reasoning. A modification to the Rete algorithm for inference engines that enforces reentrancy by incorporating techniques from bit-vector encoding, which discriminates data matches as defined in scoped rules.

An evaluation of a reentrant inference engine that is based on practical use cases. The evaluation shows that given the structured knowledge representation of heterogeneous clients, scoped rules are much easier to formulate and understand, and the inference engine itself can process a larger number of access requests at a faster rate than the traditional rule engine.

1.6 Supporting Publications

This work was a result of a number of primary and secondary publications.

1.6.1 Primary Publications

The publications that constitute the primary basis of this thesis are outlined below.

- **Harnessing Community Knowledge in Heterogeneous Rule Engines.** *Lecture Notes in Business Information Processing, vol. 322.* Kennedy Kambona, Thierry Renaux, Wolfgang De Meuter.

  This paper outlined the benefits accrued from using scope-based reasoning in heterogeneous rule engines as a means to capture collective intelligence via community knowledge. Using a simulated case study, it was confirmed that the technique presents a viable approach for efficiently representing and processing community knowledge in heterogeneous environments.

- **Efficient Matching in Heterogeneous Rule Engines.** *Proceedings of the 30th International Conference on Industrial, Engineering, Other Applications of Applied Intelligent Systems: IEA/AIE 2017. Advances in Artificial Intelligence: From Theory to Practice. Lecture Notes in Computer Science, vol. 10350, pp. 394.* Kennedy Kambona, Thierry Renaux, Wolfgang De Meuter.

  The paper presented the scope-based hashing algorithm (SBH) in a heterogeneous rule-based framework that enables efficient matching in scoped rule engines based on the Rete algorithm. SBH introduces scoped hash tables in alpha memories that help in avoiding unnecessary join tests that hamper performance. The experimental results showed that SBH significantly decreases the response time of rule engines in heterogeneous environments having entities sharing the same knowledge base.

- **Reentrancy and Scoping in Multi-tenant Inference Engines.** *Proceedings of the 13th International Conference on Web Information Systems and Technologies 2017.* Kennedy Kambona, Thierry Renaux, Wolfgang De Meuter.

  The paper presented Serena, a rule-based framework that supports multi-tenant reactive web applications. The distinctive feature of Serena is the notion of reentrancy and scoping in its inference engine, which is the key solution in making it multi-tenant. The validation was performed through a simulated case study and a comparison with a similar ad-hoc approach. The results showed that Serena's flexible approach improves computational efficiency in the engine.

- **Coordinating collaborative interactions in Web-based applications.** *Proceedings of the 2015 International Conference on Interactive Tabletops & Surfaces.* Kennedy Kambona, Lode Hoste, Elisa Gonzalez Boix, Wolfgang De Meuter.

  Focused on coordination constructs for programmers of collaborative knowledge-driven applications. Such applications can today support heterogeneous collaborative interactions for multiple clients simultaneously, but lack programming language abstractions for coordinating these complex interactions. The paper presented a framework for dedicated coordination programmer constructs that blends techniques common in complex event processing and group communication.

1.6.2 Secondary Publications

The publications that served as indirect research avenues for reinforcing the ideas behind this dissertation are outlined below.

- **Serena: A scalable middleware for real-time Web applications.** *Proceedings of the 30th Annual ACM Symposium on Applied Computing 2015.* Kennedy Kambona, Lode Hoste, Elisa Gonzalez Boix, Wolfgang De Meuter.

  Focused on using an inference engine to support real-time processing on the web. The position paper presented a middleware for real-time web applications that utilises a rule-based approach achieve real-time constraints, giving instantaneous feedback. The middleware also abstracts the underlying infrastructure needed to support real-time communication.

- **An evaluation of reactive programming & promises for collaborative Web applications.** *Proceedings of the 7th Workshop on Dynamic Languages and Applications 2013.* Kennedy Kambona, Elisa Gonzalez Boix, Wolfgang De Meuter.

  This work investigated the expressiveness of approaches that tackle problems with inversion of control in event-driven code, namely reactive extensions and promises. The paper presented a case study and subsequent analysis consisting of an online collaborative drawing editor modelled using the aforementioned techniques. From the analysis, the work proposes a roadmap of how to improve support of programming event-driven Web applications.

1.7 Outline

An outline of the chapters that this dissertation presents are enumerated below.

1. *Introduction.*
2. *Background & Motivation.* Presents the basic principles of rule-based systems, and their justification for use on the Web.
3. *Related Work.* Delves into the related work event processing systems having support for knowledge-driven applications.
4. *Serena: Cloud-based RBS.* Introduces the Serena framework, a Cloud-based rule engine based on the Rete network.
5. *Heterogeneity in RBS.* Examines the effects of heterogeneity in multi-user RBSes.
6. *The Serena^s Scoped Rule Language.* Presents *Serena^s*, a modern rule-based language with scopes.
7. *Serena^s: Reentrant Cloud-based RBS.* Presents enhancements to the Serena engine with scope-based reasoning.
8. *Evaluation.* Describes the setup and evaluation of the proposed heterogeneous rule engine.
9. *Conclusion.* Concludes with a summarised recap, presenting some hindsights and possible future extensions.

Code listings, information boxes and appendices

Throughout this document we introduce shorter code examples as listings. Code with more details or with multiple parts are deferred to the later sections in the appendix. Information boxes will be used to provide context or place emphasis on specific ideas in the discussion.

2

Reactive Knowledge-driven Applications

Turn, turn, my wheel! All things must change; To something new, to something strange; Nothing that is can pause or stay…

Henry Wadsworth Longfellow, *Kéramos*, 1878

The modern software ecosystem is predominantly *online*. This phenomenon exposes a number of challenges for traditional methods of creating and processing software applications. In this chapter we outline the differences between the classical and the modern Web environments, and how they influence software development. In doing so, we motivate the need for managing real-time processing of data, and how this can effectively be handled using a mixture of classical rule engines and state-of-the-art programming using reactive techniques[1].

2.1 Introduction

Traditionally, software systems were conceptually designed to run on single, isolated architectures. With the invention of networking at the end of the 1960s and subsequently the World Wide Web in 1991, internetworking (i.e., the Internet) brought interconnected systems and information to homes and businesses alike.

Today, traditional web applications have evolved into supporting dynamic, data-driven and *reactive* applications with large numbers of users and client devices that require real-time responses. This has become evident as connectivity has continued to become cheaper and faster, leading to more powerful ways of supporting modern systems running on the Web. An example of such systems is the information repository known as the Knowledge Graph that enhances Google Search's real-time engine results. Since its inception in 2012, the graph processes more than 70 billion facts [Pay16].

[1] Some observations described in this chapter have been published as [KBD13] and [Kam+15].

Fundamentally, the changes in the Web architecture have affected modern software systems in various ways. The next section presents an account that indicates the changes that programming and processing in the Web underwent over the years, highlighting their significance on the current web environment.

2.1.1 The Rise of the Dynamic Web

The ideas behind a global Web used to discover and retrieve information can be traced as far back as 1946 in a short by Murray Lenister[2] titled "A Logic Named Joe" [FS09]. The article, in some ways, predicted the rise of the internet and interactive computing by a '*Logic the size of a breadbox sitting on a desk linked with other commonplace logics in homes*'[3]. The Lenister short envisioned the idea of an information network today accessible from people's homes. It would be concretised later in September 2, 1969 with the imminent deployment of ARPANET to four universities in the USA for research. Later on in 1991, the use of the internet as the backbone for connectivity was presented to the public domain with Tim Berners-Lee's *The Information Mine* global hypertext project known as the World Wide Web. The Web was conceived as an information retrieval initiative aiming to "give universal access to a large universe of documents".

The induction of the Web to the general public fuelled the need for various programming and processing techniques that developers and engineers would be able to develop general-purpose applications that would run on this new architecture. What followed was a number of technical and infrastructural evolution spanning over three *ages* starting from the initial 'Static Age' to the current 'Dynamic age' [Dri11], as we illustrate in Figure 2.1. We summarise the ages next.

- **Static Age** (1990 - 1999)

  The original idea of having a 'big, virtual document system in the sky' by Berners-Lee resulted in a network of static hard-coded files that made up the Web. These files were programmed in the largely static HTML1, 2 & 3.2 languages and formed the content on the Web after *publishing* on web servers. Classic web browsers such as Viola and Mosaic/Netscape were used to connect to the servers to retrieve, format and display files to users, who were simply consumers of web content.

  The features of static server access and retrieval of information made the Web to be considered as a static, read-only medium. The processing was lacking due to limited server processing performance and bandwidth of the medium – long pages often slowed down the entire site during retrieval. The Web 1.0 retronym is often used to refer to this Age.

- **Semi-dynamic Age** (1999 - 2009)

  From some of the deficiencies of the Static Age came the Semi-dynamic Age, where information on the Web was additionally actively contributed by users. Content on the Web evolved from publishing to *participation* by users: often through blogging, online bids and early social media platforms. Consequently, this changed the landscape of the technologies used to program and process websites. Web pages were programmed increasingly using languages such as PHP and ASP.

[2] A *nom de plume* of William Fitzgerald Jenkins

[3] Lenister is said to have a more accurate description of the current PCs and the Web than the often-quoted Vannevar Bush in [Bus+45]

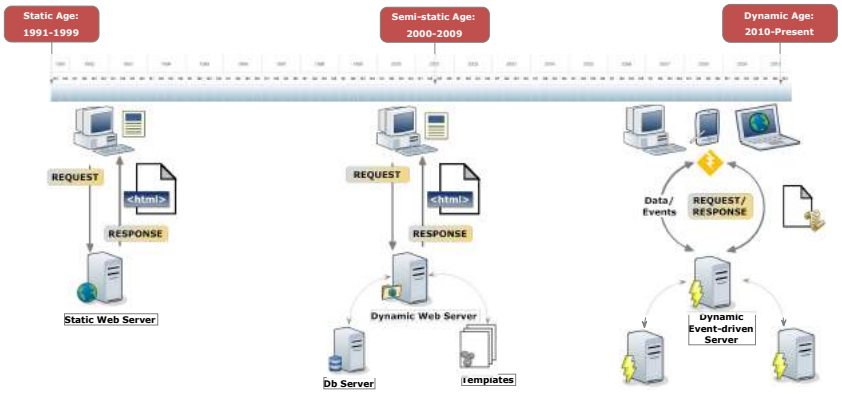


Figure 2.1: Changes in the Web over the ages – The earlier stages of the Web were based on static files but today it has evolved to embrace dynamic technologies.

The usual processing pattern in this age was to retrieve server-based content by use of predefined templates to prepare documents using content from databases, which were processed using server-side technologies like CGI [Wor14], and Servlets [Ora11]. Files were then usually presented statically in HTML4.01 & XHTML to the client. The architecture of the applications in this era followed a 3-tier web-stack. This age was referred to by the term Web 2.0.

- **The Dynamic Age** (2009 - Present)

  This third generation of the Web has been transformed into a more seamless and interoperable whole rather than separate units. In this age, the Web is characterised by a richer network of interactive contextualised information in order to make it easy to be dissected, processed and shared between cooperating web applications efficiently. The focus is shifting from simple websites to fully-fledged interactive *web applications and services*. This is significant because enriched and interconnected information greatly increases its value. This has led to the more intelligent processing of web content by server technologies, such as recommender systems [Kum+98], micro-formats [Biz+13], search using natural languages [WLY04], etc.

  Also significant is the improvement in broadband connectivity, influencing a rise in mobile internet access and streaming services. As a result, advanced distributed applications have emerged spanning desktop, mobile and other *smart* devices. These applications are not just programmed as simple webpages but as fully interactive, event-driven web applications through which information moves using various event streams. The most popular programming language for web applications is JavaScript and HTML5[4].

From this account of the changes the Web has experienced over the years, two influences are evident. The first is that cheaper connectivity and the rise of the number of connected smart devices or things has rapidly *increased the need of online processing of large quantities of data*. The second is that there is an increase in the assimilation of *newer programming paradigms* that remain robust given the dynamicity of the modern Web. The

[4]Some other languages can compile to these languages

waterfall effect of this is increasing adoption of Web architecture as a platform by traditional "desktop" software developers. We discuss the repercussions of these two influences in the upcoming sections.

2.1.2 Foundations of Cloud-based Heterogeneity

The changes that the Web platform has undergone were driven by the support of internet applications that were previously exclusively running in isolated environments. With cheaper networking hardware and the subsequent rise of the Internet, developers realised the potential of utilising the Web as a platform. Accordingly, software systems that leverage the Web platform started being increasingly deployed to online software servers.

The need for a model that allows for availability of configured computing resources on demand that resides on online servers (given the name "*The Cloud*") subsequently arose. The model was referred to as Cloud Computing [Arm+10]. Cloud computing provides (seemingly infinite) computing resources and services that can be easily provisioned, configured and deployed online with little management required.

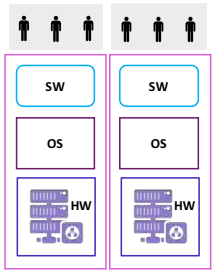
Cloud Service Models

As the Internet matures, the trend of modern institutions seeking more complex software solutions for their operations is shifting to cloud service providers. Cloud service models generally consist in three forms: Infrastructure as a Service (IaaS) is actual hardware provided by service providers available to users on demand. Platform as a Service (PaaS) is when different services on the cloud provider's platform offer computing resources to end users. Software as a Service (SaaS) includes direct applications that exist as cloud services offered by the provider remotely.

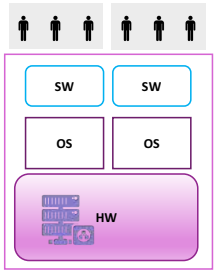
Utility Computing SaaS is becoming the more prevalent Cloud service delivery model, primarily because its underlying technologies are maturing much faster and newer approaches supporting SaaS applications are becoming more popular [So11]. Of the diverse descriptions that are attributed to the Cloud, the most distinguishing characteristics are virtualisation, scalability and a utility model for service provisioning. The latter is best known as *Utility Computing* with a pay-as-you-go model that takes advantage of cost reduction through resource sharing of online services. Utility Computing provides a way in which modern software systems can share online services and resources in a cost-effective manner. It represents the modern take on *time-sharing*, a prominent computing model used in the 1970s during the computing era of mainframes. Time-sharing initiated the idea behind envisioning computing as a utility by allowing more efficient means of using mainframe resources by jobs from different connected users.

Multi-tenancy In today's Cloud, however, utility computing has surpassed the simple models of time-sharing due to the advanced techniques employed in the *multi-tenancy* domain where economies of scale play a more significant role. Multi-tenant architectures enable a number of customers, *tenants*, to transparently share system resources [Kab+15] via a Cloud service provider. An example is Salesforce, which mainly provides a database architecture shared by several tenants [Sal15].

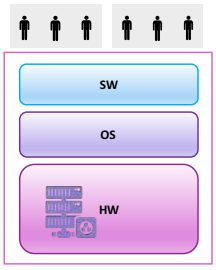
Multi-tenant architectures can generally be split into two types, illustrated in Figure 2.2 [Pat+11]. The Figure illustrates different hardware and software configurations for



(a) Single-tenant



(b) Multiple-instance multi-tenancy



(c) Native multi-tenancy

Figure 2.2: Comparison of types of multi-tenancy with single-tenant architecture – Multi-instance multi-tenancy exhibits duplication of resources, while native multi-tenancy or true multi-tenancy utilises fully shared resources among tenants.

providing Cloud services. *Multiple-instances multi-tenancy* is where the tenants have their own dedicated application instance over shared hardware or operating system, while *native multi-tenancy* or *true multi-tenancy* is where the tenants have a single shared application instance over various hosting resources. The two differ in terms of engineering and operational costs, customisability and scalability. Comparably, multi-instance multi-tenancy solutions scale to support several dozen tenants, native multi-tenancy solutions scale much better and can support several hundreds of tenants using similar resources. They however do have some disadvantages, the forefront being lack of control, thereby leading to problems with data security. The multi-tenant systems we shall focus on consist of a single instance that is shared across all clients, or tenants, from that instance. Overall, however, multi-tenant solutions simplify the deployment of applications for customers online. This is particularly significant when observing the emergence of technologies supporting modern Web applications.

2.2 Trends in the Dynamic Web

The rise of the dynamic Web has made it inadequate to rely on traditional methods of web application development. This is evidenced by the emergence of newer programming paradigms and supporting processing techniques for servers on the Web. In this section we intertwine the two, discussing the effects of newer programming paradigms on the processing backends and vice-versa. To facilitate the discussion, we begin with a foundation of how the processing of large data sets or *big data* has influenced trends in the modern web environment.

2.2.1 The Web and Big Data

Given the adoption of the internet, there are currently an estimated 6.4 billion devices, with 20 billion devices (from ordinary computers to sensors and smart devices) projected to be inter-connected by 2020 [Dav11]. These devices contribute to larger systems (e.g. within 'smart' homes and offices via Smart Spaces [Wan+02]) by gathering and pushing an unprecedented amount of data to a back-end web infrastructure that is required to perform high-level processing like data aggregation, monitoring and analysis.

By and large, the modern Web has been noted as the primary conduit of the staggering scale of the quantity of data and processing that is observed in current systems. This large amount of data contributed by devices occurrs at a record rate, and has been aptly named *big data*. Big data has been classified by researchers as having the following characteristics [VOE11]:

- **Volume** – The high capacity of data produced from varied event sources. The volume of data greatly affects efficient processing of information.
- **Veracity** – The level of trust that can be attributed to the data. Veracity goes hand-in-hand with the data quality and can be extended to confidentiality and security as well.
- **Velocity** – The rate of data generation from event sources and transfer to processing servers. The velocity is also intertwined with the rate of response by the processing entity as this affects the responsiveness of the system as a whole.
- **Variety** – The different types of data collected from different event sources such as devices with sensor and GPS data. Variety introduces challenges in isolation and convergence of big data in different contexts.
- **Value** – The possibility of discovering hidden values from its raw form. The value of big data is especially significant because the large scale greatly reduces the conventional methods used to extract meaningful information.

Technologies that support these big data characteristics can be categorised into two: online and offline. Offline big data technologies usually focus on post-processing numerous amounts of collected data for analytical purposes that spans all or most of the data in static batches e.g., in techniques such as Hadoop [Apa09]. In contrast, online big data technologies focus on real-time processing where incoming data is promptly ingested and possibly cached, as seen in online processing approaches such as Flink [Car+15].

This dissertation focuses on real-time detection and processing of data sets, which falls in the general area of online big data technologies. Detecting and processing online data sets in larger quantities, however, poses a great challenge that is distinct from those faced by offline approaches. These challenges are discussed in the coming sections.

Orchestrating Real-time Events in Online Big Data

Events play a major role in driving the main functionality of applications that rely on online big data technologies: Facebook, Twitter, and LinkedIn together are estimated to process more than 12 million events per second [Pel+15]. These applications have a much higher level of interactivity, i.e., the responsive communications between users and system components. Various entities produce data collectively (albeit intermittently) and expect prompt responses whenever an event they have subscribed to occurs.

Ma in [Ma11] noted that a major challenge of current services is the inability to adapt to the highly interactive modern Web environment, like providing *real-time* or near real-time services. Real-time services provide *reactive content* to a number of clients as soon as it is readily available in the form of events and return immediate feedback in form of notifications. From today's era of environments such as interactive multi-user online communities (in social networks, crowdsourcing), newer demands for developing web applications have emerged. A practical example is in detecting composite events from simpler but *disorderly* events, used to compose interactions in approaches such as in synchronous web-based applications for Groupware [RG92]. This is indeed the key distinguishing feature of current trends from previous methods in the earlier ages and offline big data processing technologies: the necessity of the continuous processing of data from a large number of event sources to instantaneously return timely but accurate data. The research challenge is in investigating supporting technologies for the detection, processing and orchestration of events from such interactive, dynamic web applications – all in real time. In this dissertation we term such applications as *reactive web applications*.

2.2.2 Value Extraction in Reactive Web Applications

Value is a significant feature when employing reasoning within big data to discern patterns and is one of the main principles of this dissertation. Huge amounts of data are virtually useless if the data does not provide value to the application and its end-clients. A major challenge in related research is coming up with techniques that extract value from the reactive data to *infer* potential higher-level knowledge that it may uncover before the data decays.

Significance of Value in Heterogeneous Data

In reactive big data, discovering interesting trends and patterns from massive, dispersed or entangled pieces of data greatly improves its quality. A good analogy for this is the ancient *Parable of the Blind Men and An Elephant*, that originated from the *Rigveda* collection in ancient India [JB14]. The parable describes a number of blind men sizing up an unknown object, a large elephant. Their goal is to try and determine what the object or creature is, and project the result to the others. Each man feels only one part of the elephant's body and is then required to describe the elephant from this basis. Since each man has only a limited, local perspective of the elephant, they come up with different conclusions of what the object can be: for instance, one says the elephant could be a wall, a spear or a rope depending on the limited region they can access (respectively, the side, the tusk or the tail), Figure 2.3. The fable ends with the blind men in complete disagreement of what the object is.

In the same way, *value* from data contributed by a variety of sources or clients on the Web can potentially be increased if it is extracted or processed collectively. This will significantly increase its value because the information is fed into a deterministic process where patterns within a set of collective information can be ascertained. One field where this is of practical significance is in the organisational intelligence domain, where the result of the process is often termed as collective knowledge or collective intelligence [All97]. In this dissertation we will use the term *community knowledge*. Community knowledge is obtained by a process of extracting a concise collection of useful information sourced from different clients. In practical scenarios the clients are often grouped according to logical or physical similarities. Community knowledge is important in the context of multi-tenant Cloud architectures due to the vastness and diversity of the types of information that

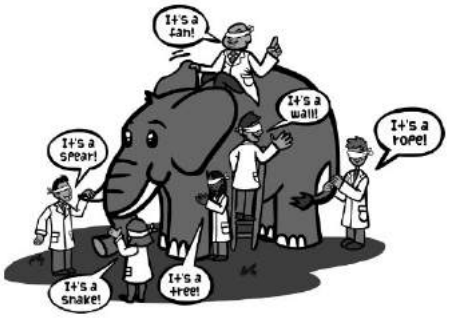


Figure 2.3: The Blind Men and the Elephant – Each blind man has a limited view of the elephant and comes up with different conclusions of what the object is (Image sourced from [Com]).

can be produced, collected and processed from different clients. This type of composite knowledge in such heterogeneous environments therefore encompasses the value and variety characteristics of reactive big data.

Inferring higher-level knowledge from large amounts of data forces a majority of techniques to require that such processing be performed offline [Gha+13; Hu+07; Yan+09]. Research in techniques that support faster and efficient ways of processing incoming data in real-time for knowledge extraction in the Cloud are therefore needed in the modern dynamic Web ecosystem. This is further complicated by today's need for discovering richer community knowledge from diverse data sources. We refer to applications that require extraction of value in the form of community knowledge through the online, reactive processing of diverse sources of data as *reactive knowledge-driven applications*.

2.2.3 Reactive Knowledge-driven Applications (RKDAs)

Reactive knowledge-driven applications (RKDAs) are modern applications that collect information produced by multiple distributed sources and need to process it in a timely manner in order to extract new knowledge in the form of community knowledge. The knowledge extraction follows the techniques in traditional knowledge-based applications and is driven by a reasoning process. The collective information consists of event data that is usually sourced from users, client devices, sensor activity, etc., and is sent to a reactive event-driven server for processing.

RKDAs can be observed in several domains. One is in the traditional *complex event processing* domain, where data from multiple sources is correlated or combined to infer events and patterns that are meaningful for quick responses [Luc02]. An approach on the Web is *stream reasoning*, a newer area where techniques are used to integrate data streams and reasoning systems in the Semantic Web [Val+09]. Examples of practical implementations of reactive knowledge-driven applications include:

- Package monitoring – RFID-based inventory management performs continuous analysis of RFID scans to track paths of shipments in order to capture possible irregularities while in transit [WL05]. Providing tools for determining problems in shipments sourced from various suppliers that end up in customers in different area/time zones can be a challenge.

- Network management – These systems analyse network packets in real-time to ascertain

traffic flow patterns for efficient routing, bandwidth monitoring and network security. An example is intrusion detection, used to promptly detect and possibly anticipate attacks in a network and to generate alerts when a known sequence of port scanning and flooding occurs [ALB11], [Cra+03]. In this case, detecting an elusive robot port scanner can prove to be elusive if the agent performs the attack from different IPs using a set of proxies.

- Environmental monitoring – Data from sensors deployed in different zones needs to be processed to acquire information about the observed zones – to detect anomalies or to predict various emergencies as soon as possible [Res+09]. Designing an early warning system that collects data from different observed area zones to provide interventions to different action zones through a number of aid agencies can be a challenge.

- Automated manufacturing – A system can analyse scanned items on-the-fly for real-time logging purposes and to detect anomalies in the manufacturing process, e.g., missing items in the conveyor belt [ZDZ09]. More superficial problems can be detected from analysing outputs from multiple conveyor flows as opposed to a single one.

- Fraud detection – Banks and other agencies need to detect occurrences of fraud by processing and inspecting real-time streams of credit card transactions [Wid+07]. It is increasingly becoming a requirement for multiple agencies to liaise with each other in order to crack down more intricate money laundering schemes.

- Stock and currency monitoring – Some financial applications require a continuous analysis of stocks to discover correlations, identify trends/spikes and forecast stock/currency prices [Luc02] for trading with short or long positions. An example is in the cryptocurrency space, where trades of one reference currency e.g., Bitcoin, can have an effect on other cryptocurrencies as well; so movements in multiple markets may need to be analysed. Fast processing of such data is significant because more recent results increase the value of possible gains [SÇZ05].

The next section outlines some of the most common characteristics of RKDAs.

2.2.4 Characteristics of Reactive Knowledge-driven Applications

Knowledge-based reasoning model Reactive knowledge-driven applications require extracting value from incoming raw data in unpredictable sequences. The data from various clients on the Web has to be processed to extract knowledge, forming a knowledge base that will enrich the functionality that modern applications are required to provide. As stated, this dissertation focuses on applications where the clients can send data intermittently and out-of-order. In general, the complexity of processing this kind of data is beyond the evaluation of simple filter predicates checking whether certain values are within pre-defined bounds. In RKDAs, the reasoning process can be extremely complex due to the non-deterministic nature of incoming data, e.g., events from sensor devices transmitting data intermittently. A knowledge-based reasoning model can be used to detect these complex events in this way instead of relying on programming predefined sequences or steps which often becomes convoluted.

Event-driven communication model The types of RKDAs on the Web that this dissertation focuses on operate via a central server, with various distributed users and devices

contributing data to the shared server. To support the distributed nature of the environment and the potential large number of clients that send data intermittently to the system, interactions in RKDAs are best suited to be based on the event-based model. The event-based model is said to be memory-efficient because there is less context to store than the traditional threaded model, and time-efficient since the expenses of context switching are greatly reduced. In the modern Web environment, these features enable the model to continuously process and respond to the large number of client requests that RKDAs observe.

Responsive processing model Related to the dynamic properties of today's online big data, RKDAs also need immediate processing to provide responsive feedback to relevant parties, with minimal latency: e.g., detection of an intrusion in an alarm system should be prompt. Dealing with RKDAs therefore requires a responsive processing model, preferably employing incremental processing techniques to reduce latency. This becomes important in RKDAs that are modelled around data that exhibits decaying features, i.e., rapidly losing its value over time [Kai+13]. A responsive processing model is therefore needed by RKDAs to produce immediate actionable knowledge. This is evident in the proliferation of responsive Web systems such as social applications like Twitter and Facebook where clients require prompt processing of real-time updates.

Active feedback mechanism An additional characteristic of RKDAs is that regardless of the amount of data that is contributed and processed, there is need for responsive feedback to relevant devices or end-users with minimal latency. This is evident in the proliferation of dynamic web systems and other distributed systems requiring real-time monitoring, such as package trackers and online home security applications. In these cases, immediate feedback is needed in order to receive real-time updates that may affect future actions of supported applications. For instance, a responsive feedback can allow an online re-configuration of the running system, an attractive albeit challenging feature in today's dynamic web architecture.

Collective intelligence through community knowledge Even though online value extraction is significant in RKDAs, what makes them stand above similar approaches is that not only is the data to be processed is 'big', but the realisation that a diverse range of autonomous sources contribute the data in one integrated system. The state of this multi-tenant system is subsequently shared by all the autonomous sources. This has a profound effect on the complexity and the relationships that can be unearthed in this shared system. This aspect is especially significant in RKDAs that share a common model for representing knowledge.

2.3 Driving Scenario

This section presents an example scenario of a reactive knowledge-driven application that requires prompt detection of events from different sources. The application scenario is inspired by the concept of having ubiquitous devices in Smart Spaces [Wan+02]: in this scenario, a security system is deployed in an office environment to monitor different access patterns using scanning devices.

The higher management of a software consultancy company *Kimetrica* has embarked on improving the security of its internal physical office environment. They have passed

regulations that require the security department to monitor accesses of all staff and other members while inside the company via security protocols. The regulations stipulate that location accesses should be immediately reported to the responsible security body for quick analyses and responses.

Throughout the company premises, the board has purchased and installed devices that scan employee badges at major access points. Permanent and contracted employees, upon registration, are issued identification badges, and are required to wear their badges at all times. The badges are contactless smart cards that can be read by the access devices. To gain access to any part of the company, a staff member is required to scan their badge on the device.

For this simplified scenario we now illustrate a few of the security protocols that have been drafted by the security department (from the regulations). These are only a few of the protocols that can be enforced in the company and will be applicable to all the members in the working environment.

- **Protocol 1**

  Company employees have access to their offices during working hours

- **Protocol 2**

  Permanent staff in the systems development department have car access to parking on the weekends from 10am to 4pm

- **Protocol 3**

  Non-staff members have access to corridors if they have a pre-existing 4-digit code issued by mid-level employees

- **Protocol 4**

  Interns are allowed access to the intern cubicles in restricted times from 8am until no later than 8pm

This example is representative of a reactive knowledge-driven application. An application supporting this scenario would typically exhibit the characteristics of RKDAs outlined in Section 2.2.4. A supporting monitoring system should be capable of capturing the requests from all employees automatically at varied times whenever a request is made, the data sent for processing can grow to be a huge amount. The number of access devices connected to the processing entity to send request data is high, littered all over the company's physical environment. Furthermore, the data is sent in realtime. Therefore the security system needs to handle a large amount of data that comes from different access requests at different times. It needs to detect those requests against the said regulations in a timely manner, by correlating low-level data read from the badges and the devices to high-level situational replies to the access requests. With this functionality, the system can detect actionable information that can be used for logging or decision-making. In a nutshell, transforming this raw data into useful knowledge is significant in achieving the security monitoring regulations specified by higher management.

We will frequently refer to this driving scenario throughout this dissertation as one instance of a reactive knowledge-driven application.

2.4 Programming and Processing Paradigms for RKDAs

In general, RKDAs can be developed using common methods, i.e., via conventional imperative programming techniques in languages such as Java and C# for programming (server-side) applications. Given the fact that in the modern setting much of the creation of the large amounts of data is occurring rapidly and in a non-deterministic fashion – generated from sensors, timers, user interactions etc., – these approaches are known to lead to unnecessary convoluted code when detecting these irregular events [Sch+11]. There have therefore been attempts that tackle these kinds of problems. This section outlines some of the proposed state-of-the-art solutions, in light of supporting reactive knowledge-driven applications. In particular, it focuses on how to program them using suitable definitions to capture events of interest, and how systems can process requests promptly providing immediate feedback.

In principle, technologies for developing applications that process real-time data sets should be able to adequately support all their characteristics, from coping with its high volume to discovering hidden values. Processing large quantities of data in real-time, however, presents unique challenges as opposed to processing the data offline. Developing web applications that process content dynamically and supporting the processing of such kinds of content can be done using a number of recent approaches. The next sections describe the common methods and identify some of their limitations.

2.4.1 Reactive Data & Complex Events

In order to support the processing and detection of non-deterministic events, servers supporting RKDAs need to be responsive: aside from managing more concurrent connections, the processing systems should be able to promptly respond to many requests while remaining robust. In this case, the traditional thread-per-connection model is known to have limitations such as scalability issues due to huge load, and increased risk of race conditions when authoring modules due to complexity of orchestrating multi-threaded applications [Ous96]. As mentioned in Section 2.2.4, the event-driven model is designed to make it easier to handle such issues – delegating incoming work to background processes thus being able to handle larger loads.

In traditional systems the execution of a program is modelled around the call stack. We present an example in the code of Listing 2.1 that calls a function performing a database query and returning the results. When the function `db` is called, the caller on line 7 has to wait until the database query is complete and the function's return value is received – in this case `rows`. Execution is *synchronous*, and only continues when the return value is received and the context is restored.

Listing 2.1: Conventional synchronous database request

```
1 function db(a,b){
2   var rows = database.query(query);
3   return rows;
4 }
5
6 var query = "Select count(*) from employees";
7 var result = db(query); /* execution waits forthe result */
8 console.log(result);
```

In contrast, event driven architectures do not logically use the concept of call stack [Hoh06]. Their execution is based on primitives known as **events**. An event is an abstraction of an

interaction. We show the same database query code in asynchronous event-driven style in Listing 2.2. The execution of the main code and the function call `database.query` are decoupled. The function call is *asynchronous*, therefore the caller needs to provide a handler or *callback* that will handle the response of the database call in line 3. Execution can now continue without waiting for the expensive database request to complete. When the request completes, an event is fired and the callback is executed.

Listing 2.2: Event-driven asynchronous database request

```
1 var query = "Select count(*) from employees";
2 /* database call with callback */
3 database.query(query, function(err, rows) { //
4       var result = rows;
5       console.log(result);
6 });
7 /* execution continues without waiting */
```

The event-driven model is said to be memory-efficient since there is less computational context to store, and time-efficient because the expenses of context switching are greatly reduced [HO06]. As a result the model is well-suited for distribution as it provides a decoupled model that is highly scalable and requires little coordination. The event-driven approach has been identified by researchers to be a natural fit for reactive processing in todays software environment [Dab+02; Dun+06; Lev+04; Mey+09]. The approach has contributed to the popularity of a number of technologies on the Web. For instance, its style embracing asynchronous events has led to the rise of the event-based JavaScript language. Its execution semantics makes the paradigm performant and scalable, and has contributed to the growth of the event-driven Node.js server. Given this, we present the first requirement for support of reactive knowledge-driven applications.

- **[Requirement 1] Event-driven processing & communication model**

  Modern Web applications make heavy use of fine-grained requests to the server during their operation [Sch+08]. This is more evident in RKDAs, where client requests may arrive at any time (as intermittent events) and from multiple distributed sources. There is therefore need for a viable concurrency model that offers an efficient and scalable means to support RKDAs. Furthermore, because they operate in a distributed environment, server engines supporting RKDAs must be able to consider how to handle a number of clients connecting to the server to transfer and receive data as well as to process these requests. The server should be able to support connections that allow clients and devices to send reactive event data intermittently, and allow the server to send both feedback and notifications to clients after processing client data. In the driving scenario from Section 2.3, for instance, in their day-to-day operations any number of employees can make random access requests at different times of the week. When an access request is made, it is sent to a server and the device awaits immediate response in order to allow or reject the request made by the employee. These requests need to be sent to the monitoring system through a model that requires the least overhead in coordination and performance when receiving and processing them.

  As presented in this section, the event-based paradigm provides a model that enables such a supporting system to be able to provide these types of requirements in an efficient and scalable way. An event-driven server provides the required execution and communication model that enriches the applications it supports by making the system responsive to user requests in a decoupled manner. As a result, the model

enables the server receiving requests to be able to process requests from a large number of clients, typically observed in RKDAs.

2.4.2 Detecting Complex Events

Event-driven reactivity is a natural programming model for web applications [Mey+09], and more-so for RKDAs. RKDAs are characterised by events that need to be orchestrated using event sequences that deliver reactive data at non-deterministic intervals. In these cases, the conventional programming abstractions available for detecting correlated or more complex events make developers run into various challenges when orchestrating these events. The most notable problem that has recently received high recognition is the *callback hell*: deeply-nested callbacks that have dependencies on data returned from previous asynchronous invocations.

To expound on the problem we illustrate an example that applies conventional techniques in a simple RKDA (also presented in [KBD13]). The example is of an online drawing editor wherein a programmer intends to detect a simple dragging of a shape on the canvas. In the design of the code a programmer would need to detect three events: a `mouse down` on a shape, a `mouse move` and an eventual `mouse up`. Detection of the start `mouse up` and the end `mouse down` state occurs only once per drag operation, with `mouse move` events in between.

Listing 2.3: Programming a shape moved operation

```
1 shape.detectMouseDown(function(e1){
2   //mousedown state logic
3   shape.detectMouseMove(function(e2){
4     //mousemove state logic
5     shape.detectMouseUp(function(e3){
6       updateShape(e3);
7     });
8   });
9 });
```

The code example in Listing 2.3 shows how to detect the shape drag operation (i.e. a user clicks on a shape and then holds and moves the mouse) using conventional callbacks[5]. As we see from the example, having to extract higher-level events – a simple drag operation in this case – creates a nested programming flow that is convoluted, making them harder to understand and test as the number of events to correlate increase [Mey+09]. These complex events are are also prominent in the driving scenario where advanced employee access patterns need to be detected for the various security protocols. As the complexity of the events to be detected increases, more advanced programming paradigms are needed.

2.4.3 The Reactive Paradigm

In this section we introduce the reactive programming (RP) paradigm, which builds upon the deficiencies of traditional imperative programming paradigms by abstracting time-varying events for their consumption by a programmer. This inverts the normal control flow of the program: it relinquishes its control to only *react* upon incoming events. A programmer can then define functions that will be invoked upon any 'external' event that is detected by the system.

[5] We omit code for updating the canvas *during* the drag for clarity.

Reactive data vs streamed data

Note that the type of incoming data we reference is different from the commonplace definition of *streaming* data, which refers to the transfer of data at steady, high-speed rates with strict quality of service (QoS) requirements [Wu+01]. Streamed data usually supports linearly-ordered real-time applications such as the timely delivery of the playback of high-definition video content over the Web. Our focus is on *reactive* data, which is usually produced at irregular or diffused intervals. We explore this concept further in Chapter 3.

In [KBD13] we investigate the use of the reactive programming paradigm for developing reactive programs such as RKDAs with such timely requirements on the Web. Most RP approaches support two kinds of abstractions that model continuous and discrete time-varying values. Continuous values, usually referred to as *behaviours* or *signals*, represent uninterrupted or fluid flow of incoming data from a steady event e.g., a timer. More relevant to the distributed setting are discrete values (usually referred to as *event streams*); asynchronous abstractions for the progressive flow of intermittent data coming from a recurring event e.g., events from interacting with a mouse. The abstractions are first-class values and can be passed around or even composed within the program.

In Listing 2.4 we show the previous example of detecting a drag event composed from consecutive mouse actions. We use the syntax of Flapjax [Mey+09], a popular reactive language extension for JavaScript. In the code we model one complete drag gesture as a single composed event of a mouse down followed by a number of mouse move events, ending with a mouse up in line 9. Unlike the nested callbacks from the previous Listing 2.3, dragE in line 14 is a first-class entity that represents a composed event stream consisting of the merged streams of the three mouse actions that represent a complete drag event. Whenever the complete interaction is detected then `dragE` will send data to its handlers as shown in line 14-16.

Listing 2.4: Shape drag operation using reactive programming

```
1  function mouseDownAndMoveE (canvas) {
2   return extractEventE(canvas,"mousedown").mapE(function(md) {
3    return extractEventE(canvas,"mousemove").mapE(function(mm) {
4     //update shape position during move...
5    });
6   });
7  }
8  function mouseDownMoveAndUp(element){
9   var downMoveAndUpE = mouseDownAndMoveE(element).mergeE(mouseUpE(element));
10  return downMoveAndUpE;
11 }
12
13 var dragE = mouseDownMoveAndUp(canvas);
14 dragE.mapE(function(e) {
15  //update shape position after drag...
16 });
```

These kinds of abstractions greatly reduce the complexity of dealing with timely event occurrences prevalent in the Web domain. For instance, RxJs [Man15] is an industry-strength RP library that is used in large web applications to capture and compose reactive streams. With imperative approaches, whenever an event occurs programmers have to explicitly perform re-computations and ensure data dependencies from changes brought about by the event are methodologically updated. With the RP approach programmers

need not explicitly trigger a re-computation of time-varying data: the framework provides automatic re-computation of reactive values in the program. [KBD13] illustrates how such RP constructs can be used to build interactive RKDAs by providing some useful abstractions for reasoning about events in such applications. Rather than force explicit ordering of program flow as with conventional programming techniques (resulting in problems such as the callback hell), reactive programming provides constructs to react whenever an event is triggered.

Reactive Programming for RKDAs

Even though the reactive paradigm provides abstractions that have been proposed for applications that react to external events, it exhibits some limitations. We explain the impact of these limitations specifically for supporting RKDAs.

As mentioned in Section 2.2.4, in addition to reactivity, RKDAs also require value extraction from events. A central idea to this is the relationship between the information from certain events: e.g., an alarm can only be raised if an intrusion happens when the door was locked from the outside or at night time: indicating respectively that the owners are either not present or could be asleep. As a result, these relationships between values brought forth from various occurrences of events are vital when determining the event patterns that are of interest to RKDAs. In reactive programming there is a lack of suitable constructs to capture such relationship patterns in events. Some reactive programming languages do indeed allow one to define custom operators: for instance FlapJax contains `receiverE` and `sendEvent` to create custom event streams and to send event data. But as we present in [KBD13], these operators increase complexities for the programmer when detecting more advanced interactive event patterns, evident in RKDAs.

Consider the code previously shown in Listing 2.3 that detects a drag operation for a single user. If several users are using the application at the same time, then additional code is needed to make sure that each detected event is processed for each corresponding user (cf. the variety characteristic in Section 2.2.1). A programmer would need to combine two `mousedown` operations as shown in Listing 2.5. The programmer is then forced to add `if..else` statements (e.g., lines 5 and 16) in the handlers of detected event streams to further specify specific patterns (in this case, that the `mousedown` operations came from different clients in Line 5 and whether the two events happened simultaneously in line 16). Further constraints would lead to an increase in the complexity of the code. This forces the programmer to specify *how* the system should work rather than simply *what* is desired: significantly complicating the reasoning process.

Listing 2.5: Shape drag operation using reactive programming

```
1  function twoClientMouseDownsE(canvas) {
2   var evt = receiverE();
3   extractEventE(canvas,"mousedown").mapE(function(md1) {
4    extractEventE(canvas,"mousedown").mapE(function(md2) {
5     if (md1.client !== md2.client){
6      //a two—client mouse down
7      evt.sendEvent(md1, md2);
8     }
9    });
10  });
11  return evt;
12 }
13
14 var mouseDownsE = twoClientMouseDownsE('editor');
15 var mouseDownsESameTimeE = mouseDownsE.mapE(function(md1, md2){
```

```
16   if (md1.time — md2.time <= 100){
17     //mousedowns happened simultaneously
18     //...
19   }
20 });
```

This leads us to the next requirement for supporting the development of RKDAs.

- **[Requirement 2] Knowledge encoding via declarative definition of event patterns**

  The *No Silver Bullets* paper by F. Brooks [Bro87] noted that the most prominent gains in software development are made by removing unnecessary artificial barriers that have made programming various tasks 'inordinately hard'. A declarative style of programming places emphasis on specifying what needs to be done rather than the sequence of steps of how to do it. By avoiding implementation details, well-written declarative code is easier to understand, modify, and maintain [And13].

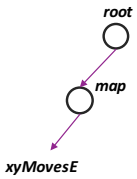
  The most significant part of designing RKDAs that deal with disorderly events is specifying and designing the dynamic or interactive concepts of the application. Hence, a framework supporting development of RKDAs should expose knowledge-encoding syntax and semantics that directly enforces the intent of the developer thereby reducing the complexity of writing code. In the case of RKDAs, this is code that is expressive enough to detect real-time events from a continuous stream given a number of related criteria or constraints, and to compose them to higher-level events. For example in the driving scenario, the supporting framework needs to expose expressive constructs that can detect various access requests against the said regulations by correlating low-level data reads from the badges, devices, etc., to high-level situational replies to the access requests. Although RP provides useful abstractions that keep time-varying values and their dependencies consistent, these types of computations can be cumbersome to express using RP while retaining algorithmic clarity [SM14] (and, as we discuss later, reasonable performance). Other research implementations that embrace declarative semantics within event-based systems (through stream processing) exist, and often follow *relational structured query languages* [Bar+09; Zhi15], *pattern matching syntax* [BM11; Bro09] and other custom techniques that borrow from these two approaches [DHW94; Han92]. These approaches will be discussed in detail in the next chapter.

Processing Reactive Data from Complex Events

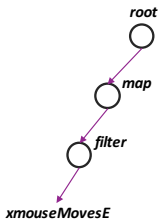
Aside from detecting event patterns, the supporting framework for RKDAs needs to be responsive enough to process data from events reactively in order to extract value from event data promptly. One of the main benefits of this is a responsive execution model and responsive feedback.

Most work on processing data that RKDAs receive in the modern context has traditionally focused on offline techniques: batch processing for data analysis or mining. An increasing number of modern approaches emphasise on processing newly-streamed data, which is considered to be of a higher contextual priority than older static data. Therefore, offline techniques have recently been mapped to processing reactive data online using techniques such as micro-batching [Cór15]. Even so, providing continuous online processing of data sets in an efficient way is still a challenge [GRC04].

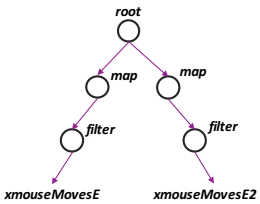
In reactive programming, the code written by a developer is usually converted to a directed acyclic data-flow graph (DAG) for processing. The DAG is a directed graph



(a) Graph for xyMovesE



(b) Graph for xMouseMovesE



(c) Graph for xMouseMovesE2

Figure 2.4: Dataflow graphs for the codes in Listing 2.6 & 2.7– The graph contains nodes that are connected according to dependencies in reactive code.

that is built with nodes and edges representing the data dependencies already defined in a program.

At runtime, most RP implementations use a proactive propagation model where changes, when detected, are immediately applied to nodes and their dependencies in the graph (i.e., push-based). This can be best illustrated with a simple example: extracting the even `x` mouse move coordinates as in Listing 2.6. The code on Line 3 maps over data received by the `mouseMovesE` event stream from the previous line. For this part, a simplified DAG consists of 2 nodes as depicted in Figure 2.4a. Incoming events start from the root node and are propagated to the first node for processing. The output is a new event stream that emits the returned `xy` coordinate values from the computations in Line 3. The next line 5 creates a new stream that contains only even `x` coordinates of the mouse gestures. RP implementations place emphasis on representing dependencies between time-varying values in the DAG built from source code. The resulting graph contains an added node representing this last computation depending on the previous one, as depicted in Figure 2.4b. Incoming event data (in this case, mouse gestures) flows through the graph and any dependent values are recomputed. RP therefore absolves the programmer from manually updating dependent computations from reactive values, since the underlying runtime will perform these computations automatically.

Listing 2.6: RP code for X Mouse Coordinates - Client 1

```
1 var mouseMovesE = extractEventE(canvas,"mousemove");
2 //mouse coordinates when mousemove is detected
3 var xyMovesE = mouseMovesE.map(function(evt){ return [evt.mouseX, mouseY]; };
4 //x coordinates of mousemove
5 var xMouseMovesE = xyMovesE.filter(function(xyMove){ return xyMove[0] % 2 === 0; });
6 //...
```

As observed in Section 2.2.4, one feature of RKDAs is that they can serve a number of distributed clients. Consider the code in Listing 2.7 which could be written by a different client, but which achieves the same purpose for detecting the even `x` coordinates made on

the canvas as that in Listing 2.6.

In the ideal situation, a reactive DAG builder would append the new client's code to the graph without duplication of nodes. This is not the case however, as most RP frameworks build the DAG for this code as shown in Figure 2.4c. This often leads to duplication of nodes, which is inefficient especially when re-evaluations occur as result of events triggering changes in large parts of the graph. In a number of such cases, nodes can indeed be reused to reduce redundancies and improve performance. The reuse is especially significant because different clients contribute a number of constraints to the the server, and therefore performance becomes a key issue when employing RP approaches to support the processing requirements of RKDAs.

Listing 2.7: RP code for X Mouse Coordinates - Client 2

```
1 var mouseMovesE = extractEventE(canvas,"mousemove");
2 //retrieve x coordinate on each mousemove
3 var xMouseMovesE2 = mouseMovesE.map(function(evt){ return [evt.mouseX, mouseY];
      ↪ }).filter(function(xyMove){ return xyMove[0] % 2 === 0; });
4 //...
```

The automatic tracking of dependencies in a reactive DAG has often led to RP implementations placing specific restrictions in the operations performed in the DAG's 'heartbeat', i.e., the immediacy of effecting a change to all dependent nodes in the graph. In practice, however, applications built with reactive abstractions are often infused with the usual imperative stateful logic [Van+17]. This brings us to the third requirement.

- **[Requirement 3] Reactive incremental processing of online/real-time events**

  A framework supporting modern RKDAs should provide efficient and prompt processing of reactive data. The *reactivity* requirement is that first, the server must be able to receive events at all times (i.e., continuously) and second, to have a highly-optimised execution engine that is able to offer event-triggered evaluation with minimal overhead to deliver instantaneous response. Optimised and efficient processing can be categorised using terms as eager, lazy, and hybrid approaches. In any case, to enforce real-time processing and immediacy of results it is necessary to monitor the data memory – to continually add new data whilst checking if there are notifications created to automatically send them to clients. With reactive processing of data, efficiently prioritising computational resources as per each input is essential.

  Frameworks supporting RKDAs, in reality, often perform extensive computations. This is because RKDAs require extracting value from real-time event data. This value extraction from events often requires a way to provide incremental processing whenever any incoming event is detected. Incremental computation avoids the entire program to be re-executed whenever small changes to input data is detected. In the driving scenario, the monitoring system should be able to store relevant intermediate data that relate to the security protocols in order to promptly process incoming requests against the said regulations and avoid re-evaluating all protocols every time a request is detected.

  One way of providing this incremental processing is through storing intermediate data (in intermediate nodes within a DAG, for example) from events that have already been observed, thus avoiding redundant computations. On the processing side, however, traditional RP approaches cannot efficiently deal with incrementally constructed values [RD17], which is vital for value extraction. The graph built behind reactive programs usually needs to be re-executed in its entirety when a change is detected.

The change needs to be propagated to the whole graph, which usually do not contain any caching mechanism that can be vital for performing more extensive computations incrementally. Incremental processing is thus important in RKDAs that require data sets from clients to be cached in order to perform the processing needed for value extraction promptly. Current examples of techniques that follow this direction of extracting knowledge in event-based systems include systems that perform set-based reasoning of events collectively and reasoning using individual events [Bas07].

With the above discussion, we proceed to explain why the rule-based paradigm is specifically significant for RKDAs, which will help us to frame the rest of the requirements in context.

2.5 The Rule-based Paradigm

This section focuses on the value extraction characteristics of RKDAs to infer higher-level knowledge. For frameworks supporting RKDAs, directly integrating a reactive, scalable event-driven server with a rule engine makes the whole system improve performance when receiving requests from a large number of clients and responsive when processing and activating client rules. The role of such a server is to wait and listen in on a stream for events from such entities and subsequently push responses back (to whomever needs it) typically with much shorter response times in a *reactive* way.

To infer higher-level knowledge,one needs mechanisms for capturing patterns from incoming data in observed events. The main problem that arises when trying to capture patterns from a large number of external events is that they arrive in a non-deterministic manner, i.e., programmers have little or no control. One of the most appealing aspects of the *rule-based paradigm* is that in principle, it offers an attractive way to escape from the complexity of problems caused by lack of control by offering formal declarative semantics. Systems employing rules to specify pattern-matching constraints are known as **rule-based systems**. Rules are modular units that expose a formalised syntax which programmers can use to define different pattern-matching criteria. The decoupled nature of the events makes them suitable for developing scalable web applications, and, the declarative semantics and efficient processing of rule-based systems provide real-time detection of patterns. This section describes the ways in which the rule-based paradigm can offer rich programming and processing semantics that web applications can use for reasoning.

2.5.1 Reactivity in Rule-based Systems

Rule-based systems were designed with a goal of general problem solving by representing knowledge in terms of modular rules that encode knowledge in the system [Hay85]. Perhaps the most famous work was the *General Problem Solver* by Newell and Simon in [SN71]. Their work demonstrated that much of human problem solving or cognition can be modelled using *production rules* whereby a rule is a small, modular collection of loosely-linked constraints. Short-term memory in the human brain is used for storing knowledge temporarily when solving problems. A *cognitive processor* finds and eventually activates the rules that have been matched by the external stimuli. The cognitive processor corresponds to a *rule engine* and the sequence of how rules are activated were conceived to be the human thought process.

The model by Newell and Simon for human problem solving is the basis of specialised rule-based systems called **production systems**. They have several manifestations, building

expert systems in domains such as multi-agent and decision support systems. Even though using specialised domain knowledge later became key in the success of such intelligent systems [GR98b], production systems still form the core of more general systems in various domains today [Fri14; Pro15].

Rule-based engines based on semantics of classic production systems perform eager evaluation of asserted facts and *react* to changes in condition states. Recent extensions of production rule systems have been extended to the Complex Event Processing (CEP) [Luc02] domain, such as the Drools [Pro15] and CLIPS [Cro11] engines. In practice, production systems provide functionality such as event monitoring and problem solving in various domains including medicine, business, manufacturing and in computer games [Lug05; WM03]. As they are the main focus of this dissertation, we will henceforth refer to production systems and rule-based systems interchangeably in the rest of its chapters.

2.5.2 General Architecture of Rule-based Systems

The rule-based system model shares the main components as other models of computation: a program, data and some execution semantics. The main difference is that for the rule-based systems programs consist of an unordered collection of rules. A typical rule-based system consists of a data memory, a rule memory and an inference engine. We illustrate the architecture of a typical rule-based system in Figure 2.5.

1. **Data memory** – The data memory is a repository of symbols that represents facts about the world. The data memory stores the current state of knowledge during problem-solving by holding the symbols that map to the facts (and goals) of the domain. The data memories are usually *global stores* as the data needs to be accessible from anywhere in the system.

2. **Rule memory** – The rule memory stores the collection of rules that will be used to process the facts. The rules use the facts to update knowledge about a task being performed. In most systems rules are added to the inference engine at compilation time and are compiled into an efficient structure suitable for the inference engine.

3. **Inference Engine** – The most significant component is the inference engine, which determines which rules are relevant to a given data memory configuration and chooses one to execute. Executing the rules is commonly referred to as *firing* rules, following the analogy of firing neurons in the human brain.

2.5.3 Programming Rule-based Systems

Facts and Rules. Facts represent data instances of real-world objects or event abstractions related to the domain of application – in some cases they can also represent conceptual objects that will fuel the problem-solving strategies. In rule-based systems, facts are composed of elements that hold simple data types. Facts in data memory can be added, modified or removed. Addition of facts into the data memory is known as **assertion** and removal is called **retraction**.

Rules can be expressed as a set of `if-then` statements. The `if` part is the condition and is also known as the **left-hand side (LHS)**, also termed the antecedent. The `then` part is the action and is also known as the **right-hand side (RHS)** or the consequent. The rule relates the facts bound in the `if` part to some action in the `then` part.
An example of a simple rule:

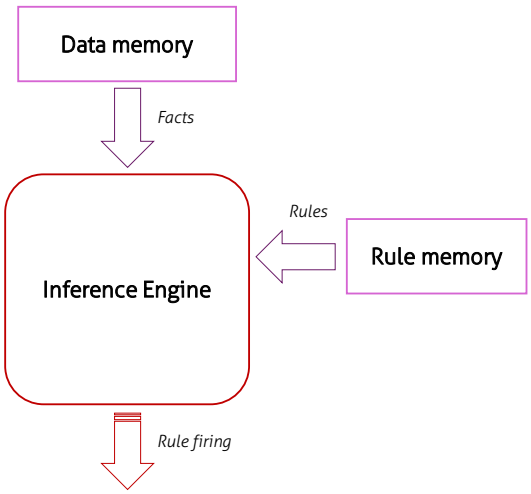


Figure 2.5: General architecture of a rule-based system – The most significant component is the inference engine. It performs the actual execution process of the rule-based system.

```
IF age > 12
AND age < 20
THEN stage is "teenage"
```

A sample fact for this rule can be:

```
age 15
```

After the rule has been executed with this fact, the new state of the system is:

```
age 15
stage "teenage"
```

The `age` fact was added into the data memory and the `stage` fact is added by the `then` part of the rule. Usually, the left-hand side of a rule is made of a combination of condition clauses, such as `AND` which combines two clauses, `OR` which defines exclusivity and `NOT` which is represents a negated clause.

The right-hand side usually contains a list of modifications made to data memory when the rule fires. In the example the `stage` is given a new value. In addition, the right-hand side can perform external operations such as printing on a display, reading from or writing to an external input/output device, signalling a system halt, etc. We now follow the architecture with a description of the runtime execution semantics of a typical rule-based system.

Processing in Rule-based Systems

Execution in a rule-based system is centred on the inference engine. The classic inference engine is a finite state machine performed by a well-known **recognise-act cycle** [For82; Mir14]. The recognise-act cycle consists of three primary states. The first two stages constitute the recognise phase, and the last is the act phase of the cycle.

1. *Match-rules* – All rules that satisfy the current state of the data memory are noted, according to the comparison algorithm of the inference engine. This is done by

comparing the left-hand sides of the rules with the current facts in the data memory. Within this cycle, the constraints in rules cannot mutate while the engine is processing rule bindings. Each ordered pair of `(rule, [relevant-facts])` is known as an instantiation of the rule. The rule matches found are all eligible for execution and collectively form the conflict set.

2. *Select-rules* – In this stage rules are chosen for execution from the conflict set. The rule-based system applies a specific pre-defined selection strategy (or possibly a number of strategies for arbitration) to find out which rules to execute.

3. *Execute-rules* – After rule selection, the system proceeds to execute the rules. Some rules, when executed, may alter the data memory and instantiate a different set of rules after performing the actions. The system then performs the match phase again, forming the recognise-act cycle.

In any rule-based system, most of the processing time is consumed by the first and most significant phase, match-rules. The term *Match* originated specifically from the concept of pattern-matching in Section 2.5 and indicates that a clause in the left-hand side of a rule is exactly the same as a data element in memory. Since most rules can be defined using variables, the variable in a rule is said to be **bound** if it can be replaced by a constant from a data element. Once bound, the variables are used consistently in that scope. An instantiation therefore includes consistent bindings for a rule in both the condition and action parts in the select-rules phase. The result of the select-rules phase, i.e., the conflict set, can be ordered and stored in an agenda. The rules are then selected in order and executed during the execute-rules phase.

Data-driven execution: During execution in a rule-based system control is hinged on the basic re-evaluation of the data states with the rules, and not by any form of static control structures explicitly defined in the program. Consequently, computation in such rule-based systems is said to be **data-driven** as opposed to conventional instruction-driven programs. The main side-effect of this is that the implementation of the execution process becomes more complex and the direction of rule application is in a top-down fashion [GR98b].

Rule application direction: The direction of application of rules during the recognise-act cycle corresponds to the type of reasoning that is used by the inference engine. In *forward-chaining* the conditions of rules determine the reasoning process. Execution begins with existing facts and derives intermediate knowledge by applying appropriate rules until a goal state is reached [Fri03]. *Backward-chaining* involves reasoning 'backward' from a goal state or from a conclusion to be proved to the facts that satisfy the goal. Backward chaining is suitable for analysis or diagnosis tasks [GR98b] and is termed as the goal-driven approach.

2.5.4 Optimising the Matching Process: The Rete Algorithm

The match-rules phase has been identified as the most intensive phase in terms of the amount of processing that the rule-based systems performs [MNM77]. Since rule-based systems usually handle a large number of rules and facts, efficiency is significant in achieving high performance in interactive domains. Rule-based systems allow complex pattern-matching across a large data memory, making the matching process a major source of inefficiency.

Unless the inference engine is optimised, successive recognise-act cycles will recompute all the matches to determine the rules that are applicable according to the conditions on

the left-hand side and the data elements in memory. Making this process fast becomes crucial and methods for performing fast matching are essential for good performance. Due to this, reducing the amount of matching in rules is the best way to make any inference engine process its cycles faster [MNM77].

The Rete algorithm: The most widely-accepted algorithm for efficiency in forward-chaining inference engines is the *Rete pattern-matching algorithm* (or just Rete), devised by C. Forgy in the late 1970s [For79]. Rete performs efficient pattern-matching [Sch+86] by relying on a special internal representation for rules added in the rule memory. The internal representation is a structure containing compiled rules with cached data, eliminating extra work performed by an unoptimised engine.

Rete variations: A number of variations of the Rete algorithm have been implemented since its invention. TREAT [Mir14] and A-TREAT [Han92] omit part of the network across cycles while maintaining the conflict set to reduce the amount of matching performed and to further reduce the memory requirements. The major benefit of the other variations Rete* [WM03] and Gator [HH93] is they provide configurable time-space tradeoffs by allowing dynamic changes in the intermediate memories to limit memory consumption. LEAPS [Bat94] is a separate but related algorithm that uses lazy evaluation to merge conflict resolution strategies during matching (as opposed to after the matching process) to achieve efficient processing.

Comparisons: Each Rete variation however is targeted for specific situations or configurations, thus in a general situation they experience inefficiencies when compared to vanilla Rete implementations. TREAT's conflict set support has a high space-time cost in the general execution of the Rete engine [Bra+91]. During normal unrestricted processing Rete* requires more space than Rete due to the overhead of the additional functionality that it provides, including maintaining dual tokens [Sch+86]. LEAPS' implementation is tightly coupled with a fixed resolution strategy, and it also needs to maintain expensive data histories to support its execution. The general Rete algorithm therefore provides efficient processing of matches in the inference engine of a more general-purpose rule-based system.

Despite its early origins, the original Rete algorithm has gone through various metamorphoses to suit different scenarios in different rule engines. The basic algorithm therefore embodies many of the underlying techniques used in a number of modern rule-based engines, both commercial (BizTalk [Wig12], Blaze Advisor [FIC12]) and non-commercial (JBoss Drools [Pro15], Jess [Fri14], CLIPS [Cro11]).

2.5.5 Rule-based Systems for RKDAs

The declarative nature of the rule-based paradigm exposes powerful abstractions for expressivity when defining patterns and for reasoning when processing the patterns – especially in developing RKDAs. In most of these applications programmers are faced with a large state space to reason about and are facing a lack of control. By hiding implementation details and enabling programmers to express program logic in terms of reactive rules, therefore, the paradigm lends itself well when specifying advanced patterns for encoding knowledge, while at the same time hiding implementation details.

We give a similar example of detecting shape drag operation as in Listing 2.8 in rule-based syntax. Lines 2-4 indicate the mouse gestures to detect, and line 5 checks the timing constraints in milliseconds. This is also illustrated in the work in [Kam+15], which

investigates the use of a rule-based approach for collaborative RKDAs. For instance, the work explains how using a rule-based language for a *collaborative drag* operation provides a natural way of detecting such high-level events. Additionally, this method is easy to extend – detecting even higher-level events simply involves adding conditions on the left-hand-side.

Listing 2.8: Rule for a shape moved operation

```
(rule "detectShapeMove"
  (Event (function "mouseDown") (shape ?shape) (args ?a1) (time ?t1))
  (Event (function "mouseMove") (shape ?shape) (args ?a2) (time ?t2))
  (Event (function "mouseUp") (shape ?shape) (args ?a3) (time ?t3))
  (test (and (time:before ?t1 ?t2) (time:between ?t1 ?t3 500)))
  =>
  (assert (shapeMoved (shape ?shape) (args ?a3))) )
```

The rule-based paradigm is also a natural fit for providing the reactive processing requirements of RKDAs. From defined rules, data-driven approaches create a graph similar to that of RP frameworks. The difference with graphs built by rule-based systems is that they are especially geared towards reactive, knowledge-based processing of data. The execution of such graphs is highly dependent on the patterns in the rule definitions, and are usually built and executed according to a strict execution sequence, e.g., using the recognize-act cycle. Classically, the strict requirements on execution and performance of these graphs resulted in their compilers creating optimised graphs that were mostly static during execution. This brings us to the fourth requirement.

- **[Requirement 4] Hot-Swapping dynamic changes to client constraints**

  As mentioned in Section 2.1.1, a framework supporting RKDAs needs to represent specific client behaviour. Among other benefits, this enables the clients to flexibly customise their specific share of the service that any framework exposes. For example, a change in the security situation may result in a need to update the behaviour of detecting threats in real-time access patterns. Therefore, it is typically the clients that initiate the adaptations in their behaviour on the server, based on changes in their environment. In this way, RKDAs are essentially dynamic applications that can undergo adaptations according to changes in the environment initiated by the clients themselves during execution.

  Due to the dynamic environment that RKDAs operate in, these client customisations also need to be accomplished dynamically without stopping those parts that are unaffected by the change. This *dynamicity* should extend to the constraint definitions that capture different event patterns at runtime for connected clients. In addition, RKDAs can serve a variety of clients at any one point in time. It is therefore unsuitable for the supporting framework to experience downtime thus limiting availability to other clients and experiencing performance bottlenecks. Classic approaches in rule-based systems (and even in reactive programming) construct static graphs at compile time for their operations, making their approaches unsuitable for runtime reconfigurations.

  There currently exist several approaches for providing dynamicity that vary in terms of the restrictions or extent of changes when performed during execution. On the one hand there are approaches that embrace a *stop-start* mechanism. This approach, however, requires the system to be taken offline before appending updating. On the other hand, there are also *on-the-run* approaches that are capable of safely appending updates without taking the system offline. In between the two extremes, there exist work that proposes *hot-swapping* approaches [Aln+13]. In coarse-grained hot

swapping, rather than performing an append, large parts of a system's state is cloned and quickly replaced at some suitable point e.g. at the end of an execution cycle. In fine-grained hot swapping these parts are replaced at more specific areas that are affected by the change, sometimes based on a heuristic. Ultimately, a solution that reduces downtime in a system supporting RKDAs by allowing changes to be made without recompilation is desirable to adequately support dynamicity.

2.5.6 Rule-based Systems & the Cloud

In Cloud-based services, one area that is gaining momentum is in providing complex services for knowledge representation. Instead of hard-coding all the functionality using conventional techniques, we have shown how developers often encode this knowledge in the form of rules to specify detection logic. e.g., IBM ODM Decision Server [Det+14], Amazon IoT Rules [Ama15] and Waylay [Way15]. In order to support RKDAs, these rule-based systems in the Cloud are required to interface with existing web server technologies.

Shelling Classical Rule-based Systems One significant aspect to determine the feasibility of incorporating rule-based systems to support RKDAs is to observe how they have traditionally been utilised in other systems. Rule-based systems have been classically added only as supporting components to existing systems, i.e, they have innately been viewed as black-boxes separate from the domain model [Ros03]. This was the result of traditional knowledge-based systems being incorporated into larger systems, but installed as *expert system shells*. The expert system shells were often seen to have a more separate and complex execution semantics by normal application programmers. A direct consequence of this is that the rules were often not intrinsic to the rest of the system model, and sometimes the application logic was either duplicated or needed to be kept consistent with the rest of the application model. Being installed as an isolated, external component was also a factor that contributed to rule engines being commonly used for offline analytical processing, rather than for real-time processing as required by RKDAs. This leads us to the next requirement.

- **[Requirement 5] Simplicity via transparent symbiosis with server execution**

  In RKDAs, the traditional approach of installing a separate, isolated instance of an offline reasoning system as the backend processing component increases architectural complexity and hampers the responsiveness of communications between components and the system as a whole. This has been coined the 'hammer and nails syndrome', identified as a limitation that reduces the responsiveness of real-time systems [Bry+09]. Identified issues include differences in data serialisation and irregular interfaces. The type of integration with web server execution influences the architecture and design of the resulting framework. For instance in Cloud-based virtualisation frameworks, homogenised environments are significant for easier adoption of higher-level applications and services [BN13]. For an RKDA framework, however, close integration with server execution makes the system performant when directly receiving requests from clients. At the same time, it makes the system responsive when processing activations and forwarding notifications to connected clients. A simple, unified framework that provides integration with operational systems goes a long way in promoting reactivity and in solving application integration challenges. In the same way, the supporting framework should be able to provide a simple unified language to provide the required semantics for an RKDA to specify knowledge constraints, communicate with the server and react to updates from the server.

Simplicity is therefore vital in an RKDA framework: externally, it should be seen as a single abstract entity with a unified interface for application code – even though the underlying architecture or runtime in reality consists of several components [Gro00]. This aspect is closely related to the reduction of the essential complexity that a developer must master to adhere to the various application requirements set forth, rather than the accidental complexity that they face when integrating different incompatible interfaces, each designed with its separate goals [Bro87]: which can be seen for example when plugging a web server with a rule engine, a database backend, separate messaging frameworks and execution runtimes for application logic. This simplicity is especially desirable in such situations where programmers face a number of accidental challenges by being forced to interact directly with different internal component interfaces when developing RKDAs.

2.6 Summary: Requirements for Supporting Reactive Knowledge-driven Applications

We now present a summary of requirements for frameworks supporting RKDAs, distilled from revelations in the previous sections in this chapter. In particular, our focus has been less geared towards reliability issues such as availability and fault-tolerance or security-oriented issues authenticity and fungibility, but more towards the development and processing requirements as discussed.

R1. Event-driven processing & communication. Need for a viable concurrency model that offers an efficient and scalable means to support RKDAs.

R2. Knowledge encoding via declarative definition of event patterns. Exposing knowledge-encoding syntax and semantics that directly enforces the intent of the developer thereby reducing the complexity of writing code for RKDAs.

R3. Reactivity via incremental processing of online/real-time events. To enforce real-time processing and immediacy of results in RKDAs it is necessary to provide technologies that support a form of incremental processing.

R4. Hot-Swapping dynamic changes of client requirements. A solution that reduces downtime in a system supporting RKDAs by allowing changes to be made without recompilation.

R5. Simplicity via transparent symbiosis with server execution. Simplicity in symbiosis with the server is significant for responsive execution and easier adoption of RKDAs and dependent services.

2.7 Chapter Summary

Web applications have been undergoing a metamorphosis from their static, isolated foundations to more dynamic, responsive and composite functionality. One of the ever-increasing demands of today's applications is the need for real-time detection of patterns within collective data. Technologies to meet these demands do, to some extent, support offline techniques of detecting patterns in static or persistent data. Research in meeting demands for online discovery of patterns in large data sets from different event sources is however limited. We refer to the types of applications with such characteristics as reactive knowledge-driven applications (in short, RKDAs).

We then presented an example that epitomises a typical RKDA and distilled the requirements that a software framework supporting RKDAs should fulfil. We proceeded to show why conventional techniques in meeting these requirements of RKDAs fall short in various critical areas. The next chapter proceeds to investigate various related research that fits into the identified requirements, but, as we will show, none of the related work on its own is suitable to support all the requirements.

3

Related Work: Reasoning in Event Streams

Individual events. Events beyond law. Events so numerous and so uncoordinated that, flaunting their freedom from formula, they yet fabricate firm form.

John Wheeler, *Frontiers of Time*, 1994 (Cited)

In Chapter 2 we pin-pointed a number of requirements that a software framework that purports to support RKDAs as defined in Section 2.2.3 should satisfy. Over the years, there has been a body of research work in technologies relatable in supporting applications meeting these criteria [BV07; Che+03; Cór15; Cro11; Wid+07; ZDZ09]. The purpose of this chapter is to provide a survey of the current state-of-the-art in technologies and techniques that can facilitate support for the requirements of RKDAs. Because the work is relatively recent, some of the technologies presented will only exhibit properties that can be indirectly mapped to support these types of applications.

Having identified the 5 requirements for supporting RKDAs, we begin the chapter with a discussion of real-time event processing technologies in Section 3.1 with the aim of supporting reactive processing of content. We split the related research areas into two fundamental categories for independent and sequenced events, presented in Sections 3.2 and 3.3. We proceed with an analytical overview of the identified work, and draw general observations for the conclusion of the chapter in Section 3.4.

3.1 Reasoning in Event Processing Systems

From the Web's dynamic age, data-intensive applications that require the processing of data that is generated or sourced from real-time events started to gain prominence. Event data can be continuously sent to the system in various forms in real time. This fundamentally differs from traditional systems in the earlier ages where the processing of data was mostly done on finite, static datasets.

The idea of supporting the reception of dynamic events with the aim of processing reactive data has led to the emergence of research areas in *event-processing systems* [EN10].

One of the earliest work dedicated to similar systems in event processing was in the 1990s by D. Luckham to analyse event-driven simulations of various architectures using the `Rapide` engine [Luc02].

Recent advancements have given birth to the development of systems that combine both the benefits of event processing and traditional reasoning systems. With regards to this reasoning over events, researchers have further identified distinctions between events that occur independently from those that occur in sequence [Bas07; Luc06]. **Independent events** can happen at the same time or at different times with no possible relations between each other. They are thus often viewed as an unordered set. On the other hand, **sequenced events** have a causal relationship between them and are usually viewed as an ordered set. A set of independent events are referenced individually and this is thus termed as an **event cloud** and those of sequenced events are referenced as an ordered set are **event streams**. The two distinctions have direct consequences on the processing model of the underlying system. Following this dichotomy, work in the event processing domain such as in [Ali+15] has categorised processing engines that support these two concepts as:

1. **Computation-oriented event-processing systems (CEPS)** – These are event processing systems that are suited for processing events as collections of data, i.e., as ordered event streams. Classical terminology refer to these types of systems as being *set-oriented*, but more recent terminology refers to them as having *set-at-a-time* semantics [FRS93]. They employ SQL-like syntax for defining queries over event streams.

2. **Detection-oriented event-processing systems (DEPS)** – Process events as independent entities forming event clouds. Classically, these types of systems were referred to as being *instance-oriented*, but they are today more commonly referred to as having *event-at-a-time* semantics [FRS93]. These systems are known to use traditional production rule-like syntax for definitions.

> To contrast the two categorisations consider the following example. Suppose we have the instance-oriented pattern `AND(E1,E2)`. Whenever $E1$ arrives it becomes of interest and should be stored, because any other event $\texttt{E}_i$ that arrives in any order can potentially satisfy `E2`. When `E2` arrives it completes the pattern. On the other hand, if we have the set-oriented pattern $\texttt{AVG(E}_{set_T}\texttt{,E}_T\texttt{)}$ when the set of relevant `E` events of type T arrive, the evaluation is performed on the entire set. The current `AVG` can be reported on the existing set, but is incomplete, because any other incoming `E` of type T will update the value of the `AVG`.

We discuss the related work of this dissertation spanning the two categories, while restricting ourselves to systems applicable to supporting reasoning in reactive event processing for reactive knowledge-driven applications as defined in Section 2.2.3.

3.2 Computation-oriented Event Processing Systems

One way to implement reactive knowledge-driven applications is to use computation-oriented event-processing systems (CEPS), which view processing of data from ordered event streams as a collection or a set. An event stream in this sense is a sequence of linearly-ordered events, where the order is often based on time: a good example is a stock market feed with a series of changing stock quotes of the trading day sent as events to the system.

Most CEPS have embraced the use of queries based on SQL termed as *continuous queries* that operate over a series of events. Continuous queries are often implemented as SQL extensions that utilise windows on streams conceptualised as relations in order to allow the use of SQLs relational operators. In this dissertation we classify CEPS into two subcategories.

The first type of CEPS have strong foundations in traditional database management systems (DBMS) specifically catered for real-time stream processing. These systems are known as **data stream management systems** or DSMS, and are known to extensively utilise continuous queries. DSMSes however interface with a foundational DBMS backend as opposed to a special or dedicated runtime engine catered for event processing from the ground up. Well-known DSMS examples include TelegraphCQ [Cha+03], SQ [SLR94] and STREAM [ABW06].

The systems in the second category also employ a SQL-like language but have a *dedicated engine* to process events as ordered collections or sets. Using the dedicated engine, they perform operations that can process continuous queries as well, using time and buffer windows over a series of events. Such systems are commonly referred to as **event stream processing systems** [BV07] or ESPs. Examples of ESP engines include Borealis [Aba+05], Medusa[Cet03], Niagara [Che+00], Cayuga [Dem+07], Esper [BV07], and more recently, Apache Flink [Car+15].

CEPS that process streams of events perform useful optimisations on the data, usually taking advantage of the order that events should arrive. A characteristic feature of CEPS algorithms is that at runtime they do not need to internally cache *all* events that arrive in a processing element, and can therefore reduce memory consumption by keeping only the 'relevant' events in the processing element. An example is the STREAM engine, which uses this phenomenon to optimise stream processing using techniques such as pipeline filters [ABW06]. The algorithms for processing CEPS event streams therefore tend to be faster as a result since they can compute events as they arrive and proceed to send results to the next processing element, flushing out irrelevant event data.

We next discuss the viability of CEPS to the requirements of supporting RKDAs using the STREAM engine which introduced the Continuous Query Language (CQL), and the recent Apache Flink [Car+15].

3.2.1 Event Processing in Data Stream Management Systems

Event Processing in STREAM

CEPS use methods that can support the event-processing needs of RKDAs: the foremost being the support for processing events in real-time. The STREAM engine was internally modelled to construct mappings between streams and database relations. STREAM applications are programmed using the CQL declarative language, which is based on SQL but with extensions for registering continuous queries against streams and updatable database relations.

Listing 3.1: Counting room accesses in STREAM CQL

```
1 CREATE INPUT STREAM accessreqs
2    (roomId INT, user STRING, time DATETIME, granted BOOLEAN)
3 SOURCE randomgen
4    PROPERTIES ("timeUnit" = "SECONDS", "period" = "1",
5       "eventNumPerperiod" = "1", "isSchedule" = "true" );
6
7 CREATE OUTPUT STREAM securitysys
```

```
8     (source INT, allowed INT)
9  SINK consoleOutput;
10
11 INSERT INTO STREAM securitysys SELECT roomId, COUNT(granted) as allowed
12    FROM accessreqs[RANGE 3600 SECONDS BATCH]
13    WHERE id = 5 AND granted IS TRUE
14    GROUP BY user;
```

Listing 3.1 shows an example in CQL that detects room accesses from events. Line 11 detects the number of allowed accesses per user (line 14) to a room with ID `5` after each hour (line 12, 13) in a building is shown in Listing 3.1. The granted access requests are batched together and results grouped according to requests by each user. We now use the STREAM engine with its constituent language CQL [ABW06] (using the example), to evaluate how CEPS can meet the requirements of RKDAs as defined in Section 2.6.

- *R1: Processing & Communication Model* – The available version of STREAM provides a client-server architecture that is loosely based on a traditional threaded model. The STREAM engine provides an execution model that can efficiently capture a variety of streams and process any user-defined relations. The client-server architecture of STREAM allows a number of clients to access the server, which exposes functionality to query and receive results using its threaded model. This model is therefore geared towards serving a relatively small amount of complex queries [Ara+04]. For RKDAs to take advantage for the event-driven model for processing and communication, developers are required to implement the interfaces with the STREAM engine manually.

- *R2: Declarative Knowledge Encoding* – STREAM's CQL language proposes abstract declarative semantics for continuous queries that is based on event streams and database relations as data types. The operators of these types span three classes: stream-to-relation, relation-to-relation and relation-to-stream, which transform one type as input to another as the output. Using the relation/stream class operators, the engine maps the data into a form that the underlying database can process. This approach fares well for simpler queries to capture patterns in incoming data streams (lines 1,7 and 11 of Listing 3.1). However, its complexity rapidly increases whenever more advanced querying is needed, where the extent of achieving the desired functionality forces one to mix relations with streams, subqueries, aggregations, windowing and the semantics therein [ABW06]. Most event-driven applications tend to only use CQLs stream-to-relation semantics, where at any point in time it views a part of a stream as a relation in order to define queries that capture events, avoiding class conversion complexities. Furthermore, existing implementations of CQL in STREAM do not support features such as subqueries in the `WHERE` clause limiting its expressiveness [Ara+04; ABW06] when encoding knowledge. The alternative proposed for these unsupported features is expressing them using intermediate named queries, but this further increases the complexity of developing RKDAs using CQL in STREAM.

- *R3: Reactivity* – Upon initialisation, the STREAM engine always evaluates incoming events according to the defined CQL queries. A fundamental property of CQL is that it evaluates incoming streams using windows. The windows in STREAM can be calculated using either a time-based or tuple-based processing model [Jai+08]. For instance, in a time window of 2 seconds, with the following streams consisting of tuples `(value, timestamp)`:
  `(50,1) (60,1) (50,2) (60,2) (40,2)`
  The time-based model picks only one value per timestamp non-deterministically (e.g.,

`(50,1) (60,2) or (60,1) (50,2)`. The tuple-based model picks all five tuples for query processing. However, during joins the two models show different behaviour. Consider the join of the streams
`Stream1 = (50,1) (60,1); Stream2 = (60,2) (40,2)`
The time-based model contains the four combinations of tuples. The tuple-based model however uses total order temporal resolution, where tuples in a given timestamp are implicitly given a rank which affects processing (but is inaccessible from query semantics). The arrival order of the tuples therefore determines the output of the query, with `4!` possible outputs. As these examples show, the two models have different semantics in the way they process incoming events using windowed intervals, affecting their reactivity semantics especially due to the 'evaporating tuples' effect [Jai+08]. This is especially the case when considering larger intervals.

- *R4: Dynamicity* – In STREAM physical query plans are generated from textual queries written in CQL and are optimised using *query plan merging* via a monitoring and adaptive query processor StreaMon. StreaMon ensures that plans and memory structures are efficient and can generate multiple continuous queries internally for performance [ABW06]. StreaMon thus applies lightweight adaptation strategies to the query plans such as runtime modification of the resource allocation and scheduling policy for performance purposes. CQL applications are however first compiled, and the system then generates special query plans for the engine. This implies that STREAM only supports adaptivity of queries: runtime additions of continuous queries would cause a stop-start recompiling process that halts the system to (re)generate modified query plans [Zhi15].

- *R5: Simplicity* – The original version of STREAM was released as a standalone component, forcing users to develop their applications using a separate server, STREAM engine and client code, exposing them to handle nuances of interfacing all these components. This made the development process difficult and introduced bottlenecks since integration became complex. Subsequent revisions of the system have been released with the aim of unifying an Apache Storm server functionality with the STREAM engine, and to improve client interfaces [Zhi15].

3.2.2 Event Processing in Stream Processing Systems

Event Processing in Apache Flink

Apache Flink [Car+15] is an open-source engine with a universal dataflow platform that can process both stream and batch data. It uses data stream processing for real-time, continuous processing of unbounded streams and batching techniques for processing finite data sets in the execution engine. Flink however treats batch processing as a special case of streaming computations. Flink provides benefits for distributed event processing systems such as fault-tolerance, machine learning components and computation over large data sets provided by tools such as Kafka [Apa14].

The Flink architecture is based on layers. At the highest layer, users can submit applications in Java or Scala which are then converted into directed acyclic graphs (DAG) by the lower layers. In the DAG, an operation such as a `map` or `filter` is represented by a node with edges representing data flow. The graph can be optimised by the query optimiser for efficient distribution of nodes with jobs. At the lowest layer, Flink supports writing to sinks, which can be persistent storage in terms of connectors to files, HDFS or JDBC.

In Flink, the event stream processing dataflow engine is used to partition, transform, and aggregate data stream sequences using windows. The Flink documentation also states that it provides limited support to out-of-order records, but these are entirely left to the burden of the programmer to provide semantics of how to handle them when they occur.

Listing 3.2: Room accesses for an Employee in Apache Flink

```
1  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
2  DataStream<Event> input = ...
3
4  Pattern<Tuple3<Integer, Long, Employee>, ?> pattern =
5   Pattern.<Tuple3<Integer, Long, Employee>>begin("A").followedBy("B").within(Time.seconds(3600));
6
7  DataStream<String> result = CEP.pattern(input.keyBy(0), pattern)
8     .flatSelect(new PatternFlatSelectFunction<Tuple3<Integer, Long, Employee>, String>() {
9
10    public void flatSelect(Map<String, Tuple3<Integer, Long, Employee>> map, Collector<String>
          ↪ collector) throws Exception {
11       Tuple3<Integer, Long, Employee> a = map.get("A");
12       Tuple3<Integer, Long, Employee> b = map.get("B");
13
14       /*check that employee instances a and b have at least 1hr between*/
15       if (a.f2.getId() == b.f2.getId()) {
16          /*custom date formatter, date output from milliseconds*/
17          String msg = "Accessed at " + DateFormatter.formatDate(a.f1) + " and " +
                ↪ DateFormatter.formatDate(b.f1);
18          collector.collect(msg);
19       }
20    }
21 });
22 result.print();
23 env.execute("Employee accesses within the hour.");
```

The code example in Listing 3.2 shows an Apache Flink program that detects access requests for an employee within 1 hour. In line 4-5 a pattern is made to detect two events that occurred within 3600 seconds. The main logic is between lines 11-18, where every two events that come from the same employee are collected (lines 11, 12, 15) and the results are shown as a printout (line 22). Since it is a relatively new project, Apache Flink suffers from infancy problems such as limited API and partial implementation of advanced features.

- *R1: Processing & Communication Model* – Event processing in Flink is performed by nodes acting as Processing Elements (PEs). When submitted, code from applications can be efficiently distributed among these nodes for simple, high-throughput processing. Flink in its core utilises the Netty communication framework [The15], an event-driven communication framework commonly used in the rapid deployment of high performance protocol servers. With Netty, Flink can support the exchange of data between node sources and sinks concurrently using pipelined intermediate streams and enables the framework to expose asynchronous event-driven functionality in processing events. This also allows Flink to support operations that cover the event-driven communication needs for streaming applications, e.g., connectivity between clients and the server, as well as between nodes assigned with various tasks. To receive feedback, clients are however required to manually poll the main processing element.

- *R2: Declarative Knowledge Encoding* – Flink offers a multitude of APIs that allow programmers to write programs in Java or Scala with windowing and aggregation functions. All user-written code is then automatically compiled into a common data flow graph,

so the programmer does not need to explicitly wire the processing graph in similar approaches such as Apache Storm [Apa15]. In addition Flink is (at the time of writing) working on a CEP library that will reduce complexity of Flink application development via the framework by including pattern-matching [Car+15]. The code in Listing 3.3 detects 3 accesses `start` `middle` and `end` with the `middle` access happening at least once (line 10). The `Pattern` construct can contain event types as well as regular expressions for pattern-matching on event tags. However, out-of-order events need to be manually orchestrated and the framework provides no dedicated API or support for dealing with these types of streams. This is because most operators provided by Flink rely on some sequencing semantics, making it difficult for developers to perform reasoning over independent, unordered events.

Listing 3.3: Detecting Consecutive Employee Accesses in Apache Flink

```
1  Pattern.<Event>begin("start").where(new SimpleCondition<Event>() {
2    public boolean filter(Event value) throws Exception {
3      return value.getName().equals("e1");
4    }
5  })
6  .followedBy("middle").where(new SimpleCondition<Event>() {
7    public boolean filter(Event value) throws Exception {
8      return value.getName().equals("e2");
9    }
10 }).oneOrMore()
11 .followedBy("end").where(new SimpleCondition<Event>() {
12   public boolean filter(Event value) throws Exception {
13     return value.getName().equals("e3");
14   }
15 });
```

- *R3: Reactivity* – Flink is said to utilise a *Kappa architecture* that forms the basis of using reactive stream processing as its primary processing method. Flink programs contain reactive code which is converted into a graph that assigns tasks to various nodes [Car+15]. Tasks are instances of jobs from user-defined programs which reactively process incoming events from neighbouring nodes using operators that transform input streams, producing new streams. This is in contrast to similar technologies such as Apache Spark that use the *Lambda architecture* where batching is the main method utilised for processing [Fer+15]. Recent versions of Flink can perform a form of incremental processing using delta iteration on only the parts of the data that contain changes [Car+15]. The result is that user programs in Flink usually produce results more promptly than those of other architectures, albeit with less accuracy.

- *R4: Dynamicity* – As mentioned, queries in Apache Flink are translated into DAGs. The DAGs are created from user programs and operations distributed as tasks. This process occurs during a compilation process where optimisations and scheduling is performed. Particularly for the CEP Flink API, there is currently no support for dynamically changing some pre-defined CEP patterns. Proposed workarounds such as changing the way operators are shipped to the schedulers and using Flink savepoints that start and stop jobs on nodes at runtime are complicated and can lose incoming event data during the start-stop process.

- *R5: Simplicity* – Even though it lacks native integration with server systems, Flink was designed to operate in environments supporting distributed event processing frameworks and architectures. Flink offers simple integration with various schedulers and

allows web-based scheduling to manage tasks and the overall system. For instance, its initial design included integration with Hadoop for its batching functionality, where it carefully manages resources needed during compilation. The design further integrates well with components such as HDFS and Kafka that are used by various event processing frameworks for high throughput and advanced scheduling. Finally, Flink can internally run tasks written for other event processing frameworks like Apache Storm using compatibility APIs.

3.2.3 Limitations of Computation-oriented EPS for RKDAs

In this section we summarise the core (in)competencies of computation-oriented event processing systems (or CEPS), as to whether they can offer suitable real-time technologies to support the development of reactive knowledge-driven applications.

In general, CEPS technologies tend to be more focused on continuous queries for high-speed querying of event streams and apply (usually aggregate) relational operators to event data. Continuous queries typically subsume SQL and expose useful constructs that are mainly used for defining complex aggregate functions and windowing.

Even though CEPS have useful technologies that can support a number of applications that require real-time computations, they are however not optimal for the problems identified in Section 2.2.3 that involve supporting the development and execution of RKDAs. Specifically, CEPS contain fundamentally distinct key features from those identified as supporting the requirements.

Most CEPS provide mechanisms for supporting the detection and processing of event sequences and do not intrinsically support capturing independent events or events that arrive out of order. This is because CEPS approaches aggregate similar event types per a given time-based or property-based constraints. Data arriving out of order is a problem for CEPS approaches due to their ordered, set-based processing: CEPS focus on set-at-a-time semantics for the detection and processing of event streams and have little support for event-at-a-time processing. As illustrated in this section, CEPS need to assign incoming event data a temporal or similarly-ranked order so that abstractions such as windowing can be effectively applied. Therefore, because RKDAs usually receive events in a non-deterministic and intermittent fashion, CEPS are not well-suited to captured such types of events. Detection of random employee accesses to determine invalid access patterns requires an approach that views employee accesses as independent events rather than ordered event sequences. As mentioned in Section 2.2.3 this dissertation focuses on such types of value extraction by processing event data individually as it arrives in the system to discern actionable knowledge as a result of the addition of new events.

Developing for RKDAs would force the programmer to mix semantics of reasoning with that of relational data streams via windowing to discern patterns from events. This is because of the interplay concepts (e.g. streams and batches or streams and relations) in CEPS, which tend to complicate expressing and processing of independent events due to different processing semantics. CEPS usually restrict processing of events to a certain window of concern, focusing on a subset of recent statements in the stream. This means that events that were observed previously can be ignored and because of this, CEPS can have the effect of reducing the immediacy of receiving real-time results, e.g., in windowing or batching operators that need to aggregate on the complete sets of histories. Another significant downside of CEPS is that most approaches are limited in providing hot-swapping functionality to processing new queries from clients, and require a start-stop mechanism to update processing elements at runtime. Finally, although the CEPS approaches seek better integration with server execution, they lack the proper technologies to support com-

(a) 'Set-at-a-time'

(b) 'Event-at-a-time'

Figure 3.1: Differentiating between 'set-at-a-time' and 'event-at-a-time' processing – The two distinctions form the basis of computation and detection-oriented event processing systems respectively.

munication with a variety of connected clients, e.g., process events and send push-based notifications.

3.3 Detection-oriented Event Processing Systems

The kinds of event processing that are not ideal for Computation-oriented Event Processing Systems are the very same problems that Detection-oriented Event Processing Systems (DEPS) focus on supporting. In the Web, one cannot always assume that events arrive in an orderly fashion due to its decoupled and distributed nature. As a result, in many situations a total ordering of any two events will not be available, especially those that are intermittent or sporadic in nature.

As opposed to reasoning over event stream sequences that are usually ordered, DEPS usually focus on supporting *order-independent* definitions and execution semantics [Luc06]. Additionally, DEPS provide abstractions that allow discovering patterns within (multiple) incoming events events as individual, independent entities (aka, event-at-a-time), as illustrated in Figure 3.1. Unlike CEPS, DEPS are not primarily focused aggregating event sequences over a property-based or time-based constraint. They are however more suited for reasoning through pattern-matching, comparing out-of-order independent events and for applying decisions and reactions to event patterns.

Viewing incoming events as an unordered independent 'cloud' rather than an ordered sequence is more appealing in the distributed, heterogeneous context. This is attractive for supporting RKDAs because in order to benefit from a richer value extraction process, the applications needs to uncover complex relationships between individual events. DEPS place emphasis on patterns of events and extracting knowledge by abstracting, composing and correlating information in the patterns.

A downside of DEPS as compared to CEPS is that they require much more memory and are often slower than their CEPS counterparts, because DEPS tend to 'remember' all events inserted in order to find the complex relationships that have been defined. DEPS have the ability of providing alternative solutions, dealing with noisy input streams, non-determinism, different preferences, conflicts and constraints. All these features are often computationally intensive.

On the other hand, the advantage is that DEPS can be applicable to a different, more relatable set of applications. Specifically, we can compare the practical scenarios of DEPS implementations to those of CEPS in supporting stream reasoning and KDAs [Bas07]. In a weather station scenario, even though both technologies can be used in monitoring climatic conditions, CEPS are suited for checking the average temperature recorded at a particular station while DEPs would be better suited to reason about the changing

weather conditions. In monitoring stock movements, CEPS are often used to provide summaries of financial transactions of the day, DEPS are better suited for detecting insider trading or more complicated fraud mechanisms. These cases show that in addition to summaries, one may need to detect which events caused other events or which events happened independently.

To perform an analysis of DEPS, we focus on rule-based approaches because these are commonly used for reasoning about knowledge bases (as mentioned in Section 2.5.3). In this dissertation we group DEPS into two categories based on the abstractions they expose and their processing backend. Congruent to DSMS we begin with **active databases** that primarily use database technology, and similar to ESPS we follow with in-memory **rule-based systems** that primarily process events via production rules and a dedicated inference engine.

3.3.1 Event Processing in Active Databases

Active databases were the first attempt at evolving passive traditional database management systems that only executed queries and transactions explicitly submitted by external entities into more *active* systems [DHW94]. In such contexts, the most significant limitation of conventional DBMS was that they were inadequate when supporting high-volume stream processing with low latencies. Active database technology was therefore conceptualised to support these requirements and toolkits were implemented as extensions to conventional database systems or as independent systems. Examples include Ariel [Han96], HiPAC [Day+88], Starbust [Wid96] and Postgres [SRH90].

An active database system monitors incoming events and triggers an appropriate timely response whenever these events occur. Defining this kind of behaviour is expressed in special event-condition-action rules or *ECA rules* of the form

Listing 3.4: Structure of an ECA Rule

```
on <event>
if <condition>
then <action>
```

The *event* part emerged from the implicit assumption that during rule processing, a rule is triggered only when the event to be monitored is detected. The *condition* is used to examine the context in which the event has taken place. Finally, the *action* describes the task to be carried out by the rule if the relevant event has been triggered and the condition is evaluated to be true. After definition, ECA rules are stored in the database. Stored ECA rules can be shared by a number of applications and can be internally optimised by the implementing database system.

In active database systems processing of ECA rules is tightly integrated with conventional database activity via queries and transactions. This activity causes events that trigger the ECA rules and the level of complexity they expose for events, conditions and actions vary. A number of database systems today employ specific technologies from the approaches borrowed from active database techniques.

We proceed to discuss the viability of active databases to the requirements of supporting RKDAs. Since active databases use rules to process events independently regardless of their order, these systems tend to be more relevant when investigating their support of the reasoning process required by systems geared towards RKDAs. We thus compare two prominent implementations of active database systems, Ariel and Postgres.

Event Processing in Ariel

Ariel is an active database technology designed to be tightly integrated with specific database systems [Car+86]. The implementation of Ariel focuses on improving condition testing via the the Rete algorithm (introduced in Section 2.5.4). Because Ariel is integrated with a database system, it utilises a tuned TREAT [Mir14] Rete algorithm. TREAT places more emphasis on intra-condition tests and focuses less on tests between conditions during execution.

An example of an ECA rule in Ariel for adding a default access level of a new employee is shown in Listing 3.5. The rule is triggered by inserting data into the staff table. Upon detection of the insertion of a new employee record (line 2), the action sets the default access level of a new employee in line 5 to the minimally-accepted level acquired from line 4.

Listing 3.5: Ariel rule for default access level of a staff member

```
1 define rule SetDefaultAccessLevel
2 on insert to staff
3 then begin
4  min_level := (select min(level) from accesslevels);
5  update staff set(level = min_level) where staff = new.id
6 end
```

- *R1: Processing & Communication Model* – In Ariel, ECA rules triggered by events are often raised using database functionality. Actions of triggered rules can only update the database structure, and mechanisms are needed to send updates to external entities. Therefore any event-based reception and feedback mechanism has to be orchestrated by a programmer manually. Furthermore, Ariel offers no built-in mechanism for supporting distributed event sources or clients. As with most other active database systems, a programmer is often at the mercy of the supporting database for connectivity. In Ariel this means that programmers are required to provide manual mechanisms for connecting to the system and for receiving updates resulting from actions of triggered rules.

- *R2: Knowledge Encoding* – Ariel decouples events and condition filters using different parts of an ECA rule. When an event occurs, the rule is triggered and the condition is checked on the data. Even though ECA rules offer restricted pattern matching semantics to perform reasoning (see below), the condition constructs availed are ample to define constraints needed to capture data from events as they define predicates over persistent data in the database. The integration of Ariel with the underlying database semantics, however, make programming ECA rules more complex. For instance, the notion of events based on transition of database transactions on tuples leads to a more complicated rule design process. This is because the developer focuses on the expression of database operations as opposed to the effects they have on the data. Consider the rule in 3.6 that avoids any employee with the rank intern from being added to the staff table.

Listing 3.6: Ariel rule for restricting interns as staff

```
1 define rule NoInterns
2  on insert employee into staff
3  if
4   employee.rank :: "intern"
5  then
6   delete employee
```

In this situation, the following *insert+update* operations in Listing 3.7 will be allowed into the database. This is clearly not the intention of the rule designer, who would be forced to additionally add other rules to ensure the requirement is met.

Listing 3.7: Operations will not trigger Ariel `NoInterns` rule

```
1 insert into staff(name='julius', age=32, rank='')
2 update staff set(rank='intern') where name='julius'
```

In addition, extracting knowledge from event patterns in Ariel is heavily dependent on the fixed structure of the event specification part of a rule where the event part (E) is the main focus of an ECA rule. A programmer can only specify the detection of a primitive event using the `on` construct in Ariel and is left to define multiple rules with actions to achieve more advanced semantics, e.g., composition. Event composition operators are not supported in event specifications, and by extension in the processing engine. As an alternative, however, a separate feature in Ariel allows the event part to be omitted from an ECA rule. In this case rule triggering is defined implicitly by a rule's condition part, in which the Ariel engine replies on tuples in the database state to process such rules. In essence, this exposes a slight difference in semantics because the data has already been persisted in the database, which is different from evaluating data from incoming events at runtime: this is observed in the example stated previously.

- *R3: Reactivity* – As they happen, events are captured as specified in the `event` part of an ECA rule. Database activity, queries, transactions etc. can all emit events. Even though the specification language assumes disorderly, independent events, data modification commands in the core database can be packaged as set-oriented operations as transactions known as *transitions*. ECA Rules can only be triggered after every database transition. Programmers designing database modifications have control over the manner in which transitions occur. Therefore the granularity of processing events in Ariel can be a set of transition-based tuple operations by the database, known as a *logical event* [Han96], rather than one operation. Ariel uses logical events to force programmers to reason about the effects of operations rather than their expression – this is actually due to its core execution being influenced by the underlying database. This issue can negatively influence reactivity due to the effects of fine-grained client events in RKDAs.
- *R4: Dynamicity* – Ariel's ECA rules are defined internally as metadata in the schema, together with tables, views, integrity constraints, etc. Defined rules are compiled into a discrimination network that evaluates the rules according to data that is integrated with the underlying database system. As such, once the rules in Ariel are defined and compiled they cannot be modified without an extensive recompilation process to relink them to the database internal structure.
- *R5: Simplicity* – Although initially Ariel was developed as a separate extension or plugin to an existing database system, subsequent revisions discovered the importance of intertwining condition testing with execution of the database system, and newer versions of Ariel feature an engine tightly integrated with a database system. Because it does not support distribution, however, Ariel has to be added to a Web server system as a separate, isolated component.

Event Processing in POSTGRES

Unlike Ariel which is primarily an active component tightly integrated with its database system, POSTGRES [SK91] is a database management system that contains an internal

package that enables active rule system functionality.

Historically, the inspiration for active functionality in POSTGRES arose from the need to satisfy requirements of more advanced business applications, i.e, from the usual object management, POSTGRES should also support knowledge management. The system is therefore able to utilise a collection of rules that form part of the logic of an application to allow derivation of data that is not directly available in the database. POSTGRES is offered as a Software-as-a-Service (SaaS) by a number of cloud services providers. Most active database functionality in modern versions of POSTGRES is provided in two forms: first as a restricted trigger language using standardised SQL syntax, and second as an internal optimiser that modifies queries into rules using the query planner for planning and execution [The12].

Refer back to the default staff access rule in Ariel shown in Listing 3.5. The implementation for POSTGRES will contain similar syntax. If a set of newly-hired employees are added, in POSTGRES' tuple-oriented system the rule is triggered once per employee. In a set-oriented rule system such as Ariel, the rule is triggered only once in the same situation. The execution model of the two systems can change the semantics of applications in some cases, e.g., when calculating moving averages.

- *R1: Processing & Communication Model* – POSTGRES provides similar event processing functionality as Ariel where rules are triggered from database effects. The initial POSTGRES implementation did not have facilities for executing requests from external components. More recent versions have support for command processing drivers that applications can install and use to submit commands using events. Using these drivers, POSTGRES provides a messaging system that can accessed through dedicated commands: `NOTIFY` which sends a notification, `LISTEN` which opens up the system to receive notification and `UNLISTEN` halts the `LISTEN` command. Push-based connectivity is also available using the `NOTIFY` command availed by the engine, which creates a notification event that adds an entry to the `pg_notify` table to be sent by a special *postmaster* service. Interfacing with these systems can however only be implemented through a configuration process.

- *R2: Knowledge Encoding* – When focusing on how knowledge in RKDAs can be deducible from events in ECA rules, there is need to investigate the syntax of the event definitions because these expose the processing and execution semantics. Initial versions of POSTGRES were primarily accessed by users through its query language POSTQUEL. Later versions however feature a set-oriented SQL-based query language. The language further supports user-defined functions/operators, arrays and path expressions [SK91]. Because users can define views declaratively in an SQL-based format (which are then internally converted into ECA rules), POSTGRES provides useful abstractions for reducing the complexity of event definitions (with similar syntax as that shown in Listing 3.5). Unlike in Ariel, however, the event part of a POSTGRES ECA rule is non-optional, and the event specification does not allow basic event operators such as disjunction or conjunction. Although users are allowed to define custom rules in ECA format, the POSTGRES rule system is further considered to be conservative due to its tight coupling with the database semantics [DHW94]. For instance, it only allows explicit triggering on selected events, which may be updates, removal, or retrieval operations. The extent of functionality exposed by ECA rules in POSTGRES is therefore limited when compared to other active database systems. Due to this, even though the language exposes powerful SQL-like abstractions, it does not support useful techniques for RKDAs such as pattern-matching for advanced reasoning.

- *R3: Reactivity* – As mentioned earlier, one aspect of active databases that affects the reactivity of the system is event granularity. In POSTGRES the rule processing is performed immediately after an event is raised, and the consequences of the event can take effect instantaneously using tuple-oriented rule processing. POSTGRES therefore allows for a reactive response when rules are activated. Furthermore, it supports specific semantics for rule activation that specify that a rule should be activated immediately upon occurrence of the event and those that should be deferred to the end of the modifying transaction.
- *R4: Dynamicity* – In POSTGRES, items like data-types, operators and functions can be added dynamically while in execution[SK91]. The system further allows users to define views (and custom rules) which will be internally transformed into ECA rules. POSTGRES supports dynamic user-written rules that specify data update semantics on views but not on the actual schema. If a view is to be updated (which means that the underlying ECA rule has to be changed), then POSTGRES will need to temporarily halt the system to recompile the view and generate a new ECA rule. In this respect, dynamicity in POSTGRES is available (to some extent), but internally the system resorts to a stop-start sequence.
- *R5: Simplicity* – Subsequent versions of the POSTGRES software can use specifically-built foreign data wrappers to link to other systems like other RDBMS, a file system, or even a web-service. Even though the active extension is somewhat integrated with the database, the use of foreign data wrappers require some manual maintenance when linking up with the POSTGRES main server daemon [The12].

3.3.2 Event Processing in Rule-based Systems

Section 2.5.1 gave a general introduction of rule-based systems (RBS). The discussion specifically focused on reactive rule-based systems that operate using forward chaining semantics, which enables them to present rules as being triggered by events, causing actions. Here we discuss RBSes in the vein of how their unique processing capabilities make them suitable for supporting reactive knowledge-driven applications (RKDAs), thus providing avenues that may provide a solution to the various research challenges for real-time event processing that were observed in Section 2.6.

Rule-based systems specifically perform inference through pattern matching and unification. As discussed in Section 2.5, RBSes rely on rules as modular problem-solving units. Production rules typically react to changes in condition state and have no explicit reification of events as in other event processing systems. However, fact updates from external sources form new fact instances that can be dynamically added to the fact base as instances of event declarations – an approach used by modern rule engines [BM11; Bro09; FIC12]. Compared to other detection-oriented processing systems, the conditions in production rules are used to specify these events implicitly (unlike the explicit event definitions in ECA rules).

RBS therefore conceptually consider updates caused by events as changes that can trigger rules based on conditions. This gives the programmer full control over the detection patterns that can be expressed in conditions. In addition, there is always a link between the data that matches the condition of a rule and the behaviour defined in the actions of the rule. Condition variables are bound to data items in the working memory that satisfy the conditions and are accessible in the actions of the rule. Therefore rule-based systems offer attractive features that make them especially suitable for the development of RKDAs: today there exist several open-source and commercial RBSes that are in use in academia and industry with similar goals [Bat94; Bro09; Fri03; Ril91; XZ10].

Given the previous introduction of rule-based systems' architecture and execution semantics in Section 2.5.1, we proceed to investigate the viability of rule-based systems for supporting the development of RKDAs.

Forward-chaining rule engines can be split into sub-categories: **pure rule-based systems** and **composite rule-based systems**. Pure rule-based systems only strictly support rule-based semantics for programming applications and internal implementation for inference. They are defined by an inference engine, factbase and rulebase that receives rules and lightweight facts [1]. Composite rule-based systems (also termed as hybrid systems [DHo04; GK94]) consist of a fusion of a rule engine with a host object-oriented architecture (or some other runtime) that accepts objects instead of facts asserted into the in-memory object repository. They often use a complex hybrid of strategies or are simply developed for a specific application domain and differ in semantics of the effects of changes to host objects and fields from external sources.

It was established in Chapter 2 that rule engines are a suitable fit for reactive knowledge-driven applications. In this section we use this reference to perform a feature-based evaluation using various rule-based systems drawn from the two domains. As in the previous sections the comparison is made against a baseline of a number of requirements identified in Section 2.6. For the evaluation we employ a practical example to perform the comparison of the rule engines to be discussed. To this end, we present a rule that represents one of the access policies of the practical use case from Section 2.3.

> **Rule for Intern Access:** *Interns are allowed access to the intern cubicles in restricted times – from 8am till no later than 8pm.*

This *Intern Access* policy rule will be used as a running example for the remainder of this section.

Event Processing in Pure CLIPS & Composite CLIPS COOL

CLIPS [Cro11] (C Language Integrated Production System) is a rule-based system that is based on the Rete algorithm that was designed at Johnson Space Center, but is today maintained independently as public domain software. A separate engine based on the pure CLIPS rule-based system was developed and became the composite variation CLIPS COOL (CLIPS Object Oriented Language), which introduced procedural and object-oriented programming support.

In CLIPS facts are composed of one or more *slots* that contain attributes with their values. In CLIPS COOL the working memory can additionally contain objects with fields and field values. CLIPS rules contain conditions and actions in the left and right-hand sides respectively, with the `=>` symbol separating the two sides. Rules are defined using the `defrule` construct and variables are identifiers that start with the `?` symbol.

The *InternAccess* rule using the traditional CLIPS syntax is shown in Listing 3.8. Line 2 captures the employee and their attributes, specifying that the employee should be an intern. Line 3 and 4 capture the intern's access request and the device from where the request was made. The `test` conditional element in line 5 provides a way to define boolean expression statements, which in this case checks the time constraints of the policy. The RHS performs an assertion in line 7 that grants the access request if the conditions are met.

[1] This is a restricted definition of pattern-directed inference systems in [Jac98]

Listing 3.8: Intern Access rule in CLIPS

```
(defrule intern_access
   (employee (name ?nam) (badge ?ibadge) (level "intern"))
   (accessreq (id ?reqid) (badge ?ibadge) (time ?t) (device ?dev))
   (accessdevice (id ?dev) (location "cubicle"))
   (test (and (< ?t 20) (>= ?t 8)))
 =>
  (assert (accessrep (reqid ?reqid) (device ?dev) (allowed true)))
  (printout t "Allowed access for" ?nam " on device " ?dev crlf)
)
```

- *R1: Processing & Communication Model* – In both CLIPS and CLIPS COOL execution of the engine is synchronous in all stages. The pure CLIPS engine offered a single entry-point for fact assertions and offered little support for several event sources asserting facts into the engine. An interfacing entity therefore has to wait until rule execution is complete in order to assert more facts to the engine. This makes the rule engine difficult to orchestrate for event processing asynchronously without internally modifying the engine's codebase. The CLIPS COOL variation however has support for Java-based extensions that are useful in supporting multiple sources when performing event processing from distributed entities.

- *R2: Knowledge Encoding* – CLIPS represents heuristic knowledge using CLIPS rules. The use of declarative rules in CLIPS promotes the detection of patterns through the inference process in the engine. The seminal paper that described the CLIPS rule-based system [Ril91] contrasted code in C vs. a rule in CLIPS whose aim was to monitor a number of sensors in order to detect anomalies. The comparison showed that in such cases, a rule in CLIPS is more compact and readable than conventional code. It also contains advanced operators that are useful in defining various detection constraints such as declaring rule properties that temporarily disable rules [GR98a]. CLIPS COOL tries to maintain the same semantics across the pure and hybrid functionality in the engine for pattern detection. For instance, classes in CLIPS COOL contain slots rather than fields, and object slots maintain most of the applicable operations on normal fact-based (or non object-based) slots. CLIPS COOL further offers basic constructs for procedural programming like generic functions and object-oriented programming, enabling developing using of a mix of rules and message-passing, or rules and procedural code [Ril91]. This however tends to complicate the design of rules and eventually makes understanding execution semantics of the engine more complex.

- *R3: Reactivity* – CLIPS terminates whenever there are no items in the agenda. Pure CLIPS therefore has the limitation of not supporting continuous execution that is useful for continuous event processing out-of-the-box. One workaround can be through the use of *incessant facts* that always force an item on the agenda to reset the engine back to the select phase after activation of rules. We show such an example in Listing 3.9 that prints a counter value. The `continue` fact in line 3 is used to trick the engine to continue execution by always modifying the value of the fact, repeatedly placing the `count` rule in the agenda. Of course, aside from being inefficient by adding operations to be performed at the end of each cycle, this technique is more of a 'hack' and is highly dependent on the conflict resolution strategy that the engine enforces. CLIPS COOL supports some form of reactive semantics through its `reactive` facet specified in its initial configuration (i.e., during definition), which specifies that changes to an object slot will initiate the pattern-matching process – akin to a fact modification in pure CLIPS.

- *R4: Dynamicity* – In both CLIPS and CLIPS COOL, once the engine is in execution mode then the rules within the rulebase are used to create the Rete graph internally. CLIPS allows a change in the rulebase to be performed through an addition, removal or modification of rules by other rule definitions. But because the engine does not inherently support reactive semantics as described above, the integration of these two features (dynamicity and reactivity) is not clear. This is a limitation in supporting dynamic features for RKDAs and would require manual interventions to achieve such functionality when using the engine. Even though the engine supports other dynamic features such as dynamic salience in rules in the `select` phase of the matching cycle, it is lacking this requirement as explained in Section 2.4.3.

- *R5: Simplicity* – In order to support simplicity when developing RKDAs with CLIPS, the manner in which the engine is integrated with Web server technology is key to enable responsiveness to clients during its operation. The pure CLIPS engine was specifically designed as a stand-alone tool (a.k.a, as an expert system shell [GR98b]) with little support for such integration. The CLIPS COOL engine does support basic interfaces but applications need to manually specify a component that maps received data to CLIPS COOL objects. It exposes an API that can be used to control the engine externally: however, the allowed operations are limited to macro-operations like starting and stopping the engine, and less about controlling the execution flow of the engine.

Listing 3.9: Incessant Facts in CLIPS

```
(defrule count
   ?c <- (counter (no ?n))
   ?y <- (continue (val false))
=>
   (modify ?c (no (+ ?n 1)))
   (modify ?y (val true))
)

(defrule show
   ?c <- (counter (no ?n))
   ?y <- (continue (val true))
=>
   (printout t "Current value: " ?n crlf)
   (modify ?y (val false))
)
```

Event Processing in Jess

Jess [Fri03] is a rule engine inspired by the CLIPS engine. Jess exposes its main rule-based syntax, the LISP-based CLIPS syntax from its parent engine and the XML-based RuleML [Bol06] syntax to represent rules.

Even though the strength of the Jess engine is in its implementation of the forward-chaining Rete algorithm, it can also simulate backward chaining by controlling execution using rules programmed in a specific form to react to *goal-seeking* or *trigger* facts. Jess follows a composite implementation and can directly reason about Java objects. It represents the objects in its working memory as Java Beans akin to slots in CLIPS. Java Beans in the engine are said to be either dynamic, where changes to fields are always kept up to date in the working memory, or static, where changes to an object are ignored. The engine however can manipulate objects using typical `assert`, `retract` or `modify` constructs in rules.

Listing 3.10: InternAccess rule in Jess

```
1 (defrule rule_intern_access
2    ?e ← (employee {level == "intern"})
3    ?r ← (accessrequest {time < 20 && time > 8})
4    ?a ← (accessdevice {id == r.device && location "cubicle"})
5 ⇒
6   (assert (accessreply (id r.id) (device a.device) (allowed true))
7   (printout t "Allowed access to intern ?e.name on device ?a.device" crlf)))
```

Jess rules typically have a left-hand side and a right-hand side separated by a '⇒'. We show the *InternAccess* rule in Jess syntax in Listing 3.10. The LHS contains a list of conditions each consisting of a condition type, like `employee` in line 2 and slots like `level`. Conditions can also contain expressions in curly braces as in the expression in line 2 that checks if the employee is an intern. On the right hand side three actions are commonly used: `assert` to add new objects, `modify` to change an object's slot and `retract` to remove an object from the working memory. Other side-effects can be used on the RHS, for instance the `print` statement shown in line 7. These statements have no effect on the engine itself but can be useful for purposes such as logging in applications.

- *R1: Processing Model & Communication Model* – Jess provides event processing via the semantics of its Rete engine, which can be started by calling `Rete.run`. In Jess, rules are constantly evaluated against incoming events asserted into the knowledge base. Jess also allows the definition of one-off rules using `defquery`, which will only run once when explicitly called externally and retrieves all matches at once. This in Jess is often used when stored Java Beans objects can undergo several changes to their state from external sources, thus having the underlying Rete engine perform undesired execution cycles. It however undermines the default semantics of the Rete engine. The default `run` method to start Jess' Rete engine executes the engine synchronously. Therefore any incoming event would have to wait until all rules are fired and execution returns to continue interacting with the engine. The alternate `runUntilHalt` is thread-based and can use constructs like `wait()` and `notify()` to perform actions asynchronously whenever there are rules to be fired. Using this alternative, Jess can internally expose events through the `JessEvent` class that can be used by sources to capture rule activations using registered event listeners asynchronously. Additionaly, even though the engine does not natively support multiple clients connecting in order to add rules and to assert facts, Jess supports connectivity through manual configuration via Web servlets. Communication with clients can be therefore be implemented using various frameworks; however application designers have to manage the idiosyncrasies themselves.

- *R2: Knowledge Encoding* – Jess follows a rule-based syntax that allows specifying constraints using conditions in rules. Jess consists of an expressive syntax derived from CLIPS that was specifically designed for pattern-matching in rules. The left hand side before the ⇒ describes the constraints for the working memory elements and the expressions to test for matching. The right hand side avails the actions to take after the conditions have been matched. The engine further exposes similar pattern for its symbiosis with Java objects using Jess patterns that allow mixing native expressions with Java-based OO syntax. Even though this in effect renders rule design more complicated, it nevertheless makes the engine able to perform reasoning over Java Beans objects in the working memory.

- *R3: Reactivity* – Jess heavily borrows from the CLIPS engine and employs the Rete algorithm at its core. As discussed earlier in Section 2.5.4 the algorithm is suited for

efficient data-driven processing. In Listing 3.11 we show our example of running a Jess application using the *InternAccess* rule, where rules are added into the engine inline and then facts are inserted to the working memory. Jess then requires the method `Rete.run()` (line 15) to be invoked which performs matching and activates rules. Similar to Drools, the Jess engine performs offline processing by default: it immediately stops execution as soon as there are no more applicable rules. The later version of Jess 5 provides the aforementioned `Rete.runUntilHalt()` method (Listing 3.12, line 17), which uses Java's wait/notify constructs for control, e.g., pausing the calling thread until active rules are ready to fire. As the name suggests, the `run` will only return when `Rete.halt()` is called. This way the engine can be made to support processing event streams using reactive semantics. By default, there is no synchronisation between changes to object fields when already in the Rete network and they thus do not trigger events in the system – but this behaviour can be allowed by configuration of the engine upon initialisation.

- *R4: Dynamicity* – The Jess engine performs processing offline by default using the `run` method, performing a match-execute cycle and exiting after rules are activated. During each run, Jess undergoes a compilation process that converts all rules into a Rete graph. With the `runUntilHalt` method, the engine continuously runs in a separate thread. However, when running in this mode the engine cannot receive new rules or modify existing ones. Work in [Thi07] reported a similar conclusion. The work involved investigating the extent of dynamism supported in the Jess engine and it reported that in dynamic environments, Jess is not able to support runtime modification of rules that already underwent a compilation process.

- *R5: Simplicity* – Even though Jess is based on the pure CLIPS engine, it requires symbiosis with the Java runtime and object architecture. The same deficiencies as those discussed of the Drools engine apply here, where various components are needed to interface with Web servers and applications. This makes it harder to develop, debug and maintain applications that require integrated reasoning semantics.

Listing 3.11: Running a Jess Session

```
1  import jess.*;
2  public class SecEngine {
3     private Rete engine;
4     public SecurityEngine(ArrayList facts) throws JessException {
5        /* Create Jess engine */
6        engine = new Rete();
7        engine.reset();
8
9        /* Load the intern accessrule */
10       engine.batch("rule_intern_access.clp");
11       /* Load the data into working memory */
12       engine.addAll(facts);
13
14       /* Fire the rules that apply to the facts */
15       engine.run();
16
17       // Return accesses created by the rules
18       return engine.getObjects(new Filter.ByClass(AccessReply.class));
19    }
20 }
```

Listing 3.12: Running Jess Continuously For Event Processing

```
1  import jess.*;
2  public class SecEngine {
3     private Rete engine;
4     public PricingEngine(ArrayList facts) throws JessException {
5      new Thread(new Runnable() {
6       public void run() {
7         /* Create Jess engine */
8         engine = new Rete();
9         engine.reset();
10
11        /* Load the intern accessrule */
12        engine.batch("rule_intern_access.clp");
13        /* Load the data into working memory */
14        engine.addAll(facts);
15
16        /* Fire the rules that apply to the facts */
17        engine.runUntilHalt();
18       }
19    }).start();
20    }
21 }
```

Event Processing in JBoss Drools

Drools [Bro09] is a popular business logic integration platform written in Java. It contains a forward-chaining production system that is built using the Rete algorithm. It is today part of the JBoss suite of business enterprise management and monitoring tools.

Drools follows a composite implementation of combining the use of rules with the use of objects. A repository of Java objects can be inserted into the network which represents the 'facts' or the state of the system. The objects can also be removed or their fields updated in Drools' Rete network.

Drools has a native language for writing rules known as the Drools Rule Language. The syntax follows the `when..then` structure to distinguish the left-hand side from the right-hand side. Due to its composite nature rule statements can access fields and invoke methods on Java Bean objects.

Listing 3.13: InternAccess (Protocol 4) rule in Drools

```
1  rule "InternAccess"
2   when
3     $e: Employee(level == "intern"))
4     $r: AccessRequest(badge == $e.badge))
5     $a: AccessDevice(id == $r.device, location == "cubicle", $r.time <= 20, $r.time > 8))
6   then
7    insert(new AccessReply($r.id, $e.name, $a.device, true)
```

The *InternAccess* rule in Drools' syntax is illustrated in Listing 3.13. Rules in Drools are identified by a rule name, in Line 1. A rule can refer to objects in the hybrid approach: lines 2-5 will, at runtime, bind to objects of the classes `Employee`, `AccessRequest and AccessDevice` respectively. In each `when` condition, types checks can be augmented with expression tests for its attributes with constant values or variables representing other attributes, as in line 3 which specifies that the `Employee` should have the level intern. On the `then` side new objects can be inserted, removed or modified in the engine. The action in line 7 adds an `AccessReply` that indicates that the request fulfils the conditions and has been

accepted. Such rules are then parsed and compiled by the framework's KnowledgeBuilder, and objects can be added into a KnowledgeSession.

Using the sample rule as a reference, we now evaluate Drools with respect to how it can support a reactive server engine for the Web, focusing on the identified requirements in Section 2.6.

Listing 3.14: Starting the Drools engine using `fireAllRules`

```
1 KnowledgeBuilder knowledgeBuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
2 /* Add DRL file */
3 BuilderknowledgeBuilder.add(drlFile, ResourceType.DRL);
4 StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
5 AccessDevice device = new AccessDevice("Cubicles");
6 knowledgeSession.insert(device);
7 /* ... */
8 knowledgeSession.fireAllRules();
```

Listing 3.15: Running the Drools engine using `fireUntilHalt`

```
1 /* Create a session and run continuously */
2 new Thread(new Runnable() {
3  public void run() {
4   KieContainer.newKieSession().fireUntilHalt();
5  }
6 }).start();
```

- *R1: Processing & Communication Model* – The Drools engine provides two event processing modes, STREAM and CLOUD [Bro09]. The mode is specified via a configuration parameter at startup. The STREAM processing mode assumes an ordered stream of events and uses an internal 'session clock' to provide this functionality. In this mode, Drools can also process streams using windowing, thus operating in a similar manner to stream processing systems discussed in Section 3.2. The CLOUD processing mode is the default mode, where the engine behaves similarly to other pure forward chaining engines. In CLOUD mode the engine sees all events in the working memory as unordered, independent facts. The engine applies the Rete algorithm to find matching facts and to activate rules. Therefore the CLOUD mode processing is better suited for providing support for RDKAs.

  For communication with clients, concurrent connections in the Web server running JBoss Drools are not available by default. Drools can support push-based concurrent connections through the EAP 6.3 WebSocket implementation in its parent framework JBoss [Red15]. This however requires the developer to delve into Drools source code, which can be a challenge to debug. As an example, the programmer needs to keep in mind that, as in the Java WebSocket 1.0 specification, callbacks for asynchronous writes are performed on a different thread to the thread that initiated the write [Fou15]. This will affect the orchestration of the reception and notification of client event messages from the running Drools session (which, is as subsequently explained, runs in a separate thread as well).

- *R2: Knowledge Encoding* – Rules are defined using declarative expressions specifying the constraints to match in the `when` section and the actions to take in the `then`. Procedural statements can also be written in the latter section. The rules sets are are defined in `.drl` files. Each condition can refer to a class rather than a fact type, e.g., `Employee` in line 3 of Listing 3.13 refers to an instance of the class `Employee` that has a specific level.

Drools provides pattern-matching constructs using condition expressions in the `when` part of rules. The left-hand-side of rules in Drools are matched according to bound variables and expressions. Conditions can be bound using variables denoted as starting with `$` and fields of objects bound to them can be accessed using normal dot operators (as with `$r.device` in line 5). Boolean guards can be defined in conditions, such the time expressions in line 5 that check if the access was made between 8am and 8pm. The rule language also support aggregation and negation operators. Using these constructs, the engine can internally perform inference on facts bound to variables in the rule definitions. The declarative expressions therefore allow programmers to capture knowledge via the various rule-based constructs exposed by the Drools rule language.

- *R3: Reactivity* – By default, when objects are added to the engine, Drools performs matching internally and stops executions without instantiations. In an RKDA framework, it is more appropriate for the engine to simply wait instead of halting execution after matching, because in this environment events happen continuously or even intermittently. Drools does not create and fire activations unless a special method `fireallRules` is invoked on the Drools session upon instantiation, as shown in Listing 3.16. `fireAllRules` then performs the matching process and proceeds to fire any rules that have entries in the agenda, immediately halting when there are no more rules to activate. The default behaviour can be changed by instead invoking a special method, `fireUntilHalt`. `fireUntilHalt` however requires to be run in its own separate thread, which will keep firing any instantiations on the agenda until none remains, as shown in the example in line 6 of Listing 3.15. This can complicate the mechanisms in which a programmer can add and receive notifications from the engine running in a reactive manner.

- *R4: Dynamicity* – The default `fireAllRules` primarily checks for activations in the current agenda then puts the engine in an idle state when there are no more activations, making this a single execution sequence. As we illustrated, in order to evaluate dynamic rule addition in a continuous, real-time fashion, a developer is obliged to first create a Drools session in `fireUntilHalt` mode, which runs in its own thread. Support for dynamic addition of rules requires the Drools engine to be able to add rules into the rule base (thus appending the rule within the Drools' inference engine structure) at runtime. This is however not the only requirement for dynamic rule addition in Drools. Whenever a rule developer wants to add a rule at runtime, the session needs to be given a new release ID and deployed as we show in lines 16-22 in Listing 3.16 which shows the common way to startup the engine. The `updateToversion` method will update the session with the new rule base. The reason for all this is because the deployable artefacts (or `jars`) that Drools uses to represent sessions are immutable source code artefacts. Therefore another `jar` is created in memory that updates the previous version. The whole process makes rule addition in Drools pseudo-dynamic, is fairly convoluted and can lead to unintended versioning problems arising from its manual implementation.

- *R5: Simplicity* – Evidently, from the above descriptions of how to configure and run the Drools engine for real-time events and dynamic rule addition we can see that the engine requires multiple separate components to achieve the real-time goals that we investigate in this section. Drools has native integration with JBoss, which contains a server component. However, integrating with other server frameworks shows that the engine contains the shelling limitations as discussed in Section 2.6. Furthermore, the intricate setup for starting the engine as reactive components with support for RKDAs not only makes the development and maintenance effort complex, but also increases

the communication overhead when receiving and processing multiple events from many clients.

Listing 3.16: Dynamic rule addition in Drools using Kiesession updates

```
KieServices ks = KieServices.Factory.get();
String testRule1 = /* rule1 here */;
/* Create kie session version 1.0.0 */
ReleaseId release1 = ks.newReleaseId( "org.kie", "drools-addStaticRule", "1.0.0" );
KieModule km = createAndDeployJar( ks, release1, testRule1 );

//Create a session and fire rules
final KieContainer kc = ks.newKieContainer( km.getReleaseId() );

new Thread(new Runnable() {
 public void run() {
   kc.newKieSession().fireUntilHalt();
 }
}).start();

String testRule2 = /* rule2 here */;
/* Create 2nd kie session version 1.0.1 */
ReleaseId release2 = ks.newReleaseId( "org.kie", "test-addDynamicRule", "1.0.1" );
km = createAndDeployJar( ks, release2, testRule2 );

/* Update session container to version 1.0.1 */
Results results = kc.updateToVersion( release2 );
//...
```

Although Drools fares well with its rule-based syntax of specifying declarative constraints, it falls when evaluating its preparedness for supporting concurrent clients and dynamic addition of rules to the running engine. Further, the disintegration of the rule engine with the rest of the architecture needed to support reception and notification real-time events from distributed clients makes it less appealing for RKDAs.

3.3.3 Summary: DEPS for supporting Stream Reasoning

We provide a summary discussing how proficient detection-oriented systems are and how they can be applied to reactively process events; with reference to the development and processing support of the kinds of applications that this dissertation targets, RKDAs.

Due to their symbiosis with database systems events, active databases are restricted to internal database calls. Indeed, although active databases incorporate a rule-based approach based on ECA rules, their support of capturing events from definitions is mostly dependent on the event specification languages that are tied to the larger database system semantics. Restrictions such as the event part being associated to only one relation (e.g., in POSTGRES [SK91]) hinder the expressiveness needed to capture advanced or multiple event patterns, required by RKDAs.

Active database systems differ in the level of granularity of triggering an ECA rule, whether tuple-based or set-based. Some ADS systems even support both types. Designers of ECA rules have experienced conflicts in these differences that lead to nondeterministic behaviour that is difficult to debug [Hor94]. Furthermore, their implementations are dependent on database effects semantics, to the extent that they are modelled around static database processing statements rather than on the foundations of reactive event-based processing that this dissertation targets for RKDAs.

Most of these restrictions placed on active database systems are not experienced in rule-based systems. In RBS events are triggered by *external and unordered* data streams coming from real-world sources and can include sensors, user activity, etc. Rule based systems offer expressive rule-based syntax that provides strong pattern-matching semantics for inference.

In theory, techniques that are based on the data-driven forward-chaining semantics such as the engines presented can be extended to provide event processing capability. When it comes to practical implementations, however, all the engines required substantial manual interventions to support reactive event processing.

CLIPS offers little support for Web-based integration and support for concurrent connectivity for receiving and sending prompt updates from the engine to clients and vice-versa. Drools and Jess require thread-related solutions which complicates the process of supporting concurrent connections and dynamic addition of rules and facts. Even though CLIPS COOL's integration with languages such as C gives good performance, it offers limited operations that support event-based reactive semantics without resorting to modifying the core engine's codebase.

3.4 Results of Analysis

We provide a comparison of selected systems that provide event processing with respect to supporting frameworks that enable the development of reactive knowledge-driven applications that harness community knowledge. We provide the analysis based on the requirements set forth in Section 2.6 to the technologies discussed for event processing.

We present a summarised view in Table 3.2, where the rows represent the technologies and the columns the requirements. The table contains 5-star rankings where a rank of 5 stars symbolises the most desirable (i.e., meets the requirements fully) while a 0-star ranking symbolises the least desirable (hardly meets any criteria).

From the table and the discussions we can draw several observations. If only sequence filtering and processing a high number of ordered events per given time slots is needed, then computation-oriented approaches are highly specialised to aptly provide such processing requirements. This dissertation focuses on uncovering support for technologies that support knowledge-based reasoning in RKDAs: i.e., ways in which processing of independent data streams can be merged with systems that provide complex reasoning abstractions. With this focus, computation-oriented approaches suffer from lack of abstractions and complexity in rule design aimed at these goals.

From the overview, we see that detection-oriented rule-based systems are particularly suited for their non-deterministic, demand-driven, precise querying processing models that enable caching of intermediate results. In addition, the on-the-fly pattern-matching and unification techniques of rules give rule-based systems a unique approach in deriving knowledge from incoming uncoordinated events. We observe that DEPS are the closest fit to support the kinds of applications that this work envisions. However, current solutions suffer from fundamental problems that will require application developers devote substantial programming effort to overcome. Rule engines follow a traditional select-execute cycle, and require more advanced or manual interventions to enable them to continuously process external events and provide feedback without affecting the engine cycle.

A separate point is that distributed stream processing systems approaches suffer from an increase in semantic complexity as a result of intertwining two paradigms together [Jai+08]. With active database systems the paradigms are database and production rule paradigms [Hor94], and with hybrid rule engines it was intermingling the underlying

object oriented runtime with the rule engine execution [DHo04]. Pure approaches therefore provide cleaner semantics and focus solely on supporting reasoning during event detection as well as processing, a viable option when meeting the requirements set forth.

From the analysis, we conclude that **the engines surveyed have little support for the extent of reactivity, dynamism, simplicity, connectivity and decoupling required** when supporting the continuous execution of the engine, dynamic addition of client rules at runtime, large number of distinct event sources and decoupled distributed execution needed for reasoning in reactive knowledge-driven applications.

3.5 Chapter Summary

To summarise, this chapter has provided a categorisation of event-processing systems based on computation-oriented and detection-oriented techniques. Both of these approaches are (in some ways) suited for the types of reasoning that reactive knowledge-driven applications require (c.f. Section 2.2.3).

Even though existing event processing systems can provide high-throughput stream processing, they lack complex reasoning capability that is required by RKDAs. In this respect two approaches were presented. Detection-oriented approaches in Section 3.3 are more suited for the reasoning semantics required by RKDAs than the computation-oriented engines discussed in Section 3.2. Rather than focusing on an inherent order with incoming events as in CEPS, DEPS view incoming events as unordered, independent entities.

From the analysis, we conclude that though promising, existing DEPS systems are still not well-suited to support the development and processing requirements of RKDAs. They have particularly exhibit limitations in allowing dynamic changes at runtime and providing continuous, reactive execution semantics and simple integration with server frameworks. The next chapters present our approach in tackling these limitations.

	Proc & Comm Model	Knowledge Encoding	Reactivity	Dynamicity	Simplicity
Computation-oriented Engines					
Data Stream Management Systems					
STREAM	★★★½☆	★★★☆☆	★★★☆☆	★☆☆☆☆	★★½☆☆
Event Stream Processing Systems					
FLINK	★★★½☆	★★★☆☆	★★★½☆	★☆☆☆☆	★★★½☆
Detection-oriented Engines					
Active Database Systems					
ARIEL	★½☆☆☆	★★★½☆	★★★☆☆	★☆☆☆☆	★★☆☆☆
POSTGRES	★★★☆☆	★★☆☆☆	★★★½☆	★★½☆☆	★★★½☆
Rule-based Systems					
DROOLS	★★½☆☆	★★★★☆	★★★★☆	★★☆☆☆	★★★☆☆
JESS	★★½☆☆	★★★★☆	★★★★☆	★☆☆☆☆	★★½☆☆
CLIPS	★☆☆☆☆	★★★★½	★☆☆☆☆	★★★½☆	★☆☆☆☆
CLIPS COOL	★★☆☆☆	★★★★☆	★★☆☆☆	★★★½☆	★★☆☆☆

Table 3.2: Evaluative feature comparison of event processing systems for supporting reactive knowledge-driven applications on the Web – Rule-based systems offer the most powerful abstractions of one-at-time detection and processing of events, but are weak in areas such as reactivity and dynamicity.

4

Serena: Cloud-based Rule Engine

One Orchestration to Rule them all,
One Subscription to find them,
One Port to bring them all,
and in the Engine bind them.

Charles Young, *Geeks With Blogs*

The previous chapters identified that rule engines can provide functionality that supports reasoning in reactive knowledge-driven applications. Current solutions however lack proper mechanisms to be able to be deployed in such dynamic environments. This chapter introduces the Serena rule engine, a custom rule-based system for RKDAs that provides reasoning over events. In Serena rule-based logic can be integrated with the event-driven Web environment using external callback functions in the Javascript language.

The chapter begins with a discussion of the architecture and execution semantics of the engine in Section 4.1. Next, it references the motivating scenario from Chapter 2 to delve into details of its implementation. The syntax and execution semantics are presented, and finally an evaluation of the engine as compared to the related work is discussed in Section 4.5. The chapter then concludes with a summary of observations in Section 4.6[1].

4.1 Introduction

This section presents the Serena Web framework. The main motivation of the framework is to overcome the limitations of current rule-based systems by instilling reactive event-processing behaviour in a forward-chaining rule-based system. This enables the framework to realise the computationally-intensive process of receiving and reactively processing data in order to detect complex events, together with accompanying data relevant to notify clients.

[1] Observations described in this chapter have been published as [KBD15]

With these foundations, the framework provides techniques that ease the dynamic definition of requirements by utilising a rule-based syntax, and also provides methods that enable the efficient processing of intermittent data giving instantaneous feedback. Serena further manages sending messages between the server and connected clients by abstracting the underlying infrastructure that supports reactivity using push-based communication.

Using the scenario from Section 2.3, this chapter dissects the inner workings of Serena by first defining the syntax of rules and then by illustrating its general architecture. Later, the execution semantics of the framework are explained in detail.

4.2 Serena Rule Language: Syntax and Semantics

This section describes the syntax and semantics of the framework. Serena supports dynamic custom rules by clients: clients are able to design, install and dynamically upload rules to the server rule engine. There exist rule-based implementations for the Web that use RuleML [Bol06], based on the verbose XML format. However, the Serena framework has embraced a schema that is more compact and faster to serialise for easier transportation of data as rules and events between clients and the server. The syntax of rules in Serena draws inspiration from JSON Rules [GP08] and is named the *Serena Rule Language*, SRL in short. Using SRL, clients can design and publish their own rules which will be shipped to the server in the Web's *lingua franca* for data exchange, the JSON format. SRL therefore provides mechanisms in which modern Web applications can easily be enriched with rule-based reasoning capabilities.

4.2.1 SRL Syntax

Similar to other rule-based languages, the Serena Rule Language defines templates describing the program data and rules describing constraints. The compact grammar specification for SRL is shown in Figure 4.1. The specification adopts the use of parentheses for delimiting definitions, $*$ for the usual zero or more occurrences, $+$ for one or more occurrences, square brackets for an optional occurrence and a vertical bar for selection. For more precise syntax, the complete ANTLR v.3 syntax for SRL can be found in [Kam].

A program in Serena generally represents event patterns captured using declarative specifications. In order to successfully identify these patterns, SRL consists of templates, rule and plugin definitions. We illustrate the syntax of these definitions in the next sections.

4.2.2 SRL Fact Templates

To define the structure of the data and their types in SRL, developers first explicitly define templates. As with other rule-based languages, the format of the facts is predefined using templates. Templates therefore provide blueprints that will be used to describe event instances represented as facts in the framework.

Listing 4.1 illustrates the template for access device data in the security example as a JSON object received by the server. Templates in SRL contain a name, shown in line 1, an optional comment and a number of template slot definitions. Template slot definitions are simply attribute-value pairs of slot name, slot type and a default value. The slot name is mandatory. An access device in this case consists of an id of type `int`[2], name as a `string` and location. The default type for any slot is `string`; since `location` has no explicit type (line 6), it is therefore implicitly assigned the type `string`.

[2]The type `int` is as specified in the host JavaScript language as double-precision floating point numbers

$$
\begin{array}{lcl}
P \in SRL & ::= & (t^* \mid r^* \mid p^*)^+ \\
t \in templates & ::= & \text{template } t_n\ [c_m]\ t_s^* \\
r \in rules & ::= & \text{rule } r_n\ c^+\ a^* \\
c \in conditions & ::= & c_e \mid t_e \mid b \\
c_e \in cond\text{-}elms & ::= & \text{type } t_n\ slc^* \\
slc \in slot\text{-}constr & ::= & s_n\ (== \mid \neq)\ s_v \\
t_e \in test\text{-}conds & ::= & e\ \ (< \mid \leq \mid = \mid \geq)\ e \\
 & & \mid e \\
p \in plugins & ::= & \text{plugin } p_n\ f^* \\
f \in functions & ::= & f_n\ e* \\
e \in expressions & ::= & [p_n.]f_n\ [e] \\
 & & \mid var \mid e\ \delta\ e \\
b \in binds & ::= & var\ \leftarrow\ c_e \\
t_s \in template\text{-}slots & ::= & s_n\ [\gamma] \\
\delta \in operators & ::= & +\ \mid\ -\ \mid\ *\ \mid\ /\ \mid\ \%\ \mid \ldots \\
s_v \in values & ::= & num \mid string\ \mid var \\
\gamma \in types & ::= & int \mid string \mid bool \\
a \in actions & ::= & \text{assert } t_n\ [s_n \Rightarrow e] \\
 & & \mid \text{retract } var \\
 & & \mid \text{modify } var \text{ with } [s_n \Rightarrow e] \\
 & & \mid \text{call } [p_n.]f_n \\
 & & \\
var & ::= & VariableName \\
c_m & ::= & Comment \\
r_n & ::= & RuleName \\
p_n, f_n & ::= & Plugin,\ FunctionName \\
t_n, s_n & ::= & TemplateName,\ SlotName
\end{array}
$$

Figure 4.1: Compact grammar of the basic Serena Rule Language

Each template definition is also assigned default system slots that include `sign` and `time` that represent the sign of the fact (explained in Section 4.4.5) and the time the fact was asserted into the server rule engine.

Listing 4.1: Template for access device

```
1 {templatename: "accessdevice",
2  comment: "the proximity badge scanning device",
3  slots: [
4      {name: "id", type: "int", default: 0},
5      {name: "name", type: "string"}
6      {name: "location"}
7  ]
8 }
```

4.2.3 SRL Rule Definitions

In SRL rules define the requirements of clients and contain conditions and actions. Conditions are expressed as predicates that capture facts. The SRL language views computation as controlled inference using mechanisms based on pattern-matching via *unification*.

Following the motivating example from Section 2.3, we use a simple rule implementation in SRL to explain their structure. The example introduced a security system deployed in an office environment to control access patterns.

We present the *InternAccess* rule that specifies that interns are only allowed access to their cubicle space. Remember from Section 2.3 that the rule should capture accesses made by intern employees between 8am and 8pm. Listing 4.2 shows the *InternAccess* rule as JSON received by the server. The SRL syntax is comparable to that of CLIPS (cf., Listing 3.3.2).

Listing 4.2: Rule for intern access

```
1 {rulename: "intern_access",
2  conditions:[
3   {type:"employee", level: "intern", name:"?name"},
4   {type:"accessdevice", name: "?dev", location:"cubicle"},
5   {type:"accessreq", id: "?reqid", person: "?name", time: "?t", device: "?dev"},
6   {type:"$test", expr:"(time.hourBetween(?t, 8, 20))"}
7  ],
8  actions:[
9   {assert: {type: "accessrep", reqid:"?reqid", allowed: true}}
10  ]
11 }
```

Rule structure

An SRL rule consists of a name, the left-hand side (LHS) and the right-hand side (RHS). The `rulename` identifies the rule. The LHS contains `conditions` (lines 2 to 7). Conditions specify a pattern to match against a fact in memory. The RHS contains the `actions` to be taken when the conditions have been fulfilled (lines 8 to 10).

Conditions

The first condition in line 3 defines a predicate that captures a fact representing an intern employee. The `?` operator on the same line defines a logic variable `?name` to capture the employee's name. During execution, the inference engine needs to find the appropriate substitution instance that will match with the logic variable. This process of finding concrete substitutions that match with logic variables in a predicate is called *unification* and is a powerful form of pattern-matching. Any subsequent uses of the same logic variable in the predicates of the rule, as in line 5, are syntactically the same and cannot be changed.

The next condition captures the device located in the entrance of the cubicle space (line 4). The actual request made is captured in the next line 5 with the device and the name ascertained to be the same, by use of the same unified logical variable `?dev`. In this line, `?t` is the time of the access request.

Line 6 shows a test condition that specifies a boolean expression. Expressions can use a host of arithmetic, relational and boolean operators. Several test conditions can be used in a rule, and multiple test expressions can be defined within one condition by chaining them with conjunctive and/or disjunctive operators that follow the native JavaScript syntax. Generally, tests act as computational filters for facts that only match their specified expressions on the grounded values of the facts. In this case, the expression test is successful only if the recorded time of an `accessrequest` fact occurred between the specified hours. Conditions therefore have the effect of introducing computations in rules. Normal conditions match to facts, and test conditions filter matched facts based on their values.

A condition *matches* a fact if all its literals and its consistent variable substitutions are satisfied. In SRL, the simplest condition element is a predicate that specifies a fact type, e.g., `{type:"employee"}`. If a pattern on a slot is not included in the condition, then it has no restrictions on the slot values during the matching process; therefore the above condition will match any fact of type `employee` in the fact base.

Variables in SRL rules

In SRL there are two types of logic variables: **slot variables** and **condition variables**. *Slot variables* bind to slot values and serve two purposes: 1) they unify matches by specifying unified substitutions in the rule's predicates (`?name` in lines 3 and 5 of Listing 4.3), and 2) they enable the use of concrete values that were bound on the left hand side to be referenced in the right-hand side modifiers (`?reqid` in line 9) Unlike slot variables, *condition variables* bind to whole facts. They can be used with slot names to access specific properties of the fact (as $r in lines 5 and 6).

Listing 4.3: Rule for intern access using condition variable $r

```
1 {rulename: "intern_access",
2  conditions:[
3   {type:"employee", level: "intern", name:"?name"},
4   {type:"accessdevice", name: "?dev", location:"cubicle"},
5   {$r: {type:"accessreq", id: "?reqid", person: "?name", device: "?dev"}},
6   {type:"$test", expr:"(time.hourBetween($r.time, 8, 20))"}
7  ],
8  actions:[
9   {assert: {type: "accessrep", reqid:"?reqid", allowed: true, time: "$r.time"}}
10 ]
11 }
```

Actions

When all the conditions specified in the LHS are satisfied, then a sequence of *actions* defined in the RHS is activated. In an SRL there are three main types of actions:

- **assert** adds a new fact into the engine according to the assert definition
- **retract** removes the fact from the engine
- **modify** modifies the fact according to the specified definitions
- **print** writes to the output log and can be useful for debugging purposes

In the example rule of Listing 4.3, the RHS of the *InternAccess* rule asserts that the access request has been granted by the reply, in line 9. In the action, the request is bound to `?reqid` that was bound by the condition in line 5.

The *assert* action creates the fact with slot names and default values according to its template. Then, the attribute elements specified in the assert definition are used to overwrite the default slot values. The resulting fact is then added to the fact base.

The *retract* action is used to remove fact elements from the fact base. Some facts may need to be removed in cases that they are not needed, e.g., they may cause inappropriate rule activations in the case that they have already been fired. Retract takes a condition variable as an argument and upon execution removes the fact bound to the variable.

The *modify* action internally operates as a retract and assert combination and logically modifies the specified fact with the defined values. An additional *halt* action is available that can be used as a special directive to stop execution of the rule engine, if enabled in the rule engine configuration.

4.2.4 SRL JavaScript Plugins

RKDAs sometimes need custom functionality that can be used to validate whether an event can be part of a pattern to be detected. Serena provides this functionality by allowing a number of system and custom plugins. Plugins are optional components, and simply define various user-defined pure functions (or helper functions) that can be referenced within rules. The functions are required to return boolean values and can thus be conceptualised as advanced predicates: they are used to evaluate the test expression in a `test` condition.

All plugins in Serena are declared in the framework's host language JavaScript. Installation of plugins requires a server-side configuration process of adding the `.js` file in the package's `plugins` directory before engine initialisation. An example of a default plugin in Serena is the `datetime` plugin, which consists of functions like `getHour(time)` that extracts the hour of the time input format, and the `location` plugin with `distanceBetween(x1, y1,x2,y2)` that calculates the euclidean distance between two points. An over-reliance of plugins, however, is usually an indicator that the programmer is working against the rule language and should re-evaluate the application scenario to see if a rule-based solution was indeed the best fit.

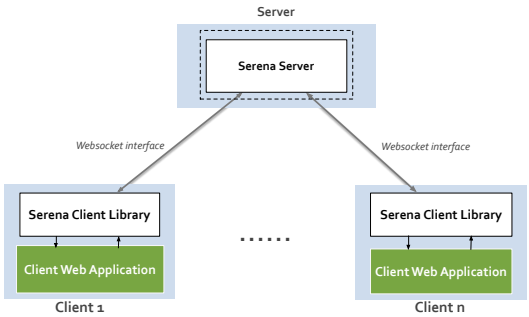


Figure 4.2: The Serena framework distributed architecture – The framework consists of a client library that uses WebSockets to connect to the rule-based server.

4.3 Serena: Architecture

Serena augments an event-driven web server with a forward-chaining inference engine that processes events reactively using rules. In Serena *clients create and install the logic reactive rules* that define the complex events *they* are interested in. Clients can also dynamically upload data to the server using facts sent in JSON format. Such data can be sent at a steady rate, but in RKDAs it is characteristically intermittent. As seen in the previous section, the rules specify which data to match, and once activated the rule can send this activation as a notification to the client.

4.3.1 Client-server Interaction

The architectural design of the Serena framework consists of client and server components, as shown in Figure 4.2. Clients are entities that are connected to the server and can consist of devices and other autonomous computational entities. The centralised server is the main processing unit of the framework. The two sides are connected to each other through *push-based* bidirectional WebSockets for low-latency communication providing fast real-time responses over the Web platform. At any given moment, a number of clients can be simultaneously connected to the server.

We illustrate a practical interaction sequence between a client and the server in Figure 4.3. Clients initially connect to the server through a client library provided by the framework. In the example scenario, the rule can be designed and uploaded by a security staff client that is connected to the Serena server. A client can thus design and send rules to the server, after which the server receives the rule and maps it uniquely to the client. The client can then start sending events, which will be added as facts to the server rule engine. If the client's rule is activated, the server will send a notification with the relevant facts to the client. The Serena client library receives the facts and calls a handler in the client application to react to the activation of the rule.

We discuss the specifics of the architecture and the execution semantics of the framework in the upcoming sections.

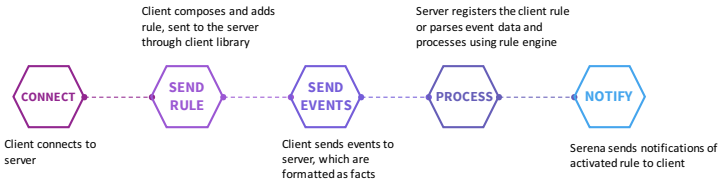


Figure 4.3: Typical client-server interaction sequence in Serena – Clients add rules and send events to the server for processing, and the server sends notifications if rules are activated.

4.3.2 Server Architecture

The Serena framework's server is the main processing entity and runs on the Node.js event-driven platform [Joy10]. It consists of several components for maintaining client connections, processing events and sending notifications. Figure 4.4 shows the server architecture. Each component is discussed in detail next, with specifics explained in the upcoming sections.

The Rule Register

Rules uploaded by clients are processed by the rule registration component. Its main function is to parse received rules and to map each rule to its client. Rules are read according to the SRL specification and are inserted into the rule engine upon reception by Serena. The rules are then parsed according to predefined templates. As described in Section 4.2.2, templates provide metadata that describe the format of the facts that are used in rule condition elements. The rule register then sends the rule to the rule engine and registers a notification channel for the client for this rule.

The Event Manager

The event manager has two main functions: supplying event data to the rule engine and managing client connections. Events are sent asynchronously by clients (e.g., mobile devices or sensors) and Serena's event manager uses an event queue to asynchronously store event data as they are received at the server side. Stored messages are dequeued in FIFO order and inserted into the rule engine for processing. To manage the often volatile client connections, the event manager maps WebSocket event channels to clients and maintains unique client session identifiers. It communicates with the client library to maintain WebSocket connections and to identify specific clients that connect to the server.

The Notification Manager

The notification manager receives notification data from the rule engine once a rule is activated. The data usually includes the facts that were involved during the instantiation of the rule. The notification is then sent to the client's notification channel with additional metadata about the notification (e.g., the notification time), and the client library invokes the relevant handler in the client code. The notification manager runs in an event loop fashion and may bundle several notifications, especially under heavy load.

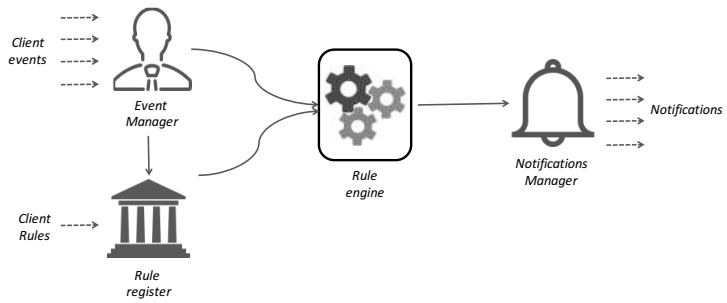


Figure 4.4: Serena server architecture – The server contains several components for managing rules and reasoning about incoming events.

The Rule Engine

The rule engine is the main reasoning component at the server-side and is used to process events according to the rules uploaded by clients. It is tasked with efficiently evaluating client events in order to determine applicable rules that can be activated. The rule engine operates in an event-loop fashion that fits well within the event-driven model of its host server, Node.js. Once a rule is activated the engine sends the accompanying data to the notification manager for the client to be notified.

4.4 Serena: Execution Semantics

Based on the architecture overview presented in Section 4.3, this section discusses the execution semantics of the server in detail. The discussion will pivot around the rule engine since it is the most significant reasoning component. The rule engine contains specific modules to achieve its functionality.

Implementing rule-based functionality requires maximum efficiency due to the complex pattern-matching techniques that are supported. Reducing the amount of matching in rules therefore guarantees faster server execution. For this reason, the Serena rule engine is based on **the Rete algorithm** to determine which combinations of facts are relevant for which rule inserted by the rule register.
A typical Rete-based system's architecture is depicted in Figure 4.5 and contains:

- *Rule base:* Rules from the rule register are added to the rule base to be used when building the Rete graph and when activating rules. The rule base thus supports the logic of applications by storing the collective knowledge that is used in evaluations of rules.
- *Fact base:* Facts that represent event data from other clients are added to the fact base. It is thus a global store that contains data that is accessible to the entire engine.
- *Inference engine:* The inference engine contains the pattern matcher and the activation scheduler which employ Rete to determine which rule to fire given the current state of the engine.

Rete provides efficient rule evaluation that is achieved through exploiting 1) **structural similarity** – sharing of the nodes when building the Rete graph, and 2) **temporal redundancy** – caching of intermediate matched data *tokens* between cycles of incoming results (at the price of higher memory usage). The next sections discuss how the inference engine provides this efficiency while performing the reasoning process.

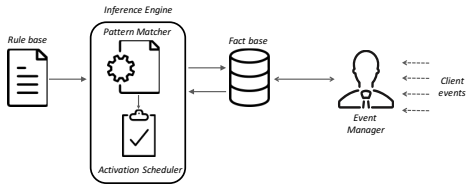


Figure 4.5: Serena rule engine components – The inference engine performs rule processing according to data from the fact base and rules from the rule base.

4.4.1 Reactive Rule Engine Execution

Classical forward-chaining inference engines follow the basis of control using a three-stage cycle: match, select and execute (Section 2.5.3). After the 3-stage cycle completes the rule engine execution halts if there are no more rules to activate. As discussed earlier, rule engines such as Drools from Section 3.3.2 and Jess from Section 3.3.2 operate using this model.

A web server requires continuous execution in order to serve incoming client data. Serena provides reactive rule engine execution by embracing the single-threaded event loop that forms the basis of operation of the Node.js server. The execution semantics of the rule engine is therefore intertwined with the Node.js server's event-loop model [MTS05], where a single thread of execution embodies it. The **event loop** uses an *event queue* to order and schedule the execution of all requests. Every concurrent interaction is then modelled as an event that is tied to an event handler. An event handler has access to a global state which is shared with other handlers; but with a single thread of execution the state is guaranteed to be consistent at any time, i.e., no concurrent modifications can be performed from other threads. Any new event will be picked from the queue by the main thread again, creating an event life cycle that continues as long as the main server process is not halted. This model is able to provide scalable processing semantics for event-driven applications, e.g., as shown in the evaluation by Chaniotis [CKT15].

We explain how this model can support sequences that are as a result of distributed clients uploading rules and sending events continuously and non-deterministically in the upcoming sections. The example of security monitoring with the *intern access* rule is used to show a typical *sequence of complex event detection (CED)* in the framework below, in stages.

1. Client uploads *InternAccess* rule R to server
2. The rule R is parsed and added to the inference engine (knowledge base update)
3. Client device at access point sends access requests as event data E (event capture)
4. Server updates fact base with E (fact base update)
5. The rule engine performs matching based on E to find client rules to fire (matching process)
6. Activation scheduler selects and fires selected rule R (rule application)
7. The server asynchronously notifies relevant client(s) rule R was fired (rule notification)
8. The server waits for more event data (engine wait), and then proceeds to stage 4.

Serena Node.js Server

The execution semantics of the stages of the inference engine embraces the *event loop* model of Node.js. The use of a single-threaded event offers a safe shared state and avoids complexities that arise from orchestrating the execution of a rule engine using several threads, as discussed w.r.t Drools, Jess, etc. (see Section 3.3.2), and has given rise to alternative concurrency models in other rule engines [Pet+14; Swa+13]. The Node.js engine has been especially optimised for event-driven execution using this model from the ground up thus limiting the drawbacks of the model itself. Furthermore, the event loop model maps onto the foundations of the host language JavaScript and to the use of the performant V8 engine by the Node.js runtime.

4.4.2 The Inference Engine and Rete

The inference engine is the heart of the rule engine. It serves as the interpreter for uploaded rules, evaluating received data according to the semantics of the rules. It is composed of several submodules:

- *The Rete graph builder* receives the rules from the clients via the rules repository, parses the rules, and builds the Rete graph based on the Rete algorithm (stage 2).
- *The matching component* is tasked with finding consistent fact bindings in the fact base (stage 5). It builds an instantiation or activation for every set of facts that satisfy a rule and places them in the queue of the activation scheduler.
- *The activation scheduler* takes the set of all rule instantiations and executes or *fires* them given an activation strategy (stage 6).

We use the example of the *InternAccess* rule to explain the concepts of the inference engine.

Building the Rete Graph

As mentioned, the inference engine match cycle is based on the Rete algorithm [For79].

Listing 4.4: Rule for intern access - revisited

```
{rulename: "intern_access",
 conditions:[
  {type:"employee", level: "intern", name:"?name"},
  {type:"accessdevice", name: "?dev", location:"cubicle"},
  {type:"accessreq", id: "?reqid", person: "?name", time: "?t", device: "?dev"},
  {type:"$test", expr:"(time.hourBetween(?t, 8, 20))"}
 ],
 actions:[
  {assert: {type: "accessrep", reqid:"?reqid", allowed: true}}
 ]
}
```

When the inference engine receives a new rule it builds a *Rete graph*. Rete compiles rules into a data-flow graph that filters incoming facts (data) as they propagate through its nodes, performing the actual matching process. We show the graph for the *InternAccess*

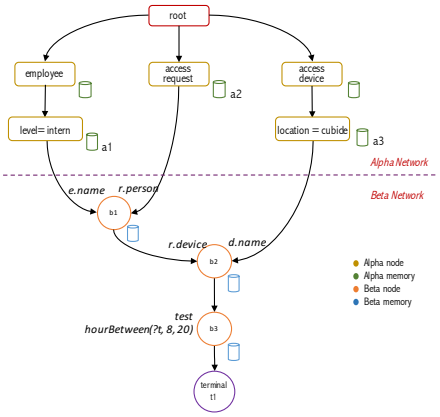


Figure 4.6: The Rete graph for InternAccess rule – The graph is a DAG that contains of two parts, the alpha and beta networks, consisting of nodes and edges in between them.

rule of Listing 4.2 (the complete rule is shown in Listing 4.4 for easier reference) after addition to the server in Figure 4.6. The graph consists of two regions, the **alpha** and **beta** network, with facts coming in from the root node.

The alpha network. The alpha network contains alpha nodes that perform intra-condition tests – such as the leftmost alpha node, that checks if a fact is of type `employee` and if the employee is an `intern`. They filter facts of that type and store them in their own alpha memory as a list.

The beta network. Below the alpha network is the beta network, which is built in the lexical order of the condition elements forming a left-associative binary tree. Two-input *beta nodes* or *join nodes* perform inter-condition tests known as join operations on their left and right inputs according to the corresponding conditions. A *beta memory* is associated with each beta node and holds the intermediate join results. The leftmost beta node $b1$ in Figure 4.6 performs joins for an `employee`'s name and the name of the person performing the `access request`, creates a *token* by appending facts that passed the test as a list, and sends it to the next beta node. It also serves as left input for successive nodes in the beta network. The second beta node $b2$ receives the token and performs joins of facts of an `accessdevice` with the device from where the `accessrequest` was made. As observed in Figure 4.6, for any join node, the right input is always an alpha node.

The beta network also hosts the nodes that represent test conditions. Test conditions produce beta test nodes, such as the node that checks for the time of the access request. The final beta node in a condition sequence represents the full activation of a rule and is called a *terminal node*. In this case the rule *InternAccess* will be instantiated once a token reaches this node.

Structural similarity. Rete takes advantage of structural similarity, where similar elements are used to enable sharing of nodes when building the network. This is because rules can share tests against the same fact type, fact slot values, relations between condition elements and entire condition elements themselves. For instance, suppose another *InternAccess* rule is to added for interns that work the third shift (also known as the graveyard shift) for overseas clients, between 10pm and 5am. The *InternAccess3rdShift* rule is shown in Listing 4.5. The rule is similar to that of Listing 4.4. It however differs in line 6 where the times for the specified shift is are checked. The structural similarity property enables Rete

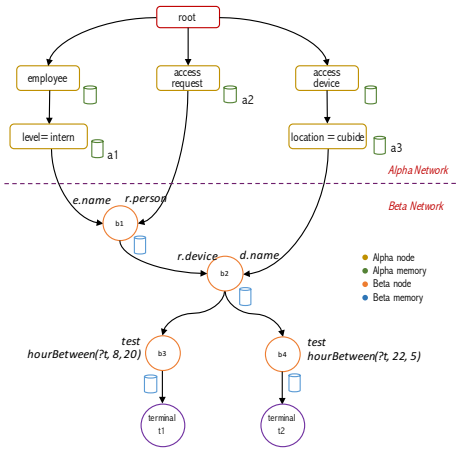


Figure 4.7: The Rete graph of the InternAccessThirdShift rule – The property of structural similarity in Rete enables the sharing of nodes across rules that have similar properties.

to reuse both the nodes and their intermediate results for similar rules thereby improving space and processing efficiency.

The resulting graph after addition of the rule in Listing 4.5 is illustrated in Figure 4.7. The graph has only changed where the test expression for the new rule is found to differ from that of the existing test node $b3$ in the graph, and only adds the new test node $b4$ and terminal node $t2$ for the second rule. In general, there exists a Rete network for a particular rule set: with more similar rule conditions, sharing is increased in the graph [Doo95].

Listing 4.5: 3rd shift intern access rule

```
{rulename: "intern_access_third_shift",
 conditions:[
  {type:"employee", level: "intern", name:"?name"},
  {type:"accessdevice", name: "?dev", location:"cubicle"},
  {type:"accessreq", id: "?reqid", person: "?name", time: "?t", device: "?dev"},
  {type:"$test", expr:"(time.hourBetween(?t, 10, 5))"}
 ],
 actions:[
  {assert: {type: "accessrep", reqid:"?reqid", allowed: true}}
 ]
}
```

The Matching Process: The Rete Algorithm

In stage 4 of the CED sequence, a fact base update triggers the next stage, namely the matching process. The matching process searches for consistent bindings between incoming facts and the existing rules.

Incoming data is received by the server and it creates an instance of a fact. The fact is then inserted into the Rete graph from the root node. The root node receives the fact and reads its sign (Section 4.2.2). A positive sign means the update is an assertion and should be added to the graph and negative means it is a retraction and it should be deleted. Facts traverse down the network as they are processed and forwarded by nodes, which store

intermediate computations in node local memories. The root forwards facts to its children, `employee`, `accessrequest` and `accessdevice` in this case.

For a concrete example refer back to the graph in the previous Figure 4.6. If a number of interns clock-in as they arrive this is captured and inserted as facts into the rule engine. The `intern` facts will be inserted into the Rete network from the root node and will eventually be stored as intermediate results in the $a1$ node. Similarly, the devices that are online will be stored in the memory of the `accessdevice` node and devices for `cubicles` are stored in the memory of $a3$. The beta node `b1` waits for an access request to continue the computation process.

Following the graph, when an intern requests access to a cubicle office space at around 12pm (this request should ideally be granted), the engine will receive the request and eventually adds it as an `accessrequest` fact to the Rete graph. The fact will enter the graph from the root node and will be sent to the `accessrequest` alpha node $a2$. This node will store the fact in its alpha memory and will send it to its child, the beta node $b1$ as shown in Figure 4.8a.

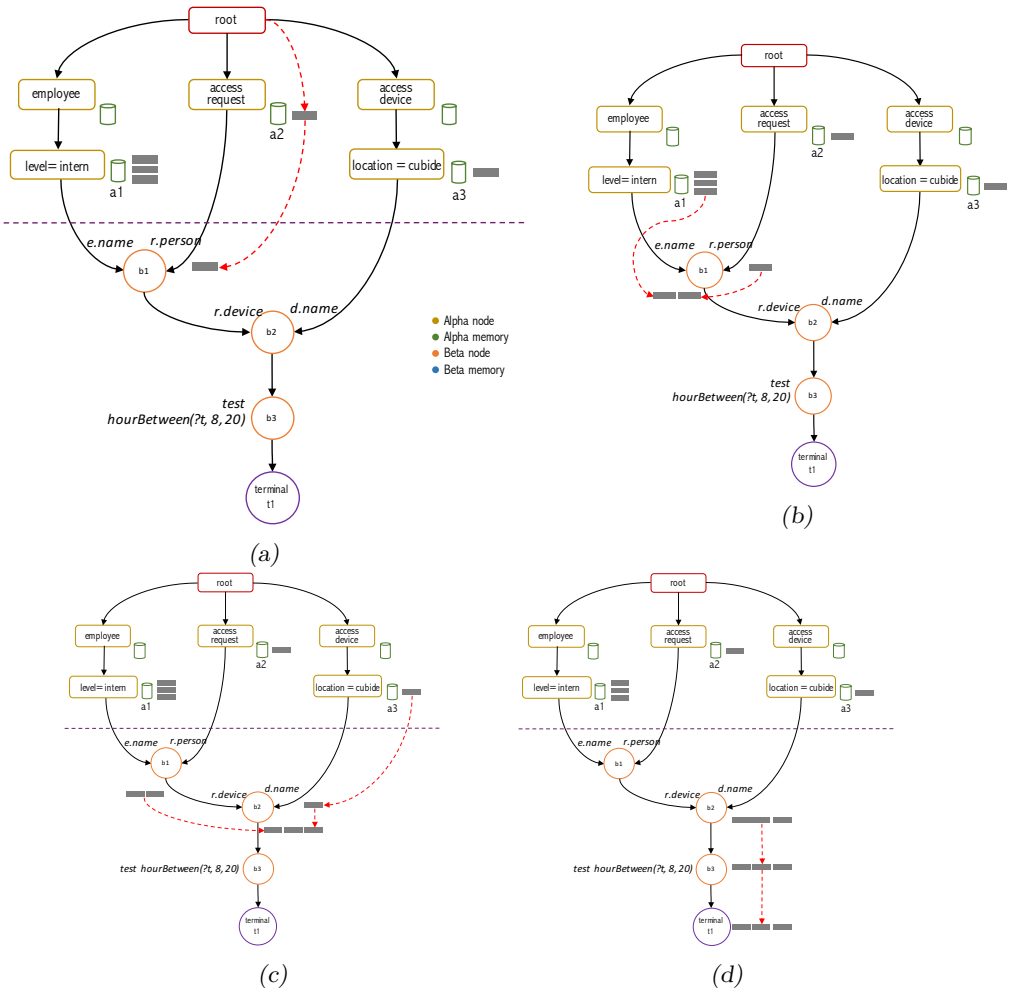


Figure 4.8: An example of a typical matching cycle in the Serena engine

Right Activation. When a fact is received at the right input of any two-input beta node, a *right activation* is triggered that issues a request for all the items in its left parent to compute consistent bindings for the fact (line 2 of Algorithm 4.1). This process is known as **matching**, and the subsequent test is called a join computation (line 4).

Algorithm 4.1 Beta Node Right Activation

```
1 function scopedBetanodeRightReceive(node : n, fact : f)
2     tokens ← n.getTokens()
3     for each token t in tokens do
4         if n.joinTestPassed(t, f) then
5             t_new tw ← n.createNewToken(t, f)
6             n.sendTokenToChildren(t_new)
7         end if
8     end for
9 end function
```

Left Activation. When a data item passes a join computation in the beta network, the Serena rule engine packages the result as a *token*. A token is logically a data structure that contains all the facts with consistent variable bindings up to that point in the rule. The facts in a token are internally represented by Serena as a list of pointers where each one points to the actual item in the alpha memory. A *left activation* is triggered when a token is received at the left input: in this case however the join test requests all items from the node's right parent which is always an alpha memory (line 2 of Algorithm 4.2). This is due to the fact that the Rete network always forms a left-associative binary tree.

Algorithm 4.2 Beta Node Left Activation

```
1 function betaNodeLeftReceive(node : n, token : t)
2     facts ← n.alphaMemory.getFacts()
3     for each fact f in facts do
4         if n.joinTestPassed(t, f) then
5             t_new ← n.createNewToken(t, f)
6             n.sendTokenToChildren(t_new)
7         end if
8     end for
9 end function
```

The beta node $b1$ will therefore request items from its left parent the alpha node $a1$. Node $a1$ will send all `intern` employee facts that it contains, and $b1$ now proceeds to perform its join test `(e.name == r.person)` that checks if the name of any of the employees is the same as that of the `accessrequest` fact.

When the person that made the request matches the employee name, then node $b1$ first creates a new token by appending the employee fact (line 5, illustrated in Figure 4.8b), stores this intermediate result in its beta memory, and in line 6 sends the new token to its children, node $b2$. The same sequence of steps occur at $b2$, but this time a left activation is triggered by node $b2$ to find out compatible access devices by performing join tests `(r.device == d.name)` on all devices from the alpha memory $a3$ as shown in Figure 4.8c.

If a compatible device is found then a token is created and sent to the test node $b3$ that checks whether the time for the request is within 8am and 8pm. The time 12pm succeeds

the test, so the token finally reaches the terminal node (Figure 4.8d), which means that the rule should be activated. In this case, the request made by an intern to enter the cubicle office space should be granted, and this rule and its bindings are sent to the activation scheduler for execution in stage 6 and eventual notification in stage 7 of the *CED*.

Observations. In summary, we list some of the main points behind Serena's execution semantics that uses the basic Rete algorithm:

- In general, at most $n-1$ beta nodes are needed to represent n conditions in a rule.
- The beta memory of node n contains matches for the first n conditions.
- The conditions of a rule are checked in order. If no combination of tokens match a sequence of conditions up to a certain point, then the remaining conditions are not evaluated.
- Rete takes advantage of structural similarity, where similar elements of different rules share nodes when building the network.
- Rete also leverages temporal redundancy by caching intermediate results of computations known as tokens as they traverse the network. Tokens reside in beta memories.
- The typical Rete network algorithm always forms a left-associative binary tree.
- Deleting a condition in the default setting requires a linear search in each node memory to delete the element in every node.

Optimising the Cost of Matching

The matching stage determines those rules that are relevant to the current state of the fact base for activation. As identified in [NGR88], a major bottleneck in the Rete algorithm is the expensive computations performed during this stage.

Concretely, as much as 90% of the execution of a Rete-based system can be spent in the match phase [Mir14], with the number of join comparisons made dominating the time the matching process takes. This was previously discussed in Section 2.5.4. For this reason, the main area of improvement when looking for avenues to speed up any Rete-based rule engine execution is join computations during the matching process. This dissertation aims to exploit this idea, by using an efficient encoding to quickly expedite the matching process in Chapter 7.

4.4.3 Rule Activation

In Serena a matching cycle can end up with an *instantiation*, which is simply a reference to a rule to be activated along with its tokens containing the facts that caused the instantiation. The instantiation is usually sent to the activation scheduler (shown back in Figure 4.5). The scheduler performs two main actions: it processes the RHS actions and sends instantiations to the notification manager to notify clients. RHS actions assert and retract, such as the assert in line 9, are queued in the event loop to be executed in the next matching cycle.

Sometimes an action can cause an existing instantiation in the scheduler to be invalidated. This is usually due to a retraction, and the semantics of retraction in Serena states that the token will eventually be retracted all the way from the alpha to the beta network to

the terminal node of the rule in one cycle (via tree-based deletion, explained in the next section). The instantiation will end up being tagged for deletion in the scheduler. Because the scheduler always first checks and removes tagged instantiations before performing actions that were already queued, the instantiation will be invalidated, maintaining consistency.

Resolution Strategies. In some cases, more than one rule can be instantiated in one cycle. Serena employs *resolution strategies* of similar engines such as Drools and Jess, which perform a last-in-first-out (LIFO) scheme for instantiations, i.e., the most recent rule is chosen for activation. The special case here is that the strategy is performed per client (otherwise only one client could dominate rule activations). Similarly, if it happens that two instantiations were added at the same time, Serena will arbitrarily choose one instantiation to queued for activation. For flexibility, developers can configure different activation schemes to be enabled at server startup: currently LIFO (default) and FIFO are supported. Custom resolution strategies that implement a pre-defined interface can also be developed and used by the activation scheduler. For example, for the evaluation of the Miss Manners benchmark in Chapter 8, the Depth strategy [Bri06] was added for use by the activation scheduler.

Refraction. Serena maintains semantics of *refraction* as in classic RBS systems [GR98b]. All rules undergo refraction, where identical rule instantiations are not allowed to fire twice in a row. Serena maintains fact identifiers of the last N instantiations ($N = 2$, by default) that fired a rule and will not activate the same rule with the same identifiers. This approach has been traditionally used to prevent possible infinite loops caused by repeated identical instantiations of a rule.

4.4.4 Client Notifications

This section outlines how client application code can be used to add rules and to receive notifications using the Serena framework.

When adding a rule using the Serena client library, the client application code specifies a handler that will be executed whenever that rule is activated on the server. By utilising the event loop model of Node.js, Serena can delegate long-running tasks such as writing to client websockets to the Node.js internal threadpool thereby reducing such overhead.

The code to add a client rule is shown in Listing 4.6. Currently the Serena client library supports Javascript-based application code, prominent on the Web. A connection to the server is made in line 1. This code is usually added as an initialisation step, e.g., in the `<head>` section of a webpage. In the background, the framework initiates and maintains a websocket connection to the server using the Socket.IO library [Rai13]. The initialisation code can optionally take a handler as the last argument, which will be invoked when a disconnection occurs. The client code then specifies the created `intern_access` rule from Listing 4.4 in line 3. Lines 4-6 and 7-10 declare the handlers to be invoked when the rule is activated and when the rule is added respectively. In line 11 the rule is sent to the server, providing the callbacks for acknowledgement and rule activation.

Listing 4.6: Adding a Serena client rule and notification callback

```
1 var serena = new SerenaClient('serverip', 'securitystaff');
2 //....
3 var rule = // json rule 'intern_access' here
4 var ackCallback = function(err, result){
5   // optional cb after rule added ack
6 };
7 var activCallback = function(rulename, facts, source){
8   // rule was fired
```

```
9   updateUI(facts);
10 };
11 serena.addRule(rule, activCallback, ackCallback);
```

On the server side, upon registration of a rule the rule registration module maps the rule to the client uploading the rule, shares this mapping with the notification manager, and triggers the client library to call the acknowledgement handler provided by the client. The notification manager maps the client's websocket to the activation handler and stores the mapping internally. When a rule is activated by the inference engine the scheduler sends the activation to the notification manager. The notification manager then identifies the client that added the rule, picks up the client's notification channel established using the websocket connection, and informs the client library to execute the `activCallback` handler on the client side.

4.4.5 Reactivity & Dynamism

From Chapter 2, the main focus of Serena is to support reasoning in RKDAs in todays dynamic Web environment. The engine thus contains further improvements to the basic Rete algorithm to fulfil these specific goals, explained in this section.

Remember from Section 2.5.3 that the traditional match-select-execute cycle assumes that when the `run` command is issued to a rule engine, the rule definitions have already been added to the rule base. This approach is obviously unsuitable for a Web server serving clients that can dynamically upload rules at any moment in time. The rule engine on the server would need subsequent re-compilation of the rules every time a client adds a rule – a process that becomes inefficient as time goes by and may introduce inconsistencies when resetting the intermediate results as cached tokens. The rule engine therefore needs to support the fact that clients can send rules and event data at any time. Serena supports this dynamism by incorporating several techniques, inspired by the goal of improving match cycle efficiency.

Reactive match-select-execute cycle – In order to allow the server rule engine to add rules and facts from clients during execution without triggering a whole recompilation process, Serena implements a process that is distinguishable from the traditional recognize-act cycle[3]. In the recognize-act cycle, *selecting a rule to execute* initiates the cycle because all facts to be matched are present in the alpha memories. This is why traditional engines contain a `run` command to begin the `select-rules` phase, after data has been added with the `make` command [Bri06]. In Serena, once the engine is already running, a cycle is initiated whenever *a change in the fact base is detected*. This is usually via an `assert` or `retract` command, which begins the `match` phase. Serena's inference engine then performs matching and then creates rule activations that arise. Any instantiations are placed in a queue to be executed by the event loop according to the activation strategy as explained in Section 4.4.3. Because Serena immediately processes incoming event data from clients, the two distinctions are the key difference in Serena's support for reactive processing in the engine.

Dynamic construction of the Rete graph – In most classical rule-based systems the engine reads all rules and creates a Rete graph through a compilation process. Even though compilation is fast, it makes addition of rules at runtime an expensive process. Any updates to the created graph often requires recompilation, thus creating a new graph. In Serena, the Rete graph is constructed dynamically. Initially when the Web server is run, the rule engine is also started. After it receives a client rule, Serena's inference engine appends the

[3]This technique was also used by a variant of the OPS5 engine [Bri06]

rule to the Rete network *on the fly* using the interpreted technique described by Doorenbos in [Doo95], rather than using the common compilation method. The technique builds the Rete graph top-down by appending nodes extracted from constructs in the parsed rule to the existing graph. Once the final terminal node is created at the bottom of the graph, Serena tags the node with the client's rule in order to map its potential activations to the client's notification channel. The graph-building process *is placed in front* of the server's event queue, before any succeeding event messages from clients are processed. This approach offers dynamism, but hampers refinement because modifying the existing rules at runtime offers limited restructuring techniques in the graph without disruption of the server execution process.

Algorithm 4.3 Adding a rule dynamically in Serena

```
1  function addRule(rule:r, rootNode:n, dummyBetaNode:b_0)
2      c_head ← r.conditions.head()
3      n_a ← createOrReuseAlphaMemory(c_head)
4      n_j ← createOrReuseJoinNode(n_a, b_0)
5      c_rest ← r.conditions.rest()
6      for each condition c_i in c_rest do
7          b_m ← createOrReuseBetaMemory(n_j, c_i)
8          n_a ← createOrReuseAlphaMemory(c_i)
9          n_j ← createOrReuseJoinNode(n_a, b_m)
10     end for
11     n_t ← createTerminalNode(n_j, r)
12     return n_t
13 end function
```

Dynamic update of the Rete graph – Certainly, the engine also needs to handle situations where there could be existing facts present in existing alpha and beta memories that can already trigger the newly-added rule immediately upon addition. When added, a new node starts from an empty state, processing inputs directed to it from its parent nodes. Therefore, tokens already created may not be able to process events that occurred earlier. Also borrowing from the said Doorenbos technique, after appending the nodes of the rule to the graph, Serena calls an optimised `update` procedure whenever each node is created (e.g. `createOrReuseAlphaMemory` in line 3 of Algorithm 4.3) that will cause all the parent nodes of the new rule to update their internal memories thus bringing the graph to a consistent state. From here, Serena can proceed in two ways. One, it can ignore any activations of the newly added rule for already-existing data. This is called the *amnesia method*, akin to a similar phenomenon in event stream processing systems [Hwa+05]. Alternatively, it can proceed with the *retrospection method*, where it executes the actions and the notifications of the new rule, causing all tokens that are sent to the node as a result of the update, to create activations. By default Serena takes the amnesia method, since retrospection has a performance penalty if rules are frequently added at runtime.

Dynamic Rule Removal – Just as rules can be added at runtime, they can also be removed in the same way. Clients remove rules by using the Serena Client Library API to remove a rule by specifying its rule name (which are unique per client). The Serena server receives the request and adds it to the event queue. The basic principle in removing a rule is based on a bottom-up process. When the inference engine dequeues the `remove` command it locates the rule's terminal node (identified from the returned value in line 12 of Algorithm 4.3) and begins the bottom-up removal process. Starting from the terminal

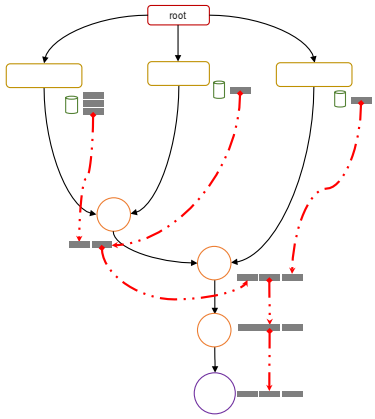


Figure 4.9: Serena's implementation of tree-based tokens in the Rete graph. Instead of performing matching during retraction of a fact the usual way, Serena just follows the path to the fact's children via its links.

node going up, the engine checks if the current node is 'in use', i.e., shares the node with another rule. If the node is not in use then its intermediate memory is removed and then the node itself is deleted, with its reference removed from the parents' `children` lists. The process then recursively proceeds in the same manner to the nodes parents and the removal is complete upon reaching the root node.

Tree-based Retractions – In Rete, retraction is often implemented by inserting a negative fact to the graph, which will trigger deletions as it percolates down the graph. This technique suffers performance limitations because it undergoes the normal matching process, performing expensive computations. For larger graphs this means that it traverses the entire network, an approach that does not scale well. Serena improves this by employing a technique from [Doo95] that *maintains pointers to any child facts/tokens*, see Figure 4.9. The pointers are created during assertions and when creating tokens as a result of a successful test either at the alpha or beta nodes. Retractions are thus simpler, because to retract a fact the engine follows the fact's child links to delete all instances initially created by that fact in the graph. This maintains the semantics of inserting a negative token as before, but speeds up the retraction process as it avoids any processing of tests or joins in the network at the cost of maintaining additional pointers in the engine.

4.5 Evaluation: Serena Rule-based Framework

The previous sections have elaborated the architecture and the execution semantics of the Serena framework. From the taxonomy in Chapter 3, it follows that Serena is a detection-oriented event processing system that contains rule-based semantics. This section proceeds to evaluate the framework on the basis of the features that enable it to fully support the development of applications with reactive knowledge-driven semantics.

4.5.1 Evaluation of Requirements & Comparison with Related Work

The Serena framework will be evaluated in the context of the concrete requirements that were introduced back in Section 2.6.

- *R1: Processing & Communication Model* – Serena is an event-based framework for RKDAs with clients sending fine-grained event data and receiving responses as event notifications. The architecture of the Serena framework makes it capable of receiving and processing events from a number of clients concurrently connected to the server, managed by both the client library and the event/notification manager at the server. The client library handles most nuances of initiating and maintaining WebSocket connections, session information and invoking rule activation handlers on the client side. At the server side, the event manager establishes and maintains connections and communication between the clients and the server, while the notification manager handles push-based client rule notifications. The connections between the clients and the server follow asynchronous non-blocking semantics, improving decoupling between the clients and the server. Notifications from the server are sent asynchronously to clients as well. The server benefits from this because it can reduce dependencies, thereby increasing the amount of clients that it can hold simultaneously, unlike traditional server-based threaded approaches supported by other rule engines. Its asynchronous nature can be seen to invert control in client notification code – better rule design can be used to mitigate such situations (e.g., via chaining of rules).

- *R2: Knowledge Encoding* – The Serena framework exposes rule-based syntax that reduces the complexity of writing code to effectively capture complex events as it is expressive and less susceptible to the non-deterministic nature of events coming from different clients. The SRL language described in Section 4.2 is inspired by similar rule-based languages and supports extracting complex events from simpler events through abstractions that provide powerful semantics such as pattern-matching and unification. Hence, SRL maximises on declarative definitions from clients, leaving specifics of computing them to the rule engine. However, SRL lacks advanced constructs such as quantification and aggregation available in engines such as Drools and Jess.

- *R3: Reactivity* – The Serena framework has been especially adapted to perform its execution continuously (Section 4.4.1). First, the server rule engine runs the Rete algorithm for a data-driven response when evaluating rules. Second, the inference engine is built with a modified match-select-execute cycle that *reacts* to incoming data rather than waiting for an explicit run instruction. Third, the inference engine also performs dynamic construction of the Rete graph, eliminating the need to have rules added beforehand using a static compilation process. Aside from enabling the engine to always receive new events and place them in the event buffer for processing, these techniques also allow the framework to build and evaluate incoming rules. It also delivers push-based responses in a reactive manner – even though responses may experience delay in heavier processing load.

- *R4: Dynamicity* – Similar to reactivity, the framework was built from the ground up with the dynamics of RKDAs in mind (as outlined in Section 4.4.5). This is particularly implemented by supporting the runtime addition of client rules to the framework – performed by adapting the inference engine to allow dynamic updates to its internal Rete graph's structure and content. Clients can therefore add and remove rules at runtime, with the option of amnesia or retrospective updates set beforehand when initialising the rule engine.

- *R5: Simplicity* – The framework's architecture and execution is integrated seamlessly with the Node.js event loop semantics. Serena uses the semantics of the event queue in Node.js to efficiently manage the processing of the inference engine and the activation

scheduler. It also delegates websocket interactions to the server for reading and writing client data efficiently. This makes the system performant when directly processing client requests, as explained in Section 4.3. To the client, though, the framework works as one unit that receives uploaded client rules and processes events. This approach however makes it somewhat harder to debug applications, and advanced debugging methods may be needed to identify sources of problems during execution.

Table 4.10 illustrates a recap of the related work outlined in Section 3.4 of Chapter 3, with the same ranking scheme. This time, the table is augmented with the ranking of the Serena framework contextualised in the rule-based engines section. Comparisons can be clearly drawn from the table. It can be observed that the limitations of classic rule-based approaches for event processing, dynamism, reactivity and simplicity that are suitable for RKDAs are suitably addressed by the Serena framework.

4.6 Chapter Summary

This Chapter introduced the Serena Framework, a blend of an event-driven web server and a rule-based inferencer. Serena,

1) eases the declarative definition of rule-based constraints by utilizing a rule-based approach,

2) efficiently processes intermittent data giving instantaneous feedback by incorporating a forward-chaining inference engine, and,

3) manages sending messages between the server and its tenants by abstracting the underlying infrastructure that supports push-based communication.

The syntax of the Serena Rule Language was first presented, paving the way for the description of the overall client-server architecture and the organisation of the server-side components. The Rete algorithm lies at the heart of the framework, and a practical example was used to showcase the execution semantics of the engine, from rule registration to notifying clients about rule activations.

Referring back to the requirements identified in Section 2.6, the chapter concluded with an evaluation of the Serena framework for supporting the reasoning semantics in RKDAs. The observation made was that the Serena framework not only offers the powerful abstractions of one-at-at-time detection and processing of events as in classical rule-based systems, but also thrives in supporting features that are significant in the modern Web landscape like dynamism and reactivity.

Even though Serena is well equipped to provide rule-based reasoning over the Web, the next chapter presents a phenomenon that rule-based systems face when exposed to multi-user environments exhibited by RKDAs. As it will be seen, this problem is unique to rule-based systems operating in such environments and is less applicable when compared to the classically-isolated rule-based systems.

	Event Processing	Knowledge Encoding	Reactivity	Dynamicity	Simplicity
Computation-oriented Engines					
Event Stream Processing Systems					
STREAM	★★★½☆	★★★☆☆	★★★☆☆	★☆☆☆☆	★★½☆☆
Data Stream Management Systems					
FLINK	★★★½☆	★★★☆☆	★★★½☆	★☆☆☆☆	★★★½☆
Detection-oriented Engines					
Active Database Systems					
ARIEL	★½☆☆☆	★★★½☆	★★★☆☆	★☆☆☆☆	★★☆☆☆
POSTGRES	★★★☆☆	★★☆☆☆	★★★½☆	★★½☆☆	★★★½☆
Rule-based Systems					
DROOLS	★★½☆☆	★★★★☆	★★★★☆	★★☆☆☆	★★★☆☆
JESS	★★½☆☆	★★★★☆	★★★★☆	★☆☆☆☆	★★½☆☆
CLIPS	★☆☆☆☆	★★★★½	★☆☆☆☆	★★★½☆	★☆☆☆☆
CLIPS COOL	★★☆☆☆	★★★★☆	★★☆☆☆	★★★½☆	★★☆☆☆
SERENA	★★★★☆	★★★★½	★★★★☆	★★★★★	★★★★½

Table 4.10: Evaluative feature comparison from Chapter 3 with ratings of the Serena framework – The framework thrives in the areas where classical rule-based systems were weak in, such as event processing & communication, support for dynamism and simplicity in integration with RKDAs.

5

Heterogeneity in Reactive Knowledge-driven Applications

If you want to go quickly, go alone. If you want to go far, go together.

African saying

One of the most distinguishing features of the Web landscape is its heterogeneity. In this chapter, we discuss the implications of heterogeneity in RKDAs, particularly on the effects of traditional processing cycles within the rule engine. In essence, heterogeneity as used in this dissertation primarily refers to multi-user contexts, but can be extended to others as well. We introduce the subject by investigating the extent of the effects of heterogeneity on shared rule-based systems in the Cloud in Section 5.2. In our view, the Cloud is a heterogeneous execution environment that leverages the Web as a platform, supporting deployment of RKDAs connected to distributed clients or devices. After identifying several criteria, we proceed to perform an analysis of the current state in techniques that offer ways to deal with problems in heterogeneous systems in Section 5.4. We conclude the chapter with a summary of the take-home points in Section 5.5.

5.1 Rule-based Systems and Heterogeneity

The evaluation made in Chapter 3 came to the conclusion that rule-based systems were most suited to support RKDAs in Section 3.4. The previous chapter 4 then introduced the Serena rule-based framework that is deployed to the Web environment. This section briefly presents the foundations of heterogeneity in rule-based systems and why this concept is important in the future adoption of RKDAs.

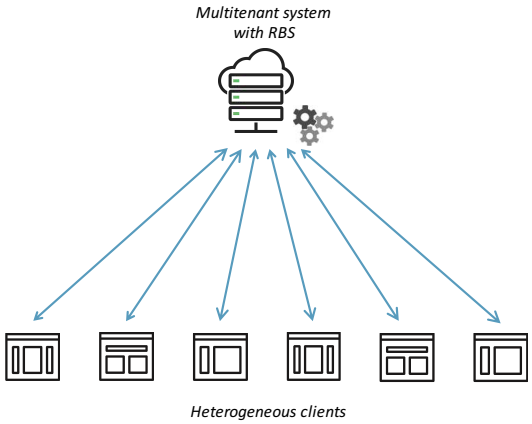


Figure 5.1: A multi-tenant system with a shared RBS serving heterogeneous clients.

5.1.1 Multi-tenant Rule-based Systems

Section 2.1.2 introduced multi-tenancy, which arose from the concept of time-sharing where different clients were connected to a server that provided a shared service. Rule-based systems (RBS) are increasingly used by Cloud service providers today to provide complex services for knowledge representation (Section 2.5.6). In these cases, multi-tenant rule-based systems can be used to accommodate the knowledge of all customers in a multi-tenant web system. RKDAs that leverage this multi-tenant system can be able to support a number of heterogeneous clients.

Aside from other benefits accrued by multi-tenancy, multi-tenant RBS can offer two main advantages. First, they offer **simplified deployment** of RKDAs. As opposed to setting-up and configuring a RBS system in-house, the provider can offer an easier way to subscribe to a multi-tenant RBS service. Second, they offer an avenue to foster **sharing community knowledge** collected from a variety of data from different clients in RKDAs. This is especially applicable to *native multi-tenancy*, where all clients use one shared instance at the application or middleware level (Section 2.1.2). Community knowledge is vital in RKDAs, which can be used to facilitate collaboration between the clients or to discern interesting 'big data' patterns that would be unable to be discovered in isolated systems.

In this dissertation we thus focus on having shared rule-based systems on the Cloud that can support the development and execution of RKDAs. Such systems are essentially native multi-tenant rule-based systems that serve a number of heterogeneous clients (Figure 2.2c). Because of this, we will refer to such systems as *heterogeneous rule-based systems*. Accordingly, RKDAs that leverage multi-tenancy are able to support a number of heterogeneous clients distributed over the Web, as in Figure 5.1. The Serena framework is an example of a heterogeneous rule-based system for RKDAs. However, these systems encounter issues when faced with a variety of heterogeneous clients contributing data. These issues are discussed in the upcoming sections.

5.2 Issues with Heterogeneity in Rule-based Systems

The driving force behind RKDA frameworks such as Serena is a rule-based system. Classic rule-based systems were conceptually designed to run in isolated configurations and were

therefore not conceived to operate in a shared, heterogeneous environment. Despite the conceptual independence of rules that promote modularity in rule-based systems, unexpected problems can arise from interactions between rules within rule engine in a heterogeneous context. The next section introduces a concrete example that will be used to discuss the issues in detail.

5.2.1 Scenario: Office Complex Security System

The motivating example from Chapter 2 showed a driving scenario that required the explicit representation, expressive detection and flexible reasoning of events. This section extends the example to highlight the unique problems that heterogeneity presents in RKDAs.

Motivating Scenario: Update

Previously, the company *Kimetrica* offices were located in one location that housed all employees. The company experienced growth in recent times resulting in more employees leading to higher day-to-day costs and a need for more office space. For cost-effectiveness, the company decides to move into an office complex that hosts several other companies.

The complex covers several floors and contains various amenities on each floor that cater to both resident employees and visitors of the companies. We show a basic logical organisation of the physical amenities of the large office complex in Figure 5.2a. The company *Kimetrica* ultimately reserves office space of two floors with about a quarter of the area per floor. It shares space with two other companies in the said floors, *Safari* Tours and *Soko* Marketing companies. It also shares amenities such as cafeteria and parking with the other companies residing in the complex.

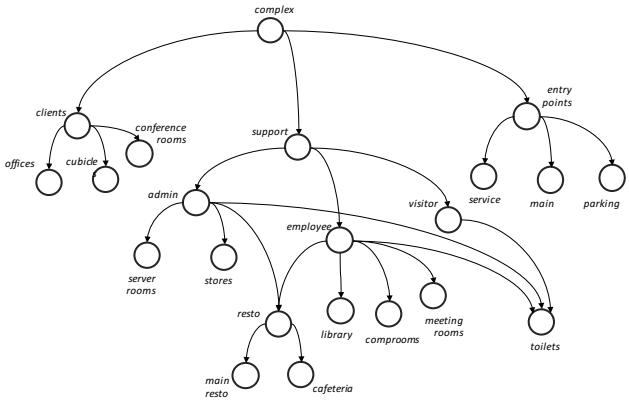
Within their respective office spaces, the arrangement of companies depends on these internal organisational structures. For reference, the organisational structure of the three companies are shown in Figure 5.2. Figure 5.2b shows the organisational structure of *Kimetrica*, Figure 5.2d shows that of the *Safari* tours company and Figure 5.2c illustrates that of the event *Soko* marketing company. In *Kimetrica*, employees of `levelA` are directors, `levelB` are department heads, `levelC` team leaders and lastly `interns` are short-time temporary hires. This shows that to an extent, the companies will be arranged according to their own predefined structures – that are expected to seldom change.

The updated version of the scenario offers a comparable analogy to the concept of heterogeneity using shared multi-tenancy concepts. In this case, the existence of several companies sharing office spaces corresponds to application instances sharing computing resources availed by a multi-tenant Cloud service provider.

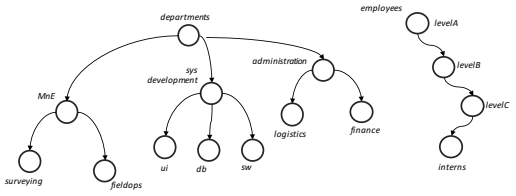
Heterogeneity comes at a cost, however. One of the main challenges is that the building faces more complex management logistics and higher maintenance costs because each company contains its own internal structures and day-to-day operations. To concretise this significant challenge we present the driving scenario of the security monitoring system in this context.

Security Monitoring System: Office Complex

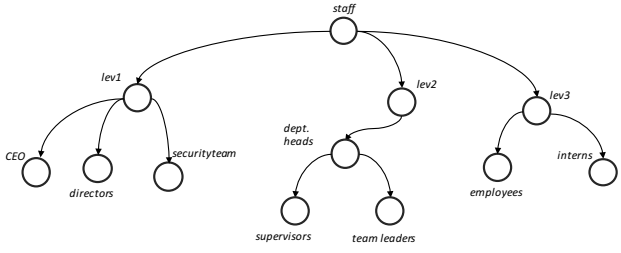
Similar to the security regulations in *Kimetrica*, the owners of the new building expect clients to adhere to strict security guidelines. The employees of client companies are issued badges that must be worn while inside the building. The office complex is a pervasive computing environment and the badges can be used to gain access to particular locations



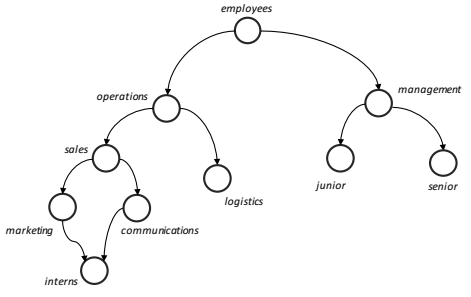
(a) Building logical layout



(b) Kimetrica consultancy structural hierarchy



(c) Soko marketing company staff hierarchy



(d) Safari tours company employee hierarchy

Figure 5.2: Structural organisation of companies in an office complex – The structure can be modelled as arbitrary DAGs with nodes having multiple parents.

in the office complex: most access points contain a scanning device that can restrict access to a location. Given this arrangement, the devices are therefore:

- associated to some node in the building's *physical* hierarchy,
- linked to some *logical* structure within the establishments.

For example, a device that restricts access to the marketing department's cubicles is linked to both the marketing team in the tours company and the cubicles of the building.

The owners of the office complex avail a security system that can regulate and monitor access patterns. In this case, the system is an online service that monitors and logs requests on security access systems for office buildings. The service can be offered by a provider in the Cloud that will host the monitoring security system, similar to the model of a Cloud Access Security Broker (CASB) [FYW15]. *Client companies formulate their own regulations of access and their security teams design rules for the service.* Using the service, the security teams can log and monitor access requests and receive updates if any of the accesses deviate from their security protocols so that relevant measures can be taken.

Because each client has its own security needs, each client can upload rules to the service. The security team of that client will then receive prompt notifications whenever the rules are triggered. When the clients start uploading rules to a shared, heterogeneous RBS, however, unexpected results can be observed. These issues are examined next.

5.2.2 Reentrancy in Heterogeneous RBS

Reentrancy is a phenomenon used to describe programs written in such a way that the same copy in memory can be shared by multiple users effectively. A program is reentrant if distinct executions of the program on distinct data cannot affect each other, whether run sequentially or concurrently [WST09]. Reentrant code is a requirement in common multi-user systems such as operating systems, where system programmers ensure that whenever a program is executed for a particular user there can be no other instructions that can modify data intended for another user[1].

Inference engines in RBS were not conceptually designed to work in the shared multi-tenant environment serving different users. This is because rule-based systems use a number of unordered rules that reference one single global working memory as can be observed in Sections 2.5.1 and 4.4. Hence rule-based systems are intrinsically non-reentrant where in this flat design space, activations could be observed from all asserted facts without discriminating their sources. To effectively exploit community knowledge, this problem needs to be addressed. This problem is exemplified in the next section with the updated example scenario.

Heterogeneous Rules in the Office Complex

Given that the security team of *Kimetrica* already designed several rules using their existing security regulations, they proceed to naïvely upload their *InternAccess* rule listed in 2.3 such as that shown in Listing 5.1. Supposing there are no rules inserted beforehand, the Rete graph as a result of adding the *InternAccess* rule is shown in Figure 5.3a.

The other companies also need to design rules to control access within their quarters in the building (refer back to their organisational structures in Figure 5.2). The *Safari* tours company has hired interns, and intends to control their access to various locations

[1]The use of the term reentrancy in this dissertation relates to the notion of reentrant procedures in multiuser systems programming and excludes those related to concurrent access and recursive method calls.

in the company premises. They therefore design and in the same way upload their own *InternCubicleAccess* rule that is shown in Listing 5.1. Evidently, this rule is identical to that of the Kimetrica except for the rule name and other variable declarations. This is because the *Safari* tours company also contains employees who are `interns`. Additionally, they capture data from `accessdevices` that have been installed (in the open-plan cubicles) by the security team of the building in their own rented area.

Listing 5.1: Tours company rule for intern access

```
{rulename: "intern_cubicle_access",
 conditions:[
  {type:"employee", level: "intern", name:"?name"},
  {type:"accessdevice", name: "?dev", location:"cubicle"},
  {type:"accessreq", id: "?reqid", person: "?name", time: "?t", device: "?dev"},
  {type:"$test", expr:"(time.hourBetween(?t, 8, 20))"}
 ],
 actions:[
  {assert: {type: "accessreply", reqid:"?reqid", allowed: true}}
 ]
}
```

The Need for Reentrancy

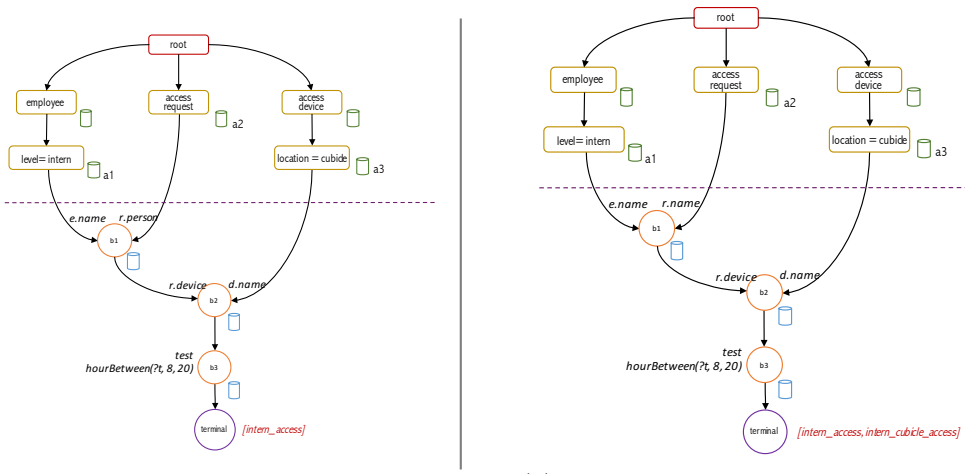
The shared Rete graph built after the addition of the rule by Kimetrica is shown in Figure 5.3a. As usual, the terminal node has been tagged with the rule so that received tokens will trigger an instantiation of that rule. When the new rule by the *Safari* tours company is inserted, Rete will reuse the nodes that are compatible with the rule definitions, as discussed in Section 4.4.2. The graph is structurally the same as before the addition of the rule, however this time the terminal node is tagged with activation of both added rules. The resulting graph is shown in Figure 5.3b.

Let us follow the execution sequence in the Rete algorithm for the graph shown after the addition of the second rule. Suppose that there are currently two intern `employee` facts in the alpha memory $a1$ representing one intern from each company, and that there is an `accessdevice` fact for entrance to the cubicle space[2]. This situation is shown in Figure 5.4a.

When an intern in the *Safari* tours company scans their badge to gain access to the cubicles, the device will send the data as an `accessrequest` to the server engine. The server will insert a fact with this info in the Rete graph. Next,

- The fact will traverse the node with $a2$ which will send it to $b1$ (Figure 5.4b) causing a right activation on the node.
- The node $b1$ will then request all the items in $a1$, in order to check whether the employee has the same name as that on the request. The intern from the tours company passes the test, and the token of `[employee, accessreq]` is made and sent to the next node $b2$ (Figure 5.4c) causing a left activation.
- Node $b2$ performs a similar process, checking if a given device matches the one in the request. If so it creates a token of `[employee, accessreq, accessdevice]` and sends it to the test node $b3$.

[2]We omit a second `accessdevice` fact for clarity since the same result of unintended activations would be observed in the terminal node



(a) First `intern_access` *rule added*

(b) Second `intern_cubicle_access` *rule added*

Figure 5.3: Resulting Rete graph after addition of rules from separate clients – The second `intern_cubicle_access` *rule reuses most of the graph that existed after the initial addition of the first rule.*

- If the request was made at an appropriate time, the test succeeds and finally reaches the terminal node as shown in Figure 5.4d .

Because the terminal node was tagged with both rules `intern_access` from the company *Kimetrica* and `intern_cubicles_access` from the *Safari* company, both rules will be activated on both clients. This in effect means that an intern from the *Safari* company has triggered an 'access granted' sequence in both companies' systems: an undesirable result! In this case one company will have a notification of granted accesses from unknown parties on its dashboards and in system logs.

> When employed in RKDAs, classic rule-based systems are said to be fundamentally *non-reentrant*. Given varied data sources, rules intended for one specific source can be activated by data from other sources.

Therefore, multiple heterogeneous data sources can lead to unexpected behaviour during the execution cycles of a shared rule engine. One undesirable consequence is that rule activations can be observed from all asserted facts without discriminating their specific source. In effect, the difficulty in localising rule control as shown in Figure 5.5 makes it hard to orchestrate the behaviour of rules in these settings. The simple example exposes the fact that in order to fully exploit capturing community knowledge in a heterogeneous RKDAs, it is vital that the system should avail mechanisms in which problems brought forth due to lack of reentrancy be suitably addressed. The next sections proceed to describe the manual ad-hoc methods that can be applied by developers of rule-based systems to try to solve this problem.

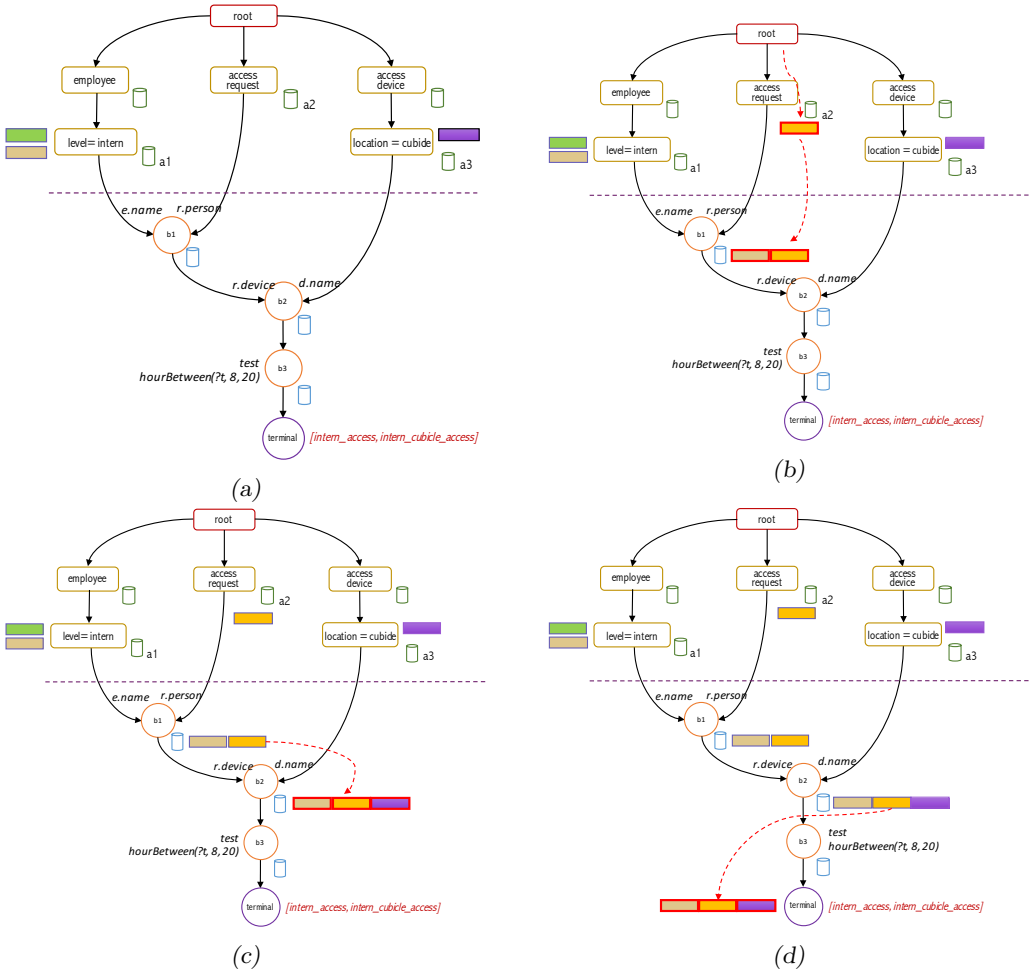


Figure 5.4: A token traversal sequence showing an unintended activation – When a token from either client is received in the terminal node both client rules case an activation.

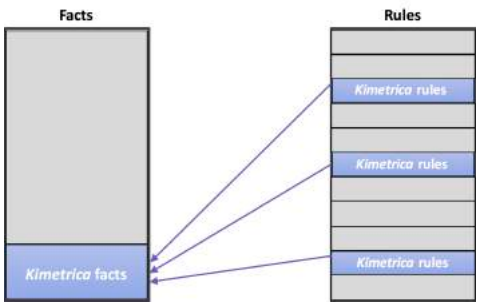


Figure 5.5: Conceptual vision for enforcing reentrancy in a heterogeneous RBS

Ad-hoc Solutions

Relation facts One way that introduces situational state is by manually adding IDs to every rule condition that will identify the source of the data. An example is having conditions in the *InternAccess* rule appended with additional `ID` slots, which contain enough information to delimit incompatible data. This method is however rigid and expects local rule designers to be aware of existence of global ID schemes.

A more flexible improvement is by using *relation facts*. Relation facts are facts asserted into the working memory that indicate a relationship between other entities in the working memory. Listing 5.2 illustrates this approach. Facts that indicate a *belongs to* relation (or similar) in the companies are added to the system, e.g., `{type:"belongsTo" entity1:"intern1" entity2:"toursco"}` relates an intern to the *Safari* tours company in the building. With this approach, the relation facts need to be added to the working memory *a priori*. Then, the rule from Listing 4.2 can be modified to bind to such facts so that one can distinguish between companies in the building. The rule is further modified by appending conditions in lines 6 and 7 to the rule, resulting in the modified rule shown in Listing 5.2.

Listing 5.2: InternAccess rule with relation facts

```
1 {rulename: "intern_access",
2  conditions:[
3   {type:"employee", level: "intern", name:"?name", company:"?empco"},
4   {type:"accessdevice", name: "?dev", location:"cubicle", company: "?devco"},
5   {type:"accessreq", id: "?reqid", person: "?name", time: "?t", device: "?dev"},
6   {type:"belongsTo" entity1:"?empco", entity2:"kimetrica"},
7   {type:"belongsTo" entity1:"?devco", entity2:"kimetrica"},
8   /* .. same as before .. */
9 }
```

Essentially, additions to the rule confirm that the intern belongs to the company *Kimetrica* and that the device is located within their premises. The rule is more flexible, i.e., when any employees or devices are changed then the preexisting relation facts can added or modified without changing the rule itself.

Consider a second example. Suppose a general rule is required in the office complex which stipulates that any person with a badge is allowed to use any toilet within their own premises in the building. Such a rule is shown in Listing 7.1.

Listing 5.3: ToiletAccess rule with relation facts

```
1 {rulename: "toilet_access",
2  conditions:[
3   {type:"employee", name:"?name", company:"?empco"},
4   {type:"accessdevice", name: "?dev", location:"toilet", company: "?devco"},
5   {type:"accessreq", id: "?reqid", person: "?name", time: "?t", device: "?dev"},
6   {type:"belongsTo" entity1:"?empco", entity2:"?c"},
7   {type:"belongsTo" entity1:"?devco", entity2:"?c"},
8   ],
9  actions:[
10  {assert: {type: "accessreply", reqid:"?reqid", allowed: true}}
11 ]
12 }
```

The resulting Rete graph for the rule is shown in Figure 5.6. Observe that the node needs to perform an additional check in beta node 4 to ascertain that the two data items `employee`

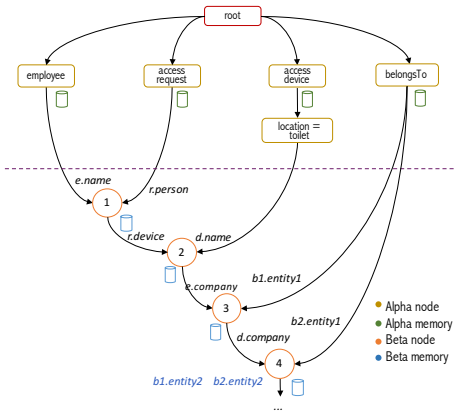


Figure 5.6: Rete graph for company ToiletAccess rule –The last join node 4 needs to perform an additional check for consistent data.

and `device` are sourced from the same company, represented by the check of the token with the fact $b1$ and $b2$. This check is extremely expensive because the matching process will perform cross-product computations of *all the data items* in the memories of the beta node and the alpha node in order to test for their compatibility.

Test Expressions A different approach takes advantage of the availability of expressions in rule conditions. The idea in this case is to require that every rule from a client to have additional discriminatory test conditions.

This can be illustrated with the same example as the original *InternAccess* rule. As with the previous method (of relation facts), first employees and devices are assigned their respective companies. This time, however, the rule contains additional expression tests that check, for instance, if the intern and the device are in the same company. Line 6 of Listing 5.4 adds a boolean test that checks if the intern and device are in the same company. This also results in a different Rete graph with an additional beta test node, shown in Figure 5.7. Note that at beta join node $b2$ the same complete cross-product joins of `employee` and `device` facts are computed.

Listing 5.4: InternAccess rule with test expressions

```
1 {rulename: "intern_access",
2  conditions:[
3   {$e: {type:"employee", level: "intern", name:"?name"}},
4   {$d: {type:"accessdevice", name: "?dev", location:"cubicle"}},
5   {type:"accessreq", id: "?reqid", person: "?name", time: "?t", device: "?dev"},
6   {type:"$test", expr:"( areInCompany('kimetrica', $e, $d) )"}
7   /* .. rhs same as before .. */
8 }
```

Discussion

As demonstrated, the above approaches are possible solutions to the reentrancy problem. The approaches however exhibit a number of limitations. From an engineering point of view, it is generally undesirable to have rule condition logic interspersed with event source

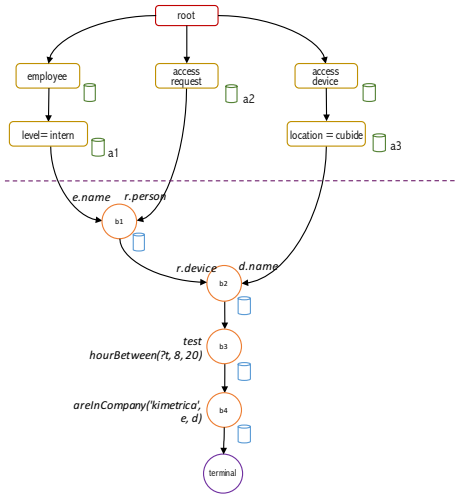


Figure 5.7: Rete graph for InternAccess rule with test expressions – The second test expression distinguishes data from other clients.

identification (also noted in [MFP06, p.131] in the context of notifications in event-based systems). This is mainly because they pollute the logical intent of the rule making it unnecessarily complex. Additionally, in these approaches clients would need to be aware of the existence and the layout of other clients. In more complex rules, it becomes tedious to distinguish which conditions need to be infused with the information that identifies clients and which ones do not.

Using ad-hoc methods forces rule designers to hard-code distinctions between clients and their data sources within the rules. This quickly becomes complex and error-prone as the number of clients and the relationships between them increase, or when the relationships become complicated to enforce using rule semantics. In a heterogeneous setup, failure to properly make these distinctions can cause unintended rule activations to leak in other clients.

> In summary, the ad-hoc approaches are problematic because they 1) draw context knowledge into application components that relates to the interaction with outside entities rather than the rule implementation 2) pollute the logical intent of the rule designer 3) complicate rule implementation and makes the process fallible. 4) impact the underlying Rete graph by creating additional nodes, requiring more computations.

5.2.3 Inter and Intra-Client Relationships in Heterogeneous RBS

The previous section showed how ad-hoc methods forces rule design to be polluted with logic to distinguish clients in RKDAs. By extension, rule complexity quickly becomes more tedious in configurations that contain complex client internal structures with multiple levels.

Refer back to Figure 5.2 that showed the structure of the companies and the office building. Usually, heterogeneous setups such as these contain complex structures that are modelled according to physical or logical **relationships** among participating entities. These

relationships can be based on various aspects, such as the principle of locality among interacting components in event-based systems [MFP06] and encapsulation in object-oriented systems. In simple situations these relationships can be based on the application-specific semantics tied to underlying structures, operations and processes of clients.

In the office complex, the *Kimetrica* company has a systems development department is composed of user interface, database and programming teams. Some rules in *Kimetrica* will want to capture events from users in the system development teams and exclude those from other levels in the company's hierarchy. Therefore aside from distinguishing between different companies, rule designers will also require distinctions between different levels of their own hierarchy. This is exemplified in the next section.

Custom Security Policies

Organisation-wide security policies are necessary but not sufficient to capture more fine-grained security policies of various clients in a heterogeneous environment. Consider the scenario where the security team of the company *Kimetrica* receive security policies that are applied in specific contexts within various departments. Some of the policies are outlined below.

- **Protocol 7** *Intern server room access*

  Interns in the systems development department can have access to the server room but only if accompanied by a department leader during working hours.

- **Protocol 8** *Financial office access*

  Only finance department employees have access to the financial records offices during working hours.

The rule for protocol 7 specifies that interns in the `sysdevelopment` department can have access to the `serverrooms` of their own departments but only if accompanied by a department leader during working hours. Listing 5.5 shows a rule enforcing such specifications using the aforementioned test expressions. The *InternServerRoomAccess* rule captures the two employees, the `intern` and the `sysdevelopment` department leader in lines 3 and 4. The `serverrooms` device is captured in line 5. Line 6 and 7 detect two access requests by the parties, indicating that both have to scan their badges on the device that enables access to the server room. To capture requests that originated from the same company, the test expression in line 9 is added. Line 8 makes sure that the access requests were made during working hours and that they were made simultaneously (specifically, within one minute) using the `time.diff` function.

Line 10 is where the main objective of the rule is specified where an expression is used to capture the various constraints of the *InternServerRoomAccess* rule. The line first checks the rank of the intern and the department or team assigned to the intern is related to systems development. The rank of the employee is then confirmed and lastly whether the device is located in the company's server room. Also, each `employee` fact needs to be modified with `dept`, and `team` fields in order to be evaluated by the expression (actually, the number of these fields depends on the levels of the organisation hierarchy).

Although flexible to updates, the approach using such expressions in the *InternServerRoomAccess* rule is complex and hard to debug if an error occurs. From the rule, it can be observed that with ad-hoc approaches, capturing such constraints increases the complexity of rule design implementation. It further muddles the logical intent of the rule designer – making them harder to analyse and understand.

Listing 5.5: Rule for intern access to server rooms

```
{rulename: "intern_serverroom_access",
 conditions:[
  {$e1: {type:"employee", name:"?intname"}},
  {$e2: {type:"employee", name:"?empname"}},
  {$d: {type:"accessdevice", name: "?dev"}},
  {type:"accessreq", id: "?reqid1", person: "?empname", device: "?dev"},
  {type:"accessreq", id: "?reqid2", person: "?intname", device: "?dev"},
  {type:"$test", expr:"(time.hourBetween(?t1, 8, 20)) && time.diff(?t1, ?t2, 60)"},
  {type:"$test", expr:"( areInCompany('kimetrica', $e1, $e2, $d) )"},
  {type:"$test", expr:"( ($e1.level == 'intern' && $e1.dept == 'sysdevelopment' &&
    ↪ ($e2.level == 'levelB' && $e2.dept == 'sysdevelopment') && ($d.location ==
    ↪ 'serverrooms'))"}
 ],
 actions:[
  {assert: {type: "accessrep", reqid:"?reqid", allowed: true, time: "?t1"}}
 ],
}
```

The resulting simplified Rete graph for the rule is shown in Figure 5.8. In the graph the second test expression node will perform expensive checks for all the constraints of the rule: the expression test will perform extensive computations to ascertain the validity of the compatibility of the tokens it receives in the network. This in-house compatibility check between data from different departments within one company is indeed a local manifestation of the reentrancy problem mentioned in Section 5.2.2.

In summary, it is common to have clients in a heterogeneous setting organised in hierarchies consisting of different components in different levels. We have seen that in a multi-tenant setup, there is need to distinguish data from other tenants. But also within the hierarchies, the clients can be organised to participate in various relationships that distinguish between various sub-hierarchies.

> We observe that in the spirit of community knowledge, exploiting the relationships between logical components in a heterogeneous configuration is a basis for addressing the engineering and management issues of these systems.

In the next section we present the requirements for addressing these issues in rule engines supporting such configurations.

5.3 Requirements for Heterogeneous Rule Engines

One of the main contributions of this dissertation is to provide a heterogeneous rule-based framework to support RKDAs. In order to suitably provide such systems, a solution needs to be uncovered to solve the various problems hitherto identified. This section outlines the requirements of heterogeneous rule engines and seeks to compare solutions drawn from related domains that can be adapted for these requirements.

5.3.1 Metadata Model for Discerning Heterogeneous Clients' Data

As discussed in previous sections, rule-based systems face challenges when deployed to heterogeneous contexts due to the existence of an all-inclusive global fact base. Current

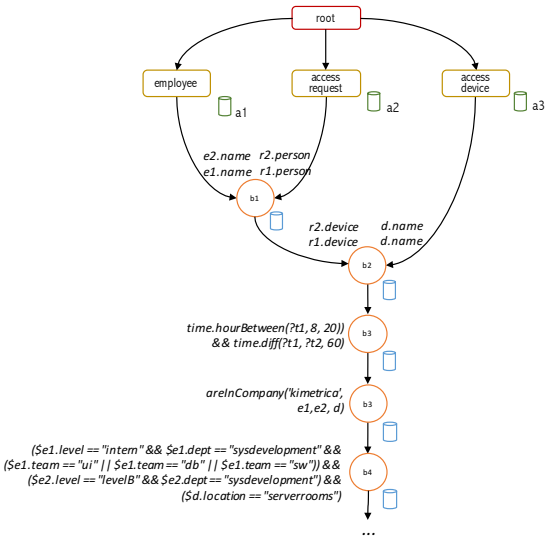


Figure 5.8: Rete graph for InternServerRoomAccess rule – The additional constraints in the test expression add excessive computations during rule execution.

approaches force rule designers to infuse logic to discriminate between the clients together with the normal logic to be captured in rules. At the same time the client facts have to be manually annotated in order to conform to the discrimination logic in the rule, e.g., each facts from an employee in the *Safari* company would be tagged with the information about the employee's company, department, team, etc. The same applies to rules uploaded by clients to the server.

The solution is to provide a uniform and consistent metadata model that supports identifying different clients as event sources. This would enable mapping client events and rules from different contexts to specific clients, e.g., data from clients in different teams at the same department. The underlying rule engine can then use this mapping to be able to ascertain the origins of different rules and events, in order determine the correct execution contexts, e.g., performing join computations with client data from the same department.

Semantically, the proposed approach should ideally utilise metadata to provide an application-agnostic solution that leaves the normal semantics of a rule-based system intact, thus absolving an application programmer from the nuances that would exist without the model. This requirement ranges from providing no model for this forcing manual interventions, to a complete model that automatically tags client metadata concerning events, rules and notifications. A simple example of this is whenever a client creates an event, the model should *automatically* tag the event with the relevant metadata before sending it to the server – instead of the client application programmer handling these specificities. Similarly on the server side, the model should be able to reason about the data model with the least effect on the underlying processing cycle of the rule engine.

Purposely, this requirement is the precursor of the next requirements.

5.3.2 Formalised Model for Grouping Heterogeneous Clients

Previous sections have demonstrated the need for a framework seeking to support RKDAs to manage the heterogeneity of clients. The main reason is the complexity due to having a

variety of clients uploading rules and data into one shared system. A well-known concept for mastering complexities in a heterogeneous setup is by composing entities into higher-level units, often called groups. Groups are semantic and syntactic abstractions of entities that share a common goal or share some commonality.

In order to manage complexity a heterogeneous RBS should therefore provide an extensible grouping model that captures the structures of clients and describes any possible compositions flexibly. Examples in the office scenario are teams which form departments that are all a part of a company hierarchy, but can also participate in other roles. One way to design this model within the framework is via a formalism that is based on the groups, and that builds upon the aforementioned metadata definitions. The formal model can be designed around clients hierarchical structure that lends itself well in progressively describing the relationships between clients. The most significant benefit of such a model is in effectively representing and managing client organisation in the system, useful during evolution. This requirement therefore ranges from having no grouping mechanism exposing rule designers to the above ad-hoc methods, to providing a complete formalised model that can be used to provide flexible ways capture and represent client organisation and compositions within rules.

5.3.3 Execution Model for Selective Computations

Rule engine execution in a heterogeneous setup considers all possible contexts, making the engine spend time performing computations that can be made more efficient (Section 5.2.2). This manner of rule engine execution is undesirable because it impedes performance. The problem is that the rule engine never provides selective matching by imposing any discrimination in rule execution: all rules are under consideration in a given match cycle. Indeed, the diversity of data sources in RDKAs requires further precision in during execution. Note that implementing this using the selection strategy in the `select` phase of rule activation (Section 4.4.3) as a solution is inadequate. This is because the `select` phase never actually depends on matching, but requires special ordering of instantiations – commonly implemented based on the recency of an instantiation. Therefore, the `match` stage would still be negatively affected.

If heterogeneity is represented in the rule engine, it can then use this knowledge to natively encode it internally in its execution rather than being infused in normal rule logic. Thus, *an extensible formalism modelled around metadata in client events and grouping models can be exploited to perform the selective execution of the rule engine, by discriminating or partitioning the data residing in the heterogeneous system.* The discriminating computations are those that try to find consistent bindings for the intended data sources to avoid unintended activations in the engine with data from unwanted sources. The implication here is that the internal structures of clients should be reflected in the runtime in order for it to efficiently process the requests within the confines of clients' configuration. The scale for this requirement ranges from providing no specialised execution, to having a model in which the rule engine can utilise at runtime to perform this data partitioning efficiently. Implementing these proposals will result in the rule engine performing selective execution of rules thereby, in a number of cases, reducing the amount of computations performed by the engine during its execution cycle. This conjuncture will be supported by the evaluations presented in Chapter 8.

5.3.4 Flexible Model for Notification Semantics

As mentioned in Chapter 2, RKDAs usually employ the event-based communication model to disseminate information as it can support a high number of clients in one instance. In a rule-based system, notifications can be sourced from the server as a consequence of rule activations. Here, a notification is a data item that reifies an event from a rule activation. The notification carries the data that accompanies the activation, but in RKDAs may also contain additional metadata such as the time of activation and the creator of the rule or the event source. One issue that arises that is exclusive to heterogenous configurations is *who to notify*, or precisely, which client(s) should receive the notification of a rule activation. This should also be determined in an efficient manner.

In the default scenario, the user that added the rule receives the notification. However, *the composition aspect can be fully embraced to group sets of clients (that share some common goal) as the basis of engineering notification contexts.* The aim of such composition is to specify boundaries for notification delivery: it semantically defines how to distribute notifications of a rule activation from the server. For the office complex, these can be drawn directly from the company organisational hierarchies presented in Figure 5.2: but in most situations they are application-specific. The boundaries should be able to be specified using a clearly-defined notification semantics, e.g., allowing or restricting propagation of notifications to related components. Of course, the ideas discussed here can reuse the extensible formalism that captures client structures as specified in previous requirements.

In most current systems the programmer is left to implement such capabilities manually on their own. This approach is undesirably static and impedes system evolution. Such semantics should be exposed as syntactical abstractions and processed at the framework level, thus promoting flexibility for notifications. Hence, for this requirement, the least desirable is leaving programmers to implement them on their own, but most desirable is to provide flexible semantics for specifying notifications. With a flexible notification model, the framework can efficiently optimise providing functionalities such as decoupling of notification channels, scheduling of server resources for piggybacked notifications, etc.

The remainder of this chapter explores how much work has been done regarding these issues in similar or related domains.

5.4 Heterogeneity in existing RBS: Related Work

The previous sections identified issues and limitations that make current rule-based systems difficult to control and maintain when deployed for heterogeneous RKDAs. This section delves into research that can be used to tackle the some of limitations of heterogeneous systems identified in this chapter. The main focus is on rule-based systems, but the discussion is extended to other related areas.

5.4.1 Decomposition in Rule Engines

In rule engines, execution begins with existing facts, deriving intermediate knowledge by applying appropriate rules without discriminating any sources. In fact, modern Rete-based RBSes have sought some kind of modularisation or decomposition of their rule base, providing ways in which rule designers can partition larger rulesets. These solutions are drawn from the concept of an analogous *rulebook*. Similar to a folder that can hold a number of files inside it, rule books can encompass multiple rules in their structure and use them to

modularise their rulebase. This section discusses the merits and demerits of these solutions with regards to the requirements for heterogeneous rule engines.

EntryPoints in JBoss Drools

The JBoss Drools engine was introduced in Section 3.3.2 and is based on the Rete algorithm. The section discussed how the Drools engine can be embedded onto a server as a back-end processing component that will process requests that are forwarded from the main server component. Citing this, the Drools research team observed that the engine will inevitably deal with event streams that are of of high volume and require correlation. Such event streams are expected to receive inputs from multiple *entry points*. More importantly, the event streams can be heterogeneous with different types of events. Entry points can be thought of a particular pipe where events from a source flood into the system. The system in this case is the rule engine and it can handle multiple pipes at any given time. Facts derived from event sources never lose their identity when asserted through an entry point. This in effect means that a fact inserted through one entry point will not match a pattern specifying another entry point.

Internally, all event streams by default use the `DEFAULT` entry point in Drools. New entry points are declared by implicitly referencing them in rule conditions. An example for the security scenario is shown in Listing 5.6, which is modified from the actual *InternAccess* rule from Listing 3.13. The example implicitly creates an entry point for the facts from the *Kimetrica* company to be selectively picked during matching. When an access request is made, the packaged fact is directly inserted into a named entry point `kimetrica` in the rule conditions. This is shown in Listing 5.7.

Listing 5.6: InternAccess rule in Drools - with Entry Points

```
rule "InternAccess"
 when
   $e: Employee(level == "intern")) from entry—point "kimetrica"
   $r: AccessRequest(badge == $e.badge)) from entry—point "kimetrica"
   $a: AccessDevice(id == $r.device, location == "cubicle", $r.time <= 20, $r.time > 8)) from
       ↪ entry—point "kimetrica"
 then
  insert(new AccessReply($r.id, $a.device, true)
```

Listing 5.7: Inserting Event into EntryPoint in Drools

```
WorkingMemoryEntryPoint accessRequestEntryPoint = session.getWorkingMemoryEntryPoint("kimetrica");
accessRequestEntryPoint.insert(newAccessRequest);
```

We evaluate this approach according to the requirements.

- *Metadata Model:* Drools does not offer a decoupled metadata model that will discern data from particular clients. Even though entry points are declared implicitly in rules, they require the programmer to directly reference them whenever they want to perform data updates in the rule engine. Further, programming updates to data objects using entry points forces the programmer to intertwine data separation logic with the normal client application logic.

- *Grouping Model:* Entry points in Drools present rather simplistic semantics for distinguishing event sources from incoming data. They do not offer any formalised model for more advanced issues such as addressing multiple entry-points as a single composable unit or relationships between them common in heterogeneous contexts.

- *Execution Model:* Entry points in Drools are perhaps the closest solution to the problem of distinguishing event data. During rule compilation, entry points in rules will be identified by the compiler as having separate event sources which internally forces it to create separate alpha nodes when building the related conditions. They can thus be used to partition the fact base depending on the nature of facts and provide a way to discriminate different facts in the fact base. When building the graph, the Drools compiler uses the default root `ReteNode` to create different `EntryPointNodes` as its children according to the entry points identified in the rules, and continues building the graph in the usual way. This internally results in as many graphs as there are entry points. Even so, the research team simply states in [Bro09] with little detail that in some unspecified cases they can reduce cross products that degrade the rule engine performance during pattern-matching.

- *Notification Model:* The entry point functionality is provided by the Drools `STREAM` streaming mode which provides special semantics for 'streamed' events that may trigger rule activations. Even so, the engine offers no notification model as a response mechanism. The programmer needs to manually implement any notification functionality if a rule of interest is activated as a result of incoming events.

Peers in Jess

Jess is a rule engine also based on Rete and was discussed in Section 3.3.2. It is a hybrid engine where the engine's working memory can contain *shadow facts* that are essentially Java objects. The engine is similar to Drools in that it is Java-based and can provide reasoning to Web servers as a backend tool. It however does not support the streaming semantics provide by Drools for specialised event processing.

Multiple instances of the rule engine in Jess can be initialised by spawning several `Jess.Rete` objects as *independent* Rete engines [Fri03]. With the multiple instances approach each `Jess.Rete` instance will execute in its own thread and will contain its own working memory, rule base, etc. In some cases, creating separate engines with their own execution environment is undesirable from a performance point of view due to duplication and redundancy. Jess therefore provides an alternative method through the use *peering* of rule engines, designed to be used in the scenario of pools of Web applications.

Conceptually, peers are an evolution of multiple `Jess.Rete` instances. One 'initial' Rete engine instance is created and rules added to it, creating the compiled Rete graph. Thereafter, multiple independent peers can be created which will share the compiled rules and templates, but each peer contains its own enclosed working memory, execution context and agenda. All peers share a similar rule set: changes to the rule set by any of the instances will thus be reflected on the other peers. Listing 5.8 shows how peers in Jess can be created. The code creates two peers from an initial Rete instance in lines 11 and 13. The peers share the same *intern_access* rule defined back in Listing 3.10, which is loaded in line 8. Remember that the engines will halt after matching the data collected from the `database`, (line 16) and to enable reactive functionality one has to write the code as illustrated back in Listing 3.12.

Listing 5.8: Creating Multiple Peers in Jess

```
1 import jess.*;
2 public class SecurityMonitor {
3    public static void main(String[] argv) throws JessException {
4       /* "initial" engine */
5       Rete engine = new Rete();
```

```
6
7       // Add intern_access_rule
8       engine.batch("intern_access_rule.clp");
9
10      // Create a peer of the initial engine
11      Rete peer = engine.createPeer();
12      // Second peer
13      Rete peer2 = engine.createPeer();
14
15      // Load the catalog data into working memory
16      database = connectToDataSource();
17      peer1.addAll(database.getItems());
18      peer2.addAll(database.getItems());
19
20      // Run the original engine;
21      engine.run();
22
23      // Run the peers
24      peer1.run();
25      peer2.run();
26    }
27 }
```

- *Metadata Model:* Jess does not provide a metadata model that will effectively manage the data from different sources. Creating instances and peers is done in alongside application code, and the developer should always specify which data will be added to which peer. The implementation of the separation of instances therefore needs to be intertwined with application logic.

- *Grouping Model:* The Jess engine explicitly exposes all peers as conceptually identical entities. So even though peers are instantiated from the initial Rete engine, Jess offers no mechanisms for managing peers that would promote advantages such as addressing multiple peers as a single abstraction in order to share common knowledge.

- *Execution Model:* The multiple instances approach of Jess is in reality similar to the independent rulebook concept. The method creates separate Rete engines each with its own components. Jess promotes the use of peers, a better approach for situations that contain large rule sets and may need to share resources. As was mentioned, the sharing of resources is however limited to rulesets and templates, and excludes the working memory and the agenda. Furthermore, to exploit community knowledge, it can be difficult for the programmer to ascertain which data will go to which peer.

- *Notification Model:* The Jess engine supports Web server extensions, and the peer system can be used to create pools of instances that can serve client requests in a request/reply fashion. It however does not expose functionality for managing notifications for responding to clients in the event of a rule activation. Therefore, even though activations from peers are distinguishable, it is up to the programmer to manually handle any response mechanism that they require.

Rule Modules in CLIPS

The CLIPS engine was introduced in Section 3.3.2 and contains two flavours, the pure CLIPS and the hybrid object-based CLIPS COOL. This section focuses on its modules for supporting heterogeneity. The engine does not provide streaming or reactive execution

natively, but as discussed earlier it can be coerced into continuous execution using incessant facts.

CLIPS provides modular management for larger rule bases through the use of *rule modules*. A rule module contains a set of rules that can be grouped together to leverage explicit control by restricting the access of the enclosed rules by other modules. Modules can therefore be used by rules to control execution. By limiting access to rules, a module functions in the same way as a rule book, allowing facts and rules to be only visible to the module. Each module has its own Rete graph and agenda for its rules.

Similar to Drools, normal engine execution actually runs in the default `MAIN` module. Custom modules can be defined using the `defmodule` construct. When the CLIPS engine is issued a `run` command, the module with the current focus is executed. Focus can be specified either by the `(set-current-module <module-name>)` command or by using `(focus <module-name>)` in the right-hand side of a rule, which shifts rule engine execution to a particular module context. At any one time, rule execution is performed in a particular module, until focus shifts to another module or the engine exits.

Listing 5.9 illustrates how the rule `intern_access` can be implemented using CLIPS module-based constructs. Lines 1 and 10 use two modules for two companies to define their different rules. The modules internally construct two separate Rete graphs. Similar to the situation with peers in Jess, two different facts would need to be inserted in both engines.

Listing 5.9: Intern Access rule in CLIPS

```
1  (defrule kimetrica::intern_access
2     (employee (name ?nam) (badge ?ibadge) (level "intern"))
3     (accessreq (id ?reqid) (badge ?ibadge) (time ?t) (device ?dev))
4     (accessdevice (id ?dev) (location "cubicle"))
5     (test (and (< ?t 20) (>= ?t 8)))
6    =>
7     (assert (accessrep (reqid ?reqid) (device ?dev) (allowed true)))
8     (printout t "Allowed access for Kimetrica " ?nam " on device " ?dev crlf)
9  )
10 (defrule touring::intern_access
11    (employee (name ?ename) (badge ?b) (level "intern"))
12    (accessreq (id ?req) (badge ?b) (time ?t) (device ?device))
13    (accessdevice (id ?device) (location "cubicle"))
14    (test (and (< ?t 19) (>= ?t 7)))
15   =>
16    (assert (accessrep (reqid ?req) (device ?device) (allowed true)))
17    (printout t "An intern in Touring " ?ename" accessed device " ?device crlf)
18 )
```

- *Metadata Model:* CLIPS exposes a simple, generic model where facts can be tagged to a particular rule module but only during fact definitions. In this manner, the event source of a data item in the graph within the engine can be determined and can be used to partition data in the working memory. The programmer needs to specify the module when defining the fact, as with the other approaches.

- *Grouping Model:* CLIPS rule modules provide modular abstractions for grouping related rules. However, the CLIPS engine does not provide any abstractions for managing rule modules that could be used as model for representing the various client structures. Describing physical and logical client relationships in a heterogeneous environment using CLIPS therefore requires a substantial amount of effort to model using CLIPS modules.

- *Execution Model:* Rule modules are a direct mapping to the rulebook concept because they contain separate, independent Rete graphs with its own execution context and

Table 5.9: Evaluative comparison of rule-based systems for supporting heterogeneity

Rule-based Systems	Metadata Model	Grouping Model	Execution Model	Notification Model
Entry Points	★☆☆☆☆	★☆☆☆☆	★★★☆☆	★½☆☆☆
Peers	★☆☆☆☆	★☆☆☆☆	★★☆☆☆	★☆☆☆☆
Rule Modules	★★☆☆☆	★★★½☆	★★☆☆☆	★☆☆☆☆

agenda. When compared to Drools' entry points, they are limiting because entry points can be used to determine event sources in conditions unlike in the module approach. This can be seen when comparing Listing 5.9 with 5.8, where different entry points can be constrained at condition-level but modules can only be attached at rule-level. Furthermore, constructs are available in rule modules that allow programmers to 'hijack' the rule engine's focus to execute a named module. This unnecessarily complicates programming heterogeneous RKDAs, because rule designers are required to orchestrate rule interactions. Absolving the rule engine from its own control of execution goes against the motivation of using a rule engine in the first place: in problem domains where there are no obvious algorithmic solutions to be found and functional specification is difficult.

- *Notification Model:* CLIPS and its invariant CLIPS COOL do not support notifications of rule activations out-of-the-box. Even though an activation from a particular rule module can be detected (as each has its own agenda), notifications to individual clients need to be programmed manually. The main reason is that the engines were not envisioned to work in a heterogenous environment and activations were intended for single-user situations only.

5.4.2 Overview: Decomposition in Rule Engines

The approaches described are related to the forward-chaining rule-based systems that this dissertation targets – and the techniques they expose that can be used to solve the identified problems. Table 5.9 summarises the approaches described in terms of the aforementioned requirements. It follows the same ranking system as the evaluation table 3.2 in Section 3.4.

The table shows discrepancies of rule-based systems for support of heterogeneous RKDAs, particularly in providing metadata and notification models. We therefore investigate support of these requirements in similar approaches found in related research areas. The areas to be examined will follow the general directions presented in the related work Chapter 3. Because these approaches do not suitably fit within the scope of rule-based systems as designated in the chapter, the discussion will only focus on their specific concepts with regards to heterogeneity.

5.4.3 Schema Sharing in Multi-tenant Databases

Drawing a connection with active databases discussed in Section 3.3.1, this section explores work that explicitly abstracts heterogeneous tenants in multi-tenant databases. In classic database management systems, several logical databases were *manually* mapped onto one physical database to enforce heterogeneity (the initial design choice of multi-tenant Cloud

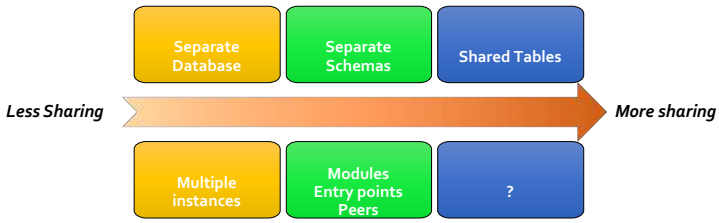


Figure 5.10: Parallels between approaches for multi-tenancy in databases and heterogeneity in rule-based systems.

service provider Salesforce). This process was found to complicate application development without supporting multi-tenancy from within the DBMS. The use of techniques such as pivot tables can help overcome these complexities, but offered limited support for features like query optimisation and indexing [Aul+11].

A multi-tenant database applies multi-tenancy concepts at the database layer by consolidating multiple tenants into a shared data-tier resource [J+07]. They are analogous to heterogeneous rule-based systems: whereas multi-tenant databases are an application of multi-tenancy at a static database layer, heterogeneous RBS construct client rules and reactively process data from multiple sources within a shared rule engine.

Multi-tenant databases are based on three main distinctions [HDX12]. In *separate database* each tenant has its own set of data that is isolated from the data for the rest of the tenants. This data model makes it easy to extend the database to meet individual needs of a particular client, but has a higher resource cost per tenant. In *shared database separate schema* several tenants can share the same database, but each tenant has its own schema that describes its own set of tables. In *shared schema/tables*, the schema is created once and different tenants are mapped directly onto it. This method has the lowest cost and can host the largest number of tenants per server, but has a higher complexity to implement data distinctions. More recent advances have proposed the use of extension tables that reify the concept of a tenant to the database layer, where the database engine can associate each request to a tenant and forwards it to the appropriate table storage.

Comparisons can be drawn between the three distinctions of multi-tenant DBMS with the approaches for rule-based systems in the previous section. A graphical representation is shown in Figure 5.10. The separate database approach is similar to the multiple instances approach – spawning several instances of rule engines corresponds to having separate databases, one for each tenant. The separate schema approach is comparable to most of the approaches presented earlier. Having a number of peers, entry points or modules to share one fact base but having several Rete graphs, agendas, intermediate memories and execution contexts.

We observe that although the shared schema/tables approach is the closest to the primary focus of this thesis, *it has no definite parallel with the aforementioned approaches for current rule engines*, as indicated in Figure 5.10.

Flavours of the presented approaches do try to come close to sharing of a single Rete instance. One example is entry points in Drools; entry points are however distinct and are, in essence, internally represented as separate Rete graphs albeit with a shared root node, and thus map to the separate schemas in multi-tenant DBMS.

It is generally accepted that support for multitenancy requires a structure for client metadata management,e.g, in extension tables [J+07]. In addition, because these databases are not inherently intended for reactive CEP, they do not particularly have avenues for

flexible client notifications, e.g., to a subset of tenants.

5.4.4 Visibility in Event-based Systems

Likewise, this section draws a connection to the event stream processing systems identified in Section 3.1 when applied to heterogeneity. For example, they have been used in the software-defined networking domain to provide information-centric networks by capturing cross-domain events in packets [Nun+14].

Conceptually, event-based systems view events received in terms of the origin, or the *event source*. With this abstraction, distributed event-based applications can be programmed at a higher level in a heterogeneous context, since the loose coupling of participating components makes it possible to design more structurally-complex systems. However, different event sources can be the cause of different event notifications, the primary communication mechanism between components. Thus the problem of *dialect* [Bat+98], i.e., of interpreting events from different event sources, needs to be tackled.

Several areas of research have tried to find ways to solve the dialect problem. The closest approach is the work about event notifications in the system `REBECA` [Fie+02] which aims to provide abstractions for structuring event-based systems. The work proposes a way to solve the dialect problem by limiting the visibility of notifications in bundled consumers. Only the components that are intended to receive notifications (i.e. the intended consumers) are able to 'see' notifications. The use of a hierarchical architecture sets this approach apart from similar efforts. To implement this, the proposed system provides a *broker overlay* that contains a set of cooperating components arranged in topologies, e.g., based on logical or administrative boundaries. Each client process contains a *local event broker* which functions similar to a proxy. The broker marshals outgoing client requests to other brokers or producers. To receive notifications, the broker contains a set of visibility roots, which is simply a list of external components that this client can receive notifications from. In the paper, the authors compare their model with other related work in event-based notification and filtering with a set of requirements, one being support for heterogeneity. The technologies compared include subject-based addressing in event zones in `READY`[GKP99], bridges for structuring JavaBeans objects [UM99] and Linda-based ActorSpaces to limit message distribution [CA94]. They conclude that unlike all the other approaches, the use of a broker architecture with visibility roots is the most suitable for managing event notifications of components organised in different multi-level compositions.

The `REBECA` system shows some differences with the identified requirements for supporting heterogeneity. The use of brokers is a form of client-side filtering where brokers can receive *all notifications* and decide which specific notifications the client can 'see' via metadata. *The semantics of placing restrictions in the actual matching process upon receiving events is not suitably discussed* (which, in comparison, is the essential part of any rule engine) and the work purely focuses on notification semantics. Furthermore, the broker architecture presupposes the existence of some form of an overlay network which requires a complex distributed management scheme that is not the primary focus of this dissertation. Nevertheless, the idea of visibility roots is a promising direction and the concept can be used as a foundation for managing heterogeneous clients.

5.5 Chapter Summary

Section 5.2 identified the main concern in the support of RKDAs is essentially harnessing community knowledge and enforcing reentrancy in the rule engine. From this, we identified

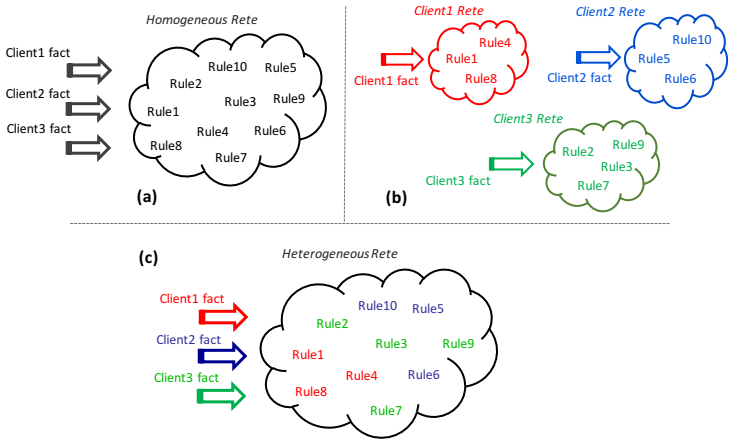


Figure 5.11: Conceptualizing a multitenant inference engine showing (a) naïve, (b) module-based and (c) heterogeneous engine approaches [KRD17b].

four key requirements that encompass the support of heterogeneity in rule-based systems. Various existing research were then discussed to discover possible avenues that can be exploited. Table 5.12 presents a summarised result of the evaluation in this respect.

From the existing research we observe that there can be three conceptual directions to take when designing a heterogeneous rule engine. The first is the naïve shared approach that causes unintended activations in rules uploaded by different clients (Figure 5.11a). The second consists of the idea of rulebooks or modules that promote the complete isolation of the rules, thereby duplicating all the rule engine instances (Figure 5.11b). The peering and entry-point approaches absolve this somewhat, but in reality do not employ sharing – instead they separate the intermediate memories and activation queues entirely in a running rule engine. The first two approaches increase rule complexity and quickly become tedious in clients with more complex internal structures like multiple departmental levels. They have a negative impact in the underlying Rete graph: additional nodes are created and more computations are required even when they could be avoided. They further pollute the logical intent of the rule designer, e.g., by adding conditions that need to enforce discrimination within the rules of all clients.

Heterogeneous rule-based systems for RKDAs require features that solve reentrancy problems and promote community knowledge as the applications they support grow in size and complexity. The following Chapter 6 proposes a model that supports RKDAs by enabling any Rete graph in shared rule engines to purely handle multiple inference states simultaneously for different sets of client rules (as illustrated in Figure 5.11c). This promotes both reentrant execution and, by extension, community knowledge by exploiting the organisation and relationships between heterogeneous clients.

Table 5.12: Evaluative comparison of RBS and other systems for supporting heterogeneity

	Metadata Model	Grouping Model	Execution Model	Notification Model
Rule-based Systems				
Entry Points	★☆☆☆☆	★☆☆☆☆	★★★☆☆	★★☆☆☆
Peers	★☆☆☆☆	★☆☆☆☆	★★☆☆☆	★☆☆☆☆
Rule Modules	★★☆☆☆	★★★★☆	★★☆☆☆	★☆☆☆☆
Multitenant Databases				
Shared schema/tables	★★★★☆	☆☆☆☆☆	★★★★☆	☆☆☆☆☆
Event-based Systems				
Visibility	★★☆☆☆	★★★☆☆	☆☆☆☆☆	★★★★☆

6

The Serena^s Scoped Rule Language

Language shapes the way we think, and determines what we can think about.

Benjamin Lee Whorf, *Language, Thought, and Reality*, 1956

The previous chapter discussed how classical rule-based systems experience challenges in shared heterogeneous systems, particularly when aiming to exploit the concept of community knowledge. This chapter presents the *Serena^s Rule Language*, a scope-based rule language intended for RKDAs. The language provides constructs that programmers of RKDAs can use to capture community knowledge by distinguishing client data using *scopes*. Section 6.1 outlines the foundations of the language. Next, the semantics of the language is presented starting from Section 6.2. The architecture that supports the language is then presented in Section 6.3. In the sections, examples of scoped rules are illustrated using the office complex scenario. The chapter concludes by presenting notification scopes in the language, used to provide selective responses to groups of clients[1].

6.1 Foundations of the Scope-based Rule Language

Current rule-based languages lack abstractions that allow rule creators to effectively organise shared knowledge. We present the solution that this dissertation proposes, through the use of *scopes*.

6.1.1 Design Factors of Serena^s Scope-based Language

In this section we introduce the inspiration behind scope-based reasoning and proceed to explain the design factors of introducing scopes in rules.

[1]Some observations described in this chapter have been published as [KRD17b].

Similarities with Temporal Reasoning in Rules

The use of scopes in rules follows a similar idea as in temporal reasoning in rules. Earlier rules lacked formal mechanisms for representing temporal semantics and were instead fused with normal rule logic [CM97]. Take the following rule, which checks the order arrival of a pizza and awards a 'gold' customer a free pizza if the time exceeds 10 minutes.

```
IF order.time - delivery.time > 600000
 AND order.id == delivery.id
 AND order.customerrank == "gold"
THEN delivery.status = "free"
```

In the first line, the temporal constraint explicitly checks the time that the delivery event occurred in the normal logic of the rule. In this case, the expression is simple, but more extensive checks make programming such logic more complex and error prone.

As a solution the work by Allen in [All83] observed that *temporal constraints in rules can be programmed without the explicit mention of time in facts, but with syntax that abstracts the temporal relationships between them.* The rule language benefits by separating the temporal logic from the rule logic and more complex concepts can be added, like interval-based time semantics [MK93; WBG08]. The next example shows the same rule with this type of syntax.

```
IF order.id == delivery.id
 AND order.customerrank == "gold"
 WITH delivery AFTER(10M) order
THEN delivery.status = "free"
```

The Notion of Scopes

Similar to earlier problems about reasoning of time in rules, current methods of data discrimination force rule designers to procedurally embed structural logic (i.e., organisation of clients) within the normal logic of the rule. This problem was illustrated in the *InternServerRoomAccess* rule of Listing 5.5 back in Section 5.2.3. The examples showed that explicitly distinguishing between sources in the normal rule logic makes it difficult to control and maintain rules in heterogeneous RKDAs.

> As a solution, Serena^s uses scope-based reasoning in rules to separate client discrimination logic from the normal rule logic. This is because structural constraints in rules can be programmed without the explicit mention of fact sources, but with formal syntax that abstracts their organisation and the structural relationships between them.

The approach taken by Serena^s is to embrace physical or logical *groupings* of clients and their relationships.

Heterogeneous Groupings RKDAs typically contain some application-specific structuring or organisation of clients (Section 5.3). Examples of groupings include research groups in universities, departments in companies, branches in conglomerate businesses, hobby categories in forums, area zones when monitoring distributed sensor networks, user lists in Twitter, and *campaigns* in crowdsourcing activities [Zam+14]. Grouping clients is a powerful concept because it can be effectively exploited for modularity in heterogeneous

rule engines. *This is the approach taken by Serena^s to orchestrate rules to discriminate or distinguish between instances of data from different clients.*

In the office complex scenario, the logical structures for clients shown in Figure 5.2 can be modelled into groups and subgroups, where each group is represented by a node in the figure. To explain the concepts, this section proceeds with a limited number of groupings shown in Figure 6.1[2]. To model the groups, a client with an *administrator* role whose main responsibility is to compose a set of groupings, or a *group hierarchy* of clients and deploy them to the server[3]. The group hierarchy defines how the clients are organised. It is defined via an adjacency list with the groups specified as a list of couples (a, b) if b is a parent of a. For example for Figure 6.1c,

$$[(directors, lev1), (securityteam, lev1), (supervisors, lev2), (employees, lev3), \ldots]$$

Heterogeneous Relationships Heterogeneous groupings are usually modelled according to physical or logical relationships among the clients – project teams can belong to (sub) departments, hobbies can be categorised into hierarchies of interest groups and sensor area zones can be contained in levels of administrative units. Section 5.2.3 gave examples such as locality as the main inspiration behind the relationships, but they can generally be attributed to application-specific semantics. These relationships are significant to exploit community knowledge. The example of the office complex groups illustrate a simple but common relationship. The *Kimetrica* company has a systems development department composed of user interface, database and programming teams. Users in the `UI` team can be said to additionally belong to the systems department teams and the company *Kimetrica* at large.

Serena^s appropriates the term scopes to represent the common relationships between groups in a hierarchy and designates that as a scope hierarchy. In Serena^s scopes define (a series of) edges traversing one or more nodes in the modelled group hierarchy. Scopes are a significant control structure for heterogeneous rule-based languages because they specify a selection of which rules that the inferencer will use at a particular time during execution.

In the next Chapter 7 we discuss the effect of using the concepts of groups and scopes on the execution of the rule engine. In the upcoming sections we present the scope-based language of Serena^s.

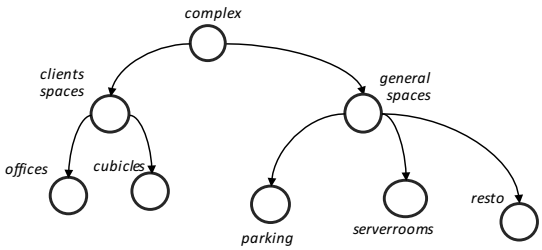
6.2 Scoped Rules in Serena^s

Serena^s employs the use of scopes to discriminate between data from different heterogeneous clients. This section outlines how scoping is used in the definition of scoped rules.

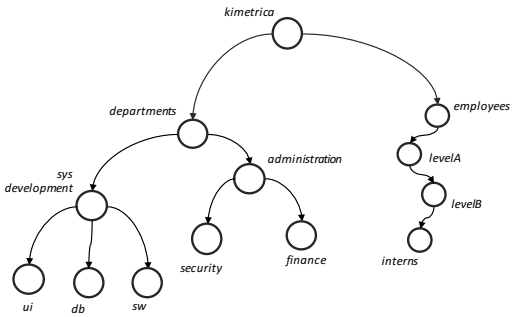
> Serena^s goes further than the concepts of *isolation* in multi-tenant, heterogeneous contexts [Guo+07; J+07] by providing ways in which clients can further exploit collective or community knowledge according to how they are organised.

[2]The evaluation in Chapter 8 is done on larger groups of clients for a similar scenario

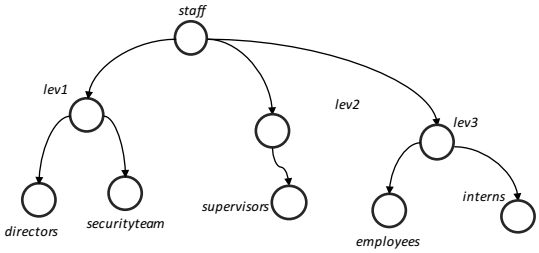
[3]The administrator role is borrowed from similar roles in the distributed event-processing domain [MFP06]. In the multitenant domain such roles are designated *tenant managers* [KKH11].



(a) Office complex groups



(b) Kimetrica consultancy groups



(c) Safari tours company groups

Figure 6.1: Structural groups of companies and physical locations

6.2.1 Defining Scoped Rules

Serena provides scope-based reasoning in rules by extending the normal rule syntax with *scope-based definitions* which specify structural constraints on the groups and the relationships between them, simply called *scopes*. Scopes are thus abstractions that assist clients in heterogeneous applications to define expressions that specify what data is applicable their rules.

The grammar of the complete Serena^s Rule Language is outlined in Figure 6.3 that contains extensions to the syntax shown back in Figure 4.1. The additions to the basic SRL are in the *scopes* and *notifies* constructs:

> *Scopes* contain scope expressions, which can be grouped by the use of parentheses. Scope expressions are always binary and can be expressed using pre-defined scope symbols, explained later in this section. They can be combined with logical operators | and &.
>
> *Notifies* contain notify expressions with unary operators used to define event notification semantics, i.e., the expressions specify the specific clients to notify once a rule is activated. Notify expressions can use scope symbols in the same way as scope expressions.

The supported scopes are illustrated in the next section.

6.2.2 Overview of Supported Scopes

In Serena^s, the scoped operations that the framework supports are presented in Figure 6.2. To clarify the differences, the operations are illustrated based on the client groups from the office complex example shown in Figure 6.1. A description of each scope follows.

- ***subgroupof***: This scope includes only the data items added by the group or any of its subgroups in the scope. This scope is ideal for a departmental rule for *sys development* that will only apply to members of that department or sub-departments (*iu*, *db* and *sw*). See Figure 6.2b.

  The dual of *subgroupof* is ***supergroupof***.

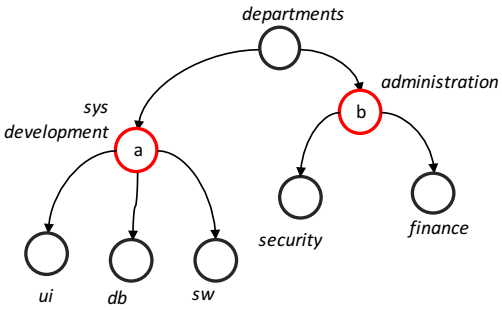
- ***private***: The private scope will exclusively capture data items from the specified group and none else (not even its subgroups or parent group). This scope is well suited for data that applies to an exact group, like in Figure 6.2c where we can specifically target data from head employees in the *levelB* group and not high-level directors in *levelA* or low-level *interns*.

- ***peerof***: Only data items that originate from specific peers will be considered in this scope. The peers include groups that are at the same level in the hierarchy. For instance the security head would want to create a rule with this scope that applies to devices stationed at the entrances of the *sys development* and *administration* departments, Figure 6.2a.

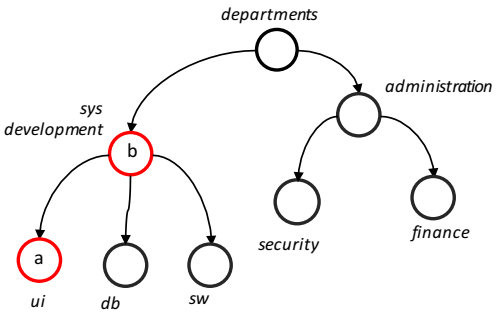
- ***visibleto***: In this scope Serena^s only captures data items from clients in groups that share the same ancestor in the hierarchy. An example is capturing data that pertains to *security* members together with all other *ui* members (Figure 6.2d).

- ***super***: In this scope the runtime captures data items from all defined groups in the hierarchy. Data will therefore be captured from all clients belonging to all the defined groups.

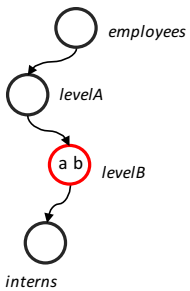
The syntax of using the above scopes for programming rules declaratively is explained in the next section.



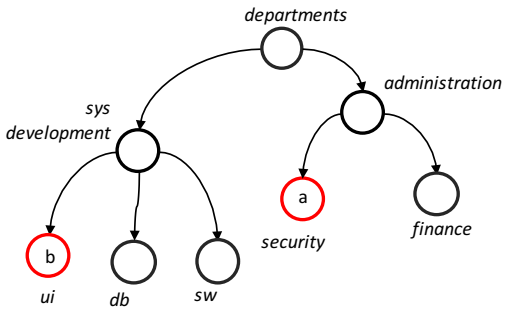
(a) a peerof b



(b) a subgroupof b, b supergroupof a



(c) a private b



(d) a visibleto b

Figure 6.2: Scopes supported in Serena – The scopes shown are in relation to the group hierarchy from Figure 6.1

$$
\begin{array}{lcl}
P \in SRL & ::= & (t^* \mid r^* \mid p^*)^+ \\
t \in templates & ::= & \text{template } t_n\ [c_m]\ t_s^* \\
r \in rules & ::= & \text{rule } r_n\ c^+\ a^*\ [s^*]\ [n^*] \\
c \in conditions & ::= & c_e \mid t_e \mid b \mid nc_e \\
c_e \in cond\text{-}elms & ::= & \text{type } t_n\ slc^* \\
slc \in slot\text{-}constr & ::= & s_n\ (== \mid \neq)\ s_v \\
t_e \in test\text{-}conds & ::= & e\ \ (< \mid \leq \mid = \mid \geq)\ e \\
 & & \mid e \\
p \in plugins & ::= & \text{plugin } p_n\ f^* \\
f \in functions & ::= & f_n\ e^* \\
e \in expressions & ::= & [p_n.]f_n\ [e] \\
 & & \mid var \mid e\ \delta\ e \\
nc_e \in ncond\text{-}elms & ::= & \text{not } c \\
b \in binds & ::= & var\ \leftarrow\ c_e \\
t_s \in temp\text{-}slot & ::= & s_n\ [\gamma] \\
\delta \in operators & ::= & +\ \mid\ -\ \mid\ *\ \mid\ /\ \mid\ \%\ \mid \ldots \\
s_v \in values & ::= & num \mid string \mid var \\
\gamma \in types & ::= & int \mid string \mid bool \\
a \in actions & ::= & \text{assert } t_n\ [s_n \Rightarrow e] \\
 & & \mid \text{retract } var \\
 & & \mid \text{modify } var \text{ with } [s_n \Rightarrow e] \\
 & & \mid \text{call } [p_n.]f_n \\
s \in scopes & ::= & s_e\ s_c\ s_e \\
s_e \in scopeexpr & ::= & s_g\ (\varphi\ \ s_g)^* \\
\varphi \in scopeoperators & ::= & \mid \mid \& \\
s_c \in scopesymbols & ::= & (subgroupof \mid visibleto \mid peerof \\
 & & \mid supergroupof \mid public \mid private \\
 & & \mid lvisibleto \mid lpeerof \mid super)\ s_e \\
s_g \in scopevalues & ::= & symbol \mid var \\
n \in notifies & ::= & (s_c\ s_e^+) \mid s_g+ \\
 & & \\
var & ::= & VariableName \\
c_m & ::= & Comment \\
r_n & ::= & RuleName \\
p_n, f_n & ::= & Plugin,\ FunctionName \\
t_n, s_n & ::= & TemplateName,\ SlotName
\end{array}
$$

Figure 6.3: Compact grammar of the Serena Rule Language with scopes

6.2.3 Examples of Scoped Rules

Scoped rules consist of the usual rules in SRL with an additional scopes construct. The scope extensions are based on a JSON representation of SRL syntax, as was explained in Section 4.2.2. We give examples of scoped rules below, using the office complex scenario.

subgroupof The example of Protocol 4's *InternAccess* rule using scope constraints is shown in Listing 6.1. The protocol specified that interns are allowed access to *Kimetrica*'s cubicle space only between working hours. The rule is similar to that shown in Listing 4.3 of Section 4.4. This time however rule utilises an additional `scopes` section in line 8.

In the scope definition, the bound condition variable `$e` from the condition in line 3 is referenced to check whether the employee that performed the access request (in line 5) is tagged to belong to the group `interns`[4]. This is done using the scope check *subgroupof* that will ascertain that the employee is tagged with the group `interns` or any of its subgroups.

Listing 6.1: InternAccessRule with Scopes

```
1 {rulename: "intern_access",
2  conditions:[
3   {$e: {type:"employee", level: "intern", name:"?name"}},
4   {type:"accessdevice", name: "?dev", location:"cubicle"}},
5   {type:"accessreq", id: "?reqid", person: "?name", device: "?dev", time: "?t"}},
6   {type:"$test", expr:"(time.hourBetween(?t, 8, 20))"}
7  ],
8  scopes:[ "$e subgroupof interns" ],
9  actions:[
10  {assert: {type: "accessrep", reqid:"?reqid", allowed: true, time: "?t"}}
11 ]
12 }
```

private A device can actually be added as a 'client' by mapping it onto the client group hierarchy `cubicles`, leading to the modification of the *InternAccess* rule to that in Listing 6.2. The modified line 8 appends the scope construct *private* that specifies that the device should originate from the specific `cubicles` group of the physical building hierarchy.

Listing 6.2: InternAccessRule with Scopes – Part 2

```
1 {rulename: "intern_access",
2  conditions:[
3   {$e: {type:"employee", level: "intern", name:"?name"}},
4   {$d: {type:"accessdevice", name: "?dev"}},
5   {type:"accessreq", id: "?reqid", person: "?name", device: "?dev", time: "?t" },
6   {type:"$test", expr:"(time.hourBetween(?t, 8, 20))"}
7  ],
8  scopes:[ "$e subgroupof interns", "$d private cubicles"  ],
9  actions:[
10  {assert: {type: "accessrep", reqid:"?reqid", allowed: true, time: "?t"}}
11 ]
12 }
```

[4]The tags are automatically added by Serena^s as specified in Section 6.3

visibleto Related to Protocol 2, the next rule in Listing 6.3 uses the *visibleto* scope in line 9 to capture all accesses made by employee vehicles via an access code on devices only for their own company parking slots.

Listing 6.3: KimetricaAccessRequests rule

```
1 {rulename: "kimetrica_vehicle_request",
2  conditions:[
3   {type:"car", no:"?no", carowner: "?onam"},
4   {$e: {type:"employee", name:"?onam"}},
5   {$d: {type:"accessdevice", name: "?dev"}},
6   {type:"accessreqcode", id: "?reqid", code: "?no", deviceid: "?dev", time:"?t"},,
7   {type:"$test", expr:"(time.hourBetween(?t, 10, 16)) && time.isWeekend(?t)"}
8  ],
9  scopes:[ "$e visibleto $d"  ],
10  /*...*/
11 }
```

peerof To monitor employee accesses at the `administration` and other departments (i.e., at the main entrances of departments) across the company, the *EmployeeDepartmentAccesses* rule below uses the *peerof* scope. To capture departmental accesses by employees at their own departments, the scope expression can add a `"$e subgroupof $d"` check.

Listing 6.4: SeniorFacultyAccess rule

```
1 {rulename: "employee_department_accesses",
2  conditions:[
3   {$s: {type:"employee", name: "?nam"}},
4   {type:"accessreq", id: "?reqid", person: "?name", device: "?dev"},
5   {$d: {type:"accessdevice", id: "?dev"}}.
6  ],
7  scopes: [ "$d peerof administration"],
8  /*...*/
9 }
```

supergroupof For a more involving example, take the *InternServerRoomAccess* rule from Listing 5.5 in Section 5.2.3. The rule implemented Protocol 7 that specified that interns are only allowed into the server rooms when accompanied by a department leader during appropriate hours. The rule implemented using scope-based syntax is shown in Listing 6.5.

Listing 6.5: Scoped rule for accompanied access to server rooms

```
1  {rulename: "intern_serverroom_access",
2  conditions:[
3   {$e1: {type:"employee", name:"?intname"}},
4   {$e2: {type:"employee", name:"?empname"}},
5   {$d: {type:"accessdevice", name: "?dev"}},
6   {type:"accessreq", id: "?reqid1", person: "?empname", device: "?dev", time:"?t1"},
7   {type:"accessreq", id: "?reqid2", person: "?intname", device: "?dev", time:"?t2"},
8   {type:"$test", expr:"(time.hourBetween(?t1, 8, 20)) && time.near(?t1, ?t2)"}
9  ],
10  scopes:[ "$e1 subgroupof interns","$e2 private levelB", "sysdevelopment supergroupof
      ↪ ($e1 & $e2)", "$d subgroupof serverrooms" ],
```

```
actions:[
 {assert: {type: "accessrep", reqid:"?reqid", allowed: true, time: "?t1"}}
]
}
```

The scope tests of the rule specify that the first employee should be an `intern` and the second should specifically be a `levelB` employee (senior departmental employees). The next *supergroupof* scope expression uses the `&` operator for defining multiple scope checks in one expression. In this case the expression specifies that the `sysdevelopment` department should encompass the departments that both employees are a part of by using the *supergroupof* scope constraint. The final scope expression checks to see whether the device is part of the `serverrooms` general group as per the specification of the protocol.

Evaluation. Compare the scoped rules with their classic versions, i.e., the *InternAccessRule* in Listing 4.3 and 6.2 and the *InternServerRoomAccess* rule back in Listing 5.5 with 6.5. In the classic implementations, distinctions between clients and their data sources need to be hard-coded *within the rule logic*, which quickly becomes complex and prone to errors as the number of clients and the relationships between them increase, or when the relationships become complex to enforce using rule semantics. Using scoped rules that logic is separated, the rules are more succinct and provide flexible ways to specify such constraints. Therefore scoped rules are a viable step forward in multi-user contexts requiring rule-based reasoning, and *can further be exploited by the rule engine to make computations faster* (this is the basis of the next Chapter 7). To give an overview of the technologies used to implement scoped rules we present the architecture of the Serena^s scoped framework.

6.3 Serena^s Architecture

6.3.1 Client-server Interaction

The client-server interaction sequence introduced in Section 4.3 remains largely unchanged. The modified interaction process is shown in Figure 6.4. An initial step for adding the parent (or main) groups of the hierarchies starts the process. The parent groups for the office complex are *Kimetrica*, *Safari tours* and *Marketing* and these can be added to the system when registering the companies as tenants in the building, for instance. The client groups are then added to the server, which then builds an efficient internal representation based on M_{ϑ_L} explained later in Section 7.1.4. The client groups are internally prefixed with their parent group, and thus the names of client groups under a parent group (e.g., one company) should be unique. This maps to the concept of a *tenant* in multitenant systems.

Thereafter, clients can at any time connect to the server through the client library as earlier, but now they are required to specify their affinity to the main group and other specific groups (Listing 6.6). Therefore a client always belongs to at least one group in the hierarchy. The Serena client library stores this information internally for subsequent communications with the Event Manager on the server.

Listing 6.6: Serena^s client connecting to the server

```
var securityClient = new SerenaClient(serverip, 'kimetrica', 'security');
```

The scoping mechanism in the client library is implemented via metadata. When the client designs a rule, the client library packages the rule and additionally tags the rule with metadata containing information of the `owner` before being sent to the server. When the

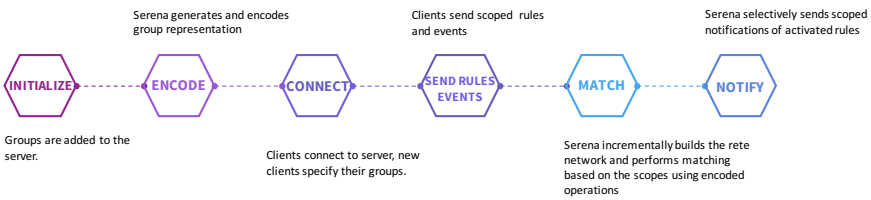


Figure 6.4: Client-server interaction sequence in Serena^s – The grouping structures are added initially, and then clients can specify the groups that they belong to when connecting to the server.

client sends rules and event data, the Serena client library tags the data with the client information stored as metadata when they initially connect to the server. Each data item is thus tagged with its respective `owner`. Data items can further be tagged with the specific `groups` that the data is applicable to according to stored metadata, e.g., a fact added by a client in `sysdevelopment`[5]. The data is then added to the rule engine together with its metadata. The scoping module can access the rule, event data and client metadata to perform its functions. The client library subsequently receives notifications form rule activations on the server and calls the suitable handlers in client code.

6.3.2 Server Architecture

The architecture of the Serena engine illustrated in Section 4.5 is now augmented to support scope-based reasoning in scoped rules. The illustration in Figure 6.5 presents a new component, the Scoping Module that references the inference engine, the event manager and rule/fact bases.

The **Scoping Module** is tasked with implementing and validating scope-based reasoning functionality in the framework. It internally represents the group structures of the clients, builds an efficient scoping mechanism and modifies the matching strategy in the inference engine to perform scope validations in rules. The Event Manager now additionally stores the primary groups that connected clients belong to as metadata for the rule engine. The Scoping Module is then used by the event manager to determine the recipients of notifications from rule activations by informing clients selectively as per the rule definitions and the stored metadata, e.g., the owner of the rule.

6.4 Localised Scopes

In a multiuser or multitenant setup, the scopes *peerof* and *visibleto* can be insufficient to define scoped rules that only capture local data (i.e., in one tenant). For instance, the scope *peerof* once applied to the office complex hierarchy would capture all data from all companies, even if the intention was to capture data only from within one company such as *Kimetrica*. The same concept is applicable to the *visibleto* scope. To solve these problems, Serena^s introduces local scopes that fulfil isolation properties. Remember that as an initial startup step, the parent (or main) groups of each client are first added to the scoping module's internal representation of client hierarchies (Section 6.1.1). Local scopes

[5]More flexible configurations can allow some clients to specify the specific groups that a data item is applicable to.

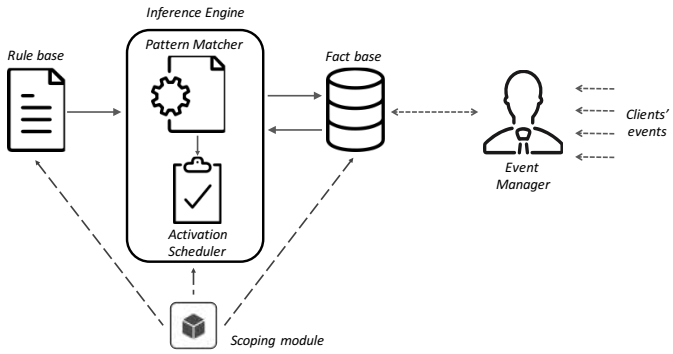


Figure 6.5: Scoped rule engine components – The scoping module interfaces with the rest of the components to implement scope-based reasoning functionality.

are therefore only applicable to a hierarchy that the data is sourced from locally, i.e., those groups that share a common ancestor that is this parent/main group. For example, in Figure 6.1 the main groups are *Kimetrica*, *Safari tours* and *Complex*.

- ***lvisibleto***: In this scope the runtime only captures data from clients in groups that share a common ancestor in the hierarchy (that is not the parent group). This is semantically equivalent to capturing data from one hierarchy within the parent group, as depicted in Figure 6.6d. An example is capturing the data that pertains to a company's *security* members in a meeting with any other employees in *departments*.

- ***lpeerof***: The data items that originate from peers under the same main group will be considered in this scope, and will exclude peers from other hierarchies. For instance a rule with this scope that applies to devices stationed at the entrances of the *sysdevelopment* and *administration* departments, Figure 6.6b.

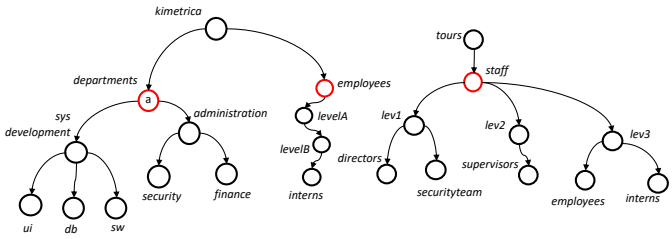
- ***public***: In this scope the runtime captures data from groups under a common parent group in the same hierarchy. The *public* scope therefore captures data within one company (Figure 6.6c). It can be used for rules that validate accesses of employees within their own company.

A side-by-side comparison of the local scopes with the normal scopes is illustrated in Figure 6.6. Note that *public* scope is a localised version of the *super* scope. As an example, the *Kimetrica* security team can use *public* to draft protocols for its members, but the security team of the office complex can draft protocols that are applicable to all its tenants via *super*.

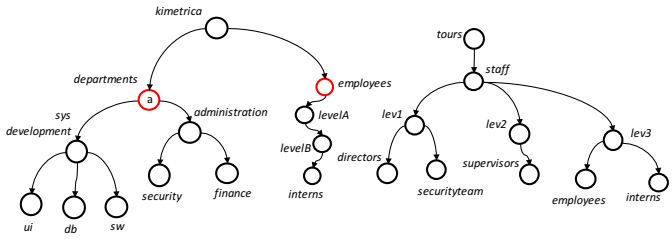
An example of a rule with localised scopes is shown in Listing 6.7. In the rule, simultaneous accesses made by two employees from the same rank in a particular room late at night in the *Safari tours* company are detected. The *lvisibleto* check in line 11 confirms if the employees are from the same rank and the *public* check confirms that the access device is installed in the *Safari tours* company.

Listing 6.7: Localised rule for night employee access

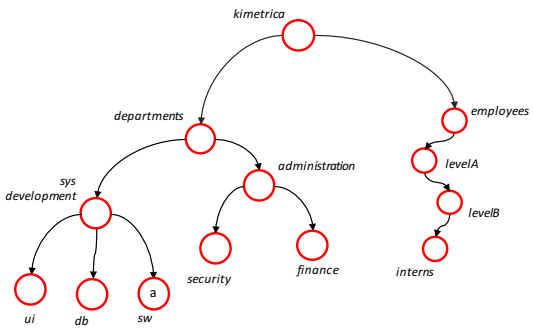
```
1
2 { rulename: "staff_member_employee_access",
3   conditions:[
```



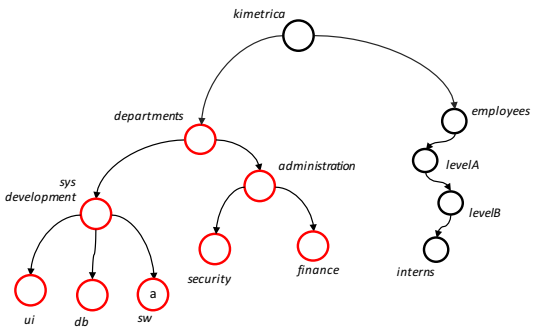
(a) peerof a



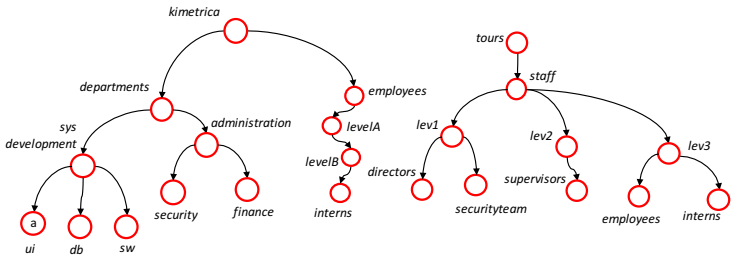
(b) lpeerof a



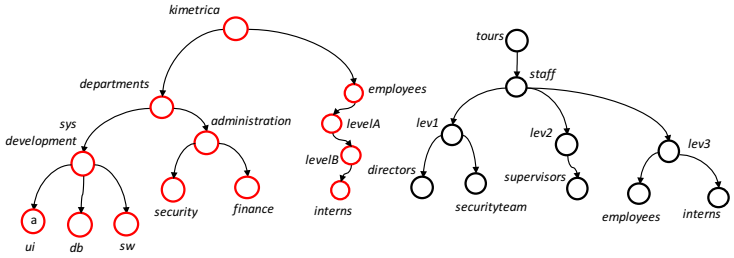
(c) visibleto a



(d) lvisibleto a



(e) super a



(f) public a

Figure 6.6: Comparing local and global scopes in Serena^s – The scopes shown are in relation to the group hierarchy from Figure 6.1

```
    {$e1: {type:"employee", name: "?nam1"}},
    {$e2: {type:"employee", name: "?nam2"}},
    {type:"accessreq", id: "?req1", person: "?nam1", time: "?t1", device: "?dev"},
    {type:"accessreq", id: "?req2", person: "?nam2", time: "?t2", device: "?dev"},
    {$d: {type:"accessdevice", name: "?dev"}},
    {type:"$test", expr:"(time.hourBetween(?t1, 0, 4)) && time.diff(?t1, ?t2, 5)"}
  ],
  scopes:[ "$e1 lvisibleto $e2", "$d public tours"],
  actions: [ /* ... */ ]
}
```

6.5 Scoped Notifications

As mentioned in Section 2.2.4, RKDAs require availability of an active feedback mechanism. When a rule is activated in an RKDA, there should be a way to notify its respective clients. *Bundling clients into groups can be exploited to limit the distribution of client notifications.* This approach is similar to that applied in event-based systems via visibility roots discussed in Section 5.4.4. For instance, in the office complex once an access request is granted then the effect should be displayed on the correct dashboards of the clients of the security team in the relevant company, as per the requirements set in the scenario in Section 2.3.

A basic approach is to notify the client that added the rule. Other application scenarios may have more stringent requirements placed on notification semantics (i.e., *who to notify*) when rules are activated: notifying a particular group rather than only the 'owner' of the rule or the client that added the rule. This is due to the structured nature of RKDAs that this dissertation targets.

Serena^s supports such notification semantics whenever a scoped rule has been activated. This is done through `notify` construct in rules, specified in the SRL syntax grammar back in Figure 6.3. Listing 6.8 shows the same *InternServerroomAccess* rule this time with the `notify` construct in Line 14. The construct specifies the clients or groups to notify once the rule is fired using *notification scopes*.

> Notification scopes are similar to the matching scopes, however in this case they enforce notification constraints to a group, related groups, or direct clients.

Notification scopes require Serena^s to retrieve and notify *all* connected clients in all the groups that are applicable to a particular notification scope definition. The `notify` construct can be used with scopes specified in Section 6.2.2 as well as local scopes in Section 6.4.

In the Listing for *InternServerRoomAccess*, the rule specifies that Serena^s should notify members of `security` group and any of its subgroups. With the `private` scope Serena^s only notifies clients that are in the specified group.

Listing 6.8: Scoped rule for accompanied access to server rooms - with notify

```
1  {rulename: "intern_serverroom_access",
2  conditions:[
3   {$e1: {type:"employee", name:"?intname"}},
4   {$e2: {type:"employee", name:"?empname"}},
5   {$d: {type:"accessdevice", name: "?dev"}},
6   {type:"accessreq", id: "?reqid1", person: "?intname", device: "?dev", time:"?t1"},
7   {type:"accessreq", id: "?reqid2", person: "?empname", device: "?dev", time:"?t2"},
8   {type:"$test", expr:"(time.hourBetween(?t1, 8, 20)) && time.near(?t1, ?t2)"}
9  ],
10 scopes:[ "$e1 subgroupof interns","$e2 private levelB", "sysdevelopment supergroupof
     ↪ ($e1 & $e2)", "$d subgroupof serverrooms" ],
11 actions:[
12  {assert: {type: "accessrep", reqid:"?reqid", allowed: true, time: "?t1"}}
13 ],
14 notify: ["private security"]
15 }
```

Notifying Direct Clients In addition, to notify one particular client, the notify construct can be prefixed with a '`#`', for example `"notify #securityhead"`. The event manager will directly send the notification to the specified client. If a rule does not contain any notification constructs, then the framework will not send any notifications when the rule is activated.

Reacting to Client Notifications In client code, in order for client applications to react to a notification from a rule with a group notification definition, the client specifies a `onGroupRuleActivated` handler with the parameters shown in Listing 6.9. The client code for receiving such notifications with scopes using JavaScript syntax in a typical web application is shown. As before, a client connects to the server in line 1. The client specifies code that will be executed whenever a notification for a scoped rule is detected, with the arguments of the `rulename`, the `facts`, and the `owner` that added rule (line 3).

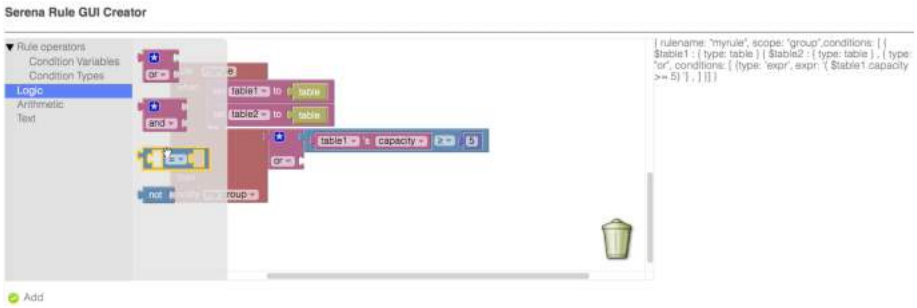


Figure 6.7: Building scoped rules graphically using SerenaUI

Listing 6.9: Serena^s reacting to notifications in client code

```
var securityClient = new SerenaClient(serverip, 'kimetrica', 'security');
//...
securityClient.onGroupRuleActivated(function(rulename, facts, owner){
  // group rule was fired
  //...
});
```

6.6 SerenaUI: Graphical Scoped Rules Builder

One of the challenges of the usurpation of rule-based systems is that end-users can have a low understanding of rule design. This is due to issues such as complexity of rules and non-existent standardised procedures of representing rules [Cla83]. The solution that most modern rule engines provide is to expose rule creators a intermediate way to design rules: usually using a domain-specific language [GDJ99] and graphical interfaces to design rules [MP90].

The Serena^s framework embraces visual programming to assist users to create scoped rules. *SerenaUI* is an interactive, graphical Web-based rule-building tool based on Blockly [Dev16], a library the provides functionality for code generation from arrangements of GUI components consisting of graphical blocks.

In SerenaUI users employ the use of interlocking, graphical and composable blocks to define scoped rules. It uses a parser that follows the syntax of the Serena rule language (SRL) shown in Figure 6.3 when generating scoped rules from graphical components. A rule designer can request the `http://<host>/serenaui` page on a browser, where after logging in Serena^s will retrieve specific groups, templates etc that are relevant to the client. For instance, a security person in *Kimetrica* can log in and the framework will deliver the relevant templates as the rule-building components on the graphical menu 'Rule Operators'. Rules can then be built using blocks that are used to build rule artefacts. Supported blocks are condition variables, literals, arithmetic or boolean operators as shown in Figure 6.7. Once the rule design is done the user can add the rule which will generate a rule in SRL that will be sent to the server via the Serena client library. The rule will then be added to the rule engine as outlined in Section 4.3. SerenaUI provides thus ways in which rules can be created and deployed graphically, which can be beneficial to a number of end-users.

6.7 Chapter Summary

This chapter presented the scoped Serena Rule Language. The foundations of the language were first introduced, giving the reasons for embracing the heterogeneity of clients in RKDAs, in order to provide constructs that can be used in scoped rules. The syntax and semantics of the framework were then presented with a focus on how rule designers can program scoped rules to capture data from heterogeneous clients. Examples of using scopes for selective client notifications were also illustrated. The SerenaUI graphical rule-building tool was presented to as an alternative method to design SRL rules. Throughout the chapter the motivating example of the office complex system was used to clearly explain the scoped rule language. Given structured knowledge representations, scoped rules provide ways in which developers of RKDAs can formulate rules to capture community knowledge. The next chapter delves into the details of how scope-based reasoning is implemented internally in the rule engine.

7

Serena^s
The *Reentrant* Cloud-based Rule Engine

Home affairs should not be talked about in the public square.

Kenyan proverb

The previous chapter introduced scoping constructs in the Serena Rule Language. This chapter presents *Serena^s*, a scoped extension to the Serena framework presented in Chapter 4. Serena^s uses the physical or logical organisation of multi-user applications to create an encoding. The encoding eases the computational workload of the inference algorithm when distinguishing client data into *scopes*. Section 7.1 outlines how the scope-aware rule engine performs its processing cycles. The following sections from Section 7.1.3 explain how scoped rules are processed in the execution cycle of the Serena^s rule engine, focusing on the motivating office complex example. A further optimisation to the approach known as *scope-based hashing* is presented in Section 7.4.2. The chapter concludes by revisiting requirements for supporting heterogeneity for RKDAs in Section 7.6 and finally presents a summary of the main points in Section 7.7[1].

7.1 The Serena^s Encoding Scheme

Rule engines require orchestration within rules to discriminate or distinguish between instances of different entities. In Serena^s this is fulfilled by using scoped rules. In addition, Serena^s exposes constructs that provide ways in which clients can further exploit relationships between them to source data with regards to their own organisational hierarchy or layout. This is in line with the vision of exploiting collective or community knowledge. This chapter discusses how the rule engine uses scopes in rules to ease the computational workload of its main processing component, the inference engine.

[1] Observations described in this chapter have been published as [KRD17a; KRD17b].

7.1.1 The Need for an Efficient Encoding

Scoped rules in Serena^s were introduced in Chapter 6. When defining scoped rules, scope expressions specify which client data is applicable to a rule. Moreover, we have seen that scopes are also useful to specify which data from *groups of clients* is applicable to a rule.

When a scoped rule is added to the server, the rule engine creates its Rete graph. We have seen that adding the rule to the graph naïvely on the server's rule engine will cause it to process all the data from all clients. This forces the execution cycle of the engine to perform excessive computations when only part of the data may be applicable to the rule. With scoped rules, the rule engine can utilise the information provided by scope expressions. The Serena^s rule engine is scope-aware, therefore when scoped rules are added, the inference engine uses the expressions to annotate Rete nodes with scope information.

During execution, the inference engine still needs a way to ascertain whether particular data applies to a scoped node. The different scopes that can be used in rules were presented in Section 6.2.2. Take the example of a node with a scope check `<$e subgroupof sysdevelopment>`[2]. To find out whether a particular data item is applicable to the node, the engine would need to transitively test whether the employee is in the group or any of its subgroups `ui`, `db` or `sw` and their children, if any. Other scopes have different relationships requiring different tests, which further complicate the process of *checking if a data item is compatible with a particular Rete node through its annotated scope check*. Instead of performing these checks using these naïve approaches, we need an efficient **encoding** that can quickly determine compatibility of data in a heterogeneous rule engine during execution. This way, the engine can *quickly* ascertain whether a particular rule scoped with a specific group should fire whenever data arrives, preferably using constant-time operations (or as defined in [Aït+89], *virtually* constant-time operations).

> The vision is to use an encoding method that rather than performing scope checks that are computationally expensive such as path traversals in hierarchical client organisation structures, performs near-constant time operations to entirely determine if a data item passes a scope check. We precompute the scope check, store and maintain it efficiently as an encoding that will be used to expeditiously process scope checks.

7.1.2 Selecting an Encoding Scheme

To select an encoding mechanism used to process scope checks in Serena^s, there are several aspects to consider.

First, the method should provide ways of capturing different arrangements of clients via client hierarchies. The encoding method should provide ways of defining how various clients are organised into groups, and the different relationships between them. Different scope tests will require different mechanisms to test for inclusion, so the chosen encoding method should encompass these factors.

Second, the encoding method can have an extensive pre-computation (e.g., longer compile time) in order to develop a more efficient runtime encoding. This is because it will run on a Web server, which typically have *run-once* characteristics that can perform lengthy initialisations for the added benefit of faster runtime execution.

[2] In this text we denote scope checks with angle brackets.

Finally, the encoding should be able to be implemented in the Rete algorithm in such a way that operations for determining compatibility of data items (as explained in the previous section) should map to the normal processing cycle of the inference engine.
A discussion of encoding methods based on these considerations follows.

7.1.3 Encoding Methods

There exist similar work to perform encodings on hierarchies, commonly referred to as *closure encoding tests* [ZG01], which have a diverse range of applications. This section briefly mentions various encoding methods for these types of tests and their suitability for use in the inference engine.

Encoding Methods

Simple methods. A simple approach is ***relative numbering*** [SPT83], a process that produces an ordered numbering of all elements. Each node stores a numbered range $[a, b]$ (the max and min numbers) of its successors, which is used to test if an element is included in the node's successors.

A separate method is ***Cohen's encoding***. With each element in a hierarchy it stores its level (or height), its unique ID in the level and an array containing all its predecessors in the hierarchy, also indexed per level [Coh91]. A predecessor test involves checking if an element exists in the array via its ID.

Advanced Methods. In the ***packed encoding*** method [VHK97] the hierarchy is modified and partitioned into *slices* where no two elements in one slice have common descendants. An array is created for each element based on the identified slices. An element is a predecessor if it is contained in another element's slice arrays.

The ***range compression*** method [ABJ89] generalises the relative numbering method through the use of multiple $[a_1, b_1], ..., [a_n, b_n]$ disjoint intervals for an element rather than a single interval. To test if an element is included in another requires a check if its assigned ID is included in any of the element's descendant intervals – making testing time dependent on these intervals.

One of the most explored areas is in the ***bit-vector encoding*** method [Aït+89; HN94; RT01]. The method is a hierarchical encoding method where ordering relations are embedded onto lattices used to encode bit vectors. Each element e is encoded as a bit vector V_e of a number of bits not larger than the total number of elements n. If in the vector $V_e[i]$ `= 1` then we say e has gene i under the encoding. Suppose another element a is encoded as bit vector V_a also using the same encoding. With the bit-vector encoding methods, to check if a is a successor of e we mask their two bit vectors to check the genes of the result: `(`V_e `and` V_a`)` `=` V_e. An encoded test using the bit vector encoding therefore uses elementary bitwise boolean operations.

Discussion. The simple methods have a limitation of only supporting single-parent hierarchies, and are unsuitable because they severely limit the types of supported client hierarchies. The various advanced methods presented also have some limitations. Aside from having restrictions on the composition of the hierarchy to be partitioned, the packed encoding and range compression methods have testing processes that become more complex with increasing hierarchies due to increasing slices or intervals. They will thus negatively affect the performance of rule engine execution.

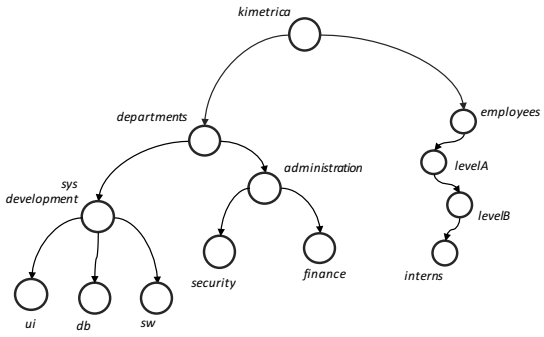


Figure 7.1: The Kimetrica hierarchy

Ultimately, the requirement of fast operations on incoming data leads to the eventual compromise between simplicity of fast encoded operations and saving space. The type of encoding can be chosen depending on the needs of the system[3]. In a standard Rete implementation however, matching cycles typically perform billions of join operations in beta nodes [Doo95]. Keeping operations simple and fast greatly benefits the fast execution of a chosen encoding, rather than having a compact space and more complex encoding tests. The bit-vector encoding method does exactly that. Its main aim is to sacrifice space for faster speed and simpler encoding tests. The bit-vector encoding method also enumerates all group elements in the hierarchy, making it possible to perform the various scope operations of the framework. The binary representations of elements also allow the framework to perform a technique similar to *specialization* in [ZG01], by using bit vectors to pre-compute values used to optimise the encoding test process even further. This is the basis of the *scope-based hashing* method which we explain later in Section 7.4. **Consequently, Serena's encoding method is based on the faster bit-vector encoding method**. The foundations of the technique is explained next.

7.1.4 The Encoding Process

The encoding method that Serena employs is based on the implementation by [Aït+89] which has formal foundations in order theory. This section highlights this encoding method and presents the specifics of the encoding process. The method follows the formal definitions detailed in Appendix A.1.1. To make the explanation more clear and concise, we utilise the hierarchy for *Kimetrica* outlined in Section 6.1.1. For reference, the hierarchy is shown again in Figure 7.1.

The Groups Hierarchy as a Poset

As explained in Section 6.3, Serena receives the client group hierarchy from an administrator. The hierarchy is a partially-ordered set (*poset*)[4]. The example hierarchy can be represented as a poset $(P, \leqslant)$ with the binary relation $\leqslant$ defined as '*is subgroup of*' or '*is part of*'.

The poset P has an element (a, b) iff a is part of b. Examples in the hierarchy elements include *(administration, departments)*, *(security, administration)*, *(sw, sysdevt)* and

[3]The choice of encoding that provides efficient operations also becomes a tradeoff between static and dynamic approaches, which are at opposite ends of a spectrum of good performance vs high flexibility. This is discussed in Section 9.3.1.

[4]For definitions, see Appendix A.1.1.

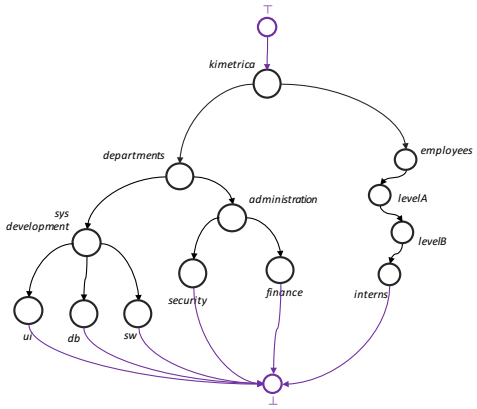


Figure 7.2: The hierarchy converted into a lattice L

(interns, levelB). With P we can perform well-defined operations such as calculating the bounds (LUB, GLB)[4] and extrema (maximals & minimals). For instance the maximal in the group hierarchy of Figure 7.1 is *kimetrica*. If that group is omitted, then the maximals are *departments* and *employees*.

Nevertheless, when processing poset operations the engine would still have to traverse all the elements of the poset. A more efficient structure to represent the elements is discussed next.

The Groups as a Lattice

The Serena^s framework converts the groups poset to a lattice L as outlined in Appendix A.1.2. This leads to the hierarchy depicted as the *Hasse diagram*[4] in Figure 7.2. The benefit of this conversion is that a lattice represents the group hierarchy in a form that is more efficient to encode and to compute than the naïve poset representation.

Encoding the Lattice

With the lattice L, Serena^s performs a customised *bit-vector encoding* process ϑ that lays its basis on the method by Aït-Kaci [Aït+89]. The formal background is outlined in detail in Appendix A.3. The process performs calculations for all groups in the group hierarchy.

Example. The calculations made to come up with the codes for elements in L depicted in Figure 7.2 are shown below. Note that the codes are in binary notation. At each step, Serena^s also calculates the level of each group represented in the matrix.

$\vartheta(\top) = 0$

$\vartheta(kimetrica) = [\vartheta(\top) \vee 2^0] = [0 \vee 2^0] = 1$

$\vartheta(employees) = [\vartheta(kimetrica) \vee 2^1] = [1 \vee 2^1] = 11$

$\vartheta(departments) = [\vartheta(kimetrica) \vee 2^2] = [1 \vee 2^2] = 101$

$\vartheta(levelA) = [\vartheta(employees) \vee 2^3] = [11 \vee 2^3] = 1011$

	⊤	kime	empl	dept	levA	admi	sysd	levB	fina	secu	sw	db	ui	inte	⊥
⊤	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
kime	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
empl	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
dept	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
levA	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0
admi	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
sysd	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0
levB	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0
fina	1	1	0	1	0	1	0	0	1	0	0	0	0	0	0
secu	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0
sw	1	1	0	1	0	0	1	0	0	0	1	0	0	0	0
db	1	1	0	1	0	0	1	0	0	0	0	1	0	0	0
ui	1	1	0	1	0	0	1	0	0	0	0	0	1	0	0
inte	1	1	1	0	1	0	0	1	0	0	0	0	0	1	0
⊥	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 7.3: The encoded matrix M_{ϑ_L} for the simple hierarchy shown in Figure 6.1b. The entire matrix for all the company hierarchies in the example is shown in A.1.

$\vartheta(administration) = [\vartheta(departments) \vee 2^4] = [\texttt{101} \vee 2^4] = \texttt{10101}$

$\vartheta(sysdevt) = [\vartheta(departments) \vee 2^5] = [\texttt{101} \vee 2^5] = \texttt{100101}$

$\vartheta(levelB) = [\vartheta(levelA) \vee 2^6] = [\texttt{1011} \vee 2^6] = \texttt{1001011}$

$\vartheta(finance) = [\vartheta(administration) \vee 2^7] = [\texttt{10101} \vee 2^7] = \texttt{10010101}$

$\vartheta(security) = [\vartheta(administration) \vee 2^8] = [\texttt{10101} \vee 2^8] = \texttt{100010101}$

$\vartheta(sw) = [\vartheta(sysdevt) \vee 2^9] = [\texttt{100101} \vee 2^9] = \texttt{1000100101}$

$\vartheta(db) = [\vartheta(sysdevt) \vee 2^{10}] = [\texttt{100101} \vee 2^{10}] = \texttt{10000100101}$

$\vartheta(ui) = [\vartheta(sysdevt) \vee 2^{11}] = [\texttt{100101} \vee 2^{11}] = \texttt{100000100101}$

$\vartheta(interns) = [\vartheta(levelB) \vee 2^{12}] = [\texttt{1001011} \vee 2^{12}] = \texttt{1000001001011}$

$\vartheta(\bot) = [\vartheta(ui) \vee \vartheta(db) \vee \vartheta(sw) \vee \vartheta(security) \vee \vartheta(finance) \vee \vartheta(interns) \vee 2^{13}] = \texttt{1111111111111}$

As indicated by the steps in Appendix A.3, each result is reversed and a `1` added to its most significant bit (leftmost column) before being inserted in the matrix. The result is a binary matrix encoding M_{ϑ_L} (read as the matrix M produced from encoding L using ϑ) of the group hierarchy shown in Figure 7.3. If i and j are indices of groups a and b respectively, and `group(`i`)` corresponds to the group a at that index, then the encoding M_{ϑ_L} has the following properties:

i) The labels on the rows of M_{ϑ_L} correspond to the groups in L; and similarly for the columns. The first row corresponds to $\top$ and the last row to $\bot$. $\top$ has only one bit value set to 1 while in $\bot$ all bit values are set to 1.

ii) An entry $M_{\vartheta_L(i,j)}$ has a `1` if `group(`i`)` = `group(`j`)` or if `group(`j`)` is an ancestor of `group(`i`)` in L, and `0` otherwise.

iii) An entry $M_{\vartheta_L(j,i)}$ has a `1` if `group(`i`)` = `group(`j`)` or if `group(`j`)` is a descendant of `group(`i`)`, and `0` otherwise.

iv) Group a at index i is a *maximal* in P iff the row $M_{\vartheta_L(i,*)}$ has a `1` only at $M_{\vartheta_L(i,i)}$ and at $M_{\vartheta_L(i,\top)}$.

v) Group a at index i is a *minimal* in P iff the column $M_{\vartheta_L(*,i)}$ has a `1` only at $M_{\vartheta_L(i,i)}$ and at $M_{\vartheta_L(\bot,i)}$.

The notation $M_{\vartheta_L(i,*)}$ indicates the row vector at index i, and likewise $M_{\vartheta_L(*,i)}$ the column vector. The properties for ii) and iii) follow from the transitive properties of the underlying poset (Appendix A.1.1). The next properties iv) and v) are validated.

7.1.1 Proposition. *Given a lattice L encoded as a matrix M_{ϑ_L}, the encoding of group g at index i in the matrix and a row vector for g, $V_g = M_{\vartheta_L(i,*)}$, then if V_g has only 2 column values set to `1` then g is a maximal in the poset P represented by the lattice L.*

Proof. It is known that one of the `1` bit values in V_g corresponds to the group g due to the reflexive properties of the poset P used to form L. Each element in a vector V in M_{ϑ_L}, is appended a bit value `1` in the encoding process to represent the topmost $\top$ element, which can be ignored since $\top \notin P$. Therefore g has no other predecessors (other than $\top$), and is subsequently a maximal in P. □

The proof of property v) follows from the Duality Principle of orders (Definition A.1.2.1).

An additional product of the encoding process is that it generates the *Level* (defined in Appendix A.3) of each group, which Serena^s stores internally. For instance,

$$\begin{aligned} \mathit{Level}(\texttt{administration}) &= \mathit{Level}(\texttt{departments}) + 1 \\ &= \mathit{Level}(\texttt{kimetrica}) + 1 + 1 \\ &= \mathit{Level}(\top) + 1 + 1 + 1 \\ &= 0 + 1 + 1 + 1 \\ &= 3 \end{aligned} \tag{7.1}$$

The levels assist us to define *local maximals*.

- A group c is a *local maximal* in P if it is an immediate descendant of a maximal, i.e., $c \prec \lceil P \rceil$.
- A group d is a *local minimal* in P if it is an immediate predecessor of a minimal, i.e., $\lfloor P \rfloor \prec d$.

Definitively, a local maximal is actually a maximal when its parent groups are removed from the poset[5]. If m_g is a maximal and m_l is a local maximal we can conclude the following from the above definition of local maximals (and equation A.1),

$$\forall m_g \in P,\ \mathit{Level}(m_g) = 1 \tag{7.2}$$

$$\forall m_l \in P,\ \mathit{Level}(m_l) = 2 \tag{7.3}$$

[5]The Duality Principle confirms the same for local minimals, but these are of little interest to our approach.

We will henceforth use the terms *global maximals* and maximals interchangeably, but we will specifically refer to local maximals (to distinguish them from global maximals).

For faster retrieval, the row indices for all the maximals and local maximals in the matrix are also stored by the scoping module in the engine (described in Section 6.3). The next sections will now show how the encoding is used to perform scoping within the inference engine.

7.2 Supporting Reentrancy via Scope-based Reasoning

This section explains how the data from encoding M_{ϑ_L} is used to efficiently implement scoping in the inference engine. The scoping module (introduced in Section 6.3) is mainly responsible for enforcing the scoping semantics in various components of the Serena server. It creates and stores the client information needed to support the scoping operations in rules in the inference engine. It internally represents the organisation of clients, and builds an efficient encoding mechanism for scopes. This encoding affects the matching strategy in the inference engine, and is used by the event manager to determine the recipients of notifications.

> The scoping module therefore performs three operations. Firstly, it uses scope definitions in rules to modify the node reuse process of the Rete graph. Secondly, it uses stored scope expressions in nodes to influence the matching process. Thirdly, it uses notify directives in rules for invocation of clients after rule activations.

7.2.1 Implementing Scoping with M_{ϑ_L}

The encoding with M_{ϑ_L} is the basis of enforcing scoping in the execution of the inference engine. To facilitate this, Serena^s adds scope tests at appropriate nodes when building the Rete network, and evaluates scoping operations in the beta network's beta join nodes during matching. These tests are based on the definitions outlined in Section 7.1.4. Given that i and j are indices of groups a and b respectively in the matrix M_{ϑ_L}, the tests include the following:

I ***visibleto*** – To perform a scope check of a `visibleto` b the Serena^s checks if the result of $M_{\vartheta_L(i,*)} \wedge M_{\vartheta_L(j,*)}$ is not $\top$ as per property i). To perform a scope check of a `lvisibleto` b it additionally checks that this result is not a global maximal as per property iv).

II ***peerof*** – To check if a `peerof` b Serena^s checks whether $Level(a) = Level(b)$ from the process of encoding M_{ϑ_L}. To check if if a `lpeerof` b it further checks if a `visibleto` b.

III ***subgroupof*** – A scope check of a `subgroupof` b is true if the result of $M_{\vartheta_L(i,*)} \wedge M_{\vartheta_L(j,*)} = M_{\vartheta_L(j,*)}$ as per property ii). Conversely, b is a *supergroupof* a as per property iii).

IV ***private*** – To find out if a `private` b it can check if $M_{\vartheta_L(i,*)} \wedge M_{\vartheta_L(j,*)} = M_{\vartheta_L(i,*)}$ and $M_{\vartheta_L(j,*)} \wedge M_{\vartheta_L(i,*)} = M_{\vartheta_L(j,*)}$ as per property ii) and iii). In practice, the check $M_{\vartheta_L(i,*)} = M_{\vartheta_L(j,*)}$ suffices.

V ***super*** – A scope check of a `super` b passes if the result of $M_{\vartheta_L(i,*)} \wedge M_{\vartheta_L(\top,*)} = M_{\vartheta_L(\top,*)}$ as per properties i) and ii).

Note that the *visibleto* scope check is equivalent to a ***public*** check if there exists a main group in every client hierarchy. Typically, groups a and b are extracted from group metadata on facts inserted into the Rete graph. We give a concrete example of how these tests are used by Serena^s during the execution cycle in Section 7.3.

To chain the operations Serena^s employs the operators `&` and `|` to designate the boolean `and` and `or` respectively. All the above tests have distributive properties, for instance[6]:
`<a subgroupof (b | c)>` translates to `<(a subgroupof b) | ( a subgroupof c)>`
`<a subgroupof (b & c)>` translates to `<(a subgroupof b) & ( a subgroupof c)>`
Consequently, by default the framework treats facts tagged with multiple groups from the hierarchy as implicitly having the `or` operator, as with the first case.

An interesting property can be observed when examining the columns of the encoded matrix M_{ϑ_L}. This property is useful for performing checks for descendants of particular groups. It is also is used by the framework for retrieving all descendants of a given set of groups, particularly useful when deciding which clients to send notifications to (discussed in Section 7.3.2).

7.2.1 Theorem. *Given the encoding of group g, $g \in L$ with index i in the matrix M_{ϑ_L} and a column vector for g $M_{\vartheta_L(*,i)}$, then the subgroups of g are the elements where $M_{\vartheta_L(*,i)}$=1.*

Proof. In the matrix a bit element at index j in the row vector $M_{\vartheta_L(i,*)}$ that corresponds to a group b is `1` if b is a supergroup of g and `0` otherwise. It follows that in every other row $M_{\vartheta_L(k,*)}$ of a group $h \in L$ a bit element $M_{\vartheta_L(k,i)}$ has a `1` whenever g is a supergroup of h, or inversely, h is a subgroup of g. Consequently, the vector $M_{\vartheta_L(*,i)}$ of any group g in M_{ϑ_L} represents all subgroups of g. □

Examples of some of the operations are presented in subsequent sections.

7.2.2 Node Reuse with Scopes

Serena^s follows the semantics of the Rete algorithm to promote *structural similarity* (explained in Section 4.4.2) with scopes. When a rule is added to the engine, as usual the engine checks if some of the conditions are compatible with existing nodes in the graph. Serena^s further takes the scope definitions in rules and uses these to place scope expressions in the graph whenever a node in the beta network is created. It attempts to find compatible nodes for reuse and, depending on the situation, can choose to reuse or append new nodes.

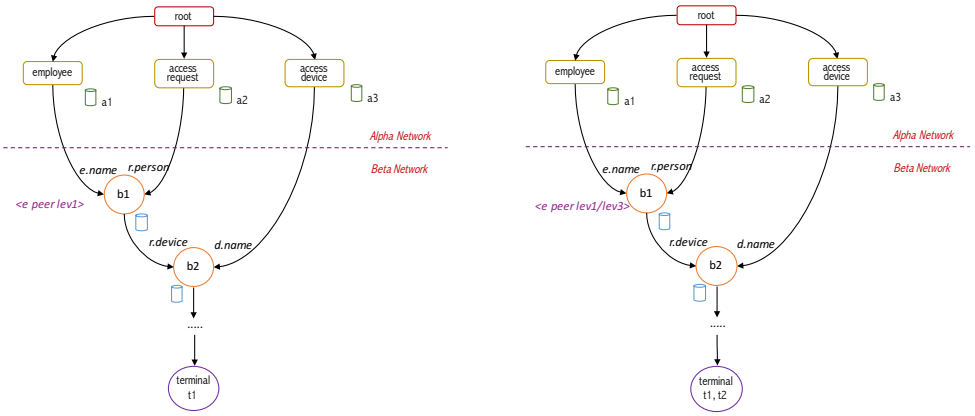
Example 1. The process is best clarified with an example. Consider the rules shown in Listing 7.1.

Listing 7.1: Example access rules

```
[{ rulename: "example_access_rule1",
   conditions:[
    {$e: {type:"employee", name: "?empname"}},
    {$d: {type:"accessdevice", name: "?device"}},
    {type:"accessreq", id: "?reqid", person: "?empname", device: "?device"},
    /*...*/
   ],
```

[6] We denote scope checks with angle brackets.

```
8     scopes: [ "$e peerof lev1"]
9  },
10 {  rulename: "example_access_rule2",
11    conditions:[
12     {$e: {type:"employee", name: "?emp"}},
13     {$d: {type:"accessdevice", name: "?dev"}},
14     {type:"accessreq", id: "?reqid", person: "?emp", device: "?dev"},
15     /*...*/
16    ],
17    scopes: [ "$e peerof lev3"]
18 }]
```



(a) After addition of first rule *(b) After addition of second rule*

Figure 7.4: Reusing scoped nodes in the Rete graph.

Figure 7.4a shows the graph when the first rule in lines 1 to 9 is added. The beta node $b1$ is annotated with the scope check for the first rule. When the second rule in lines 10 to 18 is added, Serena^s notes that aside from being structurally similar to the existing graph, the rule also has a compatible scope check in line 17. This is because of the `peerof` check in the second rule can be reused with that of the first rule: `level1` and `level3` are on the same level in the hierarchy, computed according to the definitions in the previous Section 7.2.1. The terminal node will now activate both rules. The node reuse process is explained next.

Node reuse in Standard Rete. When adding a rule to an existing graph, the traditional Rete algorithm proceeds by comparing the incoming rule's conditions to pick an existing node n for similar patterns that can be reused, starting from the root node. Once an existing node n is picked, the algorithm can proceed in one of two ways depending on the compatibility of the patterns of the identified node with the incoming conditions.

- The node n cannot be reused as the conditions are not compatible, therefore the new node will be created as a sibling of this node.
- The conditions are compatible and the current node n can be reused, thus no new node will be created and this new node will be returned.

This proceeds iteratively until all the rule conditions are processed, and ends once its terminal node is created.

Node reuse in Serena^s. As specified above, node reuse in traditional Rete proceeds to reuse a particular existing node only if the node's patterns are same as those of the incoming rule conditions. Serena^s appends an additional scoped node reuse process by *checking if the scope test of the existing node is compatible with that of the defined scope test of the rule to be added*. Compatibility of rules is based on the properties in Section 7.2.1 performed as follows:

- If the existing node is a node with a *super* scope, then it can always be reused.
- If the existing node is any of the other scopes, then it can only be reused if the incoming scope is the same and the operands are compatible as per their respective properties.

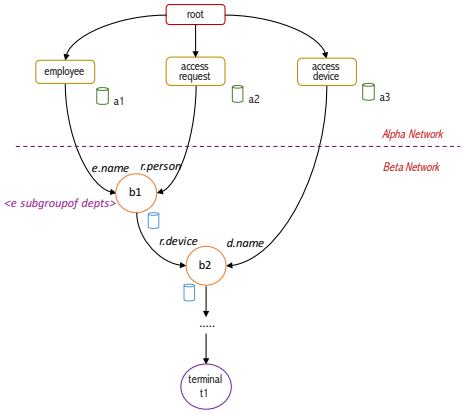
Example 2. In another example, consider the code shown in Listing 7.1. The listing also shows two rules, where the second is added after the first.

Listing 7.2: Example access rules

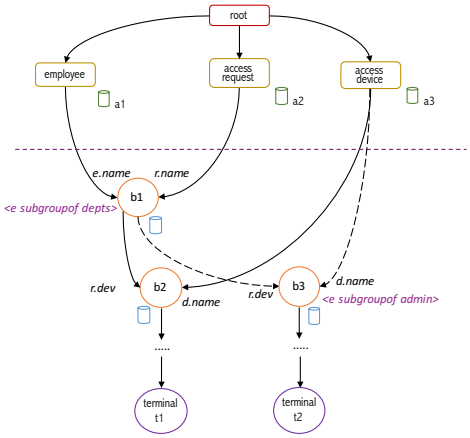
```
[{ rulename: "example_access_rule1",
   conditions:[
    {$e: {type:"employee", name: "?empname"}},
    {$d: {type:"accessdevice", name: "?devname"}},
    {type:"accessreq", id: "?reqid", name: "?empname", dev: "?devname"},
    /*...*/
   ],
   scopes: [ "$e subgroupof departments"]
},
{  rulename: "example_access_rule2",
   conditions:[
    {$e: {type:"employee", name: "?nam", badgeid: "?badgid"}},
    {$d: {type:"accessdevice", name: "?devname"}},
    {type:"accessreq", id: "?reqid", name: "?nam", dev: "?devname"},
    /*...*/
   ],
   scopes: [ "$e subgroupof administration"]
}]
```

When `example_access_rule1` is initially added, the resultant Rete graph is built as usual and is shown in Figure 7.5. Now, supposing the second `example_access_rule2` is added, then per condition, Serena^s sees that the alpha memory for `employee` and `accessdevice` can be reused. The next action is to find out if beta node $b1$ can also be reused, which follows the operations listed above. Since `adminstration` is a subgroup of `departments`, Serena^s can reuse this node. Remember that the scope check `<$e subgroupof administration>` still needs to be performed; and therefore the new node $b3$ is added as a child node for $n1$ as in Figure 7.5b.

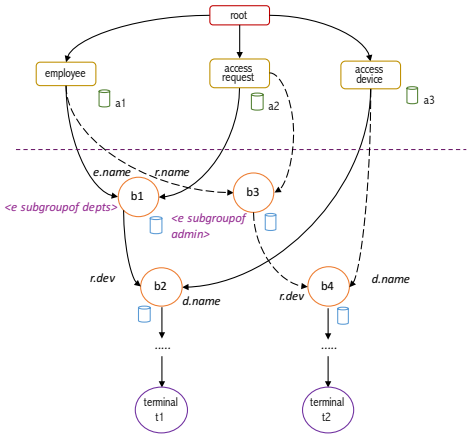
In some cases, however, the node $b1$ would not be reused, resulting in the graph shown in Figure 7.5c: we discuss such situations later in Section 9.3.4. Generally, when all beta nodes have the scope *super* then the scoping algorithm is functionally equivalent to the vanilla Rete algorithm. Moreover, due to the dynamic addition of rules, in practice the node reuse algorithm is dependent on the order that rules are added into the engine. Any newly-added rule that would result in some form of node reordering will simply be appended to the graph. This policy is preferable because it has the least effect on the



(a) First rule added



(b) Second rule added: option 1



(c) Second rule added: option 2

Figure 7.5: Reusing scoped nodes in the Rete graph – Serena can take two actions: either reuse existing nodes or append them

current execution of the engine, but some ideas on reordering the graph are discussed later in Chapter 9.

7.3 Processing Scoping Constraints

Within the inference engine, the built Rete network annotated with scopes will be used during the matching phase of the engine's processing cycles. Scopes allow for a more compact and efficient way to discriminate the tokens that a beta node is required to process (we evaluate this in Chapter 8). As mentioned, the scoping module will use M_{ϑ_L} and the outputs from its encoding process to perform the operations from the scope checks in beta nodes (denoted in previous figures with angle brackets).

7.3.1 The Matching Phase using Scopes

The matching phase with scopes is triggered by a mixture of left and right activations in beta nodes.

Left Activations with Scopes. For left activations, when a token is received in a beta node, all facts in the node's right parent, the alpha memory, need to be checked to see if compatible matches can be found. The scoped inference engine modifies the basic Rete algorithm semantics for left activations outlined in Algorithm 7.1 into that shown in Algorithm 7.2 for a scoped beta node left activation.

Algorithm 7.1 Beta Node Left Activation: Standard Rete

```
1  function betaNodeLeftReceive(node : n, token : t)
2      facts ← n.alphaMemory.getFacts()
3      for each fact f in facts do
4          if n.joinTestPassed(t, f) then
5              t_new ← n.createNewToken(t, f)
6              n.sendTokenToChildren(t_new)
7          end if
8      end for
9  end function
```

Algorithm 7.2 Beta Node Left Activation: with Scopes

```
1  function scopedBetanodeLeftReceive(node : n, token : t)
2      facts ← n.alphaMemory.getFacts()
3      for each fact f in facts do
4          if this.scopeCheckPassed(n,t,f) then
5              if n.joinTestPassed(t, f) then
6                  t_new ← n.createNewToken(t, f)
7                  n.sendTokenToChildren(t_new)
8              end if
9          end if
10     end for
11 end function
```

Right Activations with Scopes. For right activations the process is similar, except that when the alpha memory fact is received then the scoped approach needs to perform iterations through all the tokens stored in the beta node's memory. The vanilla Rete's right activation pseudocode is shown in Algorithm 7.3 and the scoped variation is show in Algorithm 7.4.

Algorithm 7.3 Beta Node Right Activation: Standard Rete

```
1  function scopedBetanodeRightReceive(node : n, fact : f)
2      tokens ← n.getTokens()
3      for each token t in tokens do
4          if n.joinTestPassed(t, f) then
5              t_new ← n.createNewToken(t, f)
6              n.sendTokenToChildren(t_new)
7          end if
8      end for
9  end function
```

Algorithm 7.4 Beta Node Right Activation: with Scopes

```
1  function scopedBetanodeRightReceive(node : n, fact : f)
2      tokens ← n.getTokens()
3      for each token t in tokens do
4          if this.scopeCheckPassed(n,t,f) then
5              if n.joinTestPassed(t, f) then
6                  t_new ← n.createNewToken(t, f)
7                  n.sendTokenToChildren(t_new)
8              end if
9          end if
10     end for
11 end function
```

The Process. The algorithm is simply modified as follows: *on a left or right activation, Serena^s first performs the encoded scope check on the fact from the alpha memory or the token's fact respectively.* If the check passes, the inference engine proceeds with the join computation. The method `scopeCheckPassed` performs scope tests for the data item arguments it receives, using the operations on the encoding outlined in Section 7.2.1. To explain the process, we next describe an example with the matching cycle using scopes.

Listing 7.3: Scoped rule for accompanied access to server rooms - revisited

```
{rulename: "intern_serverroom_access",
conditions:[
 {$e1: {type:"employee", name:"?intname"}},
 {$e2: {type:"employee", name:"?empname"}},
 {$d: {type:"accessdevice", name: "?dev"}},
 {type:"accessreq", id: "?reqid1", person: "?intname", device: "?dev", time:"?t1"},
 {type:"accessreq", id: "?reqid2", person: "?empname", device: "?dev", time:"?t2"},
 {type:"$test", expr:"(time.hourBetween(?t1, 8, 20)) && time.near(?t1, ?t2)"}
],
```

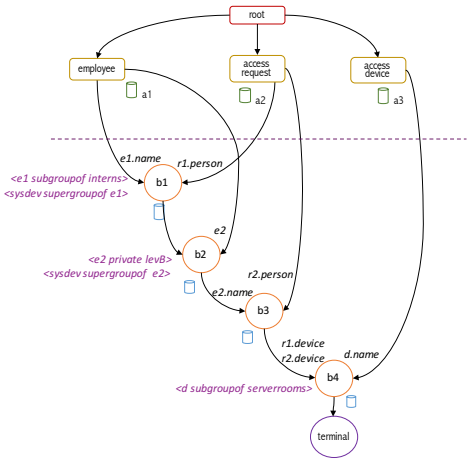


Figure 7.6: Scoped Rete graph for the ServerRoomAccess rule.

```
10  scopes:[ "$e1 subgroupof interns","$e2 private levelB", "sysdevelopment supergroupof
      ↪ ($e1 & $e2)", "$d subgroupof serverrooms" ],
11  actions:[
12   {assert: {type: "accessrep", reqid:"?reqid", allowed: true, time: "?t1"}}
13  ]
14 }
```

Building the Rete Graph

Consider the *ServerRoomAccess* rule, shown for convenience again in Listing 7.3. Remember that rule contains a *scopes* section to distinguish between data from different sources, line 10. The graph as a result of this rule is depicted in Figure 7.6, with the final test node omitted. The graph shows where Serena^s annotates the scopes in different points in the graph according to the defined scope expressions of the rule.

Scopes in the Matching Cycle

To show a definitive process, we will use the full matrix encoding of all company group hierarchies in the office complex from Figure 6.1[7]. When three employees arrive at work by clocking into their respective offices, their `employee` facts are added to the engine. Figure 7.7 shows the state of the graph with the facts added, and these are stored in the alpha node $a1$'s memory. In the node there are two *Kimetrica* employee facts (denoted with a red outline) indicating that two employees have clocked in, and one *Safari tours* fact (denoted with a blue outline), indicating one employee from that company has entered the workplace. Facts in node $a3$ also show two `devices` installed in the each of the companies.

Suppose a software developer `intern` Billie is accompanied by a `levelB` employee Zak in order to gain access to their server rooms of the *Kimetrica* company. Both the employees scan their badges using the device in the company's server room in the building. The two access requests are sent to the Serena server.

The first request from Billie will be inserted into the Rete graph from the root and will be sent through the alpha node $a2$ to the beta node $b1$ as shown in Figure 7.8a. This se-

[7]The full matrix of the entire groups of the office complex hierarchy is illustrated in Appendix A.4.

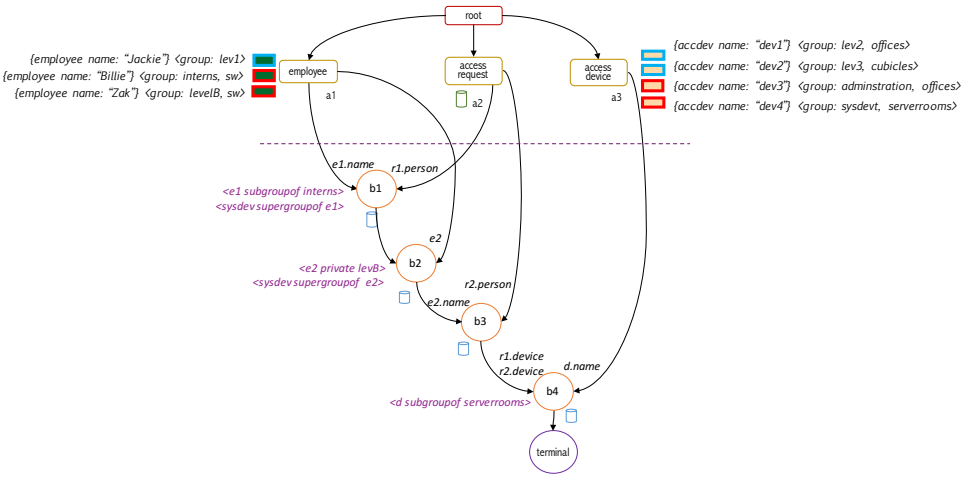


Figure 7.7: Scoped graph for the ServerRoomAccess rule with initial facts.

quence will cause a right activation in node $b1$[8]. The right activation follows Algorithm 7.4, where Serena^s will need to perform two scope checks:

1. `<$e1 subgroupof interns>`

2. `<sysdev supergroupof $e1>`

using the guidelines outlined in Section 7.2.1 for *subgroupof* and *supergroupof*. The checks will be performed on all the facts in the memory of the alpha node $a1$ with the full matrix encoding.

Scope check fails. At this point, let us first show an example of an unsuccessful scope check. If, for instance, Jackie from `Tours` (represented by the first fact in $a1$) was to make an access request, the request will be sent from the root to $a2$ to $b1$. At $b1$, the `employee` fact representing Jackie will trigger the following operations for the scope tests of the node. If l, i, sy and t are indices in M_{ϑ_L} corresponding to the groups $levB$, $interns$, $sysdevt$ and $\top$ respectively,

$$M_{\vartheta_L(l,*)} \wedge M_{\vartheta_L(i,*)} = M_{\vartheta_L(i,*)} \qquad M_{\vartheta_L(sy,*)} \wedge M_{\vartheta_L(l,*)} = M_{\vartheta_L(sy,*)}$$

lev1	1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		sysd	1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0	
inte	1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 &		lev1	1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 &	
T	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		T	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	

In the scope check `e1` represents the `employee` fact for Jackie, which is tagged with the group `lev1`. The tests fail on both accounts, because the actual results (i.e., $M_{\vartheta_L(t,*)}$) differ from the expected results (i.e., $M_{\vartheta_L(i,*)}$ and $M_{\vartheta_L(sy,*)}$) as shown by the two operations at the top. In practice, if the first fails then the second will not be performed – the implicit `and` operation between subsequent scope tests in one node is short-circuited. Therefore the fact for Jackie is incompatible and does not proceed to the join computation. The fact from Zak will similarly fail the scope test.

[8]The fact from Billie will also be sent to the node $b3$ but as it does not have any data items yet, no test will be made.

Scope check passes. Now Billie from *Kimetrica* makes an access request via the fact representing Billie with groups `interns, sw`. If k and sw are indices in M_{ϑ_L} corresponding to the groups *kimetrica* and *sw* the operations for the first scope test of $b1$ are:

$$M_{\vartheta_L(i,*)} \wedge M_{\vartheta_L(i,*)} = M_{\vartheta_L(i,*)}$$

```
inte|1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0
inte|1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 &
----------------------------------------------------------------------
inte|1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0
```

$$M_{\vartheta_L(sw,*)} \wedge M_{\vartheta_L(i,*)} = M_{\vartheta_L(i,*)}$$

```
 sw |1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0
inte|1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 &
----------------------------------------------------------------------
kime|1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The test passes because the first operation fulfils the first scope check as a whole. The second scope check of $b1$ is now performed:

$$M_{\vartheta_L(sy,*)} \wedge M_{\vartheta_L(i,*)} = M_{\vartheta_L(sy,*)}$$

```
sysd|1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
 inte|1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0
----------------------------------------------------------------------
kime|1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

$$M_{\vartheta_L(sy,*)} \wedge M_{\vartheta_L(sw,*)} = M_{\vartheta_L(sy,*)}$$

```
sysd|1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
 sw |1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 &
----------------------------------------------------------------------
sysd|1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
```

The check also passes due to the success of the second operation (remember a fact tagged with multiple groups is implicitly treated as an `or`) and the token for the two facts is created. With this, the normal join test for node $b2$ (for the name and person of the `employee` and the `request`) is assured to operate on compatible data. The join test also passes, and then the token is stored in the intermediate memory of the node $b1$ (as shown in Figure 7.8a) and sent to its child $b2$.

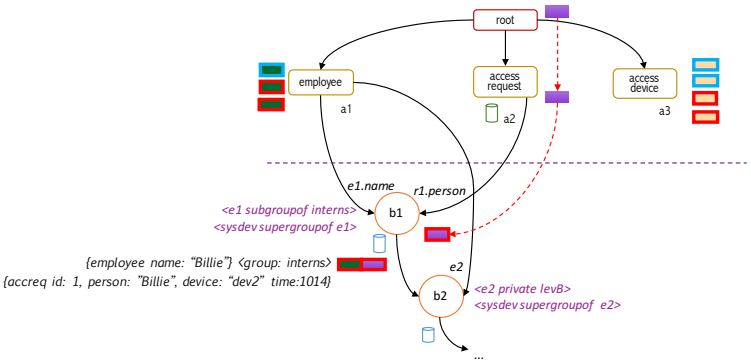
Subsequent tests. At $b2$ a left activation occurs, which will first perform the scope checks on the node $b2$ with the items from the right alpha memory $a1$. The scope checks for $b2$ are `<$e2 private levB>` and `<sysdev supergroupof $e2>`, and will once more use M_{ϑ_L} to calculate the compatibility. Again, the tests follow the ones shown for $b1$ using the rules in Section 7.2.1. The scope tests pass for the token, and the results are appended to the token with `e2` as the fact for `employee` Zak, as shown in Figure 7.8b.

When the second request from Zak comes in Figure 7.8c, the fact will eventually be sent to the node $b3$ (the request will also be sent to $b1$ but will fail the first scope test of that node). The node has no scope tests and the normal join test is performed, capturing the `employee` with the same name as the request and appends the fact representing Zak to the token. This new token is then sent to node $b4$, causing a left activation.

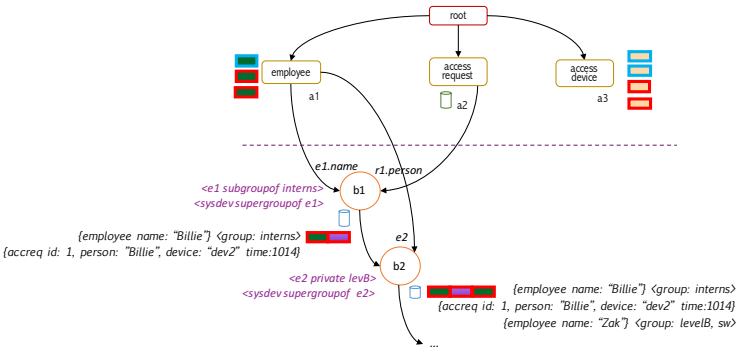
The node should now perform the scope test `<$d subgroupof serverrooms>` on all devices. The scope test for device `dev4` will succeed (since it is tagged with the group `serverrooms`) and will proceed to perform the normal join test of the node, which checks if the names of the devices are the same. The join test does succeed, and the full token shown in Figure 7.8d will eventually be sent to the terminal node for the rule *InternServerRoomAccess*, which will activate the rule since all its conditions (and scope requirements) have been met.

Observation. Observe that in Figure 7.8d all facts are sourced from the same company *Kimetrica*, which are depicted with a red outline. The facts from the *Safari tours* company in blue do not participate in the final token that activated the rule[9].

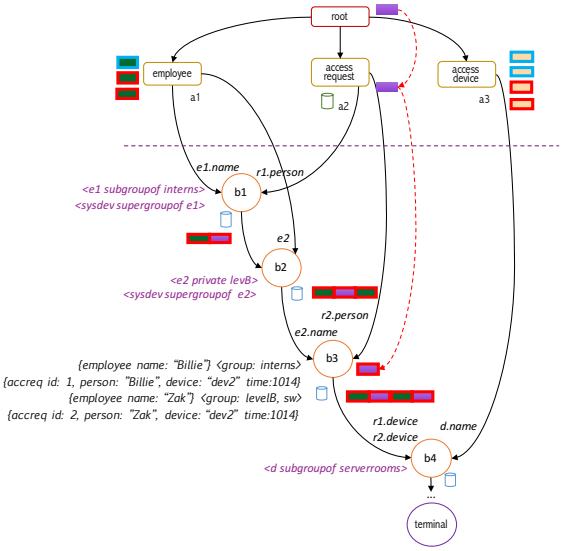
[9] For much larger matrices, it is possible to employ research that proposes offsetting matrix computations to GPU elements [LM01].



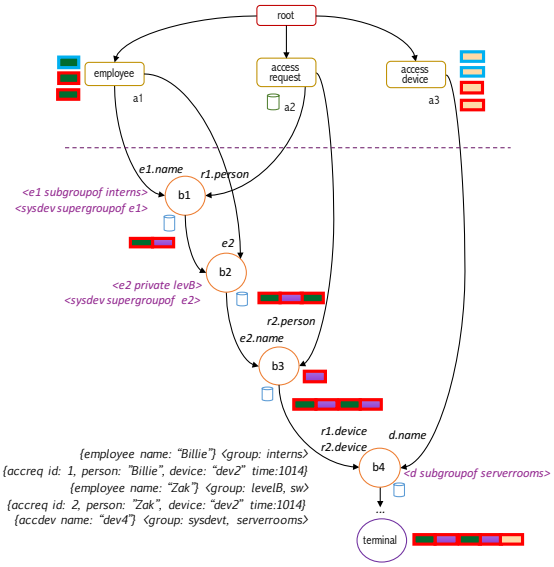
(a) First access request sent to node b1.



(b) Token received by node b2.



(c) Second access request sent to node b3.



(d) Subsequent traversal to b4 and finally the terminal node.

Figure 7.8: Scoped graph for the ServerRoomAccess rule with data – The graph shows sequences after the first and second access requests are made.

Insight: Scopes as Guards

In essence, the scope checks act like *guards* that expeditiously check to see if the data is compatible before performing an actual join in scoped rule engines. If any of the tests fail then the scope check fails and the rest of the computation will not be evaluated. The benefit achieved is that first, tests for a large number of incompatible data are avoided and second, the scope checks are performed in an efficient manner. In the end, scopes can assist to efficiently 'separate the blues from the reds' for reentrancy and in other cases combining them into 'purple facts' for community knowledge, if required.

7.3.2 Scoped Notifications

With the encoding of M_{ϑ_L}, Serena^s can support such notification semantics whenever a scoped rule has been activated. This is done through the `notify` construct in rules, specified in the SRL syntax and described back in Section 6.5. Listing 6.8 shows the same *InternServerRoomAccess* rule this time with the `notify` construct in Line 14. The construct specifies the clients or groups to notify once the rule is fired using notification scopes that specify which clients to send notifications to.

In the Listing for *InternServerRoomAccess*, the rule specifies that Serena^s should notify members of `security` group and any of its subgroups. With the `subgroupof` scope notification definition Serena directly retrieves the entry $M_{\vartheta_L(*,sec)}$, where *sec* is the index of the group *security*, and notifies clients in the groups which have an entry of `1` – which in this case is only the clients of the group `security`.

Notifications with Localised Scopes With localised scopes, the approach outlined for identifying all groups applicable is not as straightforward. Finding all groups applicable to scopes such as *lpeerof* and *lvisibleto* requires extra computations. Suppose a notify construct defines a notification scope of `notify lpeerof lev1`, which is applicable to the *Safari tours* hierarchy. The groups to be notified in this case are clients directly tied to the `lev1, lev2` and `lev3` groups, and excludes groups in other hierarchies like *Kimetrica*'s `departments` (compare this in Figure 7.9). One way Serena^s can discern this information is by iterating through all groups and compare their *Level*, which becomes inefficient as the number of groups increases.

To improve this Serena^s uses indexing generated during the encoding process (Section 7.1.4) for these notification scopes. Remember from Section 7.1.4) that the indices of global and local maximals are stored. For `notify lvisibleto ui`, if u represents the group `ui`, the solution is to retrieve the row entry for $M_{\vartheta_L(u,*)}$ and return the descendants of all row entries of local maximals that are set to 1.

Notifications for local peers. For *lpeerof*, the event manager uses a third index based on the groups at each level per global maximal. Levels are obtained when calculating each group's bit vector in the process outlined in Section 7.1.4. These indices are primarily maintained to capture all peers *within a hierarchy of a main parent group*. For this purpose, a data structure H is maintained by Serena^s is shown in Figure 7.10. The structure H is a hash with the buckets representing levels in the hierarchy: each entry has its key representing a level and the values point to a *peers hash*. The *peers hash* has contains keys as indices of maximals (or main groups) in the hierarchy and values as a binary vector representing all peers of that hierarchy at that level. The entries of the vector correspond to the indices of corresponding elements in the encoded matrix M_{ϑ_L}. Therefore if a rule contains a `notify lpeerof lev1`, then Serena^s event manager retrieves its level `Level = 3` and accesses H at key 3. It finds the entry for `Safari tours` (maximal index 2), retrieving the vector `00000011100000000000000000000000` (which corresponds to the groups *lev1*, *lev2* and *lev3*). The hash H is also significant for the scope-based hashing method, described in the next section.

Summary. To summarise, suppose e, f, u, b are indices corresponding to the groups *employees*, *finance*, *ui*, and $\perp$. The following operations are used to apply notification scopes in rules:

- To calculate `<subgroupof employees>` Serena^s retrieves the groups that are set in the column bit vector $M_{\vartheta_L(*,e)}$. In this case the groups are all descendants of the group `employees` in Figure 7.9.
- To calculate `<supergroupof finance>` it retrieves the groups that are set in the row bit vector $M_{\vartheta_L(f,*)}$, i.e., the ancestors of `finance`.
- To calculate `<private sw>` it retrieves the clients in `sw` group via its unit vector.
- To calculate `<lpeerof lev1>`, Serena^s uses H to retrieve the level of `lev1` within its parent groups and returns the vector representing groups at that index, `{lev1, lev2, lev3}`. To calculate `<peerof lev1>` it retrieves all peers of the index of `lev1` in H.
- To calculate `<visibleto ui>` it retrieves in $M_{\vartheta_L(u,*)}$ the column vectors of the global maximals set to 1. To calculate `<lvisibleto ui>` it retrieves in $M_{\vartheta_L(u,*)}$ the column vectors of local maximals set to 1. Recall that maximal indices are known from the

encoding process. In this case the only local maximal column vector represents all groups under `departments` in Figure 7.9.

- To calculate `<super db>` it retrieves the groups in the bit vector for $\perp$, $M_{\vartheta_{L(b,*)}}$.

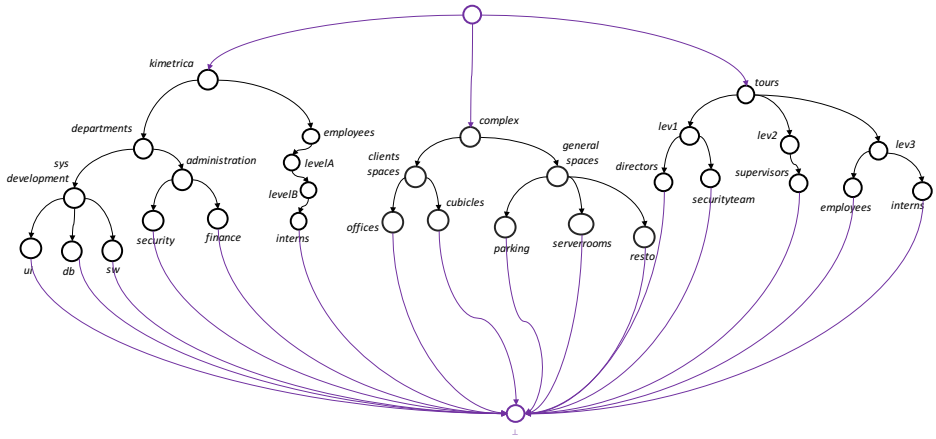


Figure 7.9: Lattice representing the groups in the office complex – The lattice is the result of the operations described in Section 7.1.4 on all the group hierarchies.

Listing 7.4: Scoped rule for accompanied access to server rooms - with notify

```
{rulename: "intern_serverroom_access",
conditions:[
 {$e1: {type:"employee", name:"?intname"}},
 {$e2: {type:"employee", name:"?empname"}},
 {$d: {type:"accessdevice", name: "?dev"}},
 {type:"accessreq", id: "?reqid1", person: "?intname", device: "?dev", time:"?t1"},
 {type:"accessreq", id: "?reqid2", person: "?empname", device: "?dev", time:"?t2"},
 {type:"$test", expr:"(time.hourBetween(?t1, 8, 20)) && time.near(?t1, ?t2)"}
],
scopes:[ "$e1 subgroupof interns","$e2 private levelB", "sysdevelopment supergroupof
  ↪ ($e1 & $e2)", "$d subgroupof serverrooms" ],
actions:[
 {assert: {type: "accessrep", reqid:"?reqid", allowed: true, time: "?t1"}}
],
notify: [ "subgroupof security"]
}
```

With the notification constructs, Serena^s uses a combination of indexing and binary operations similar to those in Section 7.2.1 to determine which groups to notify.

7.4 Scope-based Hashing (SBH)

In this section, we present an improvement to the matching efficiency of the scope-aware RBS Serena^s. The technique involves an inventive optimisation to the Rete algorithm during the matching process.

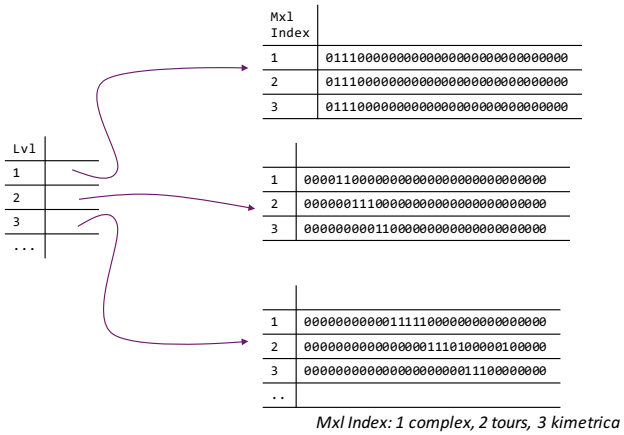


Figure 7.10: Hash H for calculating lpeerof – The primary indices represent the levels of the hierarchy, while the secondary key represent the indices of maximals in M_{ϑ_L}.

The scope-based hashing or *SBH* approach utilises scoped hash tables and group metadata in asserted facts to efficiently cherry-pick compatible data that is relevant for computing joins in the Serena^s engine.

7.4.1 Improving Scope Test Performance

Consider Algorithm 7.2, that showed the left activation process with scopes. With the iteration starting from line 3 that iterates over all the facts from the alpha memory, it is clear that with each left activation of a beta node, scope checks **are still performed on every fact in the alpha memory**, regardless. This shows that the approach performs encoded tests on every fact to determine their compatibility.

To this end, we investigate ways how to improve the efficiency of the process. In the dissertation in [Doo95], Doorenbos stated that for a variety of general-purpose applications where a large number of rules are affected by an assertion, left activations occur often during the matching process stage in Rete networks. Due to the sharing of nodes in the network, a fact that affects a large number of rules when asserted triggers one right activation of a shared join node between the alpha and beta networks. However, as the data propagates down the network from that node, it causes multiple left activations of other nodes, and the number of such left activations increases with the number of rules if there is high node reuse in the system.

We present an approach that makes such activations in Rete networks more efficient, by introducing the scope-based hashing algorithm (SBH). We revisit the office complex scenario to explain the concepts.

Example with the Office Complex Scenario

In the graph of the *InternServerRoomAccess* rule shown in Figure 7.8a, when the `accessrequest` is added, it triggers a right activation in beta node $b1$ (following the red dashed lines in the figure). If the token passes the tests, then it is sent to $b2$ (Figure 7.8b), where it causes a

left activation. If it passes the tests for $b2$ it will also cause a left activation in $b3$ where it will wait for the next request (Figure 7.8c). Once that request arrives and passes the tests the token will trigger a final left activation in $b4$. The sequence shows there are several left activations in the graph representing one *InternServerRoomAccess* rule.

Let us take the state in Figure 7.8c when the second access request from Zak has been received, and the resulting token has passed tests in node $b3$. The new token is sent to $b4$, show in Figure 7.10 (reproduced from Figure 7.8d for convenience) and it causes a left activation. Following the process in Algorithm 7.2, the node will retrieve all items from the the alpha memory $a3$. In the scoped approach the scoping module will then perform scope checks on each of these `device` items for compatibility according to the test `<$d subgroupof serverooms>`. Of course, this is not desirable because as the number of devices increase, the tests become even more expensive to perform.

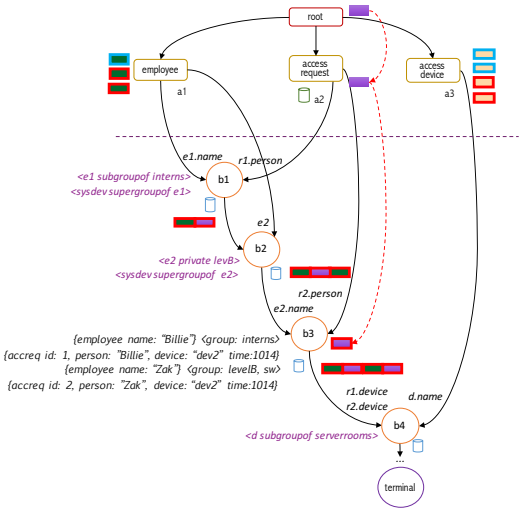


Figure 7.10: Traversal of token to node $b4$ from a second access request in Figure 7.8.

A better approach would be for the scoping module to be able to *immediately retrieve all the items in $a3$ that are compatible with the scope check defined in the node $b4$*. In this case, the module would, according to the defined scope, know that it needs to only retrieve the facts that are relevant. The module would thus retrieve facts that 'the group serverrooms is a supergroup of' from the relevant alpha memory. Enforcing this would require the alpha memory to change the means in which it stores the facts that it receives. SBH enables Serena^s to provide this by *restructuring the alpha memory to use hashing based on defined groups.*

7.4.2 Group Hashing the Alpha Memory

Alpha memories in Rete can be viewed as nodes that store facts of a particular type, e.g., the memories of `employee` and `accessdevice` nodes in Figure 7.6. One of the purposes of alpha memories is to supply beta nodes with fact items. As event data is added to the engine, the cached alpha memory dataset increases. This effect is more profound in shared heterogeneous rule engines. Hence, although scope-based rule engines offer efficient guards before computing joins, the rule engine still suffers when performing scope checks for every

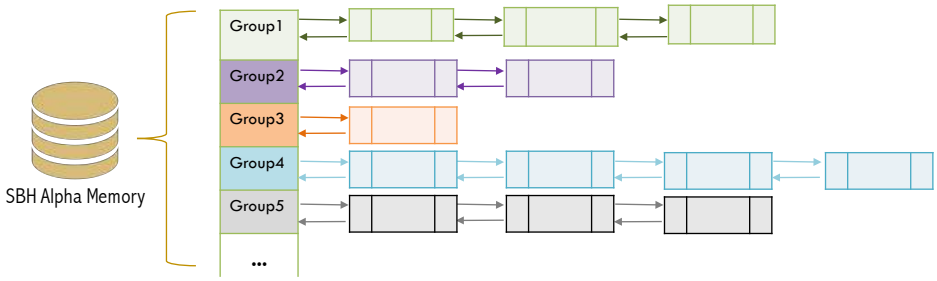


Figure 7.11: Alpha memory with SBH – The node dynamically creates hash buckets partitioned according to facts belonging to different client groups.

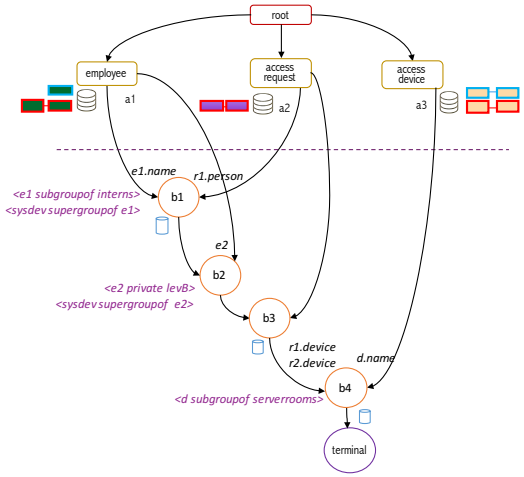


Figure 7.12: The InternServerRoomAccess Rete graph with SBH – The main difference is in the hashed alpha memories where they are partitioned according to groups (colour-coded with outlines) in the hierarchy.

data item added in the alpha memory.

SBH presents an approach that improves this scope-checking process. The technique constructs a hash table that dynamically assigns buckets based on client groups in the group hierarchy (e.g. the groups in Figure 6.1). Each bucket points to a list and each group holds a set of facts of that group. As facts are added to the system SBH assigns each fact to the correct bucket dynamically. For instance, for a device located at the entry point of a server room the fact will be added to the *serverrooms* bucket of the `accessdevice` SBH hash table. The encoding method prefixes group names to make sure groups will be mapped into unique identifiers, therefore each hash bucket will logically contain items of a unique group as in Figure 7.11.

The Rete graph structure for storing facts in SBH alpha memories is shown in Figure 7.12. In this case, the outlines show facts that belong the the same groups linked together. The SBH algorithm further modifies the `getFacts` method of a left activation (line 2, Algorithm 7.2) to allow the alpha memory to accept a list of groups as an argument. The new method will collect all data in the groups via direct access from the hash table,

explained in the next section.

7.4.3 Matching with Scope-based Hashing

Matching in scoped engines involves updating the beta network with scope guards that check compatibility of left and right inputs. SBH introduces a way to efficiently determine which fact items are compatible with an incoming token at the left input of a beta node to be subsequently used in the join test of the node. SBH modifies the scoped left activation process from Algorithm 7.2 to the one shown in Algorithm 7.5.

Algorithm 7.5 BetaNode Scoped Left Activation with SBH

```
1  function hashedBetanodeLeftReceive(node : n, token : t)
2      groupsCode ← this.calculateCodeFromScopeGuards(n.scopeTests, t)
3      groups ← this.getGroupsFromCode(groupsCode)
4      scopeFacts ← n.alphaMemory.getFacts(groups)
5      for each fact f in scopeFacts do
6          if n.joinTestPassed(t, f) then
7              t_new ← n.createNewToken(t, f)
8              n.sendTokenToChildren(t_new)
9          end if
10     end for
11 end function
```

Example Take the example of the state of the Rete graph in node $b4$ as presented in Figure 7.12. The scope guard specifies that Serena^s will check if the device `$d` is located at a subgroup of the `serverrooms` group. The token triggers a left activation on node $b4$ as usual. Instead of performing the check with every `device` fact in the alpha memory $a3$, SBH retrieves the matrix codes for all the subgroups of `serverrooms` via `calculateCodeFromScopeGuards` in Algorithm 7.5, line 2. This process is described next (with the scope guard `<$d subgroupof serverrrooms>` of beta node $b4$).

Calculating SBH Vector V_n Let n be the total number of elements of a row in the encoded matrix M_{ϑ_L} (Figure A.1). `calculateCodeFromScopeGuards` constructs a bit vector V_n with all elements having a bit value `0`. Conceptually, the V_n represents all groups in the hierarchy. At this point, no groups have passed the scope check (all have `0`s as entries in V_n). SBH then performs operations that assign a group element `1` iff it satisfies the scope test for the current node. For this case, SBH will use the test `<$d subgroupof serverrooms>` (which is part of the argument `n.scopeTests`) to retrieve the groups which `severrooms` is a supergroup of. If s is the index that corresponds to the group *serverrooms* in the matrix, then the method thus retrieves the column vector
$M_{\vartheta_L(*,s)}$= `00000000000010000000000000000001`
which represents all the subgroups of the group `serverrooms`. Because the method *retrieves all groups that apply to a particular scope definition*, this approach is equivalent to the process outlined in the previous Section 7.3.2 for notification scopes. SBH therefore reuses those specifications for line 2, when calculating the code from the scope guards at a particular node.

Retrieving Groups G in V_n The next step in Algorithm 7.5 is to retrieve the corresponding groups in method `getGroupsFromCode` line 3 which will then be used to retrieve the items per group in the alpha memory. The method `getGroupsFromCode` retrieves the groups names which have a `1` in V_n (excluding $\perp$) as a set G. In this case G=[*serverrooms*]. Retrieving G from V_n should be relatively easy since the groups are the labels in the matrix.

Retrieving Facts of G Remember that the `getFacts` method of the alpha memory now accepts the groups G as an argument. The method uses the alpha memory's internal SBH table introduced in Section 7.4.2 to access the fact items that are pertinent to the beta node $b4$. The alpha memory will thus retrieve the facts residing in each of the G buckets. Essentially, the facts retrieved are *a subset of all of the items in the alpha memory* thus avoiding computing scope checks (and subsequently the join tests) on *all* of the fact items residing in the `accessdevice` alpha memory. The rest of the code in Algorithm 7.5 proceeds normally by iterating through all the retrieved items and performing the normal join tests for the node.

7.4.4 Advanced Issues in SBH

Evaluating Scope Expressions

In reality, a number of nodes will have multiple scope expressions in one node: an example is node $b1$ which not only has `<$e1 subgroupof interns>` but also `<sysdev supergroupof $e1>`. Furthermore, scope tests can contain complex expressions – to specify "an access device that is in the `sysdevt` or `administration` departments, or the office of a `levB` member," the expression becomes `<($d subgroupof (sysdev | administration)) | $d private levB>`

One option to compute such expressions is to repeatedly call `calculateCodeFromScopeGuards` on each scope test, store multiple vectors of V_n and send their groups to the alpha memory to retrieve the items needed for a beta node's join computations. A more efficient way that SBH uses is that it performs reductions using bitwise operations, given every V_{ni} bit vector result of each scope test i of a beta node. Thus, the result is used by method `calculateCodeFromScopeGuards` to construct V_n for the scope test of beta node.

For the example `<($d subgroupof(sysdev | administration)) | ($d private levB>)`

```
<($d subgroupof (sysdev | administration)) | ($d private levB )>
= <($d subgroupof sysdev | $d supergroupof administration)
| $d private levB >
= ((00000000000000000000000100011101 | 00000000000000000000001001100001)
| 00000000000000000000000010000001)
= (00000000000000000000001101111101 | 00000000000000000000000010000001)
= 00000000000000000000001111111101
```

Note that the vector V_{ni} of a scope test of `<private` u`>` is the unit vector of u in M_{ϑ_L}. The result V_n is returned from the method `calculateCodeFromScopeGuards`. The next step computes the set G via the `getGroupsFromCode` method in line 3 of Algorithm 7.2, which in this case the set G corresponds to the groups with the bits in V_n set to `1`. G thus evaluates to `[administrative,sysdevelopment,levB,finance,security,`

`sw,db,ui,interns]`. Subsequently, the SBH algorithm proceeds normally as outlined in Algorithm 7.5, retrieving the facts of groups in G that passed the scope check.

SBH and Facts with Multiple Groups

One limitation brought about by the SBH scheme is in cases where a fact can be assigned to multiple groups. An example is a senior employee that is working on a project that teams up the `administration` and the `sys development` units. Facts for such an `employee` will be tagged with `[administration,sysdev]` group metadata. In the alpha memory of the `employee` node, SBH will assign the fact into the `administration` as well as `sysdev` buckets. During retrieval with the alpha memory's `getFacts(`G`)` method this will lead to duplication if the scope guard of a child beta node requires items from facts of the two groups. A common fix is to check for duplicate items whenever a fact from multiple groups is retrieved. This method however reduces the efficiency of the SBH algorithm execution, due to the repeated lookups that each check needs to perform to guarantee retrieved items are unique.

To remedy the problem the SBH algorithm exploits a property that most data-driven rule engines already enforce: each asserted fact has a unique handle or ID. If an engine fulfils this property then SBH uses the `getFacts(`G`)` method efficiently by employing a temporary hash with fact IDs as keys. To find whether a retrieved item is a duplicate, it checks the fact ID in the hash: if it exists then the fact has already been retrieved and is not added to the values to be returned, if not then its ID is added to the hash. This process is improved by adding the requirement that only multigroup facts require this special check. With this technique SBH provides a more efficient way to avoid duplicates when retrieving facts tagged with multiple groups from the SBH table.

Hashing Thresholds in SBH

Like most hashing techniques, SBH is becomes more effective as the number of elements increases, avoiding the need to perform scope tests in every alpha memory fact. If the items in an alpha memory are few, however, then it can be expensive to maintain hashed memories as opposed to the normal alpha memories. Therefore it is suitable for a rule engine to be able to flexibly use the SBH scheme to achieve high efficiency.

SBH provides a group hashing threshold to be set, which will activate and enable alpha node hashing only when the number of groups is above the threshold. The hashing threshold may be a configuration parameter set by a client with the *administrator* role, becoming transparent to rule designers and the RKDAs that utilise Serena^s.

Another type of hashing threshold can be set for all computations that have `length(`G`)=t`, where in extreme cases `t` is the total number of all groups representing clients, and in others `t` represents a small subset of these groups. A higher number of Gs having all client groups indicates that SBH scheme tends to degenerate to a generic rule engine without scope semantics. In such cases, the traditional approach can be faster than the scoped approach with SBH. Hence, Serena^s can incorporate such an evaluated code length threshold that turns off SBH checks in join nodes.

7.4.5 Summary: Scope-based Hashing

In the vanilla implementation, the scoped engine of Serena^s has to search through all facts in a particular alpha memory. This gets particularly inefficient as more facts are asserted in the system's lifetime. Using group-hashed alpha memories limits this retrieval of elements

to those that can actually pass the scope test. With this setup, Serena^s can take full advantage of left activations to optimise lookup. This optimisation does not affect the set of complete production matches that will be found; the semantics of the scoped Rete algorithm remains unchanged.

7.5 Maintainability of Scoped Rules and Other Issues

This section discusses effects of maintaining the scoping method on the runtime execution of the rule engine in Serena^s.

7.5.1 Retraction and Modification of Facts

The retraction of a fact in the Serena^s framework closely follows that of the normal semantics of retraction in Serena presented in Section 4.4.5. There, the *tree-based deletions* approach of having references in asserted facts in the Rete network's alpha memories that point to children in the beta network constituting beta tokens are used. In this scheme, a fact can be deleted by following these references and deallocating them up until the terminal node, if necessary. This is beneficial as it bypasses the join tests as before, but more importantly in this case it also avoids performing the scope tests as well (which would otherwise need to be performed).

7.5.2 Changes in Group Structure

The main argument for an encoding of the structural elements in client logical and physical organisations is efficiency of various operations. This feature also becomes its weakness because even though *client structures are expected to seldom change, modifications tend to have some effects on the encoding* through the addition and removal of groups.

The normal way when dealing with changes is for the framework to re-encode the matrix from the ground up. This of course implies that the system needs to be stopped and reinitialised. However, as noted in [Aït+89] if there is a change to the underlying poset at an element a then *only the elements that subsume a via the relation need to be re-encoded.* In Serena^s, not only does the encoding require to be changed, but also the state of the graph – it will be inconsistent because the tokens in the beta network may need to be reevaluated.

Therefore, Serena^s takes a two-pronged approach to updates. When encoding the matrix during initialisation, Serena^s uses the technique described in Section 7.1.4 with all groups. For dealing with changes at runtime, Serena^s provides a more restricted functionality – to reduce the effect of the change in the Rete network's existing tokens. The framework allows piecemeal changes to the hierarchy, through the runtime addition and removal of groups only if the groups *are/will be minimals in the underlying poset* (definition in Appendix A.1.1). This way, the required changes to the encoded lattice can be easily added or removed to the latter rows of the encoded matrix.

Take an example for the matrix in Figure 7.3. If a *sales* group representing a new department is added to the hierarchy as the child of the `departments` group, the matrix only needs to be updated as shown in 7.13a. The process to update the Serena^s encoding is:

- Add a row and column in the matrix before the entry representing $\perp$.
- Zero-fill the added column with `0`s.

- Duplicate the row for its parent *departments* onto the new row (fulfils the transitive property).
- At the last index s representing the new group *sales*, set $M_{\vartheta_L(s,s)} = 1$ (fulfils the reflexive, antisymmetric property).
- Add the group into the peer hash H as outlined in Section 7.3.2.

For group deletion, a similar reverse process occurs. Deletion further requires that the relevant tokens in the Rete network should be deleted as well. This deletion is performed using *command tokens*, which are meta tokens that are percolated through the scoped Rete network with the intention of deleting a group and its references in the network. The token `{remove-group "sales"}` is sent to the root node which will forward to all the alpha nodes. When an alpha node receives the token, it will search its memory for facts that have this group and will delete them (thus deleting all the child references in tokens via tree-based deletion). The alpha node will also forward the token to its children. If a beta node receives a `remove-group` command token, it will only forward to child nodes if it contains a scope test with this group. Eventually, a command token received on a terminal node indicates that the rule associated with this node should be deleted, because its scope test is now undefined. Thus the rule containing the scope check with the group is acquired from the terminal node and is scheduled for removal. The group is also deleted from the peer hash H via its level and main parent group. A final step for removing all clients that belong to that group is performed by the event manager and finally deallocates any socket connections from the server.

SBH-related Changes

If the scope-based hashing method from Section 7.4 is used, then the alpha memories need to be updated according to the changes of a group. One way is to deal with addition of a new group is to use the command token `{add-group "sales"}` to traverse the alpha network of the Rete graph. Any memory of an alpha node with SBH will then add a bucket for the group in its hash. Alternatively, the entry can be added by the alpha node dynamically when a fact of the new group is received by the node.

For deletions, the network will use the aforementioned command token `{remove-group "sales"}`. Instead of an alpha node searching its memory for facts with the indicated group, using SBH the group can directly be acquired from the hash and all the items removed using the tree-based deletion. After this, the hash entry for that group is finally removed. This actually improves the efficiency of removing groups in SBH as opposed to the normal scoped setups. Subsequently, the deleted elements will be removed from the beta network using the same manner as explained in the previous section.

7.5.3 Negation

Formally, production rule systems such as CLIPS and Jess do not explicitly refer to incoming data as events (as in, say, active databases), but can assimilate them into memory as fact assertions. In Serena, the left-hand side semantically models event conditions and the right-hand sides models their actions and/or effects. In essence, this kind of reasoning has influenced the support of negation in the engine.

Traditional production rule engines discussed in Section 3.3.2 support the classical negation model [Wag03]. In this model, using *not* in rules expresses negation with complete predicates that are subject to a completeness model having a **closed-world assumption**,

	⊤	kime	empl	dept	levA	admi	sysd	levB	fina	secu	sw	db	ui	inte	sale	⊥
⊤	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
kime	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
empl	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
dept	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
levA	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
admi	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
sysd	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0
levB	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0
fina	1	1	0	1	0	1	0	0	1	0	0	0	0	0	0	0
secu	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0
sw	1	1	0	1	0	0	1	0	0	0	1	0	0	0	0	0
db	1	1	0	1	0	0	1	0	0	0	0	1	0	0	0	0
ui	1	1	0	1	0	0	1	0	0	0	0	0	1	0	0	0
inte	1	1	1	0	1	0	0	1	0	0	0	0	0	1	0	0
sale	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0
⊥	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(a) Addition of a group sales to the encoded matrix M_{ϑ_L}

	⊤	kime	empl	dept	levA	admi	sysd	levB	fina	secu	sw	db	ui	inte	sale	⊥
⊤	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
kime	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
empl	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
dept	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
levA	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
admi	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
sysd	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0
levB	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0
fina	1	1	0	1	0	1	0	0	1	0	0	0	0	0	0	0
secu	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0
sw	1	1	0	1	0	0	1	0	0	0	1	0	0	0	0	0
db	1	1	0	1	0	0	1	0	0	0	0	1	0	0	0	0
ui	1	1	0	1	0	0	1	0	0	0	0	0	1	0	0	0
inte	1	1	1	0	1	0	0	1	0	0	0	0	0	1	0	0
sale	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0
⊥	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(b) Deletion of group sw from the encoded matrix

	⊤	kime	empl	dept	levA	admi	sysd	levB	fina	secu	db	ui	inte	sale	⊥
⊤	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
kime	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
empl	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
dept	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
levA	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0
admi	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
sysd	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0
levB	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0
fina	1	1	0	1	0	1	0	0	1	0	0	0	0	0	0
secu	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0
db	1	1	0	1	0	0	1	0	0	0	1	0	0	0	0
ui	1	1	0	1	0	0	1	0	0	0	0	1	0	0	0
inte	1	1	1	0	1	0	0	1	0	0	0	0	1	0	0
sale	1	1	0	1	0	0	0	0	0	0	0	0	0	1	0
⊥	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(c) Final matrix after addition and deletion

Figure 7.13: Showing the effect of modifications to the hierarchy on M_{ϑ_L} – Only allowed at runtime if the group to be added or deleted is a minimal in the underlying poset.

i.e., the engine assumes that all facts are available *a priori* and so negative patterns can be evaluated immediately. A problem with this model is that this *a priori* requirement is not guaranteed by the distributed streaming semantics of the rule engine in Serena. Consequently, the engine does not intrinsically support the semantics of negation when applied to streamed events in the framework.

Other classical approaches circumvent these problems in several ways. A number of rule engines that consume events use temporal operators that allow negation on timed events only within time intervals or *windows*, as Drools in STREAM mode [Bro09]. The systems in [WBG08] and [Ber02] semantically conceive facts and events separately within the engine, opening avenues for negation semantics and garbage collection in different scenarios.

The classical negation model as implemented in [Doo95] is supported in Serena only in specific cases where the engine is not intended for streamed data (i.e., where completeness is satisfied)[10]. In these cases, Serena expresses *not* by essentially testing the absence of an item in the fact base, as in similar non-streaming rule engines such as Drools in CLOUD mode. A negated condition in a rule will create a negative beta node that behaves similar to a beta node. The main difference is that a negative beta node will forward tokens to its child nodes only if its join computation fails.

In its implementation, however, Serena^s does not support having scope definitions on negated conditions in rules. This is because they conceptually represent the absence of a fact – therefore a non-existing fact does not contain the metadata that the scoping module can use to perform scope tests. A promising direction is in *negated scope expressions* where the scoping module could perform a scope test as usual, but subsequently allow the join computation on its beta node to be performed only if the scope test (boolean result) fails. But more research needs to be done regarding the effects of the technique on the performance of scope tests: because there tend to be a larger number of items to retrieve, the cost of creating and performing these operations may greatly hinder execution of the inference engine cycles. We discuss related issues in future research avenues in Section 9.3 about cost estimation models.

7.6 Requirements Revisited

Having presented the inner workings of the Serena^s framework, this section evaluates the goals of the scope-based approach in supporting heterogeneous rule based systems.

7.6.1 Evaluation of Requirements

Supporting client rules in classical rule-based systems in heterogeneous contexts was shown to exhibit problems identified in Section 5.2.2, with the main concern being essentially enforcing reentrancy in such rule engines. A related problem was the support for capturing community knowledge in client rules. The requirements set forth in Section 5.3 were therefore meant to provide a structure in which such problems can be tackled.

Serena^s as introduced in this chapter enforces these requirements to support heterogeneous rule engines in the following ways. They are presented in the same manner as the related work back in Table 5.9.

- *Metadata model* – Serena^s provides a Web-based framework that is suitable for supporting heterogeneous clients. Its scoping model hides the metadata required by the scoping

[10] This is applied to some of the rules in the Miss Manners' benchmark as discussed in Section 8.5

module and the client library about rules, clients and asserted facts into the rule engine on the server. As discussed in Section 7.2.1 the main reason is to offer high-level constructs that aid in managing the shared knowledge of heterogeneous systems. Serena^s does not require one to pollute rules by manually encoding such logic but instead relegates this to the engine. The model is non-intrusive with respect to rule activation semantics, in the sense that if the scoping functionality is removed from the framework or disabled, then the underlying semantics of rule engine execution is preserved.

- *Grouping model* – The framework places grouping of clients according to their organisation and their representation at the heart of its scoped approach. The groupings can be used within scoped rule definitions to specify the data that the engine should use when performing the rule computations. Clients with special roles add their logical or physical organisational structures to the engine. Thereafter, other clients are identified according to the groups they belong, and facts are transparently tagged with scoping metadata. Section 6.1 specified the need for these heterogeneous groupings and relationships (unlike in approaches that use rule modules and its variations) and Section 7.1.4 presented its formalised model using concepts from order theory. Hence, in Serena^s, physical or logical groupings of clients are captured and modelled internally in the engine as a hierarchy that is used to discriminate or distinguish between instances of data from different clients.

- *Execution model* – To realise the addition of scopeful constructs in a rule-based system would, when done naïvely, negatively impact the runtime computation of the inference engine. As mentioned in Section 7.1 this is mainly due to the possibly large set of possibilities that need to be verified when determining compatibility. Scoping in Serena^s (and by extension using SBH) provides an internal representation is that precomputed efficiently as an encoding that is be used when processing constraints and to enforce reentrancy during rule engine execution. In this regard, the next Chapter 8 evaluates the efficiency of this approach.

- *Notification model* – Once rules are activated in a multi-user RKDA setup, there is need for a way to directly identify the users that should be notified. As explained in Sections 6.5 and 7.3.2, the Serena^s framework exploits the aforementioned concepts of metadata about clients, their grouping structures and the efficient computation of scopes to limit the distribution of client notifications. Notifications in Serena^s are flexible expressions that can define constraints to notify a group, subgroup, or even direct clients.

The suitability of Serena according to the applicability of the heterogeneous requirements is presented in table 7.14, compared to the existing state-of-the-art as discussed in back in Section 5.4. The table shows that Serena^s provides suitable abstractions and models that can tackle the problems brought about by heterogeneity in rule-based systems.

7.7 Chapter Summary

This chapter presented the implementation of scoping in the Serena^s framework. The importance of using an efficient encoding to model organisation of clients was discussed, and the use of the bit-vector encoding method was consequently chosen. A further improvement known as scope-based hashing was presented to improve the efficiency of determining compatible data when computing joins during left activations, making the engine more efficient

Table 7.14: Evaluative comparison of Serena^s scope-based approach with other approaches

	Metadata Model	Grouping Model	Execution Model	Notification Model
Multitenant Databases				
Shared schema/tables	★★★★☆	☆☆☆☆☆	★★★½☆	☆☆☆☆☆
Event-based Systems				
Visibility	★½☆☆☆	★★½☆☆	☆☆☆☆☆	★★★★☆
Rule-based Systems				
Entry Points	½☆☆☆☆	½☆☆☆☆	★★★☆☆	★½☆☆☆
Peers	½☆☆☆☆	★☆☆☆☆	★★☆☆☆	★☆☆☆☆
Rule Modules	★★☆☆☆	★★★½☆	★★☆☆☆	★☆☆☆☆
Scopes	★★★★½	★★★★★	★★★★★	★★★★½

when processing scope tests. More advanced issues regarding the extent of how the framework supports the addition and removal of facts, rules and client groups were discussed. Throughout the chapter the motivating example of the office complex system was used to clearly explain the concepts presented, but other scenarios with similar characteristics can also be supported by the framework.

The chapter then concluded by revisiting the stated requirements of a heterogeneous framework for supporting the processing of events in RKDAs. Through the scenario and by comparisons against the requirements, the scope-based approach of the Serena^s framework was observed to provide suitable abstractions and models that support the execution of scoped rules in RKDAs backed by a common rule-based system.

8 Evaluation

Reasoning draws a conclusion, but does not make the conclusion certain, nor does it remove doubt, unless the mind discovers it by the path of experience.

Roger Bacon, *The Opus Majus of Roger Bacon*, 1897

This chapter evaluates the scope-based approach of this dissertation through its concrete implementation, the Serena^s framework. The goal of the evaluation is to examine the computational efficiency of the Serena^s framework's rule engine. The approach was centred on investigating the following points:

1. Whether a scope-aware rule engine as presented has significant computational benefits over a traditional rule engine in which scoping is manually encoded in the rule (Sections 8.2 & 8.5),
2. Comparison of the integrated scoped technique of this dissertation with one that fosters total isolation through the independent rulebook approach (Section 8.3),
3. Whether a scoped rule engine with SBH experiences significant improvements in efficiency compared to one without SBH (Section 8.4).

For each evaluation we begin with an introduction of the evaluation cases and proceed to illustrate the configuration, method and results of the actual experimental process. We then complete each section with a discussion of the observed results. In the cases we use a comprehensive simulation of a scenario that is functionally equivalent to the motivating example presented in Section 5.2.1. The simulation is used to provide a comparative evaluation with the classical approaches.

8.1 Evaluation Scenario

In this section we explain the setup of the evaluation, presented as published in [KBD15] and [KRD17a]. To highlight the requirements that such a system should meet, we present

the scenario that is akin to Cloud service providers for monitoring security systems [FYW15]. In this case, the provider is a service that monitors and logs requests in a university-wide security access system.

8.1.1 Example: University Services Access Control

Scenario Narrative

The 3 universities of the Association of Universities in Brussels (Universitaire Associatie Brussel) have passed a resolution that requires monitoring accesses of students and staff all over their campuses. Accesses that deviate from the policies in place should immediately be reported to the responsible security bodies for quick responses.

All the universities have proximity ID-card readers at major access points in their campuses. Students and staff wear identification badges that contain their issued ID cards upon registration (technically, the ID cards are contactless smart cards also known as proximity cards). To this effect, students and staff scan their issued ID cards to gain access to various locations in the campuses.

University Security Policies

We illustrate some of the policies for the universities below.

- **Policy 1** *Student Classroom Access*

  All students at all levels have access to classrooms during class times on weekdays

- **Policy 2** *Car Parking Access*

  Only registered student and staff cars are allowed entry to underground parking on their campuses

- **Policy 3** *External Staff Code Access*

  External staff are allowed access only if they have a pre-authorised access code, issued by higher level administrative staff

For this motivating example we have enumerated a number of policies. These policies are generally applicable to the whole university. Universities usually have an overall arrangement of organisational structures, roles, and functions. From a security viewpoint, we present the following university structure focusing on three core planes:

- The physical structures collocated within the university, consisting of a number of buildings and other edifices, such as the campus restaurant, parking and bank ATM.
- The different faculties of institutes of research that perform specific academic functions such as teaching and promoting innovation.
- The various administrative bodies that offer professional support services such as management and overall strategic planning. These include core personnel, part-time or external personnel.

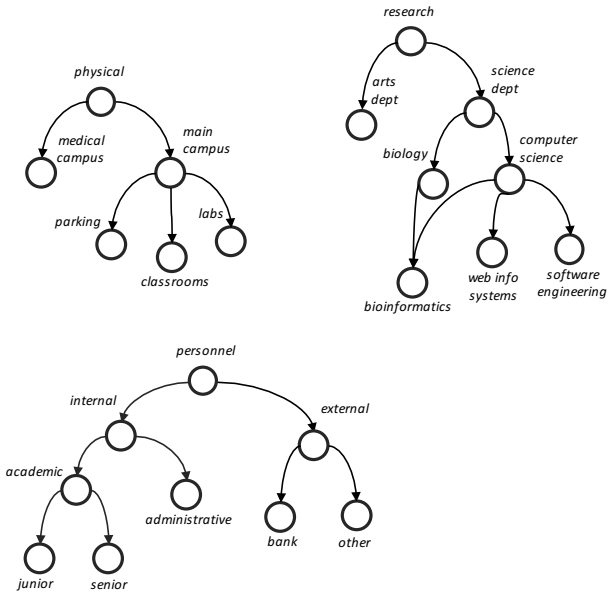


Figure 8.1: Example structures in universities – They are grouped into three hierarchical structures: one based on physical location, one on department, and one on type of personnel.

A simplified structure is shown in Figure 8.1 with the personnel, research and physical structures. The figure shows the various groups as vertices and the relationships between them as edges that depict the overall structure of the university[1].

Students, staff and strategically-placed ID-card readers can be linked to any group at any level of the structure.

Custom Security Policies

In the association, a university has the structure shown in Figure 8.1. One university develops specialised custom policies for its various departments given its layout. To this end, specific departments and units define custom access policies:

- **Policy 4** *Biology students lab access*

  Biology department students are allowed access to all labs of its (sub)departments during the weekends if accompanied by senior academic staff

- **Policy 5** *Campus bank office access*

  Only campus bank employees and consultants have access to the bank back office during working hours

These custom policies apply to different levels of the university structure, for instance within a department and its sub-departments.

[1]The hierarchy diagram is not necessarily a tree, since some nodes can have multiple predecessors.

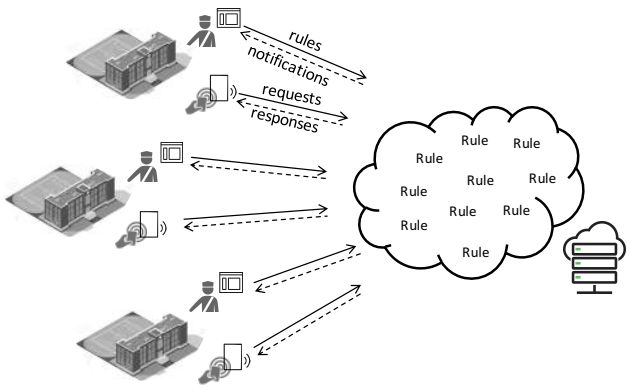


Figure 8.2: Evaluation scenario of monitoring university security – Policies added as rules to the server are used to determine the validity of access requests.

Scenario Modelling

For the experimental simulation, we enumerated around 40 security access policies. The final model contains 3 universities and 61 faculty, administrative and physical groupings (the simplified depiction of structure in Figure 8.1 was derived from this final model).

The final model also contains students, staff and devices belonging to one or multiple groups. The access devices are ID-card readers that are installed at strategic physical structures and academic departments. The staff belong to various administrative personnel categories with different access levels. Both parties (the students and staff) gain access to various locations and at various times by scanning their badges using the access devices.

Security Monitoring Service Provider

A security monitoring service provider provides an RKDA framework for the universities. The service receives all the defined security policies that should be evaluated whenever a request is made. A routine use case of a typical access request is enumerated below, and a general illustration is shown in Figure 8.2.

1. A student or staff initiates an access request to a particular location in the university by scanning his ID-card on an access device restricting access to the location.
2. The device collects the request data (time, badge, location) and needs to immediately send it to the university access device server.
3. The server in turn copies and sends the data to the monitoring service.
4. The service logs the request and instantaneously computes whether the access request is within the defined security policies.
5. The service sends the feedback as notifications to the dashboards of the relevant authorities for prompt analysis.

Student Classroom Access Example
For instance in policy 1, when a student in a university accesses a classroom during class times, the monitoring dashboard would show a status to indicate whether the access is acceptable or otherwise.

Table 8.3: Specifications for the Evaluation Case Server

Specification	Value
Operating System	Ubuntu Server 15.04
Web Server	Node.js v5.6.0
Processor	AMD Opteron 6272
Processor Speed	2.1Ghz
Total RAM	96GB DDR3
RAM Speed	1600 MHz
Allocated RAM	20GB*
Allocated Disk Space	40GB* HDD

* Allocated per Node.js process

8.2 Evaluation: Scoped & Ad-hoc Approaches

In this section we evaluate our scope-based approach by implementing the university scenario. For this evaluation we investigate whether introducing the scoping metadata architecture and semantics within the rule engine as presented in the previous chapters has significant computational benefits over a traditional approach using a vanilla rule engine that requires a manual encoding of the same knowledge.

8.2.1 Setup and Methodology

Experimental Setup

The example scenario was implemented as a simulation running on an event-driven web server. The final application has a total of 61 groups in hierarchies, 40 access rules, and 73 concurrent clients across 3 sample universities. The Serena server runs a Node.js web server, and all clients are connected to the server concurrently through websocket connections managed by the framework. The specifications for the server and the Node.js processes are shown in Table 8.3.

Methodology

The general evaluation of the university scenario focuses on investigating the differences between two approaches:

- The first follows the mechanisms that current rule-based systems expose i.e. the use of vanilla/traditional rules using manual methods such as expression tests, to enforce reentrancy and data discrimination. For this we used a JavaScript Rete-based rule engine for Node.js, JsRete, that was inspired by a similar implementation Nools [C2F14]. We refer to this as scoping using an *ad-hoc* approach.
- The second approach uses Serena's built-in approach that this thesis proposes: using the Serena framework's scoped rules realised in the inference engine's graph. We refer to this as the built-in or *scoped* approach.

The general methodology for the simulations involved splitting the evaluation process into parts:

Analogous simulation – The first part involved performing *one session of the simulation using identical access requests* for both ad-hoc and built-in scoping approaches from predetermined clients (students, staff etc.) in similar locations. This was in order to perform a direct comparison of the performance of both approaches.

Randomised simulation – The second part involved performing a more *extensive randomised simulation*. We randomly designated various clients to different levels within the university hierarchies and generated random access requests (at different times) from those clients. This was significant to come up with statistical observations of the differences between the general performance of the two approaches.

We simulated the analogous and randomised access requests from clients using different access devices throttled in ranges of between 1-5 seconds, to model practical delays in subsequent access requests by clients at access points. Each simulation was running for a total of 12 hours. The aim of these requests was to model real-world access patterns of students and staff from different departments or personnel levels accessing various university locations at different times.

For the randomised simulation, we ran 35 iterations of simulations in each category (scoped and ad-hoc): approximately 420 hours simulation runtime for each.

During both categories of simulations we logged the number of join computations performed in the inference engine's Rete graph, the RSS/Heap memory used and the cumulative number of activations observed by the server. We show the results and discuss the findings in the next section.

8.2.2 Results and Discussion

Results

We present the results in the form of several charts and graphs. In each simulation (i.e. in a single simulation run of 12hrs) an average of approximately 18k access requests were generated. Each graph compares both scoped and ad-hoc simulation categories and illustrates the results of every simulation run recorded.

From the results of the analogous simulation we present Figure 8.4, which shows the cumulative number of join tests performed, rule activations (i.e., firing rules) observed and the memory utilised for both scoped and traditional engines with the same access requests.

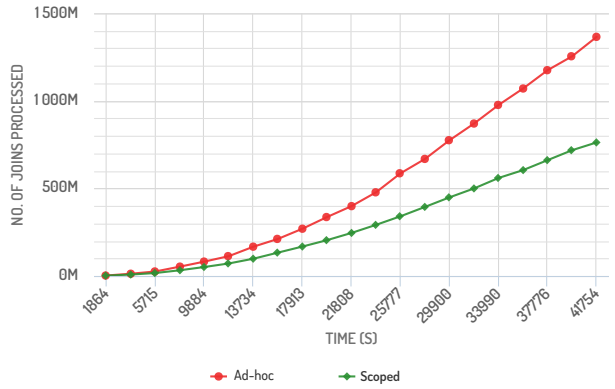
For the randomized simulation the comparative join computations, the rule activations and memory consumption of both approaches are depicted in graphs shown in Figures 8.4a, 8.4b and 8.4c respectively. The results of the runs are then aggregated the data statistically depicted using box plots show in Figure 8.6.

We next discuss the observed results of the different simulations performed.

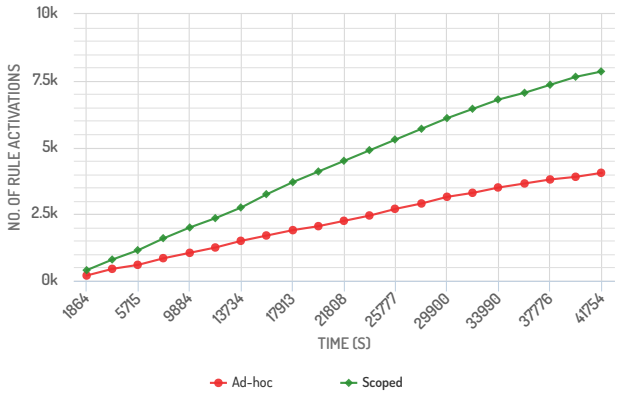
Discussion

Analogous simulation results: From the cumulative results of the analogous simulation we analyse the performance of both scoped and traditional engines using the same configurations (i.e. similar rules and fact assertions).

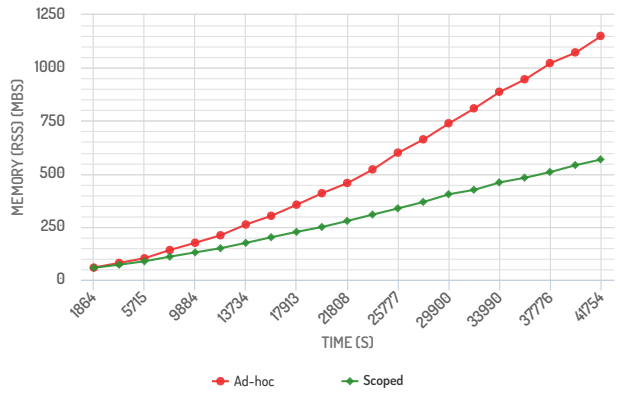
From the graphs in Figure 8.4 we observe that the traditional Rete graph built from manual ad-hoc methods in rules suffers a marked increase in the number of joins computed compared to Serena's scoped engine. The ad-hoc approach spent more time processing the expensive join operations in the engine. Serena's scope tests act as guards that use the encoding to perform efficient tests to restrict incompatible data, leading to better perfor-



(a) Comparison of number of joins processed

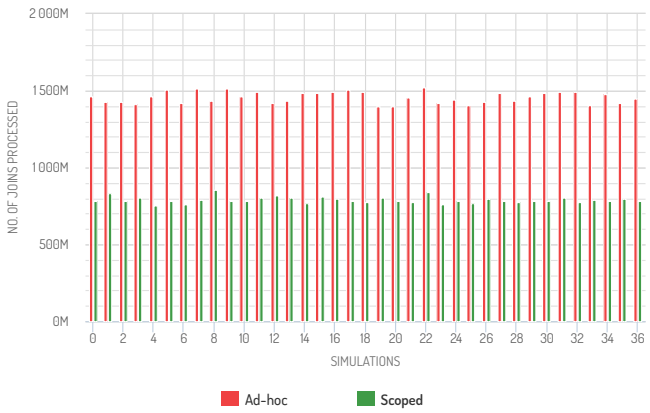


(b) Comparison of number of rule activations



(c) Comparison of memory consumption

Figure 8.4: Cumulative results for the single analogous simulation run of 12hrs – The results show that under similar conditions the scoped approach reduces the number of join computations performed (in favour of efficient encoding operations) leading to a higher number of rule activations observed in (b). The scoped approach also uses a considerably less amount of memory.



(a) Join test results for all randomised runs in both scoped and ad-hoc approaches.

(b) Activations observed over all the randomised runs for both scoped and ad-hoc approaches.

(c) Memory (resident set size) consumed by both approaches over all simulation runs.

Figure 8.5: Results of all the randomised simulation runs show a similar trend as the analogous run: the ad-hoc approach performs more join computations (a) and uses more memory (c) resulting in less activations and slower responses to access requests (b) from clients.

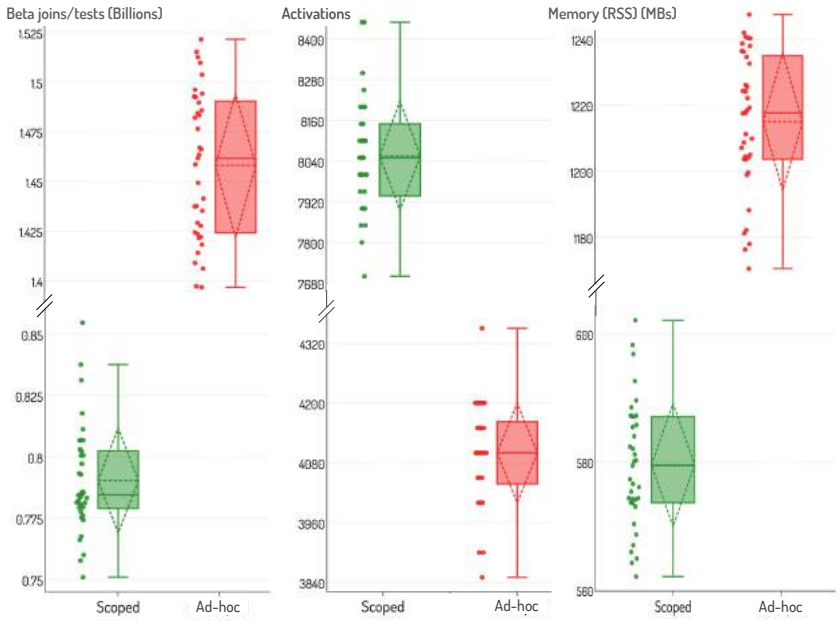


Figure 8.6: Aggregated results of all the simulation runs – The results summarise the trends observed in the previous Figure 8.5. The box plots indicate that a scoped rule engine offers computational benefits compared to an ad-hoc approach.

mance, and consequently to a higher number of activations recorded (by approximately 31%) within the same analogous simulation run.

Randomised simulation results: The aggregated results in Figures 8.5a, 8.5b and 8.5c show evidence of a better overall performance of the scoped engine. Compared to a traditional ad-hoc approach, Serena's scoped engine on average improves the computation of scope tests and total memory consumption, increasing the average number of rule activations of all randomised simulation runs as indicated by the box plot of Figure 8.6.

The reduced memory consumption of the scoped engine is an interesting result: given the matrix encoding of the hierarchy one would expect a higher memory consumption in the scoped approach. This is indeed true when we closely observe the initial memory consumption of the single analogous run of Figure 8.4c. This is illustrated in Figure 8.7. Eventually, however, the ad-hoc approach surpasses the scoped engine after about an hour of asserted facts. A reason for this is the framework internally performs space optimisations of scoping metadata as opposed to the traditional approach. For example, in the traditional approach each fact would contain a dedicated `source` slot object, taking up the same space as other slots like `name` and `age`. Serena utilises the lattice hierarchy labels to tag the data as opposed to creating slots in facts thus saving up space occupied in each asserted fact. Ultimately, the ad-hoc approach leads to a higher memory consumption, due to redundant information and inefficiencies brought about by the complexity of using the manual methods in client rules.

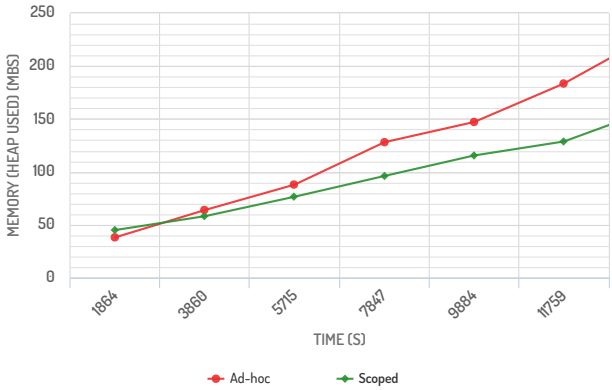


Figure 8.7: Initial cumulative memory consumption of the analogous simulation run – Serena's scoped approach begins with a higher memory usage, but is later surpassed by the engine with the ad-hoc approach.

8.3 Evaluation: Isolated Rule Engine Instances

In this section we perform a separate simulation using the isolated rulebook approach having separated instances of Rete rule engines for the university security scenario. The server configuration remains as specified in Table 8.3. We outline the process next.

8.3.1 Setup and Methodology

For the setup, we used the 3 sample universities from the first simulation and incorporated the same policies. However, in this case we distinctly duplicate and/or split the policies according to the relevant university. As expected, some policies were duplicated; for instance, the `classroom access` protocol that applies collectively to all universities needs to be replicated over the university instances.

For this setup we did not incorporate the use of scopes – and as such groups are not captured for encoding internally. However, the universities themselves do contain a hierarchy of research groups, personnel entities and physical structures. Consequently, the access rules perform data discrimination between these internal structures using normal test expressions.

Similar to the previous setup, we designed and implemented a simulation in which the universities receive the same intermittent access requests belonging to the respective universities. The requests were not randomised; they were modelled in the same way as those in the previous analogous simulation. Instances of the traditional Rete engines were spawned representing the 3 sample universities. Each simulation was modelled with the same intermittent requests limited to 12-hours similar to the previous analogous scope-based simulation.

The aim of this evaluation was to investigate differences of two approaches: on one hand we have the approach of scopes in a shared inference engine, and on the other hand we have the approach having isolated rule engine instances. We thus recorded the resource usage and computations performed during the simulation and compared the results.

8.3.2 Results and Discussion

We present the results recorded in comparison with the results of the built-in scoped approach. Each graph compares the previous built-in scoped and manual ad-hoc results of the analogous simulation augmented with the results observed from this simulation with separated instances. To analyse the approaches comparatively, we show the results of the simulation run. Figure 8.8a shows the total number of join computations recorded in the instances, Figure 8.8b shows the number of activations observed. Finally, Figure 8.8c shows the RSS (Resident Set Size) memory used by the simulation.

From the graphs we observe that our scoped engine approach has similar performance as the approach with isolated instances. Furthermore, the multiple instances generally perform better than the single-instance ad-hoc approach, but they consume much more memory than the other approaches.

In processing join tests, Figure 8.8a shows that the joins computed in the separate instances are less than those of the ad-hoc approach – this is because the ad-hoc approach still needs to perform its joins on all client data in one Rete instance. The scoped approach has fewer join computations because it instead uses the matrix encoding to perform scope checks. Remember that join computations are the most expensive computations in Rete-based rule engines as discussed in Section 4.4.2.

When analysing the activations in Figure 8.8b, we observe that within the same time interval the isolated engine instances observed a much higher number of rule activations than the ad-hoc single instance approach. Furthermore, comparing the results of the isolated instances with the scoped approach showed a slightly less number of activations. In other words, the multiple instances performed computations that resulted in processing nearly the same number of activations as the shared instance scoped approach within the same time interval of 12hrs. Indeed, having separate instances tends to have faster computations than the traditional ad-hoc approach. The limitation of the approach having separate instances manifests itself when observing the memory consumption in Figure 8.8c. Here, the memory used by the separate instances of Rete engines is significantly higher compared to all other approaches. The reason is that the isolated instances have increased redundancy, and this leads to the duplication of the working memory, graph nodes, intermediate memories and activation queues utilised by the inference engines. A single Rete instance takes advantage of structural similarity and temporal redundancy as discussed in Section 7.2.2 leading to reuse of shared resources in the same way as utility computing in the Cloud (Section 2.1.2).

8.3.3 Conclusion

In this evaluation we implemented the university security scenario using isolated Rete instances for each of the universities. We presented our methodology of having policies of each university implemented in its own instance using the same data as the analogous run in Section 8.2.1 . Within a fixed time interval we recorded the results and compared them with the scope-based and ad-hoc approaches.

The results showed that the separate instances exhibited, comparatively, the same performance as our scoped approach in terms of the rule activations processed but outperformed the traditional ad-hoc approach. The main cost of the approach with separate instances is however the high amounts of memory utilisation due to duplication of resources.

From the analysis of the results we conclude that in terms of processing the scoped shared instance and separate engine instances both offer similar performance when dealing with heterogeneous data. Furthermore, we observe that even though the use of separate

Rete engines has better performance than ad-hoc methods, the technique of having separate engines suffers from higher usage of resources – especially as more tenants are added to the system. In comparison, our scoped approach enjoys both the benefits of good performance that utilises less memory during execution. Sharing of rules in one instance further allows Serena to exploit community knowledge, unlike in separate instances.

8.4 Evaluation: Scope-based Hashing

This section evaluates the Scope-based Hashing (SBH) approach. The main focus is to find out whether a rule engine with SBH experiences significant improvements in efficiency compared to the 'vanilla' scoped approach. The evaluation was based on the complete university security scenario staged introduced in Section 8.1.

8.4.1 Setup & Methodology

We performed our evaluation in a setup consisting of a web server running Serena^s. The server was similarly configured as in the previous simulations.

In this simulation, clients and devices were initialised and connected to the rule server via WebSockets. Similar to previous evaluations the mobile clients were configured to generate access requests at intervals of 1-5 seconds and devices received reactive feedback. A security console received push-based updates of accesses to entry points. In this simulation, each access request was randomised, with a random client belonging to any group(s) making an access request at a device from a random location in the university hierarchy.

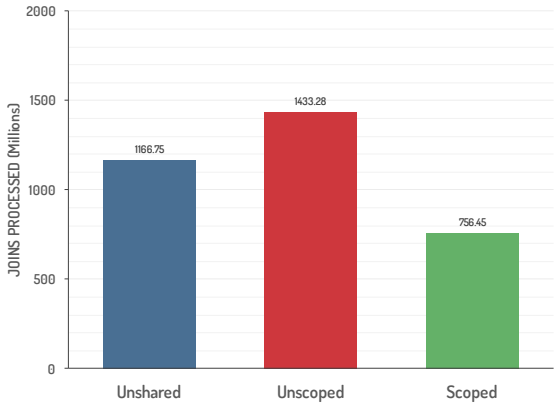
We split the experiment into two test categories. One category included running a 'vanilla' scoped rule engine, and the other involved the Serena^s rule engine with the scope-based hashing algorithm (Section 7.4): i.e., Serena^s without SBH and Serena^s with SBH. The number of simulation runs for this case was increased to a total of 62 sessions for each category, with one session running for a duration of 12 hours. The total experiment therefore spanned 124 sessions and 1488 hours runtime.

8.4.2 Analysis of the Results

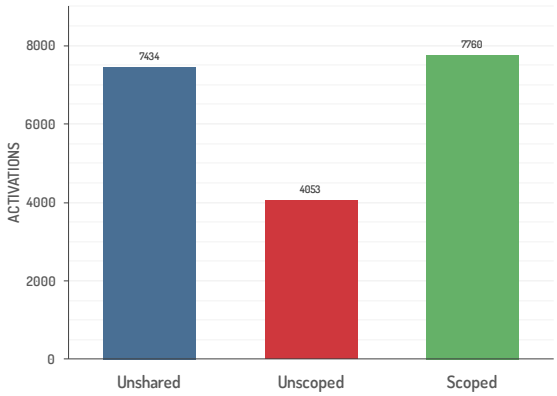
During the experiment the activation times (comparable to response time) and the memory used were logged and compared. The results for this scenario are charted using graphical bean charts that show the quartiles as well as the density estimates in Figure 8.9a. It shows the results of the activation times of the vanilla scoped engine without SBH and the rule engine with SBH. Here, rule activation time is the time it takes the engine to perform a matching process, between assertion and rule activation (Section 4.4).

The chart shows that, on average, the vanilla scoped rule engine showed higher rule activation times than the engine with SBH. SBH exhibits an advantage over the vanilla scoped approach, with reduced rule activation times of up to 80%.

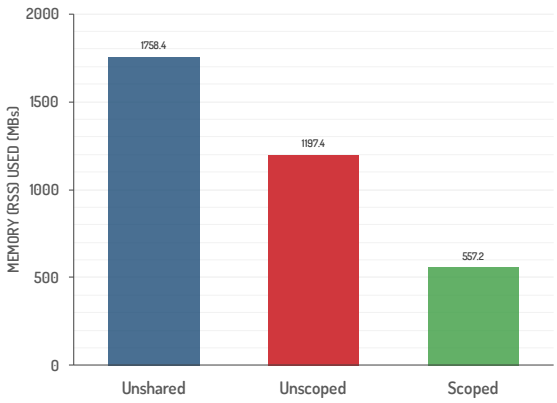
Figure 8.9b shows the results of the recorded memory consumption (measured by resident set size, RSS) averages of each category. The scoped engine showed a lower amount of memory consumed by reducing and optimising redundant information used for computing scopes. On average, the SBH approach was observed to consume up to 30% more than using the vanilla scoped engine. From this result we see that the alpha memory hashing of SBH leads to a more complex node memory structure that needs more space than the conventional node memory. In addition, the setup of the security monitoring service



(a) Comparing the number of joins processed in all module instances.

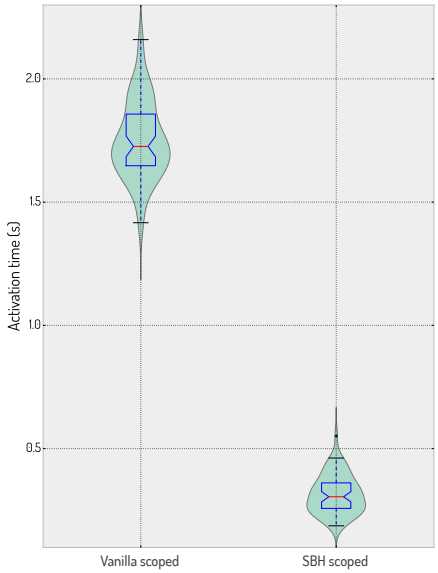


(b) Comparing the number of rule activations observed per 12-hr simulation period.

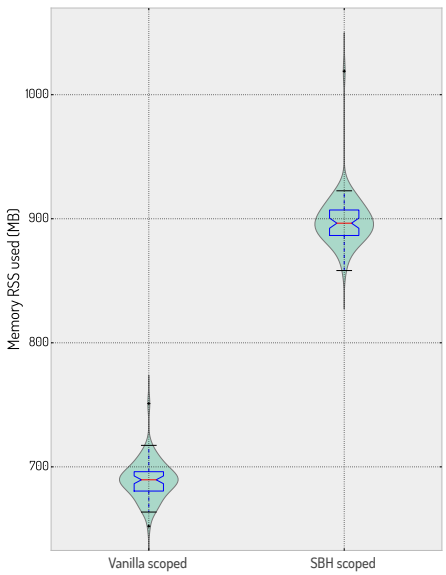


(c) Comparing the total memory consumption (RSS).

Figure 8.8: Results of the simulation for the university access control service with separate rule engine instances augmented with the other approaches – When analysing performance, our scoped Rete engine approach performs comparatively better than using isolated Rete engine instances. The approach with isolated engines performs better than the ad-hoc approach with a shared engine, but consumes more memory due to the separate instances and duplication of resources.



(a) Bean plot of the results of rule activation times – The results show that the SBH approach offers much quicker rule activation times than the plain scoped engine.



(b) Bean plots of results of memory used (Resident Set Size) – SBH was found to consume more memory than a plain scoped approach.

application consisted of a number of entities belonging to multiple groups in the hierarchy, and this therefore leads to fact duplication in the SBH table, increasing memory usage[2].

From the results we observe that for a scoped rule engine for RDKAs, adopting the SBH algorithm leads to faster execution of the engine's matching process (due to direct access in the hash table) resulting in lower activation times. We therefore find that SBH offers significant efficiency benefits for heterogeneous rule engines over vanilla scoped approaches, at the price of a higher space requirement.

8.5 Evaluation: Rule Engine Benchmark with Miss Manners

In this section we proceed to analyse the performance of Serena using a known standard in benchmarking rule engines. The de-facto academic benchmark for Rete-based rule engines is arguably the Miss Manners benchmark, explained next.

The Miss Manners Benchmark

The Miss Manners benchmark is described as follows. Miss Manners wants to throw and host a dinner party. She does not want her guests to get bored; so she intends to place them at specific parts of available dinner tables. She designs a simple solution: she attempts to match people that share a hobby and are of the opposite sex to be seated at one table.

The design of the Miss Manners benchmark provides a *combinatoric evaluation*: it specifically stress tests the beta network of a simple Rete implementation [Bra+91]. Conceptually, the problem employs a depth-first search approach that tries to find a solution of matching people that share a hobby seated at a table. The runs cause the engine to perform millions of join test computations. The benchmark is not without criticism, however, where most rule engine vendors are able to detect and 'optimise' its results. Nevertheless, given proper analysis and setup, the benchmark can give a reasonable indication of rule engine performance when computing expensive joins.

8.5.1 Setup & Methodology

Setup

We setup the Miss Manners evaluation for this work by comparing the rule engine of the Serena^s framework with the Drools Engine introduced in Section 3.3.2.

We performed a customised setup for evaluating the benchmark, since the original version of the benchmark was tied to the specifics of the execution of the OPS5 engine [Bri06]. Firstly, we implemented a similar resolution strategy as Drools' depth strategy that places priority on instantiations with more recent or higher number of facts. Secondly, because Serena is a streaming engine, we first assert the guest and setup facts before asserting the context fact that initiates the benchmark.

For this evaluation, we resolve the Miss Manners problem using a range of between 5 and 50 guests, each with a maximum of 3 hobbies. We performed the benchmark on an Intel Core i7@2.5 GHz (4870HQ) processor with 16GB DDR3@1600 MHz RAM.

[2]We discussed in Section 7.4.4 the situations that can lead to duplication of facts in the SBH table.

Methodology

We present our methodology for evaluating the benchmark. The main aim of this evaluation is to determine the benefits of the proposed scoping mechanism. We therefore perform the benchmark for a comparison between the results of the Serena engine with the Drools engine.

To effectively measure the efficiency and other benefits of scoping, we split the Miss Manners evaluation into two parts: **identical** and **related** hobbies benchmarks.

Identical hobbies benchmark

In this benchmark the goal is to place the guests matched according to a partners of different sex that have the same exact hobby. The identical hobbies benchmark therefore corresponds to the classic Miss Manners benchmark which places guests that have the same hobbies adjacent to each other. We therefore perform this evaluation with the guests having a maximum of 3 hobbies, and guests with the *exact same* hobbies are placed together.

The rules required from the identical hobbies benchmark are derived from the classical OPS5 benchmarks for Miss Manners. The rules facilitate the matching of guests on various seats. As an illustration, we now show the `assignFirstSeat` rule that is responsible for creating the initial seating arrangement and for setting the next context. To give an indication of the different rules of the benchmark we show the left-hand sides of the rule for SRL (Serena Rule Language) in Listing 8.1 and for DRL (Drools Rule Language) in Listing 8.2. All the rules for the Miss Manners benchmark in SRL are listed in Appendix C.

Listing 8.1: Miss Manners: SRL guest-seating rule as JSON

```
{rulename: "find_seating",
 conditions:[
    {$c1: {type:"context", state: "assign_seats"}},
    {type: "seating", seat1: "?seat1", seat2: "?seat2", name2: "?n2", id:"?id", pid:
      ↪ "?pid", path_done: true},
    {type:"guest", name: "?n2", sex: "?s1", hobby: "?h1"},
    {type:"guest", name: "?g2", sex: "?s2", hobby: "?h1"},
    {$c3: {type:"count", num: "?c"}},
    {sign: "not", type: "path", id: "?id", name: "?g2"},
    {sign: "not", type: "chosen", id: "?id", name: "?g2", hobby:"?h1"}
 ],
 actions:[
  /*...*/
 ]
}
```

Listing 8.2: Miss Manners: DRL rule for seating guests

```
rule findSeating
  when
     $c : Context( state == Context.ASSIGN_SEATS )
     $s : Seating( pathDone == true )
     $g1 : Guest( name == $s.rightGuestName )
     $g2 : Guest( sex != $g1.sex, hobby == $g1.hobby )
     count : Count()
     not ( Path( id == $s.id, guestName == $g2.name) )
     not ( Chosen( id == $s.id, guestName == $g2.name, hobby == $g1.hobby) )
  then
    //...
```

```
12 end
```

The `findSeating` rule determines the seating arrangements. The rules in Serena and Drools are shown in Listing 8.1 and 8.2 respectively. Both rules contain a name, left-hand side (`conditions` for SRL and `when` for DRL) and a right-hand side (`actions` for SRL and `then` for DRL). The rule finds a table with one guest (line 5) and tried to match it with another compatible guest: where the sex of both guests are different but share the same hobbies (line 6) and the guest has not been already chosen for another table (lines 8-9).

Aside from the rules facilitating the matching of guests, the benchmark also requires a number of facts representing the guests with their hobbies, sex etc. and other context information when searching for compatible guests. In the template for the `guest` fact, for instance, hobbies are of type `integer` and each guest can have up to 3 hobbies in this benchmark.

Listing 8.3: `findSeating` Rule for Related Hobbies

```
1  {   rulename: "find_seating",
2      conditions:[
3          {$c1: {type:"context", state: "assign_seats"}},
4          {type: "seating", seat1: "?seat1", seat2: "?seat2", name2: "?n2", id:"?id",
             ↪ pid: "?pid", path_done: true},
5          {$g1: {type:"guest", name: "?n2", sex: "?s1"}},
6          {$g2: {type:"guest", name: "?g2", sex: "?s2"}},
7          {type: "$test",  expr: "(?s1 != ?s2)" },
8          {type:"count", num: "?c"},
9          {sign: "not", type: "path", id: "?id", name: "?g2"},
10         {sign: "not", type: "chosen", id: "?id", name: "?g2", hobby:"?h1"}
11     ],
12     scopes: [ "$g1 visibleto $g2"],
13     actions:[
14         /*....*/
15     ]
16 }
```

The Related Hobbies Benchmark

In this part we performed a modified evaluation of the Miss Manners benchmark, where the guests can have several hobbies and can be seated adjacent to others with related hobbies. The narration goes as follows: Miss Manners considers that it is limiting to only seat guests of the opposite sex with *identical* hobbies – some guests may share *similar* or *related* hobbies and would still hold interesting conversations together.

Related hobbies are topics of interest that have common or general heritage (in comparison, identical hobbies are the exact same topics). The related interests benchmark therefore corresponds to the basic benchmark with different hobbies forming a practical taxonomy that connects general hobbies related to each other e.g. a general *arts* category to a related *watching the opera* hobby.

We can represent the hobbies structured as a hierarchy of topics of interests as depicted in Figure 8.11. Miss Manners can thus ask the guests to choose their hobbies at any level of the hierarchy. The modification to the benchmark using related hobbies increases the computations that the basic Miss Manners benchmark performs. This is because the engine needs to check whether the guests now share a common interest: for instance if they both

generally like the arts or sports (rather than just if both like the opera or hockey in the identical interests benchmark).

The facts of this benchmark were modified: we remove the slot of the `guest` facts that represented a hobby. Each guest now becomes a user in Serena, and we designate each guest a number of groups from different levels of the hierarchy, representing their hobbies.

Guests can then receive notifications if any guest 'matches' a person with related hobbies at the party. This required modifying several rules for this benchmark. In Listing 8.3 we show the modified `findSeating` rule. The rule is similar to the identical hobbies benchmark, except for the scopes in line 15 that specify that the two guests should have general interests in the hierarchy using the scope check of `visibleto`.

8.5.2 Results & Discussion

Results

We ran the separate benchmarks on both Drools and Serena engines and logged the time it took for the engine to seat all the guests correctly. Figure 8.10a shows the results of the Drools engine and Figure 8.10b shows the result of the Serena framework. Both graphs show the results of the benchmark using identical interests (without groups) and related interests (with groups) benchmarks.

Discussion

The results of the two Miss Manners benchmarks show that in the classical settings, the Serena engine is, on average, 8 times slower than the JBoss Drools engine. This result is because the Drools engine is known to have several Rete-based optimisations in its core that make it outperform the scoped Serena engine [Pro13]. Furthermore, the Serena framework contains additional components such as the event manager that maintain connections with clients and manage receiving and sending events/notifications, that have a negative impact on the overall performance of the engine. Drools, in contrast, has no notion of maintaining multiple clients and their connections, and does not need to maintain such components internally. The results can be viewed in Figure 8.10a.

When analysing the results of both identical and related hobbies benchmarks, we discover a trend where the Drools engine exhibits degrading performance when comparing its performance changes with increasing number of guests. To get a better insight of this trend, we first take the total time taken by both engines in both experiments. The total time taken by Serena engine is thus t_{serena} and for Drools is t_{drools}. We then use this to tabulate the percentage that each engine recorded for a particular number of guests over its total time.
For example, in the benchmark of 5 guests for Drools we have,

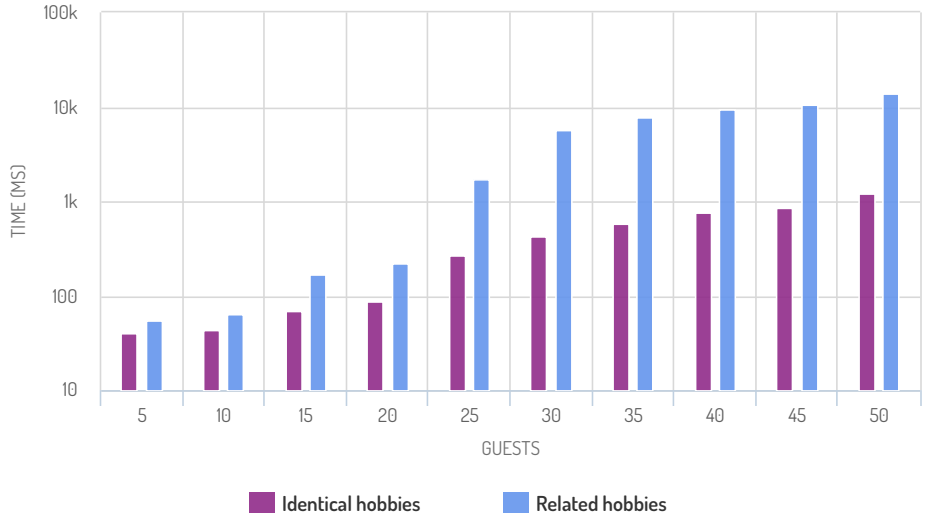
$$perc_{ide5} = \frac{t_{ide_5}}{t_{drools}} * 100$$

$$perc_{rel5} = \frac{t_{rel_5}}{t_{drools}} * 100$$

$$\Delta_{drools5} = perc_{rel5} - perc_{ide5}$$

where for the 5 guest benchmark, t_{ide5} the time taken by Drools, $perc_{ide5}$ is its percentage for the identical hobbies benchmark, $perc_{rel5}$ is its percentage for the related hobbies

(a) The results for the Serena framework



(b) The results for the JBoss Drools engine

Figure 8.10: Results of both types (identical and related) of Miss Manners benchmarks – The graphs show the time taken to find seating for guests, as the number of guests increase. We see that the Serena engine is generally slower than the commercial Drools engine by a factor of about 8x. Drools however exhibits a faster performance degradation when applying the benchmark on related hobbies.

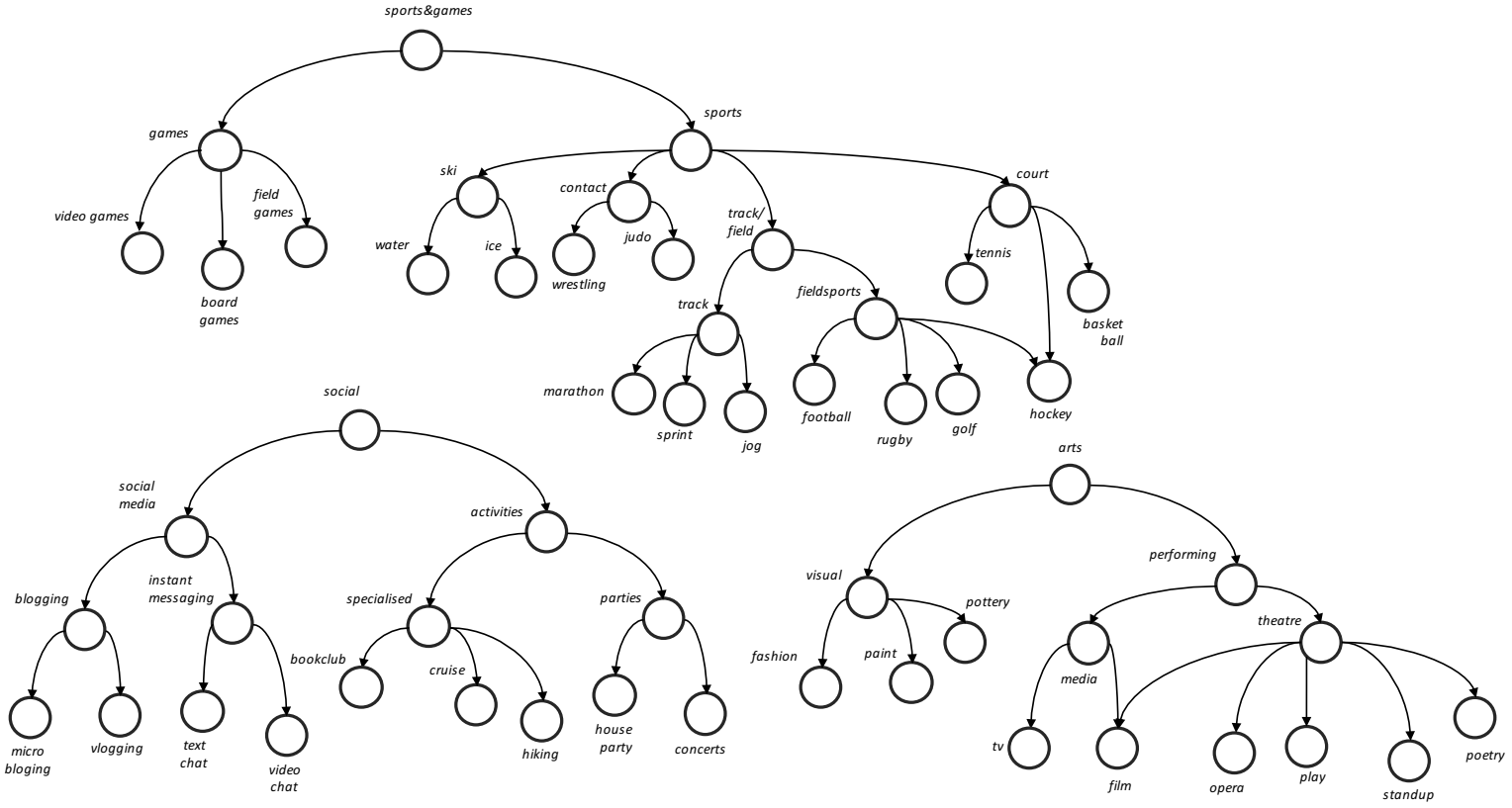


Figure 8.11: The hobby topics hierarchy used on the related interests Miss Manners benchmark – This hierarchy relaxes the restriction of seating guests with identical interests together to seating those with common general interests. The relaxation however increases the complexity of the evaluations that the benchmark performs.

Table 8.12: Comparison of the effect of the modified Miss Manners benchmarks on the performance of both engines – The table displays the time taken as a percentage of the total in both identical (ide) and related (rel) hobbies benchmarks. The increase (Δ) shows that Drools suffers from a higher performance degradation with increasing number of guests as compared to Serena.

Guests	5	10	15	20	25	30	35	40	45	50
Drools										
$perc_{ide}$	0.08	0.08	0.13	0.16	0.49	0.78	1.09	1.41	1.58	2.26
$perc_{rel}$	0.10	0.12	0.31	0.40	3.18	10.49	14.50	17.75	19.58	25.53
Δ	0.02	0.04	0.18	0.24	2.69	9.72	13.40	16.34	18.00	23.27
Serena										
$perc_{ide}$	0.09	0.15	0.25	0.45	0.98	1.92	4.50	8.74	14.74	25.04
$perc_{rel}$	0.10	0.17	0.27	0.45	0.90	1.66	3.42	5.62	11.81	18.76
Δ	0.01	0.02	0.02	0.00	-0.09	-0.26	-1.08	-3.12	-2.92	-6.27

benchmark and $\Delta_{drools5}$ is the percentage difference between the two benchmarks. A similar calculation is performed for Serena.

The results are shown in the rows of Table 8.12. From the table, we observe that the Serena engine shows a smaller change in performance degradation as the number of guests increase when comparing the two benchmarks. This can be seen in the Δ rows of the table with increasing number of guests. The reason for the better relative performance of the Serena engine is because with the increase in the number of guests the engine utilises efficient computations to determine whether two guests are compatible with each other using the encoding of the different hobbies. As the guests increase, the amount of computations to be performed to determine this compatibility increases, making the Serena engine to perform better than the classical benchmark that uses normal join tests when placing guests in table seats.

In contrast, and as seen in Figure 8.10b, the Drools engine suffers from a decrease in performance as the number of guests are increased when comparing deltas of both identical and related hobby benchmarks. The reason here is that with the higher the number of guests the engine needs to perform more combinations of tests to determine which guests with related hobbies can be seated together. Consequently, the different beta computations to ascertain whether such scope checks pass take their toll on the established Drools engine's performance.

8.6 Chapter Summary

In this chapter we have evaluated our approach of introducing scopes to a rule-based framework supporting RKDAs. We performed the evaluation using two strategies to compare our proposed benefits:

- To illustrate the practicality of use, we modelled an extensive real-world scenario (code in Appendix D); and,
- To investigate its efficiency with the state of the art, we performed a comparison with a well-established inference engine (code in Appendix C).

The first strategy involved designing and implementing a real-world scenario of a university access monitoring service in the Cloud interacting with its clients over an extended period of time. The second strategy involved reproducing a known benchmark with the established Drools engine.

From the results and their analyses we confirmed that our technique is useful for RKDAs to capture the inherent organisation of various knowledge representations, and maintains efficiency by considerably lowering the amount of time the inference engine takes to perform its most expensive computations.

The university access monitoring service simulation illustrates that our model controls the number of computations performed by the engine and can also reduce the memory the engine uses internally, compared with a traditional approach with isolated instances. The improvements introduced by the scope-based hashing approach further resulted in significant efficiency benefits over both traditional and purely scoped approaches, with a tradeoff for higher resource usage.

Although the Miss Manners benchmark exposed that Serena was on average slower than the state-of-the-art (i.e., heavily-optimised Drools), the analysis of the results confirmed that current engines suffer substantially when processing structured, shared knowledge in form of groups. This is indeed significant, because structured, shared knowledge is common in heterogeneous RKDAs, as discussed in Chapter 5.

In conclusion, introducing scoping in rule-based frameworks for RKDAs results in efficient computations due to scoped rules. This improves the responsiveness of the RBS as a whole. This in effect means that given the structured knowledge representation, rules in reactive knowledge-driven applications are easier to formulate and understand; and the supporting framework can therefore process a larger number of access requests at a faster rate. As a result, scoped rule engines for RKDAs exhibit increased response times and greater efficiency than using classical forward-chaining rule engines to encode the same knowledge.

9 Conclusion

True creativity often starts where language ends.

Arthur Koestler, *The act of creation*, 1964

The current Web landscape consists of a dynamic architecture that supports the assimilation of technologies that provide complex online services to a larger number of distributed end-users. This dissertation presented our work of using scope-based reasoning in frameworks supporting reactive knowledge-driven applications (or RKDAs). In this chapter we revisit our initial problem specification in order to assess the extent to which these goals were met. The chapter then follows by discussing the main contributions of this research work, and finishes by drawing out the limitations of the approach coupled with some possible solutions as the future work.

9.1 Revisiting the Problem Statement

The Web platform has evolved into a dynamic architecture that processes content from intermittent events [Dri11]. Modern techniques that try to support these changes include reactive programming, which builds upon the deficiencies of traditional imperative programming paradigms by managing data dependencies from various events for their consumption by a programmer [KBD13]. In such techniques however, it eventually becomes difficult to express complex operators for event correlation, composition and other orchestration methods for real-time *complex* events (Chapter 2).

An approach well-suited for this is rule-based systems that provide rule-based syntax for definitions and a rule-engine for processing. Rule-based reasoning leverages a declarative style of programming rather than the common imperative style. This way, an application programmer is offered a way to tackle the complexity of problems caused by lack of control of incoming events from different sources. In a similar manner, the rule engine is capable of capturing, processing and detecting higher-level complex events from such simple but disorderly events.

Rule-based systems are particularly well-suited for expressing *community knowledge*, that uses techniques which combine data derived from a variety of distributed sources in such way that the information can be used collectively to infer higher-level knowledge. This has become particularly significant in todays Web platform, where the sources are users that contribute to such a shared heterogeneous environment. Applications that typically collect events produced by multiple distributed sources and process it in a timely manner in order to extract community knowledge are known as reactive knowledge-driven applications (RKDAs).

However, heterogeneity, when coded manually, has negative implications on the execution cycle of the main component of a rule-based system, the inference engine. Classical rule-based systems are indeed inherently non-reentrant as they can spur rule activations from all asserted facts without discriminating their disparate sources. Reentrancy describes programs written in such a way that multiple users can share the same copies of data in memory consistently. Using current rule-based approaches programmers enforce reentrancy through manual interventions, by hard-coding distinctions between different data sources within the rules. This becomes difficult to enforce and orchestrate using traditional rule semantics as the number of clients and the relationships between them increase. Failure to properly make these distinctions causes unintended rule activations in other clients.

RKDA frameworks therefore require orchestration within client rules and within the inference engine during processing to discriminate or distinguish between instances of different entities. This problem manifests itself in other domains such as multi-tenant databases and distributed event-based systems (Chapter 5). Ultimately, however, even though rule-based systems are suitable for supporting the development of RKDAs, abstractions that attempt to solve these problems are lacking in current approaches.

9.2 Summary & Contributions

9.2.1 Summary

The goal of this dissertation was to provide a reactive rule-based system for RKDAs that solves reentrancy problems by implementing *scope-based reasoning*. The scope-based approach supports scoping within rule definitions and in rule engine execution to enforce data discrimination between instances of data from different clients and to promote community knowledge. The motivation for this dissertation was organised into two parts.

Serena: Rule Engines for RKDAs

The first part discusses how modern Web applications have changed from their static, isolated foundations to more dynamic, responsive and composite functionality. The functionality has enabled Web applications, denoted in this dissertation as **reactive knowledge-driven applications** or RKDAs, which require programming and processing constructs to collect information produced by multiple distributed sources, and to process it in a timely manner in order to extract new knowledge. The requirements identified for technologies that should support these types of applications are summarised in Table 9.1.

While presenting a survey of the current state-of-the-art that meets requirements (Chapter 3), the discussion concluded that most current approaches were lacking when it comes to meeting demands for programming online discovery or detection of patterns in large data sets. The work further showed why most conventional techniques of meeting demands for online processing of events in reactive knowledge-driven applications using event

Table 9.1: Recap of requirements for support of knowledge-driven applications

Requirement	In detail
Event-driven model	For efficient online processing of client events.
Knowledge encoding	For providing knowledge extraction in reactive data.
Incremental processing	To provide real-time processing and immediacy of results
Hot-swapping	To enable the dynamic addition of client constraints at run time
Simplicity	For providing transparent symbiosis with server execution

Table 9.2: Recap of requirements that foster the support for capturing community knowledge in rule-based systems

Requirement	Detail
Metadata model	Metadata for managing heterogeneous clients
Grouping model	Formalised model for grouping clients
Execution model	Execution model for selective computations
Notification model	Flexible model for notification semantics

processing systems are unsuitable. This dissertation consequently identified a promising research area in the rule-based paradigm (Chapter 4). Specifically, detection-oriented rule engines that process events independently were found to be suitable to capture definitions for heterogeneous data from clients.

The first part therefore culminated in a custom detection-oriented rule-based inferencer, Serena, that meets the identified requirements. Clients can upload and install rules to the server and receive reactive feedback in the form of push-based notifications. Serena as a whole provides dynamic definition of rule-based constraints, efficiently processes intermittent data through a forward chaining inference engine and manages connections and message sends between the server and clients.

The Challenges of Heterogeneity in the Web

The second part focused on heterogeneity in the Cloud, particularly the implications of this heterogeneity in rule-based systems. Using a practical example, we showed the problems of heterogeneity on a shared instance in the traditional processing cycles of a rule engine. These effects can be attributed to lack of reentrancy in rule-based systems, making them difficult to control and maintain when they are deployed in heterogeneous settings. The problem becomes more complex when community knowledge is taken into account. To frame a solution, requirements to solve these problems were presented (Chapter 5). The requirements are summarised in Table 9.2.

Given these requirements we investigated related research in rule-based (and other) domains, focusing on heterogeneous solutions with shared instances or resources. Even though specific rule-based systems contain generic abstractions for managing heterogeneity such as rule modules and entry points, these abstractions are not suited for multiple clients sharing a single engine instance. Additionally, they have no support for targeted notifications for clients or a well-designed metadata architecture for managing heterogeneity. In related domains, abstractions such as schema sharing in multi-tenant databases and notification filtering in event-based systems exist with the aim of designing and engineering

for heterogeneity. Schema-sharing techniques in multi-tenant databases were lacking in providing a grouping model and did not support notification mechanisms for client events. Event-based systems had promising research in visibility roots for notifications but lacked in providing a suitable execution model.

Scope-based Reasoning

Chapter 6 introduced scope-based reasoning in Serena. Serena provides a viable solution for heterogeneous rule engines and exposes abstractions that promote capturing community knowledge. Serena uses this scope-based approach and is built using two main program design and engineering artefacts, scoped rules and a scope-aware rule engine, using a unified framework that spans both the server and client architectures.

Scoped Rule-based Language Scoped rules contain an extended rule-based syntax that allows rule designers to define scope constraints. Scopes are a control structure for heterogeneous rule-based languages because they specify a selection of which (subset of) rules that a scope-aware inferencer will use at a particular time during execution. They specify which data to which the rule is applicable when matching. Additionally, scoped rules also specify who to notify and what information is sent with the notification. To ease the learning curve of rule design for a number of users, rule creators can optionally design rules for the framework using a provided graphical user interface named SerenaUI, that exposes a visual programming editor to build client rules.

Scope-aware Rule Engine In shared heterogeneous multiuser environments, the use of scopes allows shared rule engines to cope with the means of determining which rules are applicable to a data token within the engine at runtime. This is not only important for enforcing reentrancy, but also for supporting capturing community knowledge in shared multiuser architectures, where a rule can be applicable to a group of clients or applications. As discussed in Chapter 5, determining the relationships between the data being processed at runtime efficiently becomes an important factor for the successful operations of a heterogeneous rule engine, because having multiple sources can negatively affect the performance of performing rule computations. Consequently, Serena utilises the concepts of physical or logical *groups* of tenant clients and their relationships in its implementation. The framework internally captures and converts sets of clients into an internal representation computed efficiently via an encoding. The encoding generated is used to perform operations efficiently in order to determine compatible data during rule engine execution. The encoding is a modified application of the method for efficient implementation of lattice operations in [Aït+89]. The process and results of this encoding are used when computing scope operations by scope guards installed on the beta nodes within the Rete graph generated by the engine. The scope guards are invoked to determine whether the join computation that the node should perform on the data inputs will use compatible data. The chapter also presented a further optimisation with the *scope-based hashing* (SBH) technique, that extends the work on scoped inference engines with scoped hash tables of alpha memories in the inference engine. The hashes are computed according to client groups in asserted facts added using the scoped metadata architecture. The SBH approach utilises the scoped hash tables and fact metadata to exclusively and efficiently compute the compatible inputs that will be relevant when computing joins during the match cycle of heterogeneous rule engines.

Scoped Notifications Finally, the notification model in the framework further provides flexible means to limit the distribution of client notifications whenever client rules are activated. Notifications can define constraints to notify a group, subgroup, or even direct clients. Section 6.5 explained how the framework exploits metadata about clients, their grouping structures and the encoding process to limit the distribution of client notifications.

The benefits afforded by Serena were validated in the evaluation Chapter 8. In the evaluation, a practical scenario representative of an RKDA in a heterogeneous environment was modelled and tested in an extensive experimental simulation, and the results were analysed. Together with the previous evaluation of the related work, it was confirmed that the scope-based technique is useful to capture the inherent organisation of community knowledge representations (i.e., qualitative evaluation), and is efficient by considerably lowering the amount of time the inference engine takes to perform its most expensive computations (i.e., quantitative evaluation). The SBH improvement increases the efficiency of the scope-based approach during the matching process, albeit with a higher memory consumption. When comparing Serena's implementation to an industry-strength forward-chaining rule engine, it was further confirmed that classical rule engines suffer substantially when processing structured knowledge in a heterogeneous configuration. The scoped approach on the other hand showed negligible differences in performance in the same settings. In the end, Serena^s fulfils the big data characteristics of volume, velocity, variety and value via its scale, reactivity, heterogeneity and community knowledge features. Of course, the requirement is that client structures have to be defined to be internally represented in the framework for these benefits to be realised.

Summary. Serena^s therefore fulfils the requirements for capturing community knowledge in rule-based systems, as summarised below.

- *Metadata model* – Serena^s provides a Cloud-based framework that is suitable for supporting RKDAs using scoping model that operates on metadata about rules and facts asserted into the rule engine on the server, to aid in managing shared knowledge in heterogeneous systems.
- *Grouping model* – The framework places grouping of clients according to their structural organisation at the heart of its scoped approach. The groupings can be used within scoped rule definitions to specify the data that the engine should use when performing the rule computations.
- *Execution model* – Scoping in Serena^s, and by extension using SBH, provides an internal representation is that precomputed efficiently as an encoding that is be used when processing constraints to check for data compatibility during rule engine execution.
- *Notification model* – Once rules are activated a multi-user setup needs some way to identify the users that should be notified: the Serena^s framework exploits the aforementioned points of metadata about clients, their grouping structures and the efficient computation of scopes to limit the distribution of client notifications.

Scoped rules allow rule designers to focus on distinguishing constraints on clients as sources of events independently, isolated from from the logical intent of the rule. This *separation of concerns* feeds into a solution of the problems of shared heterogeneous data and at the same time promotes the capture of community knowledge separate from how the computational aspect of the rule is designed.

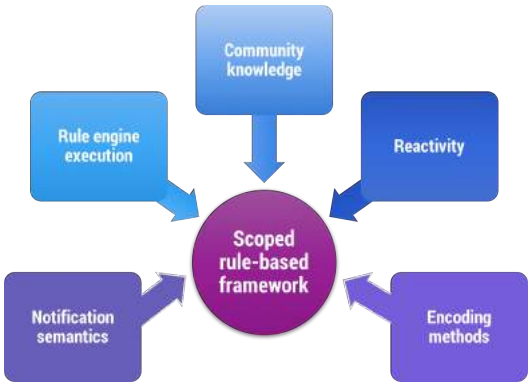


Figure 9.3: The resulting constituent parts of a scoped RBF – The approach merges techniques from different domain areas fused to form a scoped rule-based framework.

The ideas that this dissertation has presented encompass research from distinct fields brought together by changes of the traditional landscape of the Web to the current dynamic and versatile architecture. Its artefact, Serena^s is a unified framework that contains concepts amagalmated from several fields (Figure 9.3), including:

- rule engine execution for capturing complex events through the forward-chaining Rete algorithm,
- research in collective intelligence through harnessing community knowledge in heterogeneous environments,
- work in the prompt processing of data sets from intermittent sources continuously using online, reactive and dynamic semantics,
- research in efficient computation of an encoding method that entirely determines data relationships in heterogeneous environments,
- event-based notification semantics to determine the recipients of notifications of events once constraints are fulfilled.

The framework uses these concepts to provide an application-agnostic solution to solving problems in heterogeneous rule based systems and to provide means in which community knowledge can be exploited.

9.2.2 Restating the Contributions

From the summary of the direction and the methodology that this dissertation has embraced, we revisit its main contributions as they pertain to the ideas that it represents in the following summary.

A study of the open issues, limitations and shortcomings of supporting reactive knowledge -driven Web applications in multiuser environments that require reasoning semantics. This work has performed a literature study in the domain of knowledge-driven systems for the Web after identifying a number of criteria that systems aiming to support RKDAs need to fulfil.

A unified reactive, rule-based framework for heterogeneous environments that can support reactive knowledge-driven applications. The framework supports RKDAs by presenting a detection-oriented rule-based framework built on the Rete algorithm. The framework allows runtime definition of rules by clients to be dynamically added in the engine during execution. The engine allows the framework to incrementally evaluate new events with newly-asserted definitions and reactively send feedback to notify clients selectively.

A rule-based language augmented with extensions to define scoped rules for expressing community knowledge. The extensions are applied to rules in order to support scope-based constraints in heterogeneous rule-based systems, known as scoped rules. Scoped rules allow programmers to capture collective intelligence in the form of community knowledge by augmenting the rule language with scope-based semantics that define scoped rules. They allow programmers to specify orchestration within rules in order to distinguish between instances of different client entities as well as capturing data according to relationships between groups of clients.

A reentrant inference engine that embraces scope-based reasoning. The work presented an extension to the Rete algorithm that efficiently enforces reentrancy by incorporating techniques based on the bit-vector encoding method, which discriminates data matches as defined in scoped rules. The implementation uses a scope-based architecture in the inference engine to determine compatible inputs to a beta join node in the engine during matching. With the scope-based hashing approach the alpha network can also be a party to the efficient implementation of scope-based reasoning in the engine.

A qualitative and quantitative evaluation of a reentrant inference engine. The qualitative evaluation has shown that given the structured knowledge representation of heterogeneous clients, Serena^s scoping constructs are useful in expressing heterogeneous community knowledge. The quantitative evaluation showed that the scoped engine can process incoming events at a faster rate than a classical rule engine. A further evaluation of the scope-based hashing approach showed that an improvement in efficiency was realised using hashed alpha memories, albeit with a slightly higher usage in memory.

9.3 Limitations & Future Research

This section identifies some of the limitations of the work presented, and proposes the future avenues that the ideas behind it has unearthed.

9.3.1 Support for Custom Scopes

Serena^s uses a bit-vector encoding method for representing the common patterns needed to represent community knowledge. The downside is that the process and the encoding itself is mostly static and consequently, the types of scopes it can support is limited. It would be useful to avail a more dynamic or custom scope-based language in which clients can compose their own scope definitions which can be used in scoped rules, which can then be internalised in the engine's execution process. One challenge with this research path is the choice of scope-based language to be exposed to the clients. Similar languages are

prevalent today in the field of graph query languages. Indeed, the problems with expressing various knowledge representation systems was source of major inspiration in languages supporting the graph data model (GDM) [Kun87], as it needs expressive and flexible techniques to represent models as well as derive them from queries. Some GDM research has been applied to the Web, such as in AllegroGraph [Aas06]. The interest here, however, is in the expressive power of structural property graphs that can be queried using a declarative query language. Todays such query languages are particularly common in modern *graph databases*, where the data (and its schema) is represented entirely as a graph. Unlike in those based off of traditional databases, graph-database query languages (GQLs) support paths, neighbourhoods and connectivity [AG08] – which in our case would be useful when constructing custom scopes in client structures. Examples of promising GQLs in this environment include newer languages such as Cypher/Gremlin [HP13], GraphQL [HS08], UnQL [BFS00], SocialScope [ALY09], that have gained notable traction from older languages GraphDb [Güt94], GRAM [AS92], G [CMW87], and GOOD [Gys+94].

Listing 9.1 shows a precept of defining a simple dynamic scope in a rule definition in Serena. The snippet returns siblings that share the same immediate parent through the `siblingOf` definition.

Listing 9.1: Example of a Dynamic Scope Definition

```
1 DEFINE SCOPE siblingof AS
2 MATCH (group1)—[:parent]—()—[:child]—(group2)
3 RETURN group1, group2
```

Another challenge would be the design of a specification language for scopes in rules. The language would be semantically similar to the aforementioned GQLs thus improving its expressiveness in defining and capturing different scopes. Despite offering expressive functionality for custom scoping, this approach however has a significant disadvantage of having two different matching semantics within one rule: rule-based and graph query-based. As a result, this increases the complexity of understanding the rule logic as a whole. A separate challenge is how existing custom scope definitions would handle the addition and removal of groups at runtime. This would need a traversal of all the definitions to find the ones affected and perform suitable actions on them according to some predefined semantics.

Even with dynamic scope definitions and querying, the implementation described above would need to offer efficient checking of these definitions whilst the rule engine is in execution. This implies that the approach would follow implementations of *in-memory graph databases* such as imGraph [JR13] to achieve high-speed processing requirements. Ultimately however, the support of static vs. dynamic scopes leads to a general tradeoff between flexibility and the range of optimisations applicable; aspects that can be usually chosen depending on the needs of the serving application.

9.3.2 Antiquated Data in Rete

The Rete algorithm at the heart of the Serena framework is an in-memory algorithm for efficient matching in rule-based systems. From its temporal redundancy feature, the algorithm needs to store intermediate results in its internal nodes to avoid redundant computations across cycles. Due to this, it is vulnerable to the known limitation of increasingly becoming lethargic as the amount of data stored in its intermediate memories continually increases from assertions. In such cases the rule engine will inevitably store and process a large number of data items per node eventually becoming a major performance bottleneck (due to the increasing amount of stored state), reducing the response times of the framework as a whole.

This problem has already been investigated by several works. One of the earlier work is by the TREAT algorithm (introduced in Section 2.5.4). TREAT takes advantage of *conflict set support* which explicitly retains the conflict set across cycles and thus beta memories do not need to be retained, reducing overall memory consumption. This research direction led to work that investigated tradeoffs between a high memory usage and processing in the beta network, in Rete* [WM03] and Gator [HH93] (also discussed in Section 2.5.4).

Even with these modifications, the engine will inevitably run into memory management issues, albeit at different rates. To solve this, in work related to this thesis PRete [Ale15] investigated ways in which Rete can persist (parts of) its intermediate state to a rapidly-accessible solid-state drive. While doing this, the network would be able to remove the 'least recently used' tokens during a predetermined number of matching cycles in the inference engine. With this scheme, it was observed that the Rete engine can increase its workload capacity, as well as making it more resilient in the event of a server crash.

The PRete method can be extended to tailor to specific situations in a heterogeneous context. Work in [FRS93] proposed an adaptive algorithm that evaluates rules and returns the set of 'most profitable' relational expressions to be maintained for a good compromise between a rule engine with and without intermediate beta memories. The algorithm was based on an analysis of the flow of data in a rule engine. If fused with a scope-aware engine it is possible to analyse the effects of different scope definitions in rules to determine which data from which client groups is superfluous and can be persisted to a rapid-access disk, to improve engine execution.

Separate work by PARTE [Swa+13] involved introducing relative temporal semantics in rules, as well as in rule engine execution. This can be useful for garbage collection of antiquated facts, by offloading unnecessary data that cannot partake in current rule temporal patterns. These techniques can be incorporated into Serena, as scoping retains the basic semantics of the Rete algorithm.

9.3.3 Cloud Models for Heterogeneous Rule-based Systems

Serena is designed to work in a Cloud environment. It is therefore subject to other issues that were not the primary focus of this thesis, such as virtualisation and providing cost models.

For virtualisation, work on distributing the Rete algorithm over several clusters or virtual machines exists. Recent approaches concentrate on the actor and message-passing models to separate Rete graph nodes [Swa+13; Wan+14]. Since Serena preserves the semantics of the Rete algorithm, these approaches can be integrated into the framework, taking care to keep a scope check and its distributed node together. In addition, the encoding should be present in all nodes with a scope check, and any changes to the encoding should be propagated to the relevant nodes.

For cost models, we observe that execution cycles in the Serena rule engine can be dominated by rules from one client (or one tenant). In such cases it becomes useful to be able to calculate the amount of resources that a client uses for accountability. These can be used for billing purposes or to offer various flexible payment models to registered customers. This is important to clients as well, to confirm that the provider has adhered to the level of performance often agreed upon in the Service Level Agreement, or SLA (which can play a part in the *veracity* aspect of big data).

To calculate the usage of resources by a particular client, modern Cloud providers often offer a flat cost (or sometimes, no cost) for the base application, but the client incurs an additional price for the extent of usage on particular aspects such as service customisation and runtime resource usage. This is particularly attractive for a heterogeneous platform

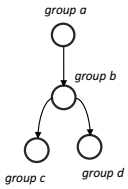


Figure 9.4: Sample client group structure.

because demand for its services varies over time [Arm+10]. For instance, detecting access requests on various rooms in the office complex will drastically reduce during working breaks and over the weekends.

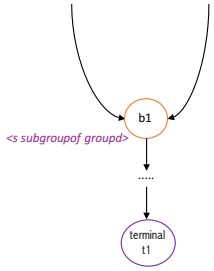
In this respect, a vital challenge for the shared instances of a heterogeneous rule based system is how to determine (or estimate) the cost models associated with a particular client, groups of clients or company. Models often seen in cloud computing incorporate bandwidth, CPU hours and disk storage used. For distributed entities, the number of nodes or VMs occupied can be included. Solutions for calculating such metrics for scoped engines can include calculating more specific metrics to particular features. Aside from the number or groups of clients per parent company, other indicators of rule engine execution can be used to give specific memory usage metrics such as the number of rules per client or groups of clients, and the number of facts asserted. More elaborate metrics can also be incorporated to calculate the execution cost model, including the number of join computations, beta node activations, rule activations, etc. Network metrics can include the number of events received and notifications sent to and from specific clients or groups of clients.

9.3.4 Other Research Avenues

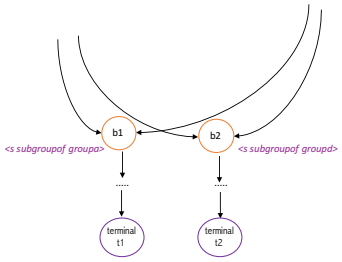
Rule Optimisation and Graph Reordering

Serena supports dynamic addition of rules from clients to the server. Internally, whenever a rule is added to the Rete graph built by the rule engine, existing nodes are either reused or new nodes are appended to the graph. Appending nodes makes the runtime process of adding rules faster in general but may reduce the potential for possible optimisation steps that can be exploited.

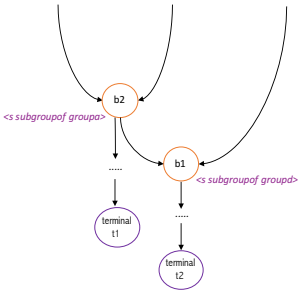
For instance, consider the simplified layout of client groups illustrated in Figure 9.4. Suppose a rule $r1$ contains a scope definition $s1$ `<$s subgroupof groupd>` for a condition $r1_c$. The graph for $r1$ can be as shown in Figure 9.5a. If a rule $r2$ with a similar condition $r2_c$ that contains scope definition $s2$ `<$s subgroupof groupa>` is added by a client **after** $r1$, the rule engine tries to find nodes to reuse for $r2_c$. If none is found, it adds $r2$ to the graph by simply appending its nodes to reduce processing downtime during rule addition. The result of adding rule $r2$ with an existing graph of rule $r1$ is shown in Figure 9.5b. For better optimisation, a graph *reordering* can occur. In this case, having a node for $r1_c$ with scope check $s1$ for `groupd` can be rearranged having the node for scope check $s2$ for `groupa` inserted before that of $s1$, as node $b2$ shown in Figure 9.5c. The challenge of this idea is whether reordering offers a more optimised approach when compared to the downtime needed to reorder and re-update the new graph with existing tokens. Borrowing from ideas behind condition-reordering in Rete-based systems [NGR88], the approach can be fine-tuned to achieve different results.



(a) The current Rete graph for rule $r1$.



(b) Addition of $r2$ without reordering.



(c) Addition of $r2$ with reordering.

Figure 9.5: Optimisation via scoped graph reordering at runtime

Right Activations in Scope-based Hashing

The scope-based hashing approach (Section 7.4) offered a reasonable improvement over a scope-based engine. It was however only applied to left-activations, and although these can dominate right activations in a high number of cases, they do not substantively cover all the join computations performed by the engine.

Future research for SBH in right activations during matching can be promising to capitalise on the foundations of scoped alpha memories in order to further improve the performance of the algorithm as a whole. This implies that the beta memories should also contain hashed memories by scope, unearthing research questions regarding the semantics of hashing token compositions based on client groups and existing scopes. Furthermore, implementing this approach can further increase the complexity of processing retractions and changes to client group structures at runtime.

9.4 Concluding Remarks

This dissertation presented the Serena framework, which blends an event-driven web server with a rule-based inferencer. Internally, the rule engine and its components are contained in a shared instance that can serve multiple tenants. Currently, there are no rule-based approaches that offer rich semantics that developers can use to exploit community knowledge contributed by distributed, heterogeneous clients in a shared multi-tenant instance. This solution is useful in a number of multi-client applications to deal with the problem of orchestrating data patterns in a heterogeneous setting, and to exploit the fact that much of the community knowledge significant when performing reasoning and deductions can

be structured hierarchically. Rule designers can utilise scoped rules to detect patterns in real-time data and to realise grouping structures in RKDAs, backed by a common rule-based system. Scoped rules provide ways in which clients can further exploit relationships between them to source data with regards to their own organisational structure or layout.

From the evaluation we have confirmed that our technique is a viable approach for capturing community knowledge. The evaluation showed that our model precisely controls the amount of deduction or computation performed automatically by the framework (as collective information from clients flows into the system) in a both expressive and computationally effective manner when compared to classical approaches. The requirement is that the clients need to specify their group structures to fully take advantage of the proposed framework: the groups to be encoded in the client hierarchy have to be defined and added to the engine, so that the benefits provided by the encoding scheme can be fully realised.

To conclude, this dissertation presented our vision garnered from the research areas of rule-based systems, reactive technologies and collective intelligence. We believe scope-aware rule engines provide a collegial approach that advances of the current state-of-the-art in allowing modern multi-user RKDAs to orchestrate, as well as capitalise on, community knowledge deduced from heterogeneous event sources.

A

Serena's Matrix Encoding

A.1 Definitions

A.1.1 Posets

A partially-ordered set (or simply a poset) $(P, \leqslant)$ is a set P and a binary relation $\leqslant$, such that for all $a, b, c \in P$, the following properties always hold:

1. $a \leqslant a$ (reflexivity)
2. $a \leqslant b$ and $b \leqslant c$ implies $a \leqslant c$ (transitivity)
3. $a \leqslant b$ and $b \leqslant a$ implies $a = b$ (antisymmetry)

Two elements in a poset are defined to be *comparable* if either $a \leqslant b$ or $b \leqslant a$. If two elements are not comparable they are said to be *incomparable*. Unlike a totally-ordered set, it is not a requirement that all elements in a poset be comparable.

Poset Operations

Bounds: Given $A \subseteq P$, an element $b \in P$ is called an *upper bound* of A if $a \leqslant b$ for all $a \in A$. b is a *least upper bound* or LUB if $b \leqslant a$ whenever a is an upper bound of A. The dual of the least upper bound is known as the *greatest lower bound* or GLB. Conversley, the LUB $\vee$ of P is also known as the *join* or *suprema* of A. The GLB $\wedge$ is the *meet* or *infima* of A.
Extrema: A maximal element (or just the *maximal*) of a poset P, which we denote as $\lceil P \rceil$, is an element $m \in P$ that is greater than any other element in P according to the relation $\leqslant$. More formally,

$$\lceil P \rceil = \forall b \in P, b \leqslant m \tag{A.1}$$

If there is one unique maximal element in P, we call it the *maximum*. The dual of the maximal is known as the *minimal*, $\lfloor P \rfloor$ and a unique minimal is known as the *minimum*.

A.1.2 Lattices

If in a finite poset P every pair has at least an LUB $\wedge$ and a GLB $\vee$, then the poset P with the features $(P, \leqslant, \wedge, \vee)$ is said to be a *lattice* L.

A.1.1 Lemma. *Given a finite poset P with a set of maximals M, if an element $\top$ is added such that $(m \leqslant \top)$ for all $m \in M$ then the new element $\top$ is the LUB of all elements in P.*

Proof. Assume the newly added element $\top$ was not the LUB of an element $b \in P$. This means that there was no $m \in M$ that was an upper bound of b. But if b has no upper bound in P it means b is itself a maximal, and thus b would be in the set M. so, if $b \in M$ then the relation $(b \leqslant \top)$ makes $\top$ a LUB of b. □

With Lemma A.1.1, the same proof construction can be made for the addition of a GLB $\bot$ to all minimals in P.

A.1.2 Lemma. *Given a finite poset P with a set of minimals N, if an element $\bot$ is added such that $(\bot \leqslant n)$ for all $n \in N$ then the new element $\bot$ is the GLB of all elements $b \in P$.*

Proof. The proof is similar to that of Lemma A.1.1. If the newly added element $\bot$ is not the GLB of an element $b \in P$, then there is no $n \in N$ that was a lower bound of b. But if b has no lower bound in P it means b is itself a minimal, and thus b would be in the set N. □

The second proof is actually confirmed by the Duality Principle for orders,

A.1.2.1 Definition (The Duality Principle)**.** *If a statement ϕ is true for all orders, then its dual ϕ^D is also true on all orders.*

This leads us to the following theorem.

A.1.3 Theorem. *Given a finite poset P, we can transform it into a lattice by adding a parent $\top$ to all maximals and a child $\bot$ to all minimals.*

Proof. Lemma A.1.1 already proves that $\top$ is the LUB of all elements in P. Its dual Lemma A.1.2 also proves that $\bot$ is the GLB of all elements in P. From the definition of a lattice, the addition $\top$ as a parent of all maximals and $\bot$ as a child to all minimals makes every element to have at least an LUB and a GLB, confirming that $(P, \leqslant, \top, \bot)$ is a lattice. □

Therefore, one way to transform the poset P into a finite lattice [BMN97] is by adding a parent $\top$ to every maximal and a child $\bot$ to every minimal in P.

The Covering Relation

We say for two elements $a, b \in P$, a is *covered* by b if b immediately follows a in the poset ordering (i.e. a is an immediate successor of b, b is an immediate predecessor of a). More formally,

$$a \prec b \text{ iff } a \leqslant b \text{ and } \nexists c \text{ s.t. } a \leqslant c \leqslant b, c \neq a, c \neq b \tag{A.2}$$

This enables us to depict a lattice in a *Hasse diagram*, where an edge goes from b to a iff $a \prec b$.

Lattice levels

In this dissertation we define the level of an element a in a lattice as the longest distance of a from the maximum of the lattice (in this case, $\top$) to the element, i.e.,

$$Level(a) = \begin{cases} 0 & \text{when } a \text{ has no predecessors} \\ & \text{in } P \text{ and,} \\ max(\{Level(b)) \mid b \succ a\}) + 1 & \text{otherwise.} \end{cases} \tag{A.3}$$

where $\succ$ is the dual of $\prec$.

A.2 Operations with ϑ

Having constructed a lattice L, the method in [Aït+89] defines a mapping ϑ from L to another lattice $(S \subseteq, \cap, \cup)$ such that for every $a, b \in L$,

$$\vartheta(a \wedge b) = \vartheta(a) \cap \vartheta(b), \tag{A.4}$$

$$\vartheta(a \vee b) = \vartheta(a) \cup \vartheta(b). \tag{A.5}$$

ϑ is thus said to be join and meet-preserving. Furthermore, the method specifies that ϑ should be invertible, making it relatively easy to calculate the GLB and LUB such that $\forall$ $a, b \in L$,

$$a \wedge b = \vartheta^{-1}(\vartheta(a) \cap \vartheta(b)), \tag{A.6}$$

$$a \vee b = \vartheta^{-1}(\vartheta(a) \cup \vartheta(b)). \tag{A.7}$$

The notion of the mapping ϑ is the basis of the matrix encoding, described next.

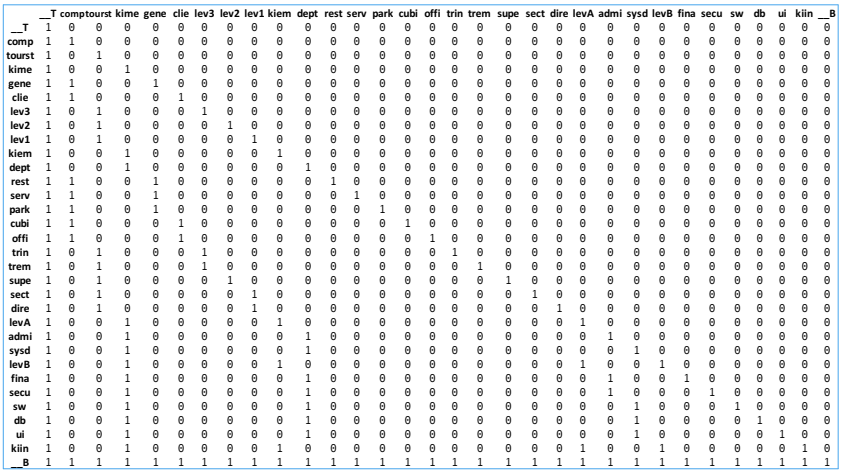
A.3 Matrix Encoding

Serena has adopted the encoding method mentioned in [Aït+89] similarly taking ϑ as the transitive closure. Given a finite poset P, the framework uses the ideas described in Appendix A.1.2 to transform P into a lattice L. Serena then uses the encoding method in [Aït+89] with a slight modification that will map L onto an encoded matrix M_{ϑ_L}. The whole process is described below.

- Instead of starting with $\bot$, start with $\top$ as the first element from L. Assign $\vartheta(\top) = 0$.
- Visit the next elements level by level *downwards* in L (using the relations in L), and calculate the bitcode of each element as a vector.
- The bitcode of an element $a \in L$ is obtained by

$$\vartheta(a) = 2^{i-1} \vee \bigvee_{a \prec x} \vartheta(x) \tag{A.8}$$

  where i is the number of elements visited since $\top$ and x represents a predecessor of a; therefore $\vartheta(x)$ is the code of each predecesor of a.



	__T	comp	tourst	kime	gene	clie	lev3	lev2	lev1	kiem	dept	rest	serv	park	cubi	offi	trin	trem	supe	sect	dire	levA	admi	sysd	levB	fina	secu	sw	db	ui	kiin	__B
__T	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
comp	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
tourst	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
kime	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
gene	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
clie	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lev3	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lev2	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
lev1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
kiem	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
dept	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rest	1	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
serv	1	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
park	1	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cubi	1	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
offi	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
trin	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
trem	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
supe	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
sect	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
dire	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
levA	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
admi	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
sysd	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
levB	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
fina	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
secu	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
sw	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
db	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
ui	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
kiin	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	1	0
__B	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure A.1: The full matrix encoding of all the company hierarchies – The groups are based on the running example of the office building complex showcased in Section 5.2.1.

- An entry in the new matrix M_{ϑ_L} for a is the reverse of the bitcode obtained by Equation (A.8) without the most significant bit, which is always a `1`.

With this encoding, Serena can perform operations in Eq (A.6) and (A.7) having $\cap$ as the bitwise `AND` and $\cup$ as bitwise `OR` in M_{ϑ_L}.

A.4 Full Matrix Encoding for Office Complex Example

The resulting binary matrix of all the groups of the office complex hierarchies after the encoding process with ϑ outlined in the previous section is shown in Figure A.1.

B

Coordinating Collaborative Interactions using Scoped Rules

This section uses the research work in [Kam+15] that identified some of the early challenges that developers of RKDAs face whenever they require mechanisms to orchestrate the large number of events from different sources. RKDAs today have the ability to provide richer interactions hitherto unrealised by running them on isolated devices. These modern applications can now support proximal and remote collaborative interactions for multiple clients simultaneously connected to each other. Most technologies however were found to lack programming language abstractions for coordinating complex interactions, such as to define, detect and combine complex events coming from multiple clients or other heterogeneous software entities. Furthermore, they lack the expressiveness required to support non-trivial levels of collaborative interactions for connected clients.

In [Kam+15] we identifed two software mechanisms that web-based mobile applications should provide to support the development of collaborative interactions: distributed event composition and group coordination. The work culminated as the Mingo framework, a precursor to Serena which provides dedicated coordination programmer constructs for these two mechanisms by blending techniques common in complex event processing and group communication. In this appendix we apply these ideas using scoped rules in the Serena^s framework.

B.1 Motivating Ex: Online Collaborative Drawing Editor

Consider an example of a traditional collaborative drawing application where several distributed participants on mobile phones are connected via the web using a server. The participants share the same canvas on multiple mobile devices. The users are participating in the same session and can therefore interact at the same time. The shared canvas can be used by the several participants connected via the web to draw and interact with (or manipulate) shapes already drawn on the canvas.

In addition to the normal functionality provided by a traditional drawing application,

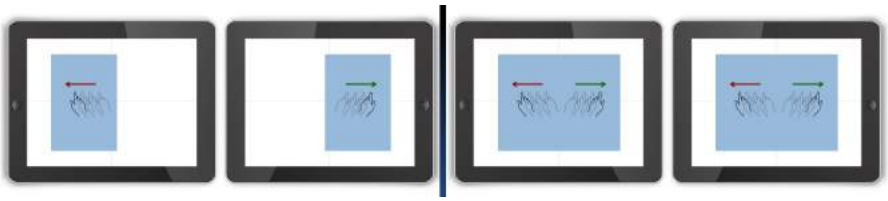


Figure B.1: The Collaborative Resize Interaction – One participant drags to the left, another to the right, forming the collaborative interaction on both canvases.

we envisioned that the collaborative web-based drawing application further allows users on mobile devices to perform advanced collaborative interactions[1]. We illustrate an example of collaborative interactions that can be applied:

Collaborative resize interaction. Consider when one participant on a mobile device starts to drag a drawn shape to the right, while at the same time a different participant in the same session drags the same shape to the left (Figure B.1). Typically an application recognizes two separate sequential interactions (e.g. drag-right and drag-left respectively), and proceeds to enforce techniques to determine ownership and which action to apply. In contrast, our sample application recognizes this as a single collaborative resize interaction. As a result, the application then reacts by causing the shape to resize according to the input of the participants.

B.2 Collaborative Interactions in Serena^s

We present the approach for coordinating complex interactions, such as the example in the previous section, in collaborative Web applications. We compare it with Mingo [Kam+15], an object-oriented framework employing web-based infrastructure to coordinate collaborative interactions. Mingo extends the JavaScript language with blended constructs for group coordination and complex event processing.

The rule shown in Listing B.1 for Mingo and B.2 is used to define the collaborative interaction for the distributed resize, and is first added to the server. Then, a number of interactive client devices connect to the server and start the application. They create or join rooms which represent a drawing session. In Serena^s, the rooms represent groups, and on the server side these groups are encoded by the rule engine. Whenever a simple event that has been denoted to comprise a complex interaction is performed, the invocation is intercepted and the event information is sent to the server. If a complex interaction has been realised, it is propagated to all clients in the session.

To clarify the coordination orchestration, we illustrate how a collaborative interaction is realized.

1. A developer writes the application in JavaScript and constructs to coordinate the application's distributed interactions.

2. The runtime sends the interactions and their relevant information to the server by means of WebSockets.

3. The server assembles the local interactions from client devices with their respective information and sends them as facts to the rule engine. Therefore, in this case, a fact is an internal representation of a simple interaction in the application.

[1]The users should already be aware of this e.g. from a description of the application features

4. The interactions are then asserted. If the interactions trigger a pre-defined rule created from constructs previously defined by the developer, it can combine the simple interactions into one composed event.

5. The composed event is then pushed back to the runtime which in turn delivers it to the awaiting client devices that defined a handler for the composed event.

Listing B.1: A Collaborative Resize Rule in Mingo

```
1 rule('collabResizeRule',
2  '(Invoked (function "touchMove") (dev ?d1) (args ?a1) (time ?time1) (room ?room))
3   (Invoked (function "touchMove") (dev ?d2) (args ?a2) (time ?time2) (room ?room))
4   (test (time:within ?time1 ?time2 1000))
5   →
6   (assert (collabResize (args ?a1 ?a2) (dev ?d1 ?d2) (room ?room))')
```

Listing B.2: Collaborative Resize Rule in Serena

```
1 { rulename: "collabResizeRule",
2   conditions:[
3    {$f1: {type:"invoked", function: "touchMove", dev:"?d1", args:"?a1", time:"?t1"}},
4    {$f2: {type:"invoked", function: "touchMove", dev:"?d2", args:"?a2", time:"?t2"}},
5    {type:"$test", expr: "time.within(?t1, ?t2, 1000)"},
6   ],
7   actions:[{assert: {type:"collabResize"  args:"[?a1,?a2]", devs:"[?d1,?d2]"}}]
8   scopes: [ "$f1 private $f2"],
9   notify: ["private $f1"]
10 }
```

The Mingo rule in Listing B.1 is defined by its name `collabResize`, actions and conditions, similar to SRL in Section 4.2.1. The conditions and actions in Mingo are delimited by a '→'. The first two conditions capture the move interaction on a shape in the same room (line 2 and 3). The next condition is a `test` condition that checks if the interaction was made at around the same time. If the interaction is confirmed, then the action taken is to assert the `collabResize` interaction with the arguments, client device IDs.

The SRL rule is similar to the Mingo rule. The difference lies when coordinating the interactions between sessions. The Mingo rule requires the logic of orchestrating room sessions to be interspersed with the logic of capturing the `touchMove` events for detecting the `collabResize` gesture. In lines 2, 3 and 6, the room logic is added to the constructs in the rule to ensure the interactions come from the same room session. Of course, in line 6 Mingo then broadcasts the `collabResize` notification to all drawing sessions, thus *in the client-side there needs to be a conditional check to see whether a received collaborative interaction is destined for that particular* `room`. In contrast, Line 8 of Listing B.2 uses the `private` scope to ensure that the two interactions occur in the from the same room group. Also, the notification to be sent to the drawing session of the room in question is performed in line 9. Even though the example mainly exposes the reentrant functionality provided by the Serena^s framework, the rules contain more intuitive scope constructs that can be applied separately from the normal rule logic. Furthermore, more flexible client notifications are supported.

C

Miss Manners' Benchmark in Serena

In this section we present a listing for the rules for the Miss Manners' benchmark explained in the evaluation chapter in Section 8.5. We performed the evaluation to compare the performance of the Serena engine with a mainstream benchmark.

In the Manners benchmark the engine needs to compute the seating arrangements for a number of guests at various tables in a dinner party. The setup of the basic benchmark contains 8 rules for determining the seating arrangements. The main crux of the benchmark is the high number of join computations in the rule engine to find compatible guests at a particular dinner table. We illustrate the rules of the benchmark used in the evaluation using the Serena Rule Language (SRL) syntax.

The AssignSeats rule

This rule creates the first `seating` arrangement and creates a `path` that will be used to match the next guests. It also sets the context to the next state of assigning seats. A counter is maintained for the number of seats that have been found.

Listing C.1: AssignSeats Rule

```
{   rulename: "assign_first_seat",
       conditions:[
           {$c1: {type:"context", state: "start"}},
           {type:"guest", name: "?n"},
           {$c3: {type:"count", num: "?c"}}
       ],
       actions:[
           {assert: {type: "seating", seat1: 1, name1: "?n", name2: "?n", seat2: 1,
             ↪ id:"?c", pid: 0, path_done: true}},
           {assert: {type: "path", id: "?c", name: "?n", seat: 1}},
           {modify: '$c3', with: {type:"count", num: "(?c + 1)"}},
           {modify: '$c1', with: {type:"context", state: "assign_seats"}}
       ]
}
```

The FindSeating rule

This is the main rule of the benchmark. It determines the compatibility of guests according to their hobbies and different genders. Internally, the rule engine will perform combinatorial tests for all the guests (excluding the guests that are already seated).

Listing C.2: FindSeating Rule

```
{   rulename: "find_seating",
    conditions:[
        {$c1: {type:"context", state: "assign_seats"}},
        {type: "seating", seat1: "?seat1", seat2: "?seat2", name2: "?n2", id:"?id",
          ↪ pid: "?pid", path_done: true},
        {type:"guest", name: "?n2", sex: "?s1", hobby: "?h1"},
        {type:"guest", name: "?g2", sex: "?s2", hobby: "?h1"},
        {$c3: {type:"count", num: "?c"}},
        {sign: "not", type: "path", id: "?id", name: "?g2"},
        {sign: "not", type: "chosen", id: "?id", name: "?g2", hobby:"?h1"}
    ],
    actions:[
        {assert: {type: "seating", seat1: "?seat2", name1: "?n2", name2: "?g2",
          ↪ seat2: "(?seat2 + 1)", id:"?c", pid: "?id", path_done: false}},
        {assert: {type: "path", id: "?c", name: "?g2", seat: "(?seat2 + 1)"}},
        {assert: {type: "chosen", id: "?id", name: "?g2", hobby:"?h1"}},
        {modify: '$c3', with: {type:"count", num: "(?c + 1)"}},
        {modify: '$c1', with: {type:"context", state: "make_path"}}
    ]
}
```

The MakePath and PathDone Rules

These two rules work in tandem to ensure all guests are seated at a table, using paths that represent the searching process of compatible guests. The MakePath rule adds a `path` for every `seating` and PathDone modifies a seating arrangement for a later check (that will see if the seating has been matched with a compatible guest).

Listing C.3: MakePath Rule

```
{   rulename: "make_path",
    conditions:[
        {$c1: {type:"context", state: "make_path"}},
        {type: "seating", id:"?id", pid: "?pid", path_done: false},
        {type: "path", id: "?pid", name: "?n1", seat: "?s"},
        {sign: "not", type: "path", id: "?id", name: "?n1", seat:"?s"}
    ],
    actions:[
        {assert: {type: "path", id: "?id", name: "?n1", seat: "?s"}}
    ]
}
```

Listing C.4: PathDone Rule

```
{   rulename: "path_done",
    conditions:[
        {$c1: {type:"context", state: "make_path"}},
        {$c2: {type: "seating", seat1: "?s1", name1: "?n1", name2: "?n2", seat2:
          ↪ "?s2", id:"?sid", pid: "?pid", path_done: false}}
    ],
    actions:[
        {modify: '$c1', with: {type:"context", state: "check_done"}},
        {modify: '$c2', with: {type: "seating", seat1: "?s1", name1: "?n1", name2:
          ↪ "?n2", seat2: "?s2", id:"?sid", pid: "?pid", path_done: true}}
    ]
}
```

The AreWeDone and Continue Rules

These rules are the terminating rules. As their name imply, the Continue rule checks if there are any `paths` that have not been explored and changes the state back to that of assigning seats, while the AreWeDone rule activates when the final seat has been assigned and changes the state to the final state of printing the results.

Listing C.5: Are We Done? Rule

```
{ rulename: "are_we_done",
    conditions: [
        {$c1: {type:"context", state: "check_done"}},
        {type: 'last_seat', seatno:"?lseat"},
        {$c2: {type: "seating", seat2: "?lseat"}}
    ],
    actions:[
        {modify: '$c1', with: {type:"context", state: "print_results"}}
    ]
}
```

Listing C.6: Continue Rule

```
{ rulename: "continue",
    conditions: [
        {$c1: {type:"context", state: "check_done"}}
    ],
    actions:[
        {modify:'$c1', with: {type:"context", state: "assign_seats"}}
    ]
}
```

The PrintResults and AllDone rules

The final housekeeping rules are used to ensure that the last seating has been done in rule PrintResults and subsequently stops the benchmark by removing the context fact from the engine in rule AllDone.

Listing C.7: Print Results Rule

```
{   rulename: "print_results",
    conditions:[
        {$c1: {type:"context", state: "print_results"}},
        {$c2: {type: "seating", seat2: "?s2", id:"?id"}},
        {type: 'last_seat', seatno:"?s2"},
        {$c4: {type: "path", id: "?id", name: "?n", seat:"?s"}}
    ],
    actions:[
        {retract: '$c4'}
    ]
}
```

Listing C.8: AllDone Rule

```

{ rulename: "all_done",
    conditions: [
        {$c1: {type: "context", state: "print_results"}}
    ],
    actions: [
        {retract: "$c1"}
    ]
}
```

D

University Security Access Rules

In this section we present a listing for some of the rules for the University Security Access scenario explained in the evaluation chapter in Section 8.1. The full set of rules, facts, templates and other assets can be found in `http://bit.ly/serena-uni-rules`

Listing D.1: Rules for University Access

```
var accessRules = [
/*Classtime access for any university hierarchy*/
 {rulename: "university-students-classtime-access",
  conditions: [
   {$s: {type: "student", name: "?nam", badgeid: "?badgid"}},
   {type: "accessreq", id: "?reqid", badgeid: "?badgid", time: "?t", deviceid: "?devid"},
   {$d: {type: "accessdevice", id: "?devid"}},
   {type: "$test", expr: "(hour(?t) > 8 && hour(?t) < 20 && (isWeekday(?t) == true) )"}
  ],
  actions: [
   {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
  ],
  scopes: ["$d visibleto $s",
   "$d private classes"]
 },
/*All vub students have access to classes during class times (can add during the week)*/
 {rulename: "vub-students-classtime-access",
  conditions: [
   {$s: {type: "student", name: "?nam", badgeid: "?badgid"}},
   {type: "accessreq", id: "?reqid", badgeid: "?badgid", time: "?t", deviceid: "?devid"},
   {$d: {type: "accessdevice", id: "?devid"}},
   {type: "$test", expr: "(hour(?t) > 8 && hour(?t) < 20 && (isWeekday(?t) == true) )"}
  ],
  actions: [
   {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
  ],
  scopes: ["$s subgroupof vubuni",
   "$d private classes"]
 },
/*Evening vub students are allowed extended access to classes in the evening (till 10pm)*/
 {rulename: "vub-evn-students-class-access",
  conditions: [
   {$s: {type: "student", name: "?nam", badgeid: "?badgid", evening: true}},
   {type: "accessreq", id: "?reqid", badgeid: "?badgid", time: "?t", deviceid: "?devid"},
   {$d: {type: "accessdevice", id: "?devid"}},
   {type: "$test", expr: "(hour(?t) > 17 && hour(?t) < 22) && (isWeekday(?t) == true)"}
  ],
```

```
  actions: [
   {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
  ],
  scopes: ["$s subgroupof vubuni",
   "$d private classes"]
 },
/*Generalize for any university or institution*/
 {rulename: "uni-evn-students-class-access",
  conditions: [
   {$s: {type: "student", name: "?nam", badgeid: "?badgid", evening: true}},
   {type: "accessreq", id: "?reqid", badgeid: "?badgid", time: "?t", deviceid: "?devid"},
   {$d: {type: "accessdevice", id: "?devid"}},
   {type: "$test", expr: "(hour(?t) > 8 && hour(?t) < 22) && (isWeekday(?t) == true)"}
  ],
  actions: [
   {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
  ],
  scopes: ["$d visibleto $s",
   "$d private classes"]
 },
/*Only ING bank employees have access to vub ING bank during working hours*/
 {rulename: "ing-staff-bank-access",
  conditions: [
   {$s: {type: "staff", name: "?nam", badgeid: "?badgid"}},
   {type: "accessreq", id: "?reqid", badgeid: "?badgid", time: "?t", deviceid: "?devid"},
   {$d: {type: "accessdevice", id: "?devid"}},
   {type: "$test", expr: "(hour(?t) > 7 && hour(?t) < 18)"}
  ],
  actions: [
   {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
  ],
  scopes: ["$s private ing",
   "$d private bank"]
 },
/*VUB staff and students are allowed car access to the university campuses after
  ↪ registering their cars*/
 {rulename: "vub-car-access",
  conditions: [
   {type: "car", no: "?no", ownername: "?onam"},
   {type: "accessreqcode", id: "?reqid", code: "?no", deviceid: "?devid"},
   {
    type: "or", conditions: [
     {$s: {type: "staff", name: "?onam"}},
     {$s: {type: "student", name: "?onam"}}
    ]
   },
   {$d: {type: "accessdevice", id: "?devid"}}
  ],
  actions: [
   {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
  ],
  scopes: ["$s subgroupof vubuni",
   "$d peerof etterbeek"]
 },
/*University staff and students are allowed car park access to their university between
  ↪ 11am and 4pm in the weekend*/
 {rulename: "uni-registered-car-access",
  conditions: [
   {type: "car", no: "?no", ownername: "?snam"},
   {type: "accessreqcode", id: "?reqid", code: "?no", deviceid: "?devid", time: "?t"},
   {
    type: "or", conditions: [
     {$s: {type: "staff", name: "?snam"}},
     {$s: {type: "student", name: "?snam"}}
    ]
   },
   {$d: {type: "accessdevice", id: "?devid"}},
   {type: "$test", expr: "(hour(?t) > 11 && hour(?t) < 16) && (isWeekday(?t) == false)"}
  ],
  actions: [
   {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
  ],
  scopes: ["$d visibleto $s",
   "$d peerof etterbeek"]
```

```
  },
/*All vub staff have access to their group's offices during working hours*/
  {rulename: "vub-staff-office-access",
   conditions: [
    {$s: {type: "staff", name: "?nam", badgeid: "?badgid"}},
    {type: "accessreq", id: "?reqid", badgeid: "?badgid", time: "?t", deviceid: "?devid"},
    {$d: {type: "accessdevice", id: "?devid"}},
    {type: "$test", expr: "(hour(?t) > 8 && hour(?t) < 19)"}
   ],
   actions: [
    {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
   ],
   scopes: ["vubuni supergroupof $s",
    "$d subgroupof $s",
    "$d private offices"]
  },
/*All academic staff have access to classes during working hours (and overtime)*/
  {rulename: "vub-academic-staff-class-access",
   conditions: [
    {$s: {type: "staff", name: "?nam", badgeid: "?badgid"}},
    {type: "accessreq", id: "?reqid", badgeid: "?badgid", time: "?t", deviceid: "?devid"},
    {$d: {type: "accessdevice", id: "?devid"}},
    {type: "$test", expr: "(hour(?t) > 7 && hour(?t) < 20)"}
   ],
   actions: [
    {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
   ],
   scopes: ["vubuni supergroupof $s",
    "$s subgroupof AP",
    "$d private classes"]
  },
/*All non-staff in KUL have access to depts if they have a pre-existing code for access
    ↪ to the corridors*/
  {rulename: "non-staff-code-access",
   conditions: [
    {type: "accessreqcode", id: "?reqid", code: "?code", time: "?t", deviceid: "?devid"},
    {type: "allowedaccesscode", code: "?code", issuer: "?staffnam"},
    {$s: {type: "staff", name: "?staffnam"}},
    {$d: {type: "accessdevice", id: "?devid"}}
   ],
   actions: [
    {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
   ],
   scopes: ["$s supergroupof $d",
    "$d private corridors"]
  },
/*All non-staff have access to ALL depts if they have a pre-existing code for access to
    ↪ the corridors and if the coe was issued by a senior administrative staff*/
  {rulename: "non-staff-code-access-admincode",
   conditions: [
    {type: "accessreqcode", id: "?reqid", code: "?code", time: "?t", deviceid: "?devid"},
    {type: "allowedaccesscode", code: "?code", issuer: "?staffnam"},
    {$s: {type: "staff", name: "?staffnam"}},
    {$d: {type: "accessdevice", id: "?devid"}}
   ],
   actions: [
    {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
   ],
   scopes: ["$s private senioradmin",
    "$d visibleto $s",
    "$d private corridors"]
  },
/*All WE students have access to WE department-level corridors, classes and student rooms
    ↪ on weekdays*/
  {rulename: "WE-students-WE-access",
   conditions: [
    {$s: {type: "student", name: "?nam", badgeid: "?badgid"}},
    {type: "accessreq", id: "?reqid", badgeid: "?badgid", time: "?t", deviceid: "?devid"},
    {$d: {type: "accessdevice", id: "?devid"}},
    {type: "$test", expr: "(isWeekday(?t) == true)"}
   ],
   actions: [
    {assert: {type: "accessrep", reqid: "?reqid", allowed: true}}
   ],
```

```
   scopes: ["$s subgroupof WE",
    "$d subgroupof WE",
    "$d peerof DINF",
    "$d private (corridors | classes | studyrooms)"]
  },
 /*Notify the secretaries that a delivery came in by a code issued.*/
  {
   rulename: "non-staff-secretary-code-access",
   conditions: [
    {type: "accessreqcode", id: "?reqid", code: "?code", time: "?t", deviceid: "?devid"},
    {type: "allowedaccesscode", code: "?code", issuer: "?staffnam"},
    {$s: {type: "staff", name: "?staffnam"}},
    {type: "accessrep", reqid: "?reqid", allowed: true},
    {$d: {type: "accessdevice", id: "?devid"}}
   ],
   scopes: ["$s private senioradmin",
    "$d visibleto $s",
    "$d private corridors"],
   notify: ["subgroupof $s",
    "private juniormgt"]
  },

 /*UNT professors get notifications of access to their labs by their thesis students on
   ↪ weekends*/
  {rulename: "notify-profs-master-students-promotor-lab-access",
   conditions: [
    {$stu: {type: "student", name: "?nam", badgeid: "?badgid", level: "master", promotor:
      ↪ "?staffid"}},
    {$stf: {type: "staff", id: "?staffid", name: "?stfnam"}},
    {type: "accessreq", id: "?reqid", badgeid: "?badgid", time: "?t", deviceid: "?devid"},
    {$d: {type: "accessdevice", id: "?devid"}},
    {type: "$test", expr: "( month(?t) > 1 && month(?t) < 9 )"},
    {type: "$test", expr: "( isWeekday(?t) == false )"}
   ],
   scopes: ["$stf subgroupof seniorAP",
    "$d subgroupof $stf",
    "$d private labs"],
   notify: ["subgroupof $stf",
    "private seniorAP"]
  },
 /*junior admins get notifications of personnel requesting access in late night hours*/
  {rulename: "log-accepted-or-denied-requests-outside-hours",
   conditions: [
    {$s: {type: "staff", name: "?stfnam", badgeid: "?badgid"}},
    {type: "accessreq", id: "?reqid", badgeid: "?badgid", time: "?t", deviceid: "?devid"},
    {$d: {type: "accessrep", reqid: "?reqid", allowed: "?allowed"}},
    {type: "$test", expr: "( hour(?t) < 6 && hour(?t) > 10 )"}
   ],
   scopes: ["junioradmin supergroupof $s"],
   notify: ["private serverrooms"]
  }
 ];
```

Bibliography

[Aas06] Jans Aasman. "Allegro graph: RDF triple database". In: *Cidade: Oakland Franz Incorporated* (2006).

[Aba+05] Daniel J Abadi et al. "The Design of the Borealis Stream Processing Engine." In: *Biennial Conference on Innovative Data Systems Research*. Vol. 5. 2005, pp. 277–289.

[ABJ89] Rakesh Agrawal et al. *Efficient management of transitive relationships in large data and knowledge bases*. Vol. 18. 2. ACM, 1989.

[Aït+89] Hassan Aït-Kaci et al. "Efficient Implementation of Lattice Operations". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11.1 (Jan. 1989), pp. 115–146. ISSN: 0164-0925. DOI: 10.1145/59287.59293. URL: http://doi.acm.org/10.1145/59287.59293.

[Ale15] Coldea Alexandru. "Prete: Persisting Rete". MA thesis. Brussels, Belgium: Vrije Universiteit Brussel, 2015.

[Ali+15] Cyril Alias et al. "Generating a business model canvas for Future-Internet-based logistics control towers". In: *4th International Conference on Advanced Logistics and Transport (ICALT), 2015*. IEEE. 2015, pp. 257–262.

[All97] Verna Allee. *The knowledge evolution: Expanding organizational intelligence*. Routledge, 1997.

[All83] James F. Allen. "Maintaining Knowledge About Temporal Intervals". In: *Commun. ACM* 26.11 (Nov. 1983), pp. 832–843. ISSN: 0001-0782. DOI: 10.1145/182.358434. URL: http://doi.acm.org/10.1145/182.358434.

[Aln+13] Dawood Alnajjar et al. "Implementing flexible reliability in a coarse-grained reconfigurable architecture". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.12 (2013), pp. 2165–2178.

[AS92] Bernd Amann and Michel Scholl. "Gram: a graph data model and query languages". In: *Proceedings of the ACM conference on Hypertext*. ACM. 1992, pp. 201–211.

[Ama15] Amazon Web Services, Inc. *Rules for AWS IoT*. http://docs.aws.amazon.com/iot/latest/developerguide/iot-rules.html. (Accessed on 12/10/2016). Apr. 2015.

[ALY09] Sihem Amer-Yahia et al. "Socialscope: Enabling information discovery on social content sites". In: *arXiv preprint arXiv:0909.2058* (2009).

[And13] John R Anderson. *The architecture of cognition*. Psychology Press, 2013.

[AG08] Renzo Angles and Claudio Gutierrez. "Survey of graph database models". In: *ACM Computing Surveys (CSUR)* 40.1 (2008), p. 1.

[ALB11] Leonardo Aniello et al. "Inter-domain stealthy port scan detection through complex event processing". In: *Proceedings of the 13th European Workshop on Dependable Computing*. ACM. 2011, pp. 67–72.

[Apa09] Apache Software Foundation. *Hadoop*. http://hadoop.apache.org. (Accessed on 01/03/2015). Mar. 2009.

[Apa14] Apache Software Foundation. *Kafka: A high-throughput, distributed messaging system*. kafka.apache.org. (Accessed on 21/04/2015). 2014.

[Apa15] Apache Software Foundation. *Apache Storm: Trident API*. http://storm.apache.org/releases/current/Trident-tutorial.html. (Accessed on 11/18/2016). 2015.

[Ara+04] Arvind Arasu et al. "Stream: The stanford data stream management system". In: *Book chapter* (2004).

[ABW06] Arvind Arasu et al. "The CQL continuous query language: semantic foundations and query execution". In: *The VLDB Journal—The International Journal on Very Large Data Bases* 15.2 (2006), pp. 121–142.

[Arm+10] Michael Armbrust et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[Aul+11] Stefan Aulbach et al. "Extensibility and data sharing in evolving multi-tenant databases". In: *Data engineering (ICDE), 2011 ieee 27th international conference on*. IEEE. 2011, pp. 99–110.

[Bai+13] Engineer Bainomugisha et al. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (Aug. 2013), 52:1–52:34. ISSN: 0360-0300. DOI: 10.1145/2501654.2501666. URL: http://doi.acm.org/10.1145/2501654.2501666.

[Bar+09] Davide Francesco Barbieri et al. "C-SPARQL: SPARQL for continuous querying". In: *Proceedings of the 18th international conference on World wide web*. ACM. 2009, pp. 1061–1062.

[Bas07] Tim Bass. "Mythbusters: event stream processing versus complex event processing". In: *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*. ACM. 2007, pp. 1–1.

[Bat+98] John Bates et al. "Using events for the scalable federation of heterogeneous components". In: *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*. ACM. 1998, pp. 58–65.

[Bat94] Don Batory. *The LEAPS algorithms*. Univ., Department of Computer Sciences, 1994.

[BV07] Thomas Bernhardt and Alexandre Vasseur. "Esper: Event stream processing and correlation". In: *ONJava, O'Reilly* (2007).

[Ber02] Bruno Berstel. "Extending the RETE algorithm for event management". In: *Temporal Representation and Reasoning, 2002. TIME 2002. Proceedings. Ninth International Symposium on*. IEEE. 2002, pp. 49–51.

[BMN97] Karell Bertet et al. "Lazy completion of a partial order to the smallest lattice". In: *International KRUSE Symposium: Knowledge Retrieval, Use and Storage for Efficiency*. 1997, pp. 72–81.

[Biz+13] Christian Bizer et al. "Deployment of RDFA, microdata, and microformats on the Web–a quantitative analysis". In: *International Semantic Web Conference*. Springer. 2013, pp. 17–32.

[Bol06] Harold Boley. "The RuleML family of web rule languages". In: *International Workshop on Principles and Practice of Semantic Web Reasoning*. Springer. 2006, pp. 1–17.

[BM11] Mr Jérôme Boyer and Hafedh Mili. "IBM websphere ilog jrules". In: *Agile business rule development*. Springer, 2011, pp. 215–242.

[Bra+91] David A Brant et al. "Effects of Database Size on Rule System Performance: Five Case Studies." In: *VLDB*. Vol. 91. 1991, pp. 287–296.

[BN13] Gerd Breiter and Vijay K Naik. "A framework for controlling and managing hybrid cloud service integration". In: *Cloud engineering (ic2e), 2013 ieee international conference on*. IEEE. 2013, pp. 217–224.

[Bri06] Timothy Brick. "OPS5: A Production Rule System for Expert Systems". In: (2006).

[Bro87] Frederick P Brooks. *No silver bullet: Essence and Accident in Software Engineering*. April, 1987.

[Bro09] Paul Browne. *JBoss Drools business rules*. Packt Publishing Ltd, 2009.

[Bry+09] Francois Bry et al. *Tutorial on Event Processing Languages*. www.slideshare.net/opher.etzion/debs2009-event-processing-languages-tutorial. (Accessed on 9/10/2016). July 2009.

[BFS00] Peter Buneman et al. "UnQL: a query language and algebra for semistructured data based on structural recursion". In: *The VLDB Journal: The International Journal on Very Large Data Bases* 9.1 (2000), pp. 76–110.

[Bus+45] Vannevar Bush et al. "As we may think". In: *The Atlantic Monthly* 176.1 (1945), pp. 101–108.

[C2F14] C2FO. *Nools: Rete based rules engine written in JavaScript*. https://github.com/C2FO/nools. (Accessed on 03/03/2014). 2014.

[CA94] Christian J Callsen and Gul Agha. "Open heterogeneous computing in ActorSpace". In: *Journal of Parallel and Distributed Computing* 21.3 (1994), pp. 289–300.

[Car+15] Paris Carbone et al. "Apache flink: Stream and batch processing in a single engine". In: *Data Engineering* (2015), p. 28.

[Car+86] Michael J Carey et al. "The architecture of the EXODUS extensible DBMS". In: *Proceedings on the 1986 international workshop on Object-oriented database systems*. IEEE Computer Society Press. 1986, pp. 52–65.

[Cet03] Ugur Cetintemel. "The aurora and medusa projects". In: *Data Engineering* 51.3 (2003).

[Cha+03] Sirish Chandrasekaran et al. "TelegraphCQ: continuous dataflow processing". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, pp. 668–668.

[CKT15] Ioannis K. Chaniotis et al. "Is Node.js a viable option for building modern web applications? A performance evaluation study". In: *Computing* 97.10 (2015), pp. 1023–1044. DOI: 10.1007/s00607-014-0394-9. URL: http://dx.doi.org/10.1007/s00607-014-0394-9.

[Che+00] Jianjun Chen et al. "NiagaraCQ: A scalable continuous query system for internet databases". In: *ACM SIGMOD Record*. Vol. 29. 2. ACM. 2000, pp. 379–390.

[Che+03] Mitch Cherniack et al. “Scalable Distributed Stream Processing”. In: *CIDR*. Vol. 3. 2003, pp. 257–268.

[CM97] Susan J Chinn and Gregory R Madey. “Evaluation and use of CLIPS for developing temporal expert systems”. In: *Proceedings of the 10th international conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*. Goose Pond Press. 1997, pp. 87–92.

[Cla83] William J Clancey. “The epistemology of a rule-based expert system—a framework for explanation”. In: *Artificial intelligence* 20.3 (1983), pp. 215–251.

[Coh91] Norman H. Cohen. “Type-extension Type Test Can Be Performed in Constant Time”. In: *ACM Transactions on Programming Languages and Systems* 13.4 (Oct. 1991), pp. 626–629. ISSN: 0164-0925. DOI: 10.1145/115372.115297. URL: http://doi.acm.org/10.1145/115372.115297.

[Com] The Baseline Company. *The Blind Men and the Elephant*. http://www.theblindelephant.com/the_blind_elephant_fable.html. (Accessed on 04/11/2016).

[Con16] Concur Technologies Inc. *The Concur Cloud Platform*. https://www.concur.nl/concur-integrated-ecosystem. (Accessed on 11/03/2016). Mar. 2016.

[Cór15] Patricio Córdova. “Analysis of Real Time Stream Processing Systems Considering Latency”. In: *University of Toronto patricio@ cs. toronto. edu* (2015).

[Cra+03] Chuck Cranor et al. “Gigascope: A Stream Database for Network Applications”. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’03. San Diego, California: ACM, 2003, pp. 647–651. ISBN: 1-58113-634-X. DOI: 10.1145/872757.872838. URL: http://doi.acm.org/10.1145/872757.872838.

[Cro11] James L Crowley. “Intelligent Systems: Reasoning and Recognition”. In: *ENSIMAG* 2 (2011), pp. 4–6.

[CMW87] Isabel F Cruz et al. “A graphical query language supporting recursion”. In: *ACM SIGMOD Record*. Vol. 16. 3. ACM. 1987, pp. 323–330.

[Cza12] Evan Czaplicki. “Elm: Concurrent FRP for Functional GUIs”. In: *Senior thesis, Harvard University* (2012).

[Dab+02] Frank Dabek et al. “Event-driven programming for robust software”. In: *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM. 2002, pp. 186–189.

[Dav11] Dave Evans. *The IoT, How the Next Evolution of the Internet Is Changing Everything*. http://bit.ly/cisco-internet-things. [Online; accessed 13-September-2014]. Apr. 2011.

[Day+88] Umeshwar Dayal et al. “The Hipac project: Combining active databases and timing constraints”. In: *ACM Sigmod Record* 17.1 (1988), pp. 51–70.

[DHW94] Umeshwar Dayal et al. “Active database systems”. In: (1994).

[DD14] Harvey M Deitel and Barbara Deitel. *Computers and Data Processing: International Edition*. Academic Press, 2014.

[Dem+07] Alan J Demers et al. “Cayuga: A General Purpose Event Monitoring System.” In: *CIDR*. Vol. 7. 2007, pp. 412–422.

[Det+14] Paolo Dettori et al. “Blueprint for business middleware as a managed cloud service”. In: *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE. 2014, pp. 261–270.

[Dev16] Google Developers. *Blockly: Overview*. https://developers.google.com/blockly/guides/get-started/web. (Accessed on 22/01/2016). June 2016.

[DHo04] Maja D’Hondt. “Hybrid aspects for integrating rule-based knowledge and object-oriented functionality”. PhD thesis. PhD thesis, Vrije Universiteit Brussel, 2004.

[Doo95] Robert B Doorenbos. “Production matching for large learning systems”. PhD thesis. University of Southern California, 1995.

[Dri11] Mike Driscoll. *Node.js and The JavaScript Age*. 2011. URL: http://gigaom.com/cloud/node-js-and-the-javascript-age/ (visited on 10/03/2015).

[Dun+06] Adam Dunkels et al. “Protothreads: simplifying event-driven programming of memory-constrained embedded systems”. In: *Proceedings of the 4th international conference on Embedded networked sensor systems*. Acm. 2006, pp. 29–42.

[EN10] Opher Etzion and Peter Niblett. *Event processing in action*. Manning Publications Co., 2010.

[FRS93] Françoise Fabret et al. *An adaptive algorithm for incremental evaluation of production rules in databases*. Institut national de recherche en informatique et en automatique, 1993.

[FYW15] Edurardo Fernandez et al. “Cloud Access Security Broker (CASB): A pattern for secure access to cloud services”. In: *4th Asian Conference on Pattern Languages of Programs, Asian PLoP ’15*. Tokyo, Japan, 2015.

[Fer+15] Raul Castro Fernandez et al. “Liquid: Unifying Nearline and Offline Big Data Integration.” In: *CIDR*. 2015.

[FS09] David Ferro and Eric Swedin. “History of Computing 2”. In: ed. by Timo Impagliazzo Johnand Järvi and Petri Paju. Berlin, Heidelberg: Springer, 2009. Chap. Computer Fiction: Towards Investigating the Importance of Science Fiction in the Historical Development of Computing, pp. 84–94. ISBN: 978-3-642-03757-3. DOI: 10.1007/978-3-642-03757-3_9. URL: http://dx.doi.org/10.1007/978-3-642-03757-3_9.

[FIC12] FICO. *High-Volume Batch Processing with the FICO Blaze Advisor business rules management system*. Tech. rep. FICO Inc., Jan. 2012.

[Fie+02] Ludger Fiege et al. “Engineering event-based systems with scopes”. In: *European Conference on Object-Oriented Programming*. Springer. 2002, pp. 309–333.

[For82] Charles L Forgy. “Rete: A fast algorithm for the many pattern/many object pattern match problem”. In: *Artificial intelligence* 19.1 (1982), pp. 17–37.

[For79] Charles Lanny Forgy. “On the efficient implementation of production systems”. PhD thesis. Carnegie-Mellon University, 1979.

[Fou15] Apache Software Foundation. *Apache Tomcat - WebSocket How-To*. https://tomcat.apache.org/tomcat-7.0-doc/web-socket-howto.html. (Accessed on 02/03/2016). Sept. 2015.

[Fri14] E.J. Fried. *Jess Website: Web Links*. http://www.jessrules.com/links/. (Accessed on 01/03/2016). Feb. 2014.

[Fri03] Ernest Friedman-Hill. *JESS in Action*. Vol. 46. Manning Greenwich, CT, 2003.

[Gha+13] Ahmad Ghazal et al. "BigBench: towards an industry standard benchmark for big data analytics". In: *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. ACM. 2013, pp. 1197–1208.

[GR98a] Joseph C Giarratano and G Riley. "CLIPS reference manual". In: *Basic Programming Guide, CLIPS Version* 6 (1998).

[GR98b] Joseph C Giarratano and Gary Riley. *Expert systems*. PWS Publishing Co., 1998, pp. 10–21.

[GP08] Adrian Giurca and Emilian Pascalau. "JSON rules". In: *Proceedings of the of 4th Workshop on Knowledge Engineering and Software Engineering, KESE* 425 (2008), pp. 7–18.

[GRC04] Matteo Golfarelli et al. "Beyond data warehousing: what's next in business intelligence?" In: *Proceedings of the 7th ACM international workshop on Data warehousing and OLAP*. ACM. 2004, pp. 1–6.

[GDJ99] Carlos J Alonso González et al. "A graphical rule language for continuous dynamic systems". In: *Computational Intelligence for Modelling, Control and Automation*. Vol. 55. Amsterdam, Netherlands, CIMCA-99. 1999, pp. 482–487.

[GK94] Suran Goonatilake and Sukhdev Khebbal. *Intelligent hybrid systems*. John Wiley & Sons, Inc., 1994.

[Gro00] Ralph Grove. "Internet-based expert systems". In: *Expert systems* 17.3 (2000), pp. 129–135.

[GKP99] Robert E Gruber et al. "The architecture of the READY event notification service". In: *Electronic Commerce and Web-based Applications/Middleware, 1999. Proceedings. 19th IEEE International Conference on Distributed Computing Systems Workshops on*. IEEE. 1999, pp. 108–113.

[Guo+07] Chang Jie Guo et al. "A framework for native multi-tenancy application development and management". In: *4th IEEE International Conference on Enterprise Computing, e-commerce, and E-Services, 2007. CEC/EEE 2007*. IEEE. 2007, pp. 551–558.

[Güt94] Ralf Hartmut Güting. "GraphDB: Modeling and querying graphs in databases". In: *VLDB*. Vol. 94. 1994, pp. 12–15.

[Gys+94] Marc Gyssens et al. "A graph-oriented object database model". In: *IEEE Transactions on Knowledge and Data Engineering* 6.4 (1994), pp. 572–586.

[HN94] Michel Habib and Lhouari Nourine. "Bit-vector encoding for partially ordered sets". In: *Orders, Algorithms, and Applications*. Springer, 1994, pp. 1–12.

[HO06] Philipp Haller and Martin Odersky. "Event-based programming without inversion of control". In: *JMLC*. Vol. 4228. Springer. 2006, pp. 4–22.

[Han92] Eric N Hanson. "Rule condition testing and action execution in Ariel". In: *ACM SIGMOD Record*. Vol. 21. 2. ACM. 1992, pp. 49–58.

[Han96] Eric N. Hanson. “The design and implementation of the Ariel active database rule system”. In: *IEEE Transactions on Knowledge and Data Engineering* 8.1 (1996), pp. 157–172.

[HH93] Eric N Hanson and Mohammed S Hasan. “Gator: An optimized discrimination network for active database rule condition testing”. In: *University of Florida.–Gainesville: CIS Departement* (1993).

[H+00] David Hay et al. “Defining business rules – What are they really?” In: *Final Report* (2000).

[Hay85] Frederick Hayes-Roth. “Rule-based systems”. In: *Communications of the ACM* 28.9 (1985), pp. 921–932.

[HS08] Huahai He and Ambuj K Singh. “Graphs-at-a-time: query language and access methods for graph databases”. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 405–418.

[HDX12] Li Heng et al. “Survey on multi-tenant data architecture for SaaS”. In: *International Journal of Computer Science Issues(IJCSI)* 9.6 (2012).

[Hoh06] Gregor Hohpe. “Programming without a call stack: Event-driven architectures”. In: *Objekt Spektrum* (2006).

[HP13] Florian Holzschuher and René Peinl. “Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j”. In: *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM. 2013, pp. 195–204.

[Hor94] Bruce M Horowitz. “Intermediate states as a source of non-deterministic behavior in triggers”. In: *Research Issues in Data Engineering, 1994. Active Database Systems. Proceedings Fourth International Workshop on*. IEEE. 1994, pp. 148–155.

[Hu+07] Jian Hu et al. “Demographic prediction based on user’s browsing behavior”. In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 151–160.

[Hwa+05] J-H Hwang et al. “High-availability algorithms for distributed stream processing”. In: *21st International Conference on Data Engineering (ICDE’05)*. IEEE. 2005, pp. 779–790.

[Jac98] Peter Jackson. *Introduction to expert systems*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[J+07] Dean Jacobs, Stefan Aulbach, et al. “Ruminations on Multi-Tenant Databases.” In: *BTW*. Vol. 103. 2007, pp. 514–521.

[Jai+08] Namit Jain et al. “Towards a streaming SQL standard”. In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1379–1390.

[JB14] Stephanie W Jamison and Joel P Brereton. *The Rigveda: the earliest religious poetry of India*. Vol. 1. 2014.

[JR13] Salim Jouili and Aldemar Reynaga. “imGraph: A distributed in-memory graph database”. In: *Social Computing (SocialCom), 2013 International Conference on*. IEEE. 2013, pp. 732–737.

[Joy10] Joyent Inc. *The Node.js Website*. 2010. URL: http://nodejs.org/ (visited on 11/09/2015).

[Kab+15] Jaap Kabbedijk et al. "Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective". In: *Journal of Systems and Software* 100 (2015), pp. 139–148.

[Kai+13] Stephen Kaisler et al. "Big data: issues and challenges moving forward". In: *System Sciences (HICSS), 2013 46th Hawaii International Conference on*. IEEE. 2013, pp. 995–1004.

[Kam] Kennedy Kambona. *Serena Rule Language ANTLRv3*. http://bit.ly/serena-antlr. (Accessed on 06/08/2018).

[KBD13] Kennedy Kambona et al. "An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications". In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. DYLA '13. Montpellier, France: ACM, 2013, 3:1–3:9. ISBN: 978-1-4503-2041-2. DOI: 10.1145/2489798.2489802. URL: http://doi.acm.org/10.1145/2489798.2489802.

[Kam+15] Kennedy Kambona et al. "Coordinating Collaborative Interactions in Web-based Mobile Applications". In: *Proceedings of the 2015 International Conference on Interactive Tabletops & Surfaces*. ACM. 2015, pp. 181–190.

[KBD15] Kennedy Kambona et al. "Serena: scalable middleware for real-time web applications". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM. 2015, pp. 802–805.

[KRD17a] Kennedy Kambona et al. "Efficient Matching in Heterogeneous Rule Engines". In: *Proceedings of the 30th The 30th International Conference on Industrial, Engineering, Other Applications of Applied Intelligent Systems (IEA/AIE 2017)*. Springer. 2017, to appear.

[KRD17b] Kennedy Kambona et al. "Reentrancy and Scoping for Multitenant Rule Engines". In: *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST)*. ScitePress. 2017, to appear.

[KKH11] Sungjoo Kang et al. "A design of the conceptual architecture for a multitenant saas application platform". In: *Computers, Networks, Systems and Industrial Engineering (CNSI), 2011 First ACIS/JNU International Conference on*. IEEE. 2011, pp. 462–467.

[Kim+14] Milhan Kim et al. "RETE-ADH: an improvement to RETE for composite context-aware service". In: *International Journal of Distributed Sensor Networks* 2014 (2014).

[Kum+98] Ravi Kumar et al. "Recommendation systems: A probabilistic analysis". In: *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*. IEEE. 1998, pp. 664–673.

[Kun87] Hideko S Kunii. "DBMS with graph data model for knowledge handling". In: *Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*. IEEE Computer Society Press. 1987, pp. 138–142.

[LM01] E Scott Larsen and David McAllister. "Fast matrix multiplies using graphics hardware". In: *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. ACM. 2001, pp. 55–55.

[LF91] Douglas B Lenat and Edward A Feigenbaum. "On the thresholds of knowledge". In: *Artificial intelligence* 47.1-3 (1991), pp. 185–250.

[Lev+04] Philip Levis et al. "The Emergence of Networking Abstractions and Techniques in TinyOS". In: *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*. NSDI'04. San Francisco, California: USENIX Association, 2004, pp. 1–1.

[Luc02] David Luckham. *The power of events*. Vol. 204. Addison-Wesley Reading, 2002.

[Luc06] David Luckham. "What's the Difference Between ESP and CEP". In: *Online Article, August* (2006).

[Lug05] George F Luger. *Artificial intelligence: structures and strategies for complex problem solving*. Pearson education, 2005.

[Ma11] Hua-Dong Ma. "Internet of Things: Objectives and Scientific Challenges". In: *J. Comput. Sci. Technol.* 26.6 (Nov. 2011), pp. 919–924. ISSN: 1000-9000. DOI: 10.1007/s11390-011-1189-5. URL: http://dx.doi.org/10.1007/s11390-011-1189-5.

[MP90] Michel Mainguenaud and Marie-Aude Portier. "Cigales: A graphical query language for geographical information systems". In: *Proc. 4th Intl. Symposium on Spatial Data Handling, Zürich*. Citeseer. 1990, pp. 393–404.

[MK93] Marcus A Maloof and Krys J Kochut. "Modifying rete to reason temporally". In: *Tools with Artificial Intelligence, 1993. TAI'93. Proceedings., Fifth International Conference on*. IEEE. 1993, pp. 472–473.

[Man15] Sergi Mansilla. *Reactive Programming with RxJS: Untangle Your Asynchronous JavaScript Code*. Pragmatic Programmers, 2015.

[MNM77] J McDermott et al. "The efficiency of certain production system implementations". In: *ACM SIGART Bulletin* 63 (1977), pp. 38–38.

[Mey+09] Leo A Meyerovich et al. "Flapjax: a programming language for Ajax applications". In: *ACM SIGPLAN Notices*. Vol. 44. 10. ACM. 2009, pp. 1–20.

[MTS05] Mark S Miller et al. "Concurrency among strangers". In: *International Symposium on Trustworthy Global Computing*. Springer. 2005, pp. 195–229.

[Mir14] Daniel P Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production System*. Morgan Kaufmann, 2014.

[MFP06] Gero Mühl et al. *Distributed event-based systems*. Springer Science & Business Media, 2006.

[NGR88] P Pandurang Nayak et al. "Comparison of the Rete and Treat Production Matchers for Soar". In: *AAAI*. 1988, pp. 693–698.

[New73] Allen Newell. *Production systems: Models of control structures*. Tech. rep. DTIC Document, 1973.

[Nun+14] Bruno Astuto A Nunes et al. "A survey of software-defined networking: Past, present, and future of programmable networks". In: *IEEE Communications Surveys & Tutorials* 16.3 (2014), pp. 1617–1634.

[Ora11] Oracle Corporation. *JSR 315: Java™ Servlet 3.0 Specification*. 2011. URL: https://www.jcp.org/en/jsr/detail?id=315 (visited on 03/09/2015).

[Ous96] John Ousterhout. "Why threads are a bad idea (for most purposes)". In: *Presentation given at the 1996 Usenix Annual Technical Conference*. Vol. 5. San Diego, CA, USA. 1996.

[Pat+11] M. Pathirage et al. "A Multi-tenant Architecture for Business Process Executions". In: *2011 IEEE International Conference on Web Services (ICWS)*. July 2011, pp. 121–128. DOI: 10.1109/ICWS.2011.99.

[Pay16] Robbie Payne. *Google Assistant At The MadeByGoogle Event*. https://chromeunboxed.com/google-assistant-steals-the-andromeda-thunder-at-the-madebygoogle-event/. (Accessed on 10/06/2016). Oct. 2016.

[Pef+06] Ken Peffers et al. "The design science research process: a model for producing and presenting information systems research". In: *Proceedings of the first international conference on design science research in information systems and technology (DESRIST 2006)*. sn. 2006, pp. 83–106.

[Pel+15] Tuomas Pelkonen et al. "Gorilla: a fast, scalable, in-memory time series database". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1816–1827.

[Pet+14] Martin Peters et al. "Scaling parallel rule-based reasoning". In: *European Semantic Web Conference*. Springer. 2014, pp. 270–285.

[Pro13] Mark Proctor. *Drools & jBPM: R.I.P. Rete time to get PHREAKY*. http://blog.athico.com/2013/11/rip-rete-time-to-get-phreaky.html. (Accessed on 04/11/2015). Nov. 2013.

[Pro15] Mark Proctor. *Drools - Testimonials and Case Studies*. http://www.drools.org/learn/testimonialsAndCaseStudies.html. (Accessed on 02/06/2016). July 2015.

[Rai13] Rohit Rai. *Socket.IO: Real-time Web Application Development*. Packt Publishing Ltd, 2013.

[RT01] Olivier Raynaud and Eric Thierry. "A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests". In: *European Conference on Object-Oriented Programming*. Springer. 2001, pp. 165–180.

[Red15] Redhat. *Websockets in JBoss*. http://bit.ly/java-websockets-info. (Accessed on 02/03/2016). May 2015.

[Res+09] Bernd Resch et al. "Live Geography–Embedded Sensing for Standarised Urban Environmental Monitoring". In: (2009).

[RD17] Bob Reynders and Dominique Devriese. "Efficient Functional Reactive Programming Through Incremental Behaviors". In: *Asian Symposium on Programming Languages and Systems*. Springer. 2017, pp. 321–338.

[Ril91] Gary Riley. "Clips: An expert system building tool". In: (1991).

[RG92] Mark Roseman and Saul Greenberg. "GroupKit: A groupware toolkit for building real-time conferencing applications". In: *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*. ACM. 1992, pp. 43–50.

[Ros03] Ronald G Ross. *Principles of the business rule approach*. Addison-Wesley Professional, 2003.

[Sal15] SalesForce. *The Force Multitenant Architecture*. https://developer.salesforce.com/page/Multi_Tenant_Architecture. (Accessed on 11/12/2016). Dec. 2015.

[SM14] Guido Salvaneschi and Mira Mezini. "Towards reactive programming for object-oriented applications". In: *Transactions on Aspect-Oriented Software Development XI* 8400 (2014), pp. 227–261.

[Sch+08] Fabian Schneider et al. “The new web: Characterizing ajax traffic”. In: *International Conference on Passive and Active Network Measurement*. Springer. 2008, pp. 31–40.

[Sch+11] Christophe Scholliers et al. “Midas: a declarative multi-touch interaction framework”. In: *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction*. ACM. 2011, pp. 49–56.

[Sch+86] Marshall I Schor et al. “Advances in Rete pattern matching”. In: *AAAI*. Vol. 86. 1986, pp. 226–232.

[SPT83] Lenhart K. Schubert et al. “Determining type, part, color and time relationships”. In: *IEEE Computer* 16.10 (1983), pp. 53–60.

[SLR94] Praveen Seshadri et al. “Sequence query processing”. In: *ACM SIGMOD Record*. Vol. 23. 2. ACM. 1994, pp. 430–441.

[SN71] Herbert A Simon and Allen Newell. “Human problem solving: The state of the theory in 1970.” In: *American Psychologist* 26.2 (1971), p. 145.

[So11] Kuyoro So. “Cloud computing security issues and challenges”. In: *International Journal of Computer Networks* 3.5 (2011).

[SK91] Michael Stonebraker and Greg Kemnitz. “The POSTGRES next generation database management system”. In: *Communications of the ACM* 34.10 (1991), pp. 78–92.

[SRH90] Michael Stonebraker et al. “The implementation of POSTGRES”. In: *IEEE transactions on knowledge and data engineering* 2.1 (1990), pp. 125–142.

[SÇZ05] Michael Stonebraker et al. “The 8 requirements of real-time stream processing”. In: *ACM SIGMOD Record* 34.4 (2005), pp. 42–47.

[Swa+13] Janwillem Swalens et al. “Cloud PARTE: elastic complex event processing based on mobile actors”. In: *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*. ACM. 2013, pp. 3–12.

[The15] The Netty Project. *Netty NIO*. http://netty.io/. (Accessed on 01/18/2016). May 2015.

[The12] The PostgreSQL Global Development Group. *PostgreSQL Documentation: The Rule System*. https://www.postgresql.org/docs/9.4/static/rules.html. (Accessed on 01/11/2016). May 2012.

[Thi07] Rajkumar Thirumalainambi. “Pitfalls of JESS for Dynamic Systems.” In: *Artificial Intelligence and Pattern Recognition*. Citeseer. 2007, pp. 491–494.

[UM99] Naohiko Uramoto and Hiroshi Maruyama. “InfoBus repeater: a secure and distributed publish/subscribe middleware”. In: *Parallel Processing, 1999. Proceedings. 1999 International Workshops on*. IEEE. 1999, pp. 260–265.

[Val+09] Emanuele Della Valle et al. “It’s a streaming world! Reasoning upon rapidly changing information”. In: *IEEE Intelligent Systems* 24.6 (2009), pp. 83–89.

[Van+17] Sam Van den Vonder et al. “Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model”. In: (2017).

[VOE11] Richard L Villars et al. “Big data: What it is and why you should care”. In: *White Paper, IDC* (2011).

[VHK97] Jan Vitek et al. *Efficient type inclusion tests*. Vol. 32. 10. ACM, 1997.

[Wag03] Gerd Wagner. "Web rules need two kinds of negation". In: *International Workshop on Principles and Practice of Semantic Web Reasoning*. Springer. 2003, pp. 33–50.

[WBG08] Karen Walzer et al. "Relative temporal constraints in the Rete algorithm for complex event detection". In: *Proceedings of the second international conference on Distributed event-based systems*. ACM. 2008, pp. 147–155.

[WL05] Fusheng Wang and Peiya Liu. "Temporal management of RFID data". In: *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment. 2005, pp. 1128–1139.

[WLY04] Hai-Feng Wang et al. *Search engine with natural language-based robust parsing for user query and relevance feedback learning*. US Patent 6,766,320. July 2004.

[Wan+14] Jinghan Wang et al. "A distributed rule engine based on message-passing model to deal with big data". In: *Lecture Notes on Software Engineering* 2.3 (2014), p. 275.

[Wan+02] Xiaohang Wang et al. "Semantic space: An infrastructure for smart spaces". In: *Computing* 1.2 (2002), pp. 67–74.

[Wan15] Yingwei Wang. "The relationships among cloud computing, fog computing, and dew computing". In: *Dew Computing Research* Nov 12 (2015).

[Way15] Waylay.io. *Waylay: Smart Reasoning for IoT*. http://www.waylay.io/blog-one-rules-engine-to-rule-them-all.html. (Accessed on 15/10/2016). Mar. 2015.

[Wid+07] Alexander Widder et al. "Identification of suspicious, unknown event patterns in an event cloud". In: *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*. ACM. 2007, pp. 164–170.

[Wid96] Jennifer Widom. "The starburst active database rule system". In: *IEEE Transactions on Knowledge and Data Engineering* 8.4 (1996), pp. 583–595.

[Wig12] Steef-Jan Wiggers. *BizTalk Server Business Rule Engine*. http://bit.ly/biztalk-engine. (Accessed on 11/08/2016). Jan. 2012.

[WST09] Jan Wloka et al. "Refactoring for reentrancy". In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2009, pp. 173–182.

[Wor15] Workday Research. *Workdays Technology Platform and Development Processes*. Tech. rep. Workday Inc., Jan. 2015.

[Wor14] World Wide Web Consortium. *CGI: Common Gateway Interface*. 2014. URL: http://www.w3.org/CGI/ (visited on 11/20/2015).

[WM03] Ian Wright and James A. R. Marshall. "The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case". In: *Int. J. Intell. Games & Simulation* 2.1 (2003), pp. 36–48.

[Wu+01] Dapeng Wu et al. "Streaming video over the Internet: approaches and directions". In: *IEEE Transactions on circuits and systems for video technology* 11.3 (2001), pp. 282–300.

[XZ10] Ding Xiao and Xiaoan Zhong. "Improving Rete algorithm to enhance performance of rule engine systems". In: *Computer Design and Applications (ICCDA), 2010 International Conference on*. Vol. 3. IEEE. 2010, pp. V3–572.

[Yan+09] Jun Yan et al. “How much can behavioral targeting help online advertising?” In: *Proceedings of the 18th international conference on World wide web*. ACM. 2009, pp. 261–270.

[Zam+14] Jesse Zaman et al. “Citizen-Friendly participatory campaign support”. In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on*. IEEE. 2014, pp. 232–235.

[ZDZ09] YH Zhang et al. “An extensible event-driven manufacturing management with complex event processing approach”. In: *International Journal of Control and Automation* 2.3 (2009), pp. 1–12.

[Zhi15] Zhiqiang. *Process of Developing CQL Applications: Huawei BigData/StreamCQL Wiki*. https://github.com/HuaweiBigData/StreamCQL/wiki/Process-of-Developing-CQL-Applications. (Accessed on 01/11/2016). Oct. 2015.

[ZG01] Yoav Zibin and Joseph Yossi Gil. “Efficient subtyping tests with PQ-encoding”. In: *ACM SIGPLAN Notices*. Vol. 36. 11. ACM. 2001, pp. 96–107.

Index