Harnessing Community Knowledge in Heterogeneous Rule Engines

Kambona, Kennedy Kondo; Renaux, Thierry; De Meuter, Wolfgang

Published in:
Lecture Notes in Business Information Processing

DOI:
10.1007/978-3-319-93527-0_7

Publication date:
2018

Document Version:
Final published version

Harnessing Community Knowledge in Heterogeneous Rule Engines

Kennedy Kambona, Thierry Renaux, and Wolfgang De Meuter

Software Languages Lab, Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussels, Belgium
{kkambona,trenaux,wdemeuter}@soft.vub.ac.be
http://soft.vub.ac.be

Abstract. Currently, there is a lack of rule-based approaches that offer rich semantics that developers can use to exploit community knowledge contributed by distributed heterogeneous clients. Such abstractions can be useful in a number of applications to deal with the problem of orchestrating data patterns in a heterogeneous setting. This work presents scope-based reasoning in heterogeneous rule engines as a means to capture collective intelligence via community knowledge. Using scoped rules, rule designers can detect patterns in real-time data and to realise grouping structures in heterogeneous applications backed by a common rule-based system. The proposed solution exploits the fact that much of the heterogeneous community knowledge significant when performing reasoning and deductions can be structured hierarchically. We evaluate our work through a simulated case study, confirming that our technique presents a viable approach for efficiently processing community knowledge in heterogeneous environments.

Keywords: Rule-based systems, community knowledge, Rete, scopes, business rules

1 Introduction

As a result of todays dynamic software environment, clients require real-time processing in modern data-intensive applications: where the clients contribute data as events and expect to receive instantaneous feedback through notifications. 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 sent continuously over distributed networks to application servers in various forms as out-of-order, partially unbounded and/or time-varying sequences. Accordingly, there is a current need for *knowledge-intensive* techniques that ease the dynamic definition of constraints in order to extract value from such continuous, *reactive* event data. This will enable the system to infer potential higher-level knowledge that contributed data may uncover. Currently, the technologies available to meet these goals is

The final authenticated publication is available online at https://doi.org/10.1007/978-3-319-93527-0

often times complicated and limiting due to the non-determinism and sheer volume of data to discern patterns on [1].

Recent advancements in tackling this problem has re-discovered the use of forward-chaining rule-based systems (RBS) in areas that embrace the use of *business rules* [2]. Rule-based systems provide a declarative approach that represents the conceptual logic of a system in an application-independent way. The fundamental power that rule-based systems expose is not only from complex inference mechanisms but also from the ability for them to embrace rich knowledge bases that reflect aspects of the real world [3].

Fundamentally, 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. In contrast, the online software ecosystem is definitively characterised by a heterogeneous environment where different types of client devices contribute vastly diverse types of data for processing. Sharing brought about by Utility Computing [4] is significant in exploiting the *collective knowledge* that can be discovered from the data that these devices contribute.

Sharing however unearths issues that can be attributed to classically-isolated rule engine design. Classical rule engines suffer from a lack of proper modularisation when installed to serve 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. They are thus said to be *non-reentrant* [1]. Most rule-based approaches provide basic techniques that do not intrinsically support the flexibility and expressiveness in customising specific client behaviour. The methods employed to mitigate these problems unnecessarily retract the gains made by using a rule-based system in the first place: they increase the complexity of application development making it fallible, and muddles the design of the conceptual logic of the application. They furthermore lack the proper mechanisms in which to exploit improvements in efficiency that can be realised in a heterogeneous setup.

In this work we augment the support the efficient processing of heterogeneous, multi-user applications through a technique that enforces the consolidation and partitioning of client constraints and data defined using special rule-based programming constructs. We present *scoped rules*, that enable rule creators to distinguish between events pertaining to different sources while keeping this logic cleanly separated from the application logic. As such, the basic purpose of the rule is not muddied with the logic required for distinguishing clients, leaving the logical intent of a rule easy to understand for a rule creator.

At the same time, scoping enables the system to exploit a number of performance optimisations in the server's rule engine during its matching process. The approach of encoding the physical, structural or other logical organisations of multi-user applications eases the computational workload of the inference algorithm. We show that the underlying rule engine can effectively use such abstractions to generally decrease the engine's overall response time and thus improving overall processing efficiency when compared to classic methods.

The document is organised as follows. Section 1.1 discusses the significance of incorporating community knowledge and presents a motivating example of a representative heterogeneous application. Sections 2-5 describe the proposed heterogeneous rule-

based system and its execution semantics. Section 6 then proceeds to evaluate the work against current methods, and Section 7 outlines the ways in which similar approaches can be used to support the capture of community knowledge in other heterogeneous environments. Section 8 recapitulates the main points and points out some limitations and possible avenues for future improvements.

1.1 The Significance of Community Knowledge

Value is a significant feature when reasoning on data contributed by client devices to discern useful patterns [4]. Discovering interesting patterns from such intermittent, dispersed or entangled pieces of data greatly improves data quality. A good analogy for this is the ancient *Parable of the Blind Men and the Elephant* that originated from the *Rigveda* collection from ancient India [5]. The fable 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 are then required to describe the elephant based on this. 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, each says the elephant could be a wall, a spear or a rope depending on the limited region (respectively, the side, the tusk or the tail) they can access, Figure 1. The fable ends with the blind men in complete disagreement of what the object is.

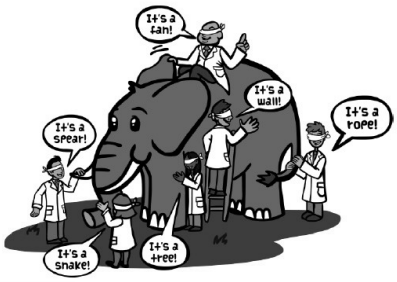


Fig. 1: The Blind Men and the Elephant – Each blind man has a limited view of the elephant and come up with different conclusions of what the object is (Image sourced from [6]).

In the same way, value from data contributed by a variety of sources or clients can potentially be increased if it is processed collectively (cf., discovering that the object is indeed an elephant). 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 manner of processing is of practical significance is in the organisational intelligence domain, where the result is often termed as collective intelligence [7].

This work goes further in defining a concise collection of useful information sourced from different clients that can be grouped according to logical or physical structures,

that we denominate as **community knowledge**. Community knowledge is important 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 different clients. This type of composite knowledge encompasses the variety of data that is often contributed in these environments.

This outlook stands above other similar approaches because a diverse range of autonomous sources can contribute reactive data in one integrated system. This has a profound effect on its complexity and the relationships that can be unearthed. For example, traditional systems were focused on relating features with a single independent entity, such as age, date of birth, gender, etc. Today, an individual's features as well as relationships form a social connection with other entities through relatable hobbies and other interests. In traditional systems, individuals are only linked if they have same basic features. In a modern, dynamic community, two individuals can be linked via these social connection relationships even though they do not share any of these intrinsic features.

1.2 Motivating Example

We now present a representative practical example that motivates the need for exploiting community knowledge in a heterogeneous environment. To highlight the requirements that such a system should meet, we describe a scenario of a security monitoring system in an university environment that monitors access patterns. The service can be deployed in a heterogeneous configuration (such as through a Cloud Access Security Broker [8]) to monitor and log access in several institutions. The example follows that explained in [1].

University Services Access Control. The security departments of universities in Brussels have embarked on improving the security of their campuses. They have passed regulations that require the security departments enforce protocols to monitor accesses of all staff and other students and staff in their institutions.

Devices that scan badges issued to students and staff have been installed at major access points throughout the university premises. The badges are contactless smart cards that can be read by the access devices. To gain access to any part of the university, a person is required to scan their badge on the device. We outline some of the security protocols that have been drafted by the security department from the regulations below.

1. Students at all levels (bachelor, master, etc.) have access to classrooms during class times on weekdays
2. Only vehicles of students and staff are allowed to enter the underground parking in campus buildings

In Figure 2 we illustrate a simplified common structure for a university consisting of different hierarchies: personnel, physical structures and research department hierarchies. As a result, specific departments and units can therefore define custom access policies:

3. Biology department students are allowed access to all labs in the (sub)departments in the weekends if accompanied by senior academic staff

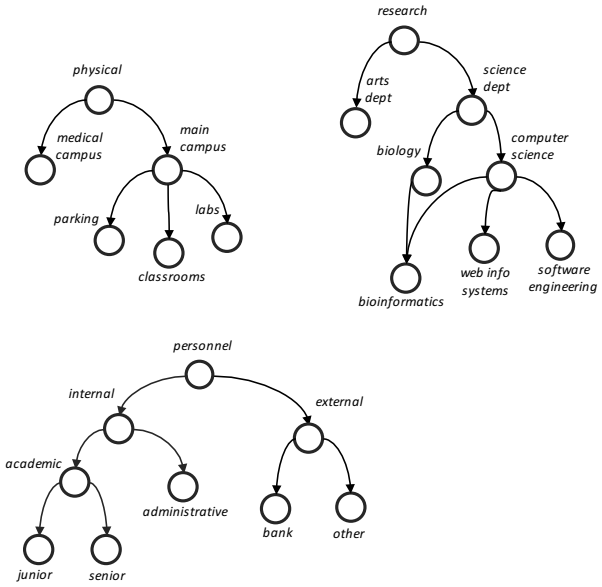


Fig. 2: Structural organisation of a university [1] – The structures can be modelled as physical locations, research groups and personnel; spanning students, staff and physical structures.

4. Only campus bank employees and consultants have access to the bank back office during working hours

A system supporting this example should be capable of defining both types of policies, and of processing the requests from all users efficiently against the protocols at varied times whenever any request is made. For instance in policy 1, when a student on a university accesses a classroom during class times the monitoring dashboard would show a status to indicate whether the access is acceptable or otherwise. Such kind of processing model is common in forward-chaining rule-based systems in the complex event processing domain.

2 The Serena Rule-based System

The example outlined in the previous section is a representative of a reactive heterogeneous application. In this work, we particularly target the dynamic design of such knowledge-intensive, data-driven applications that continuously stream data back and forth between clients and the server. These systems are required to manage the shared knowledge base reused by the various applications they support. In order to reason about data sent by client devices and to extract higher-level knowledge from it, it is vital that the value of the sent data be processed efficiently. Instead of hard-coding all this shared knowledge using conventional techniques, developers often encode this knowledge in the form of rules to specify detection logic. In such situations, a modern rule engine can be used to accommodate the knowledge for all clients of a heterogeneous configuration. This is the main idea behind the design of the Serena framework [1] .

2.1 The Serena Framework

Serena is a rule-based framework that augments an event-driven web server with a forward-chaining inference engine that processes event data reactively using rules. In Serena clients create and install the logic reactive rules that define the complex events they are interested in. The rules specify which data to match, and once activated the rule can send this activation as notification to clients.

2.2 Serena's Execution Semantics

Producing efficient rule-based functionality requires maximum efficiency due to complex pattern-matching techniques of rule-based semantics. Reducing the amount of matching in rules therefore guarantees faster server execution. The Serena runtime is based on the production systems model of knowledge representation [9], which uses data-sensitive rules rather than sequenced instructions as the basis of computation.

We use the university access control's example protocol 1 to explain the semantics of execution in a typical heterogeneous rule engine.

Rule Syntax. Rules support defining constraints that will enforce pattern-matching within conditions in order to execute some actions. The university policies from the scenario in Section 1.2 can be easily expressed in a rule-based format. We show such a rule to be added by a university security staff using a syntax similar to JSON Rules [10] in Listing 1.1 for the classroom protocol 1. Remember that the protocol specified that students are only allowed access to classrooms on weekdays. The rule in JSON format is received by the framework's server when shipped from the client.

Listing 1.1: Rule for allowing classroom accesses to a student

```
1  {rulename: "classtime-access1",
2   conditions:[
3    {type:"student", name: "?name"},
4    {type:"accessdevice", name:"?dev", location:"classroom"},
5    {type:"accessreq", id: "?reqid", person: "?name", time: "?t", device: "?dev"},
6    {type:"$test", expr:"(hourBetween(?t, 8, 20) && (isWeekday(?t) == true) )"}
7   ],
8   actions:[
9    {assert: {type: "accessrep", reqid:"?reqid", allowed: true}}
10  ]
11 }
```

The 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 2-6) that specify constraints on incoming events for event detection. The RHS contains the `actions` to be taken when the conditions have been fulfilled (lines 8-10). The LHS of the listed rule definition captures the access request from a person on an ID scanning device within the specified time periods (line 6). In the rule the '`?`' operator denotes a variable binding (e.g. `?name` in lines 3 & 5).

When all the conditions specified in the LHS are satisfied, then the actions defined in the RHS are activated. In the example rule, the RHS asserts that the access request has been granted by the reply in line 9 from the request captured in line 5.

The Rete Algorithm. When the rule engine receives the rule it builds a discrimination network via the Rete algorithm [11] in its inference engine. Rete compiles rules into a data-flow graph that filters facts (data) as they propagate through nodes performing a match-select-execute process. Rete inference engines perform efficient matching, a technique that reasons over the data to detect consistent bindings for constraints in rules that need to be fulfilled. Efficient matching is achieved by exploiting two characteristics. The first is structural similarity which involves sharing similar nodes when building the graph. The second is temporal redundancy which is the caching of intermediate matched data tokens between cycles of incoming results, albeit at the price of higher memory usage from the added caches.

The Rete Graph. We show the graph for the *classtime-access* rule previously shown in Listing 1.1 after addition in the server. The graph consists of two regions, the alpha and beta 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 `student`.

The beta network is built in the lexical order of the conditions in a rule, forming a left-associative binary tree. Two-input *beta nodes* or *join nodes* perform tests or joins between conditions on their left and right inputs. Each beta node is associated with its *beta memory* and holds the intermediate join test results. The leftmost beta node in Figure 3 from [1] performs joins for a the name of a `students` with the name of the person performing the `access request`. If the test passes, it will create a *token* of both facts in the result and send it to the next node. Each beta node can be connected to subsequent nodes in the beta network as a left input. In most beta nodes the right input is connected to the output of an alpha node's memory. In Figure 3, the second beta node receives the token and performs joins of facts from a scanning `accessdevice` with the device of the `accessrequest` made.

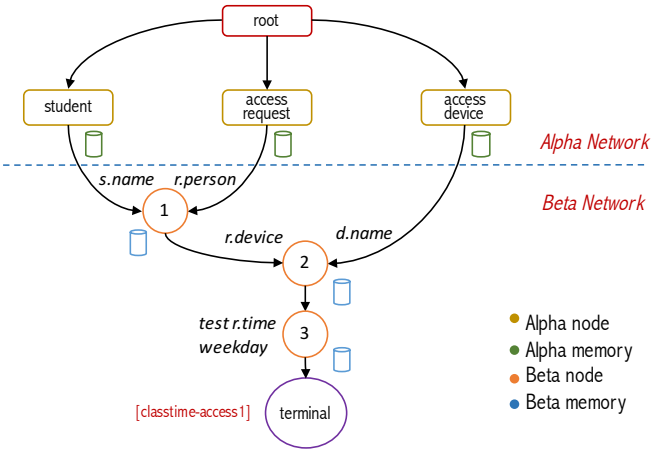


Fig. 3: The Rete graph built for the classtime access rule – The graph contains alpha and beta nodes with intermediate memories [1].

The beta network also hosts the nodes that represent test conditions as beta test nodes, e.g., the third beta node in Figure 3 that checks for the compatibility of the time that the access request was made. The last beta nodes in any Rete graph represent the full activation of a particular rule (or in some cases, rules) and is named a *terminal node*. If a token reaches a terminal node then the rule associated with that node, in this case the *classtime access* rule, will be instantiated.

The Matching Process. In Serena, every fact base update triggers a matching process. The matching process searches for consistent bindings between facts and the existing rules. Incoming data is received by the server which it creates an instances of facts. The facts are then inserted into the graph from the root node. Facts traverse down the network as they are processed and forwarded by nodes, which store intermediate computations in local memories.

If a number of students arrive at the university by accessing campus entry points, this data can be captured and inserted as facts into the rule engine. The student facts will be inserted into the network starting from the root node and will eventually be stored as the intermediate results in the `student` node. Similarly, the access devices that are online will be stored in the memory of the `accessdevice` node. The beta node `1` now will perform its join operations when an access request fact is received.

When a student requests access to a classroom at around 12pm (this request should be granted), the engine will receive it and eventually add it as an `accessrequest` fact to the Rete graph. The fact will be sent from the root to the `accessrequest` node. This node will store the fact in its alpha memory and will send it to its child, beta node `1`.

It will be received at the right input beta node `1`, causing a *right activation* which will issue a request for all the items in its left parent to compute consistent bindings for the fact. On the other hand, a *left activation* is triggered when a data item is received at the left input: where the join test will request all items from the node's right parent (which is always an alpha memory).

The beta node `1` will therefore request items from its left parent the `student` alpha node. The alpha node will send all `student` facts that it contains. Beta node `1` now proceeds to perform its join test `(s.name == r.person)`, which checks if the name of any of the `student` facts is the same as that of the `accessrequest` fact.

When the person that made the request matches the name, then node first creates a new token by appending the `student` fact with the `access request` fact, stores this intermediate result in its beta memory, and sends the new token to its child, node `2`. Thereafter, the same sequence of steps occur at node `2`, but this time a left activation is triggered to find out compatible access devices by performing join tests `(r.device == d.name)` on all devices from the alpha memory of `accessdevice`.

If a compatible device is found then a token is created and sent to the test node `3` 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, which means that the rule should be activated. In this case, the request made by a student to enter the classroom will be granted, and this rule and its bindings are sent to the scheduler for execution and eventual notification to the client.

The Cost of Matching. The matching stage determines the rules that are relevant to the current state of the fact base for activation. As identified in [12], a major bottleneck in the Rete algorithm is, unsurprisingly, the expensive computations during this stage. Concretely, as much as 90% of the execution of a Rete-based system can be spent in the match phase, with the number of join comparisons made dominating the time the matching process takes. 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.

3 Reentrancy in Heterogeneous Rule-based Systems

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 inputs cannot affect each other, whether run sequentially or concurrently [13]. 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. This way if the program is interrupted due to scheduling optimisations, for example, the program can be *re-entered* at any point in time without concern that programs that were executing during the interruption modified any of its data[1].

Rule engines were not conceptually designed to work in the heterogeneous environment. This is because rule-based systems are characterised by a uniform design space where a number of unordered rules are referencing a global working memory. When ported to heterogeneous environments such as in the multi-tenancy [1] context, these rule-based systems are revealed to be intrinsically non-reentrant where in this flat design space activations could be observed from all asserted facts without discriminating their sources. We exemplify the problem next using the university access control example.

3.1 Example: Non-reentrancy in Classical Rule-based Systems

Now the security team of the university, *University1*, has installed their rule in the Serena server running a heterogeneous RBS. The team from the second university, `University2`, designs several rules using protocols extracted from the same security regulations. We observe the situation when they proceed to naïvely upload their similar classtime access rule.

In Rete rules are technically shared in their entirety within the network. As mentioned, structural similarity promotes sharing of nodes performing the same test but corresponding to different rules. When a security person from *University2* adds their classtime access rule *classtime-access2*, this results in the Rete graph that is structurally the same as before the addition of the rule – however this time the terminal node is

[1] Reentrancy as used here relates to the notion of reentrant procedures in multiuser systems programming and excludes those related to concurrent access and recursive method calls.

tagged with activation of both rules. Because the terminal node was tagged with both rules, when a student from either university makes an access request in a classroom both rules will be activated on both clients, *University1* and *University2*. This can indeed be an undesirable result in heterogeneous setups, since now both companies can have notifications of granted accesses from unknown parties on their dashboards or in their system logs.

This simple example exposes the fact that in order to fully exploit capturing community knowledge in a rule-based system operating in a heterogeneous setup, it is vital that the system should avail mechanisms in which problems brought forth due to lack of reentrancy be suitably addressed.

Classic rule-based systems are thus said to be fundamentally non-reentrant. Given varied data sources, rules intended for one specific source or a number of sources can be activated by data from other sources. Therefore, multiple heterogeneous data sources can lead to unexpected behaviour during execution cycles of the rule engine. One undesirable consequence is that rule activations can be observed from all asserted facts without discriminating their specific sources. In effect, the difficulty in localising rule control makes it hard to orchestrate the behaviour of rules in these settings.

3.2 Common Workarounds in Classical Rule-Based Systems

To fully exploit community knowledge in rule-based systems within heterogeneous contexts, there is need to solve the lack of reentrancy brought about by sharing in the Rete network. In [1] we describe traditional techniques applied by developers of rule-based systems to enforce discrimination during execution in a Rete graph. The techniques included using relation facts to introduce situational state in the execution cycle and test expressions to perform beta tests on tokens using discriminatory test conditions in rules.

The two approaches have similar limitations. It is generally undesirable to have rule condition logic (or application logic) interspersed with event source identification as noted in [14] 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.

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, and quickly becomes complex and fallible 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 also cause unintended rule activations to leak in other clients.

In summary, these classical methods 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, and 4) in some cases impact the underlying Rete graph by creating additional nodes requiring more computations.

3.3 The Role of Client Relationships in Community Knowledge

Rule complexity quickly becomes more tedious in heterogeneous configurations. In this section we show the significance of relationships in exploiting community knowledge, especially in situations that contain more complex internal structures.

Usually, heterogeneous setups contain complex structures that are modelled according to physical or logical *relationships* among participating clients. These relationships can be based on aspects such as the principle of locality among interacting components in event-based systems [14] and encapsulation in object-oriented systems. In simple situations these relationships can based on the practical application-specific semantics tied to underlying structures, operations and processes of clients. For instance, research groups 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.

Consider the rule from protocol 3 that specified that students from a department in the university can have special access times to their (sub-)departmental labs. The rule developed for the protocol is shown in Listing 1.2. There are two access requests in lines 7 & 8 from the student and the senior academic staff, so we need to check if they come from the same department and if the department is `biology` or `bioinformatics` (line 9) as per the policy and the defined structure in Figure 2 (note that the rule is more complex if the student and academic come from different departments). The resulting Rete graph for policy 3 with test expressions is shown in Figure 4. Defining rules for such situations increases the problems brought about by a lack of reentrancy as seen in the expression in line 9, which tries to capture data from the same `biology` department or its sub-department `bioinfomatics`. In the graph, node 4 contains further tests that implement the above constraints.

Listing 1.2: Rule for biology dept. weekend lab access

```
1  {rulename: "biology_weekend_access",
2   conditions:[
3    {$stu: {type:"student", name: "?stuname"}},
4    {$stf: {type:"staff", name: "?stfname"}},
5    {$d: {type:"accessdevice", name: "?dev", location:"labs"}},
6    {type:"accessreq", person: "?stuname", device: "?dev"},
7    {type:"accessreq", person: "?stfname", device: "?dev"},
8    {type:"$test", expr:"( areInSameUni($stu.dept,$stf.dept,$d.dept) )"}
9    {type:"$test", expr:"( ($stu.dept == $stf.dept) && ($stf.dept == $d.dept) && ($d.dept ==
      ↪ 'biology' || $d.dept == 'bioinformatics') )"}
10   /* ... action ... */
11 }
```

4 Requirements for Heterogeneous Rule-based Systems

This section outlines the requirements of heterogeneous RBSes to suitably provide flexible mechanisms that can be used to exploit capturing community knowledge.

4.1 Metadata Model for Managing Client Data

Heterogeneous RBSes require an approach that imposes a uniform and consistent model supporting the identification of clients as event sources thus enabling the mapping of

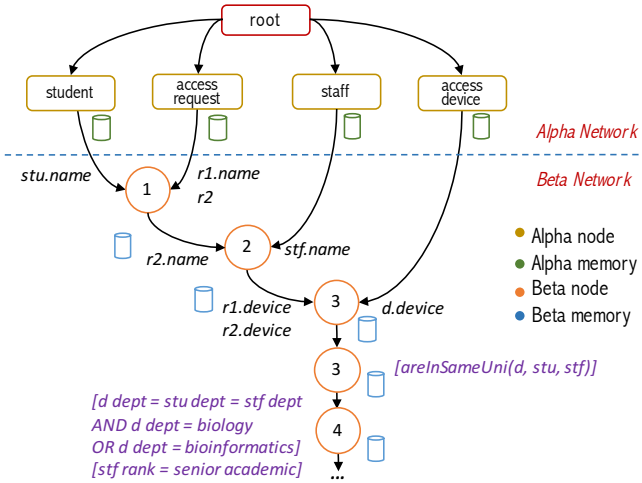


Fig. 4: Rete graph for policy 3 with additional tests for department-level checks [1]

data items (and rules) from different contexts to specific client(s). This in effect will form the basis for the underlying rule engine to be able to ascertain the sources of constraints and events; using this to determine their respective execution contexts. The proposed approach should use a metadata architecture that is application-agnostic in order to promote the normal semantics of a rule-based system thus absolving the end-user application from the nuances that would exist without the model. On the server side, the approach should also be able to reason about the data model with the least effect on the underlying processing cycle of the rule engine.

4.2 Formalised Model for Grouping Clients

One of the challenges that heterogeneous RBSes face is ways in which to partition execution contexts, brought about as a direct result of how client rules are shared within the Rete graph generated by the inference engine for efficiency purposes. Many heterogeneous environments with relationships form communities through some form of *grouping* [15]. A heterogeneous RBS can use this aspect to provide an extensible model that captures the structures of clients and describes any possible compositions dynamically. In addition, this approach can be effectively captured in a more flexible way through a formalism that is based on the aforementioned metadata definitions. The formal model should be designed around a hierarchical structure that lends itself well in progressively describing the relationships between identified application-dependent classifications, useful in different real-world scenarios.

4.3 Execution Model for Selective Computations

During execution, rule engines do not provide selective matching by imposing any discrimination in rule execution: all rules are under consideration in a match cycle. Indeed, the diversity of data sources in a heterogeneous setup requires further precision in the

execution context. Even the selection strategy of rule activation never actually depends on selective rule matching, but in reality is based on the recency of an instantiation and requires special ordering of rules. An extensible formalism can be exploited to effectively perform selective execution of the rule engine by discriminating or consolidating the data residing in the heterogeneous system. These computations are those that try to perform the "*are the tokens that we want to match originating from the same source?*"-check that is needed to find consistent bindings for the different data sources, to avoid unintended activations with data from unwanted sources. The implication therefore 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 each client's configuration or constraints. 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.

4.4 Flexible Model for Notification Semantics

In such heterogeneous contexts, a notification is a message sent to a specific client(s) that reifies an event resulting from a rule activation. The notification carries the data that accompanies the activation, but may also contain additional metadata such as the time of activation and the owner of the rule. One issue that arises that is exclusive to heterogenous configurations is *who to notify*, or precisely, which client(s) that should receive the notification of the rule activation. In the default scenario, the user that added the rule should receive the notification. However, the issue of composition can be fully embraced to group sets of clients that share some commonality or goal, similar to engineering notifications in event-based systems [14]. The aim of such composition is to specify boundaries for notification delivery: it semantically restricts the distribution of notifications of a rule activation from the RBS. The boundaries should be able to be specified using a clearly-defined notification semantics.

5 Scoping in Heterogeneous RBS

The solution presented in the Serena framework [1] introduces scoping in heterogeneous RBSes by embracing the concepts of physical or logical groups of clients and their relationships.

Serena models groups internally with the aim of using these representations to enforce data discrimination in the rule engine. It describes a structural representation that uses the notion of a group as a primitive. Serena represents the *group hierarchy* as a directed acyclic graph with the groups as the nodes with the clients connected to different groups at different levels in the graph. It also uses *scopes* to represent the common relationships between groups as a *scope hierarchy*.

5.1 Supported Scopes in Serena

Serena supports the following scope operations, depicted in Figure 5 with reference to the university security access control example. Constraints imposed by protocols in

heterogeneous setups (such as the *classtime access* rule) can be defined using these scopes.

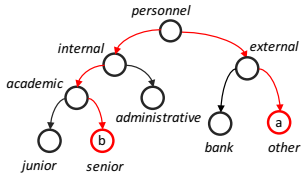
- ***subgroupof*** (Figure 5c): Only data added by the specified group or any of its subgroups are included in this scope. This scope is suitable for a departmental rule for *computer science* that will only apply to members of that department or subdepartments (*web info systems*, *software engineering*, *bioinformatics*). Its dual is ***supergroupof***.
- ***visibleto*** (Figure 5a): This scope captures data from clients in groups that have the same ancestor in the hierarchy, e.g., capturing the data that pertains to *senior* academic researchers in one project collaborating with *other* personnel within the same university.
- ***peerof*** (Figure 5b): Data items that originate from *peers*, or groups at the same level in the hierarchy, will be considered in this scope. A researcher would for example create a rule with this scope that applies to members in *computer science* and *biology* departments.
- ***private*** (Figure 5d): The private scope will exclusively source data from the specified group and none else – not even its subgroups or parent group. This scope is suitable for data that applies to a specific group, e.g., when targeting devices at the campus entrance gates and not those in its related subgroups elsewhere on campus premises.
- ***public*** (Figure 5e): The public scope captures all data from all defined groups in the hierarchy. This could be useful for collaboration in the universities by sharing security information between them for data from the devices/student/staff in all the groups.

5.2 Defining Scoped Rules in Serena

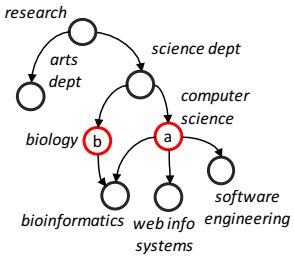
Instead of embedding logic for distinguishing clients in the main logic of the rule, Serena exposes scoped rule definitions by extending normal rule syntax with scope-based definitions. The scope definitions specify scope-based constraints on client groups and the relationships between them. A similar approach is observed when enforcing temporal logic in rules: Allen in [16] proposed rule extensions for temporal constraints for point-based or interval semantics, which have been implemented in various systems today, e.g., [17]. The example of protocol 3's *biologyweekendaccess* rule using scope constraints is shown in Listing 1.3.

Listing 1.3: Scoped rule for biology dept. weekend lab access

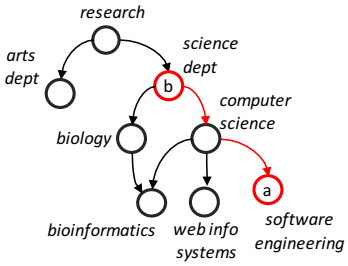
```
{rulename: "biology_weekend_access",
 conditions:[
  {$stu: {type:"student", name: "?stuname"}},
  {$stf: {type:"staff", name: "?stfname"}},
  {$d: {type:"accessdevice", name: "?dev", location:"labs"}},
  {type:"accessreq", id: "?reqid1", person: "?stuname", time: "?t1", device: "?dev"},
  {type:"accessreq", id: "?reqid2", person: "?stfname", time: "?t2", device: "?dev"},
  {type:"$test", expr:"(hourBetween(?t, 8, 20) && (isWeekend(?t1, ?t2) == true) && isNear(?t1,
   ↪ ?t2) )"}
 ],
 scopes:[ "biology supergroupof ($stu & $stf & $d)", "$stf private senior"],
 actions:[
```



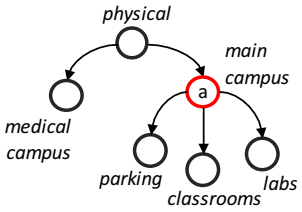
(a) *a* visibleto *b*



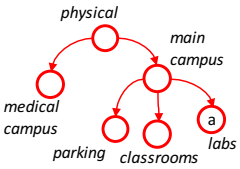
(b) *a* peerof *b*



(c) *a* subgroupof *b*



(d) private *a*



(e) public *a*

Fig. 5: Scopes supported in Serena [1]

```
12    {assert: {type: "accessrep", reqid:"?reqid1", allowed: true}}
13   ],
14   notify:[ "subgroupof administrative"]
15  }
```

The rule is similar to Listing 1.2. It however has an additional *scopes* section (line 10) where the bound condition variables in line 3, 4, and 5 in the scope constraint are referenced to check whether the student, staff and device facts are all tagged to belong to the general `biology` department using `supergroupof`. The additional scope check in line 10 enforces the constraint that the staff member be from the `senior` academic group. The rule therefore fulfils constraints of Protocol 3 that specified that lab accesses made in the weekends by a student are allowed only if they are accompanied by a senior academic staff member in the biology department, or any of its subdepartments.

5.3 Encoding the Group Hierarchy

Community knowledge requires means in which to determine the compatibility of different sources of heterogeneous data. Rather than performing computationally expen-

sive scope checks (such as path traversals in a hierarchical setting) Serena builds an encoding that aims to perform near constant-time operations to entirely determine compatibility of data in a heterogeneous setup. This is vital because during the match-execute cycle, Rete can perform combinatorial processing in its computations in the beta network as the dataset increases: therefore client group path traversals will dramatically affect the performance per cycle. The basic idea is that we precompute the scope check, store and maintain them efficiently as an encoding that will be used to expeditiously process scope constraints.

The encoding is based on the *transitive closure*, a significant component modelling most relationships in knowledge and representation systems as identified in [18] that makes our encoding suitable for querying binary relationships – precisely the kinds of operations that the inference engine performs when performing a scope check between left and right inputs. We next outline the encoding process.

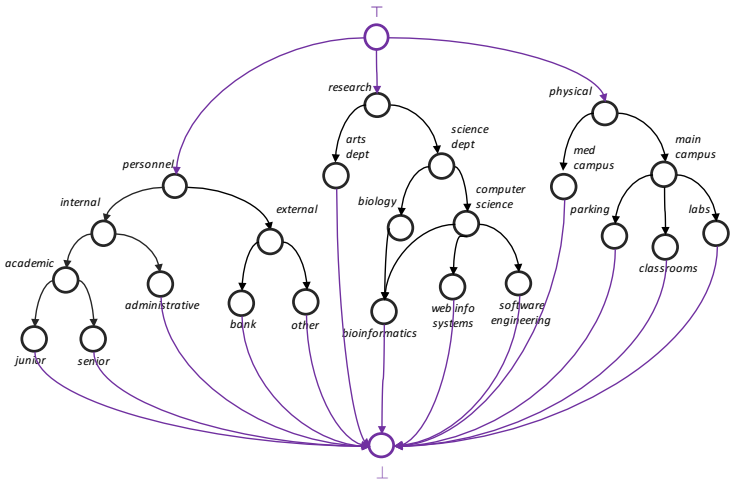


Fig. 6: The lattice Hasse diagram – Serena uses the lattice to encode the hierarchy (Image sourced from [1]).

The Groups Hierarchy as a Poset. Initially, Serena captures the group hierarchy from an administrator as a partially-ordered set (*poset*). The example hierarchy in can be represented as a poset $(P, \leqslant)$ with the binary relation $\leqslant$ defined as '*is subgroup of*', that suffices for most cases.

The poset P has an element (a, b) iff a is part of b. With P we can perform well-defined operations such as calculating the bounds (LUB, GLB) and extrema (maximals, minimals). For instance the maximal in the group hierarchy of Figure 2 can be represented as a poset $(P, \leqslant)$ with the binary relation $\leqslant$ defined as '*is a part of*' (the general $\leqslant$ relation '*is subgroup of*' is enough for most cases). The poset P has an element (a, b) iff a is part of b, so elements include *(internal, personnel)* and *(computer science, sci-*

ence dept). The poset however has a limitation of manually searching and traversing the pairs when processing scope operations.

The Groups as a Lattice. A lattice offers improvements over the poset by representing the group hierarchy in a form that is more efficient to encode and compute than the earlier poset representation. The Serena^s framework therefore converts the groups poset to a lattice L, see Appendix A.2. This leads to the hierarchy depicted as the hasse diagram in Figure 6. Other distinct hierarchies can have their own top-level element same as $\top$.

Encoding the Lattice. With the lattice L, Serena performs a customised *bit-vector encoding* process that lays its basis on the method by Aït-Kaci [19]. The description is outlined in detail in Appendix C. The process performs calculations for all groups G in the group hierarchy. The result is a *binary matrix encoding* M_ϑ of the group hierarchy shown in Figure 7, with the following properties:

	⊤	per	res	phy	int	sci	mai	aca	com	biol	adm	lab	clas	sen	soft	bioi	⊥
⊤	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
per	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
res	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
phy	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
int	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
sci	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
mai	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
aca	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
com	1	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
biol	1	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0
adm	1	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
lab	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0
clas	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
sen	1	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0
soft	1	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0
bioi	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	1	0
⊥	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Fig. 7: The hierarchy as a matrix encoding M_ϑ – The rows and columns represent groups in the hierarchy [1].

i) The labels on the rows of M_ϑ represent the groups in L; and similarly for the columns. The first row represents $\top$ and the last row represents $\bot$.
ii) An entry $M_{\vartheta(a,b)}$ has a `1` if `group` a = `group` b or if `group` b is an ancestor of `group` a in L, and `0` otherwise.
iii) An entry $M_{\vartheta(b,a)}$ has a `1` if `group` a = `group` b or if `group` b is an descendant of `group` a, and `0` otherwise.
iv) An element a is a *maximal* iff the row $M_{\vartheta(a,*)}$ has a `1` only at $M_{\vartheta(a,a)}$ and at $M_{\vartheta(a,\top)}$.
v) An element a is a *minimal* iff the column $M_{\vartheta(*,a)}$ has a `1` only at $M_{\vartheta(a,a)}$ and at $M_{\vartheta(\bot,a)}$.

The process further generates and stores the *level* or depth of each group in the hierarchy. The indexes of all the maximals can also be stored for faster reference.

Scoping with $\mathbf{M}_\vartheta$ – The M_ϑ is used as the basis of performing scope operations in the rule engine. To facilitate this Serena adds scope tests or guards at appropriate nodes when building the Rete network. The guards are used to perform scoping operations in the beta nodes during the matching process.

- ***visibleto:*** A scope check of a `visibleto` b the involves checking if the result of $M_{\vartheta(a,*)} \wedge M_{\vartheta(b,*)}$ is a maximal in M_ϑ as per property (iv).
- ***peerof:*** A scope check of a `peerof` b includes calculating if $Level(a) = Level(b)$ from the encoding process of M_ϑ.
- ***subgroupof:*** A scope check of a `subgroupof` b is true if the result of $M_{\vartheta(a,*)} \wedge M_{\vartheta(b,*)} = M_{\vartheta(b,*)}$ as per property (ii). Conversely, b is a `supergroupof` a.
- ***private:*** To find out a `private` b it can check if $M_{\vartheta(a,*)} \wedge M_{\vartheta(b,*)} = M_{\vartheta(a,*)}$ as per property (ii) and (iii).
- ***public:*** A scope check of a `public` b includes checking if we $M_{\vartheta(a,*)} \wedge M_{\vartheta(\top,*)} = M_{\vartheta(\top,*)}$ from properties (ii) and (i).

During execution, Serena performs scope operations efficiently using these operations. It retrieves the values in the matrix and performs binary operations from the encoding in near-constant time.

5.4 Scoped Execution and Notifications

The matching process stage is where any rule engine performs most of its computation. As mentioned in the previous section, matching in scoped rule engines involves updating the beta network with scope guards that check compatibility of left and right inputs. One way of building the rule in Listing 1.3 is shown in the Rete graph of Figure 8. The main difference is in the beta node 3 where we now have in place a scope check, a more compact and efficient way to discriminate the tokens for the node to process. The scoping module will use M_ϑ to perform the binary operations from the scope guards in the figure denoted with angle brackets.

On a left or right activation, Serena *first* performs the encoded scope check on the fact from the alpha memory or the token's fact respectively. If the check passes the join computation proceeds as normal. For instance, when a token reaches beta node 3, it triggers a left activation to find a compatible `accessdevice`. Serena will first perform the `supergroupof` scope check on the devices as defined in Section 5.3. For example, if the access request is made from a device `dev` in the `bioinformatics` subgroup, the engine performs the supergroup check on the alpha memory's `device` fact, which in this case succeeds:

$$M_{\vartheta(biology,*)} \wedge M_{\vartheta(bioinformatics,*)} = M_{\vartheta(biology,*)}$$

```
 1010010001000000
&1010010011000010
 1010010001000000(biology)
```

Similar operations are performed for the `student` *supergroup* check and the *private* scope check for the `senior` staff member from the academic personnel group using M_{ϑ}. If successful, we have established the facts are compatible and proceed to the join operation for node 3.

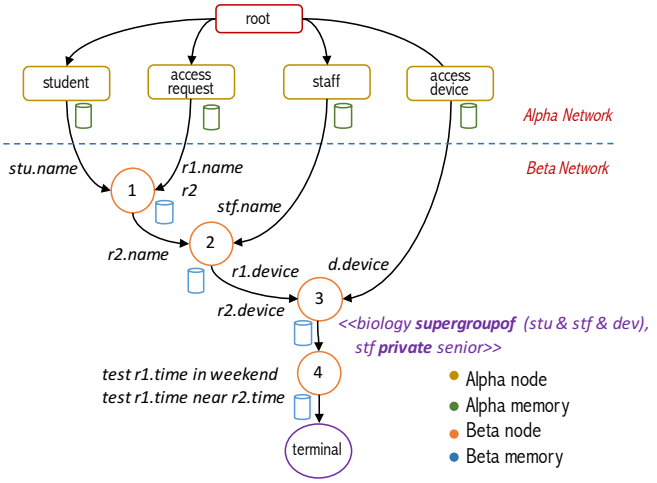


Fig. 8: The Rete graph for *biology weekend access* with scopes – Scopes act as guards that ensure compatibility of data. [1]

The final aspect of rule activation is determining *who to notify*, and specifically, which group of clients should receive a notification. Serena rules expose a `notify` construct that specifies notifications once the rule is fired. In line 14 of Listing 1.3 specifies the groups to notify once the rule is fired. The *notification scopes* are similar to the matching scopes but in this case they can enforce notification constraints to a group, subgroup, or direct clients. Serena invokes similar binary operations as in Section 5.3 to determine the groups to notify as when performing a scope check during matching.

6 Experimental Evaluation

The evaluation in [1] used the University Services Access Control scenario detailed in Section 1.2 to investigate whether the scoping metadata architecture has significant computational benefits over traditional techniques in current rule engines. The aggregated results in that work showed evidence of a better overall performance of the scoped engine compared to traditional approaches using expression test and relation facts.

In this paper we compare the scoped approach to the rulebook or *module-based* approach (see Section 7) for implementing heterogeneous applications using rule-based systems. We focus on investigating whether scoping has significant computational benefits over the module-based approach in such situations.

6.1 Setup

For the setup, we used the 3 sample universities from the security access control example in Section 1.2. We also incorporated similar protocols as prescribed for the scenario. However, as modules represent logically distinct rulebooks, there was need to split the policies according to the relevant university. As expected, in this case some policies were duplicated: for instance, the *classroom access* protocol that applies collectively to all universities needs to be replicated over the modules.

The evaluation was implemented using a simulation running on an event-driven web server. The final application had a total of 61 groups in hierarchies, 40 access rules, and 73 concurrent clients across 3 sample universities. All clients were connected to the server concurrently through websocket connections, with the Node.js server running with an AMD Opteron Processor 6272 at 2.1Ghz and 20GB RAM.

6.2 Method

In the simulation, universities receive the same intermittent access requests from various clients with the aim of computing whether the requests deviate from their own protocols as rules. Each simulation was modelled with intermittent requests in ranges of between 1-5 seconds, limited to 12-hours similar to the scope-based simulation in [1]. The requests model students and staff from different departments or personnel levels randomly accessing various university locations. We recorded the resource usage and computations performed during the simulation, and compared the results.

6.3 Results & Discussion

The results of the runs were aggregated and depicted in the graphs shown in Figure 9. The figure shows the total number of join computations recorded, the number of activations observed and the RSS (Resident Set Size) memory consumed.

The computations in the module instances were observed to be less than those of the unscoped approach – this is because of the additional checks in the unscoped rules that need to enforce data discrimination, and thus lead to more computations that need to be performed. The scoped approach certainly has fewer computations because it instead uses the matrix encoding to perform scope checks. Remember that join computations have been observed as the most expensive computations in Rete inference engines, Section 2.2.

The instances of the module based approach process a higher number of activations our scoped approach within the same time interval of 12hrs. Indeed, having separate module instances would result in fast computations than the traditional unscoped approach. The limitation of the module-based approach is unearthed by viewing its memory consumption: the memory used by the module instances is significantly higher compared to all other approaches. This is because the instances have increased redundancy, and this leads to the duplication of the working memory, graph nodes and intermediate memories utilised by the rule engine.

The results showed that the module-based approach exhibited similar performance as our scoped approach in terms of the rule activations processed: but both outperformed

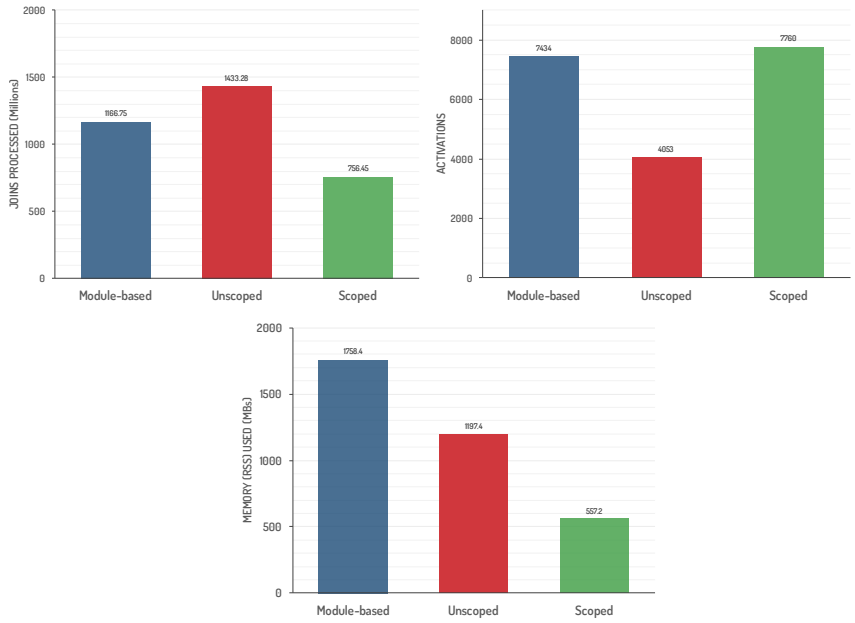


Fig. 9: Results for the experimental evaluation – The scoped rule engine approach performs comparatively similar to the module-based approach and outperforms the unscoped approach. The scoped engine also uses less resources than both approaches.

the traditional unscoped approach. The main cost of the module-based approach is however the high amounts of memory utilisation due to duplication of resources. This effectively means that the module-based approach suffers from scalability problems, since more instances would be needed as clients are added to the system, resulting in significantly higher utilisation of resources. In this sense, our scoped approach enjoys both the benefits of a relatively good performance that utilises less resources during execution.

7 Related Work

We delve into research that provide mechanisms that can be used to control rule engine execution, useful when implementing rule-based solutions in heterogeneous settings. Following the discussions, Table 1 presents a summarised result of the related work in a 5-star ranking system.

7.1 Rule-based Systems

EntryPoints in JBoss Drools. Drools [2] is a Java-based rule engine based on the Rete algorithm. The engine can deal with event streams that are of high volume and require correlation by receiving and processing inputs from multiple *entry points*. Entry points can be thought of a particular pipe where events from a source flood into the system. New entry points are declared by implicitly referencing them in rule conditions.

When inserting events data into the engine, however, Drools requires entry points to be directly referenced and it therefore lacks a proper meta architecture. Furthermore, entry points do not offer any formalised model for more advanced relationships that can be present in heterogeneous contexts as well as addressing multiple entry-points as a single composable unit. The engine also does not offer any notification model as a feedback mechanism for rule activations.

Peers in Jess. Jess [20] is a Java-based rule engine that can provide reasoning to Web servers as a backend tool. Jess exposes *peering* of its engine, designed to be used in the scenario of pools of Web applications. Conceptually, peers are an evolution of multiple 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 the same rule set: changes to the rule set by any of the instances will thus be reflected on the other peers.

Jess' peering does not provide a metadata model that will effectively manage the different data and sources. The implementation of the separation of instances can be therefore intertwined with application logic, an issue that may complicate the development of heterogeneous applications. 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. Additionally, when using the peer system in Jess, the programmer needs to ascertain which data will go to which peer. In cases that aim to take advantage of collective intelligence in heterogeneous environments, it is usually unclear to ascertain which peer(s) should receive data from a particular source. Jess also does not expose functionality for managing notifications for responding to clients in the event of a rule activation.

Rule Modules in CLIPS. CLIPS [21] is a Rete-based rule engine that was primarily written in C. It provides modular management for larger rule bases through the use of *rule modules*. A rule module can be though of as a rulebook, having 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.

CLIPS rule modules provide modular abstractions for grouping related rules. However, CLIPS exposes a generic model where facts need to be manually be tagged to a particular rule module during fact data definitions. The CLIPS engine also does not provide any abstractions for managing rule modules that could be used as model for representing the various client structures. Furthermore, constructs are available in rule modules that allow programmers to 'steal' the rule engine's focus to execute a named module. This unnecessarily complicates the programming of heterogeneous rule-based applications, because rule designers are required to orchestrate rule interactions. This

unnecessarily detracts the inference engine from its own control of execution. Finally, in CLIPS notifications to individual clients need to be programmed manually.

7.2 Visibility in Event-based Systems

Event-based systems are increasingly applied in heterogeneous context. Perhaps the closest approach is the work about event notifications in the system REBECA [22]. The designers aim 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 using broker overlays. Only the components that are intended to receive notifications (the intended consumers) are able to 'see' notifications filtered by their local event broker.

The work however, only focuses on notification semantics and does not suitably address the actual matching or processing of events. Furthermore, the broker architecture presupposes the existence of some form of an overlay network which requires a more complex management scheme and is not the primary focus of the platform that this work aims to support.

7.3 Schema Sharing in Multi-tenant Databases

Multi-tenant database systems can support heterogeneity by mapping the context of clients into the existing patterns of conventional databases. The closest work in such heterogeneous databases is the shared schema/tables approach. Using this approach the schema is create 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 much higher complexity to implement.

Table 1: Comparison of RBSes and other systems for supporting heterogeneity

	Metadata Model	Grouping Model	Execution Model	Notification Model
Multitenant Databases				
Shared schema/tables	★★★★☆	☆☆☆☆☆	★★★½☆	☆☆☆☆☆
Event-based Systems				
Visibility	☆☆☆☆☆	★★½☆☆	☆☆☆☆☆	★★★★☆
Rule-based Systems				
Entry Points	½☆☆☆☆	½☆☆☆☆	★★★☆☆	★½☆☆☆
Peers	½☆☆☆☆	★☆☆☆☆	★★☆☆☆	★☆☆☆☆
Rule Modules	★★☆☆☆	★★★½☆	★★☆☆☆	★☆☆☆☆
Scopes	★★★★½	★★★★★	★★★★★	★★★★½

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 re-

quest to a tenant and forwards it to the appropriate storage site. These database schemes are fundamentally designed to process static data and they have static configurations that can degrade in performance when ported to reactive systems with eager incremental processing as with forward-chaining RBSes.

8 Conclusions and Future Work

Community knowledge is significant in today's heterogeneous systems because of the vastness and diversity of real-time data produced by different clients. This work described a framework that supports scope-based reasoning in heterogeneous systems with the aim of supporting reasoning using community knowledge by mitigating the consistency problems that such systems exhibit.

Scoped rules contain an extended rule-based syntax that allows rule designers to define scope constraints. The work presented uses groups and common relationships between them to build an internal representation that captures the scopes present in many domains. Scopes are a control structure for heterogeneous rule-based languages because they specify a selection of which rules a scope-aware inferencer will consider at a particular time during execution. The scoped model is therefore useful to capture the inherent organisation of client representations and to define constraints in rules for the reasoner, thereby harnessing community knowledge in a shared instance. The evaluation showed that the proposed scoped approach enjoys slightly better performance that utilises less resources during execution as compared to similar module-based approaches in forward-chaining rule-based systems.

As future work we would like to investigate support for client-defined dynamic scopes, which will have an impact on the encoding method and the intermediate memories in the Rete graph at runtime.

Acknowledgements Thierry Renaux is supported by a doctoral scholarship from the Agency for Innovation by Science and Technology in Flanders (IWT), Belgium.

References

1. Kambona, K., Renaux, T., De Meuter, W.: 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
2. Browne, P.: JBoss Drools business rules. Packt Publishing Ltd (2009)
3. Lenat, D.: Feigenbaum. ea (1987) on the thresholds of knowledge. In: Proceedings of. 1173–1182
4. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: A view of cloud computing. Communications of the ACM **53**(4) (2010) 50–58
5. Jamison, S.W., Brereton, J.P.: The Rigveda: the earliest religious poetry of India. Volume 1. (2014)
6. The Baseline Company: The blind men and the elephant. `http://www.theblindelephant.com/the_blind_elephant_fable.html` (Accessed on 04/11/2016).

7. Allee, V.: The knowledge evolution: Expanding organizational intelligence. Routledge (1997)
8. Fernandez, E., Yoshioka, N., Washizaki, H.: 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)
9. Newell, A.: Production systems: Models of control structures. Technical report, DTIC Document (1973)
10. Giurca, A., Pascalau, E.: JSON rules. Proceedings of the of 4th Workshop on Knowledge Engineering and Software Engineering, KESE **425** (2008) 7–18
11. Forgy, C.L.: On the efficient implementation of production systems. PhD thesis, Carnegie-Mellon University (1979)
12. Miranker, D.P.: TREAT: A New and Efficient Match Algorithm for AI Production System. Morgan Kaufmann (2014)
13. Wloka, J., Sridharan, M., Tip, F.: 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) 173–182
14. Mühl, G., Fiege, L., Pietzuch, P.: Distributed event-based systems. Springer Science & Business Media (2006)
15. Grund, M., Schapranow, M., Krueger, J., Schaffner, J., Bog, A.: Shared table access pattern analysis for multi-tenant applications. In: Advanced Management of Information for Globalized Enterprises, 2008. AMIGE 2008. IEEE Symposium on. (Sept 2008) 1–5
16. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26**(11) (November 1983) 832–843
17. Wang, F., Liu, P.: Temporal management of rfid data. In: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment (2005) 1128–1139
18. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. SIGMOD Rec. **18**(2) (June 1989) 253–262
19. Aït-Kaci, H., Boyer, R., Lincoln, P., Nasr, R.: Efficient implementation of lattice operations. ACM Trans. Program. Lang. Syst. **11**(1) (January 1989) 115–146
20. Friedman-Hill, E.: JESS in Action. Volume 46. Manning Greenwich, CT (2003)
21. Riley, G.: Clips: An expert system building tool. (1991)
22. Fiege, L., Mezini, M., Mühl, G., Buchmann, A.P.: Engineering event-based systems with scopes. In: European Conference on Object-Oriented Programming, Springer (2002) 309–333

Appendices

A Definitions

A.1 Posets

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)

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[2].

Extrema: The *maximal* of a poset P, abbreviated $\lceil P \rceil$, is an element $m \in P$ that is not greater than any other element in P according to $\leqslant$. More formally,

$$\lceil P \rceil = \forall b \in P, b \leqslant m \tag{1}$$

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.2 Lattices

If in a 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*. One way to transform the poset P in into a lattice 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). More formally,

$$a \prec b \text{ iff } a \leqslant b \text{ and } \nexists c \text{ s.t. } a \leqslant c \leqslant b, c \neq a, c \neq b \tag{2}$$

This enables us to depict a lattice in a *hasse diagram*, where a curve goes from b to a iff $a \prec b$.

Lattice levels In this paper 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.,

$$Lvl(a) = \begin{cases} 0 & \text{when } a \text{ has no predecessors in } P \text{ and,} \\ max(\{Lvl(b)) \mid b \succ a\}) + 1 & \text{otherwise.} \end{cases} \tag{3}$$

where $\succ$ is the dual of $\prec$.

B Operations with ϑ

Having L we can define a mapping ϑ from L to another lattice $(S \subseteq, \cap, \cup)$ such that for every $a, b \in L$,

$$\vartheta(a \wedge b) = \vartheta(a) \cap \vartheta(b), \tag{4}$$

$$\vartheta(a \vee b) = \vartheta(a) \cup \vartheta(b). \tag{5}$$

[2] 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.

If ϑ is invertible, then this makes it easy to calculate $\vee$ and $\wedge$. $\forall\, a,b \in L$,

$$a \wedge b = \vartheta^{-1}(\vartheta(a) \cap \vartheta(b)), \tag{6}$$

$$a \vee b = \vartheta^{-1}(\vartheta(a) \cup \vartheta(b)). \tag{7}$$

C Matrix Encoding

We use the encoding method mentioned in [19] taking ϑ as the transitive closure, with a modification that will enable us to map a lattice L to an encoded matrix M_ϑ.

- Instead of starting with $\bot$, start with $\top$ as the first element. Assign $\vartheta(\top) = 0$.
- Move to the next elements level by level downwards in L and calculate the bitcode of each element as a vector.
- The bitcode of an element $a \in L$ is obtained by

$$\vartheta(a) = 2^{i-1} \vee \bigvee_{a \prec x} \vartheta(x) \tag{8}$$

  where i is the number of elements visited since $\top$, and $a \prec x$ represents predecessors of a; therefore $\vartheta(x)$ is the code of each predecesor of a.
- An entry in the new matrix M_ϑ for a is the reverse of the bitcode obtained by (8), without the most significant bit.

With this encoding, we can perform operations in Eq (6) and (7) having $\cap$ as the bitwise `AND` and $\cup$ as bitwise `OR` in M_ϑ.