Reentrancy and Scoping for Multitenant Rule Engines

Kennedy Kambona, Thierry Renaux* and Wolfgang De Meuter
Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium

Keywords: Rule-based Systems, Multitenancy, Rule Engines, Rete Algorithm, Reentrancy, Scoping, Business Rules.

Abstract: Multitenant web systems can share one application instance across many clients distributed over multiple devices. These systems need to manage the shared knowledge base reused by the various users and applications they support. Rather than hard-coding all the shared knowledge and ontologies, developers often encode this knowledge in the form of rules to program server-side business logic. In such situations, a modern rule engine can be used to accommodate the knowledge for tenants of a multitenant system. Existing rule engines, however, were not conceptually designed to support or cope with the knowledge of the rules of multiple applications and clients at the same time. They are not fit for multitenant setups since one has to manually hard-code the modularity of the knowledge for the various applications and clients, which quickly becomes complex and fallible. We present Serena, a rule-based framework for supporting multitenant reactive web applications. The distinctive feature of Serena is the notion of reentrancy and scoping in its Rete-based rule engine, which is the key solution in making it multitenant. We validate our work through a simulated case study and a comparison with a similar common-place approach, showing that our flexible approach improves computational efficiency in the engine.

1 INTRODUCTION

Traditionally, software systems were conceptually designed to run in isolation. With cheaper networking hardware and the subsequent rise of the Internet, various web technologies have evolved to support dynamic, data-driven and *reactive* applications that handle a massive number of users and client devices. Consequently, modern software systems are increasingly being deployed in the Cloud.

A distinguishing characteristic of Cloud-based platforms is Utility Computing (Armbrust et al., 2010) with the *pay-as-you-go* model that improves cost reduction through resource sharing. Utility Computing provides a way in which modern software systems can simultaneously support multiple clients and at the same time share resources through *multitenancy*. A multitenant application is installed on a single instance (rather than separate instances) and serves all clients, or 'tenants' from that instance (Pathirage et al., 2011). They exhibit some advantages, including reduced maintenance and increased scalability as they pertain to economies of scale. However, a large number of providers have limited support for multitenancy at the application level (i.e. native multitenancy) – only focusing on process isolation. Partitioning and securing multitenant application behaviour at this level is complex and requires a huge development effort (Guo et al., 2007).

In this paper, we focus on knowledge-intensive multitenant applications running over the web, connected to different clients sending massive amounts of events and data. These systems are required to manage the shared knowledge base reused by the various tenant applications they support. In order to reason about the data and extract higher-level knowledge it is vital that the value of the sent data be extracted efficiently, its massive and intermittent nature notwithstanding. Rather than hard-coding all the shared knowledge and ontologies, developers often encode this knowledge in the form of rules to program server-side logic e.g. as business rules (Hay et al., 2000).

In such situations, a modern rule engine can be used to accommodate the knowledge for tenants of a multitenant web system. To this end, we have augmented an event-driven web server with a forward-chaining rule engine constituting Serena, a rule-based multitenant framework that receives and reactively processes data in order to detect complex events to-

*Supported by a doctoral scholarship of the Agency for Innovation by Science and Technology in Flanders (IWT), Belgium

Kambona, K., Renaux, T. and Meuter, W.
Reentrancy and Scoping for Multitenant Rule Engines.
DOI: 10.5220/0006283400590070
In *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST 2017)*, pages 59-70
ISBN: 978-989-758-246-2

gether with accompanying data relevant to notify clients. In Serena clients can install logic reactive rules that define the complex events they are interested in and dynamically upload data. The rules specify which data to match, who to notify and what information is sent with the notification.

Conventionally, rule engines were not conceptually designed to work in the multitenant environment. These rule-based systems (such as production systems (Newell, 1973)) are intrinsically non-reentrant: they are characterised by a *flat design space* where activations could be observed from all asserted facts without discriminating their sources. Further distinctions between clients and their data sources need to be hard-coded within the rules, which quickly become complex and fallible as the number of clients and the relationships between them increase, or when the relationships become complex to enforce using rule semantics. In a multitenant application, 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 instances of different entities. Serena provides techniques for users and developers to specify *scoped rules* that detect patterns in real-time data and to realise grouping structures in knowledge-intensive multitenant applications.

Scoped rules are a custom rule representation based on a formalised description. They allow definition of scoped constraints that enable rule creators to distinguish between events pertaining to different clients, 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. This leaves the logical intent of a rule easy to understand for a rule creator. At the same time, scoping enables us to exploit a number of performance optimizations in the server's rule engine during the matching process. Our approach of encoding the physical, structural or other logical organizations of multitenant applications eases the computational workload of the inference algorithm, thereby decreasing the engine's overall response time.

In brief, the main contributions in this work are:

- A reactive, rule-based framework for multitenant architectures supporting knowledge-based applications (Section 3)
- A meta-extension to the Rete algorithm for inference engines to improve reentrancy by incorporating techniques from bit-vector encoding that discriminate data matches as defined in the rules (Section 4.2)
- An extension to the rule-based syntax in the framework to support a formalised scope-based reasoning in multitenant systems (Section 4.3)

We begin by introducing the motivation and proceed to enumerate some requirements in Section 2. We then present the Serena framework's architecture and scoping mechanism in Sections 3 and 4. We penultimately evaluate our approach in Section 5 and finally discuss the related work and conclusions in Sections 6 and 7.

2 DATA-DRIVEN MULTITENANCY

In this section we motivate the need for a data-driven solution in a multitenant rule engine. To highlight the requirements that such a system should meet, we present a scenario of a service provider in the Cloud for monitoring security systems. The service monitors and logs requests in institution-wide security access systems, e.g., in universities.

2.1 Motivating Example: University Services Access Control

Universities in Brussels have passed a resolution that requires monitoring accesses of students and staff all over their campuses and report access requests that deviate from policies in place. The universities have installed proximity ID-card scanners at most major access points, and students/staff scan their issued ID cards to gain access to various locations in the campuses. Some of the security monitoring policies that the security team design are illustrated below:

1. All students at all levels have access to classrooms during class times on weekdays
2. Only registered student and staff cars are allowed entry to underground parking on their campuses

A common university structure consists of different students and staff: research, administrative or external/outsourced; physical structures' hierarchy; and research department hierarchies. A simplified structure for a university is shown in Figure 1. As a result, specific departments and units are allowed to 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
4. Only campus bank employees and consultants have access to the bank back office during working hours

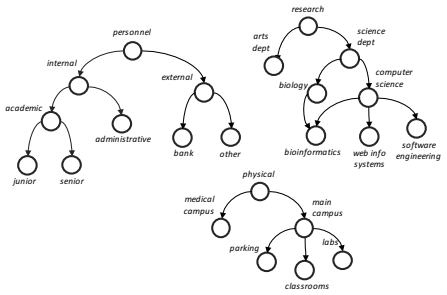


Figure 1: Example structures in a university – Client groups are based on three hierarchical structures: physical location, department, type of personnel. The hierarchies can be arbitrary DAGs and groups can have multiple parents.

For this scenario, we have enumerated around 40 security access policies. The final model contains 3 universities and 61 faculty, administrative and physical groupings – with students, staff and devices belonging to one or multiple groups. Whenever an access request is made by a student or staff the security system of the university sends the data to the monitoring service. According to the policies defined, the service logs the request, computes whether the access is within the defined security policies and displays the results on a dashboard. 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.

2.2 Requirements

The security monitoring service is a representative example of a reactive multitenant application. We particularly target the dynamic design of knowledge-intensive, data-driven applications that continuously stream data back and forth between clients and the server. The scenario illustrates some of the requirements that such multitenant frameworks should satisfy:

- *Data-driven framework for instantaneous processing of intermittent data streams* – The framework should be responsive to new inputs sent by tenants by processing them in real-time or near real-time fashion allowing the end-user application to react to the data. For instance, in the motivating example, the monitoring service provider should be able to process access requests from a large number of clients and devices promptly according to custom policies to provide immediate feedback. To send such feedback it also needs to handle persistent push-based client connections.
- *Runtime support for the definition and real-time detection of customisable constraints* – The framework should reduce the complexity of writing code that can efficiently detect real-time events from a continuous stream given a large number of criteria or constraints. This is a challenge to system developers because the intent of the developer is transcended by the accidental complexity (Brooks, 1987) of the implementation. In the example, the university security should be able to easily express and upload their own current and future policy constraints for detection of access violations using an expressive syntax.
- *Metadata architecture for multitenant partitioning* – The framework should be able to model the structures of tenants and possible compositions or relationships between them dynamically through metadata definitions that will discriminate or partition the data residing in the multitenant system. This implies that the internal structures of tenants should be reflected in the runtime in order for it to process the requests within the confines of each client's configuration: in this case the policies of each university. In addition, the internal model should be able to support other software applications from other tenants e.g., other institutions or businesses.

3 SERENA: MULTITENANT RBS

We present the Serena Web-based framework which 1) eases the dynamic definition of requirements by utilizing a rule-based approach, 2) efficiently processes intermittent data giving instantaneous feedback by incorporating a forward-chaining rule engine, and 3) flexibly supports multitenancy by adopting concepts from group theory to model tenant structures. We dissect the inner workings of Serena by first illustrating its architecture and we later explain its execution semantics.

3.1 Serena Architecture

The architecture of Serena is illustrated in Figure 2. The server is written as a Node.js package that consists of five main components. The **fact base** maintains facts asserted from events and the **rule base** manages addition and removal of client rules. The **inference engine** is at the heart of the framework and evaluates received data according to the defined rules. It contains the graph builder that builds a Rete graph (Section 3.2.1) augmented with scopes, the *matcher* that finds consistent bindings in the fact base, and the *activation scheduler* that executes or fires instantiated rules. The **scoping module** builds an efficient encoding mechanism for scopes that will affect the match-

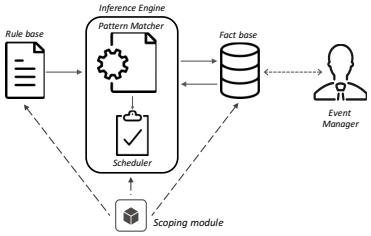


Figure 2: Architecture of the Serena server – The scoping module builds an efficient scoping mechanism that affects matching in the inference engine.

ing process. The **event manager** receives and queues event data from clients and pushes queued notifications to their recipients whenever their rules are executed.

On the client side Serena provides a library that initialises and maintains the (re)connections to the server runtime. It further manages sending of websocket messages and reception of notifications pushed from the server through the event manager.

3.2 Serena Execution Runtime

The Serena runtime is based on one of the most widely-used models of knowledge representation known as the production systems model (Newell, 1973). The distinguishing feature of production systems is the use of data-sensitive rules rather than sequenced instructions as the basis of computation.

Rule-based systems usually consist of a number of unordered rules referencing a global fact base. Similarly native multitenant architectures serve multiple clients that share a dedicated instance, accessing global resources. To support and cope with the knowledge of rules applicable to multiple clients and applications, rule engines and multitenant architectures require features for structural decomposition at the application level. Both models can benefit from modular design and structural abstractions as the systems they support grow in size and complexity.

We outline how the Serena framework embraces this approach, exemplified using the example scenario. We first begin by explaining the semantics of rules in Serena.

3.2.1 Rule-based Syntax

The university policies from the scenario in Section 2.1 can be easily expressed in a rule-based format. We illustrate such a rule to be added by a university security staff using a customised JSON Rules (Giurca and Pascalau, 2008) syntax in Listing 1 for the classroom policy 1. The rule object can be

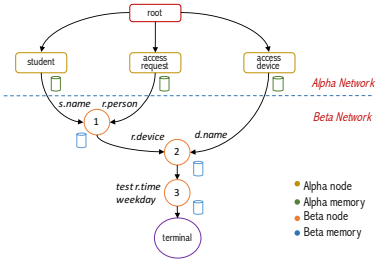


Figure 3: The Rete graph for classtime access rule – Once a token reaches the terminal node the rule is activated.

generated from a web-based graphical UI for intuitive rule definitions.

Listing 1: Rule for classtime access.

```
1  {rulename: "classtime-access",
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 }
```

A rule consists of a name, the left-hand side (LHS) with conditions for event detection, and a right-hand side (RHS) for a reaction after detection. The LHS of the definition (lines 2-6) 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. Here, we assert that the access request has been granted (line 9).

In Serena clients can dynamically add rules to the multitenant server through the framework's client library. The rules are appended to the existing inference engine's graph and define the real-time detection constraints for that client. In general, the inference engine will process and detect any events that clients are interested in and once activated will notify the relevant client(s). A client registers a handler that will be invoked once the rule has been activated.

3.2.2 The Rete Algorithm

Rules from clients are added to the server inference engine. Inference engines perform pattern-matching, a technique that reasons over the data to detect constraints that need to be fulfilled. Most current inference engines are based on the Rete algorithm (Forgy, 1982). Rete compiles rules (such as the one in Listing 1) into a data-flow graph that filters facts (data)

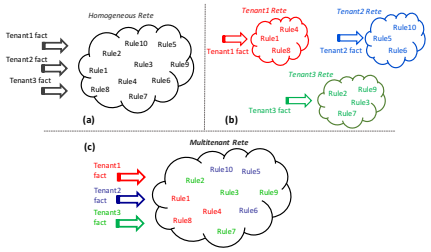


Figure 4: Conceptualizing a multitenant inference engine showing (a) naïve, (b) module-based and (c) scoped engine approaches.

as they propagate through nodes performing the actual matching process in the *match-execute cycle*. The matching process searches for consistent bindings between the facts and the existing rules. Efficient matching is achieved through exploiting 1) structural similarity – sharing of the nodes when building the graph, and 2) temporal redundancy – caching of intermediate matched data *tokens* between cycles of incoming results, at the price of higher memory usage.

In Figure 3 we show the Rete graph built in Serena after addition of the `classtime-access` rule from Listing 1. Facts enter the graph from the *root* node. In the upper alpha network, single-input *alpha nodes* perform generated type selection and intra-condition tests with an *alpha memory* node holding the results. The leftmost alpha node `student` filters facts of that type and stores them in its alpha memory.

The beta network is built in the lexical order of the condition elements forming a left-associative binary tree. Two-input *beta nodes* perform inter-condition tests or 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 in Figure 3 performs joins for a `student`'s name and the name of the person performing the `access request`, creates a *token* of both facts in the result and sends it to the next node. It also serves as left input for successive nodes in the beta network. The second beta node receives the token and performs joins of facts from a scanning `device` with the device of the `accessrequest`. For any beta node the right input is always an alpha memory node.

The final beta node in a condition sequence represents the full activation of a rule and is named a *terminal node*. In this case the rule `classtime-access` will be instantiated once a token reaches this node.

The Need for Reentrancy – In Rete rules are technically shared in their entirety within the network. Structural similarity promotes sharing of nodes performing the same test but corresponding to different rules.

As stated previously, clients can add rules dynamically to the Serena Web server. Adding rules in a multitenant setting is not, however, without its risks when using the naïve approach of having a single inference engine on the multitenant server (Figure 4 (a)) for all tenants. For example, a separate client in another university can develop a rule similar to the one in Listing 1. This will cause Rete to reuse the same graph and as a result, both universities will be receiving notifications of granted accesses in their dashboards whenever a student in either university enters a classroom: an undesirable result. In general, allowing clients to add rules in such multitenant settings brings about problems of unintended or spurious activations. We describe the common ways developers of rule-based systems attempt to solve this problem.

Rule Modules: One solution provided by a number of rule engines (as we discuss in Section 6) is to spawn a separate engine instance or *module* for each client or tenant (Figure 4(b)). This resolves the problems of unintended and spurious activations but nonetheless comes at a heavy cost: by eliminating sharing it undermines benefits of utility computing (Guo et al., 2007) and the strengths of the Rete algorithm resulting in a rapid increase in resource utilisation. Multiple separate engines make the system prone to duplication of resources e.g., nodes, working/intermediate memories and activation queues.

Relation Facts: Another solution is by asserting facts that indicate a *belongs to* relation in the universities, e.g., `{belongsTo science vub}` relates department `science` to the university `vub`. The entities would then be assigned the different departments. In this approach, the relation facts are added to the fact base *a priori*. Then, the rule from Listing 1 can be automatically modified to bind to such facts so that we can distinguish between institutions. The rule is modified by appending conditions 6 & 7 to the rule in Listing 1, resulting in the modified rule in Listing2.

Listing 2: Rule for classtime access w. relation facts.

```
1 {rulename: "classtime-access-uni",
2  conditions:[
3   {type:"student", name: "?name", dept:"?studept"},
4   {type:"accessdevice", name: "?dev",
     ↪ location:"classroom", dept:"?devdept"},
5   {type:"accessreq", id: "?reqid", person: "?name", time:
     ↪ "?t", device: "?dev"},
6   {type:"belongsTo" dept:"?devdept", uni:"?uni1"},
7   {type:"belongsTo" dept:"?studept", uni:"?uni1"},
8   /* .. action ... */
9 }
```

This new rule will create the rete graph shown in Figure 5. A new alpha node for `belongsTo` is added and needs to join with the `student` and `accessdevice` alpha nodes. When an access request is asserted it joins with the relevant student and device in nodes 1 and 2. The token reaches the join node 3

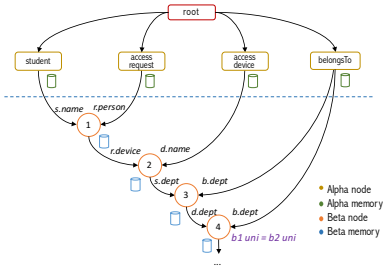


Figure 5: The Rete graph for the modified policy 1 rule – Additional nodes for performing discrimination are added.

causing a left activation. This will initiate a scan on the entire `belongsTo` alpha memory for in its right input to find compatible departments, which end up in beta join 4. Here computations are also performed for compatible departments for the device and to ensure that the request came from *the same university* as the student and the device. Certainly, the problem with this approach is that it increases additional nodes and the costly joins that the inference engine needs to perform.

Test Expressions: A better but more involved approach would require that every rule from tenants have additional discriminatory *test conditions*. We illustrate with the same example as the original rule in Listing 1. As with relation facts, we first assign students and devices to departments that they are part of. This time, however, the rule contains additional expression tests that check if the student and the device are in the same university. Line 6 of Listing 3 adds a boolean test that checks if the student and device are in the same university. This also results in a different Rete graph with an additional beta test node, shown in Figure 6. Note that at beta join node 2 the complete cross-product joins of `student` and `device` facts are still computed.

These last two approaches have similar limitations. First, they increase rule complexity and quickly become tedious in clients with more complex internal structures like multiple departmental levels (see next section). They also impact the underlying Rete graph: additional nodes are created and more computations are required. They further pollute the logical intent of the rule designer by adding conditions that need to enforce discrimination within the rules of all clients.

Listing 3: Rule for classtime access.

```
1 {rulename: "classtime-access-uni",
2  conditions:[
3   {$s: {type:"student", name: "?name"}},
4   {$d: {type:"accessdevice", name: "?dev",
     ↪ location:"classroom"}},
5   {type:"accessreq", id: "?reqid", person: "?name", time:
     ↪ "?t", device: "?dev"},
6   {type:"$test", expr:"( areInSameUni($d.dept,$s.dept) )"}
7   /* .. action ... */
8 }
```

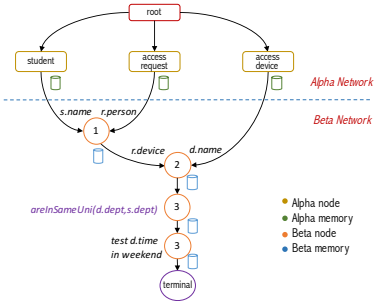


Figure 6: Rete graph for policy 1 with test expressions – A new node is added that checks compatibility of student and access device.

Support for multiple tenants can be improved by making the Rete algorithm *reentrant* such that any Rete graph can purely handle multiple inference states simultaneously for different sets of client rules, as illustrated in Figure 4(c).

Reentrant Rete – In order to enforce multitenancy during the inference engine's match cycles, it is imperative that an efficient representation be used to represent the hierarchy and to quickly determine the relationships between the data being processed at runtime. This is especially significant in a multitenant setup like the monitoring service where during a join the node needs to perform an additional "*is the token's request that we want to match with the access device originating from the same university?*"-check that is needed to determine compatible facts that apply to rules that belong to one tenant. This check tries to find a consistent binding for a device of the same exact university to avoid unintended activations with data from other tenants. Remember that the same check is also performed when facts from a different university are asserted into the monitoring service.

The numbers of these checks increase markedly when the multitenant rule engine supports relationships within and between tenants. For instance, policy 3 from the motivating example specified that students from a department in the university can have special access times to their (sub-)departmental labs. The rule for the policy is shown in Listing 4. This time there are two access requests (line 7, 8) from the student and the senior academic, 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 1 (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 7.

As identified in (Nayak et al., 1993), a major bottleneck in Rete and a number of its variants is such expensive computations during the match phase. There-

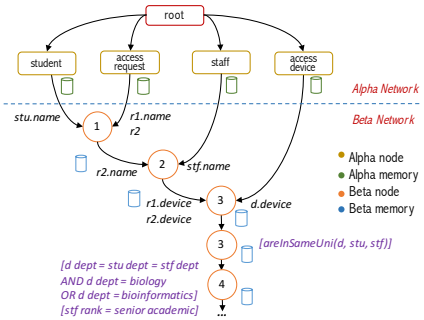


Figure 7: Rete graph for policy 3 with departmental checks – To discriminate data from different clients additional tests are needed.

fore any representation needs to be able to perform these checks in a zealously efficient manner. We next show the direction that Serena takes to solve this.

Listing 4: Rule for biology dept. weekend lab access.

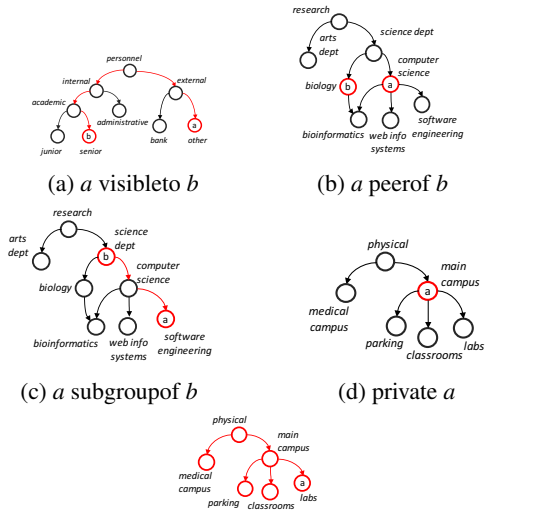
```
{rulename: "biology_weekend_access",
 conditions:[
  {$stu: {type:"student", name: "?stuname"}},
  {$stf: {type:"staff", name: "?stfname"}},
  {$d: {type:"accessdevice", name: "?dev",
   ↪ location:"labs"}},
  {type:"accessreq", person: "?stuname", device: "?dev"},
  {type:"accessreq", person: "?stfname",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

4 SCOPING THE RULE ENGINE

Serena's approach is to embrace the concepts of physical or logical *groups* of tenant clients and their relationships, common in multitenant applications (Grund et al., 2008). Examples of groups include research groups in a university, branches in an organization, hobby categories in forums, area zones when monitoring distributed sensor networks or user lists in Twitter. The framework only requires tenants to send their group hierarchies as a list of pairs and it converts and encodes the hierarchies into an efficient representation.

4.1 Representation of Tenant Structures

Serena models groups internally with the aim of using these representations to enforce data discrimination in the rule engine. We describe a structural representation that uses the notion of a group as a primitive. We showed in Figure 1 how we can conceptually structure the tenants and subtenants of the monitoring service in groups and subgroups. Serena represents the



(a) *a* visibleto *b*

(b) *a* peerof *b*

(c) *a* subgroupof *b*

(d) private *a*

(e) public *a*

Figure 8: Scopes supported in Serena – The scopes shown are in relation to the group hierarchy from Figure 1.

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.

One characteristic is that groups usually have an aspect of relationships between them – 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. We therefore appropriate the term *scopes* to represent the common relationships between groups in the hierarchy and designate that as a *scope hierarchy*. Serena adds scopes as (a series of) edges in the group hierarchy. Serena supports the following scope operations shown in Figure 8 on the client groups.

- ***visibleto***: In this scope we only capture data from clients in groups that share the same ancestor in the hierarchy. An example is capturing the data that pertains to *senior* academic researchers collaborating with *other* personnel within the same university (Figure 8a).
- ***peerof***: Only data items that originate from peers will be considered in this scope. The peers include groups that are at the same level in the hierarchy. For instance a researcher would want to create a rule with this scope that applies to members in *computer science* and *biology* departments, Figure 8b.
- ***subgroupof***: Only the data items added by the group or any of its subgroups are included in the scope. This scope is ideal for a departmental rule for *computer science* that will only apply to members of that department or sub-departments (*web*

info systems, *software engineering*, *bioinformatics*). See Figure 8c. Its dual is ***supergroupof***.

- ***private***: The private scope will exclusively source data 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 8d where we can target ID scanning devices at the campus entrance gates and not those in its subgroups such as the campus parking.
- ***public***: Here we capture all data from all defined groups in the hierarchy. The universities could, for example, collaborate in sharing security information between them so they can be interested in data from the devices/student/staff in all the groups and their subgroups (Figure 8e).

4.2 Encoding the Group Hierarchy

To enforce reentrancy and to efficiently process the various scopes within the match-execute cycle of the inference engine, the Serena framework internally converts the scopes discussed in Section 4.1 into a more efficient encoding. Our vision is to use an encoding method that, rather than performing computationally expensive scope checks such as path traversals in a hierarchical structure, performs (near) constant-time operations to entirely determine data relationships in the structure. 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 tenant 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.

We base our encoding on the *transitive closure*, a significant component modelling most relationships in knowledge and representation systems as identified in (Agrawal et al., 1989) 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.

The Group Hierarchy as a Lattice – Initially, Serena captures the hierarchy as a partially-ordered set (*poset*) (Habib and Nourine, 1994). The example hierarchy in Figure 1 can be represented as a poset $(P, \leqslant)$ with the binary relation $\leqslant$ defined as '*is a part of*' (in most cases the general $\leqslant$ relation '*is subgroup of*' suffices). The poset P has an element (a,b) iff a is part of b, so elements include *(junior, academic staff)*, *(bioinformatics, biology)*, *(internal, personnel)* and *(computer science, science dept)*.

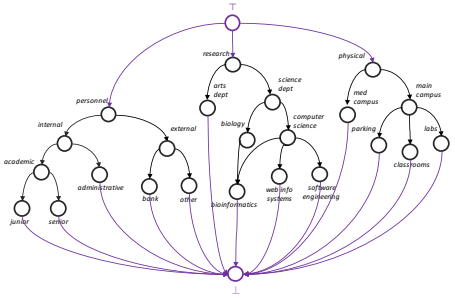


Figure 9: Hasse diagram of the group hierarchy as a lattice – The lattice is the basis of encoding the group hierarchies in a matrix.

To come up with an encoding, we convert the groups poset to a lattice L with a $\top$ and a $\bot$. This leads to the hierarchy depicted as the hasse diagram in Figure 9. Other distinct hierarchies can have their own top-level element same as $\top$. A lattice represents the group hierarchy in a form that is more efficient to encode and compute than the earlier poset representation.

Encoding the Lattice – With L, Serena performs a customised *bit-vector encoding* method that is based on the method by Aït-Kaci (Aït-Kaci et al., 1989). The result is a binary matrix encoding M_ϑ of the group hierarchy, shown in Figure 10 for our example, with the following properties:

i) The labels on the rows of M_ϑ represent the groups in L; similarly for 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` a is an ancestor of `group` b in L, and `0` otherwise

iii) An entry $M_{\vartheta(b,a)}$ has a `1` if `group` a = `group` b or if `group` a is an descendant of `group` b, and `0` otherwise

iv) 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) A is a minimal iff the column $M_{\vartheta(*,a)}$ has a `1` only at $M_{\vartheta(a,a)}$ and at $M_{\vartheta(\bot,a)}$

Additionally, the encoding process generates the *level* or depth of each group, which we store as an integer. We also store indexes for all the maximals in the matrix. We now show how the encoding is used to perform scoping within the inference engine.

Scoping with $\mathbf{M}_\vartheta$ – The encoding with M_ϑ is the basis of performing scope operations in the inference engine. To facilitate this Serena adds scope tests at appropriate nodes when building the Rete network,

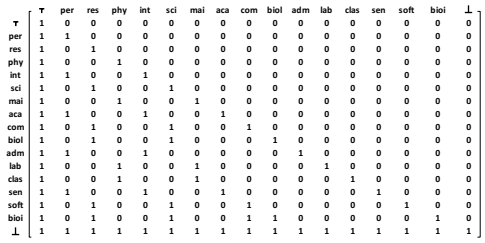


Figure 10: The group hierarchy matrix encoding M_{ϑ} – The groups are labels in the rows and the columns of the matrix.

and performs scoping operations in the beta betwork's beta join nodes during matching.

- ***visibleto:*** To perform a scope check of a `visibleto` b the runtime checks if the result of $M_{\vartheta(a,*)} \wedge M_{\vartheta(b,*)}$ is a maximal in M_{ϑ} as per property (iv).
- ***peerof:*** To check if a `peerof` b it calculates 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:*** For a scope check of a `public` b then we calculate if $M_{\vartheta(a,*)} \wedge M_{\vartheta(\top,*)} = M_{\vartheta(\top,*)}$ as per properties (ii) and (i).

With these operations, the Serena runtime can perform scope operations efficiently. It retrieves the values in the matrix and performs binary operations from the encoding in near-constant time.

4.3 Defining Scoped Rules

To expose scoped rule definitions, Serena follows a similar direction as Allen's work in (Allen, 1983) that proposes rule extensions for temporal interval constraints. Similarly, we present *scope-based constraints* by extending the normal rule syntax with scope-based definitions that specify structural constraints on the groups and the relationships between them, which we simply call *scopes*. The scopes supported are as in Section 4.1.

We illustrate in Listing 5, where we show how to define policy 3 of biology students lab accesses in the weekends from Section 2.1 using scope constraints.

Listing 5: Scoped 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    {$d: {type:"accessdevice", name: "?dev",
      ↪ location:"labs"}},
6    {type:"accessreq", id: "?reqid1", person: "?stuname",
      ↪ time: "?t1", device: "?dev"},
7    {type:"accessreq", id: "?reqid2", person: "?stfname",
      ↪ time: "?t2", device: "?dev"},
8    {type:"$test", expr:"(hourBetween(?t, 8, 20) &&
      ↪ (isWeekend(?t1, ?t2) == true) && isNear(?t1, ?t2) )"}
9   ],
10  scopes:[ "biology supergroupof ($stu & $stf & $d)", "$stf
     ↪ private senior"],
11  actions:[
12   {assert: {type: "accessrep", reqid:"?reqid1", allowed:
      ↪ true}}
13  ],
14  notify:[ "subgroupof administrative"]
15 }
```

The rule is similar to Listing 1, with an additional `scopes` section (line 10) where the bound condition variables in line 3, 4, and 5 are referenced to check whether the student, staff and device facts are all tagged to belong to the `biology` department or its subdepartments using the scope check `supergroupof`. The additional scope check in line 10 enforces the constraint that the staff member has to be in the `senior academic` group. The rule will therefore detect the constraints of policy 3, which was to capture lab accesses made in the weekends by a student that is accompanied by a senior academic staff member in the biology department and any of its subdepartments.

4.4 Scoped Execution and Notifications

Within the inference engine, the rule in Listing 5 will be built as shown in Figure 11. The main difference is in the beta node 3 where we now have in place 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.

The algorithm is modified as follows: on a left or right activation, we *first* perform the encoded scope check on the fact from the alpha memory or the token's fact respectively. If the check passes, we proceed with the join computation. When a token reaches beta node 3, for instance, 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 4.2. 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:

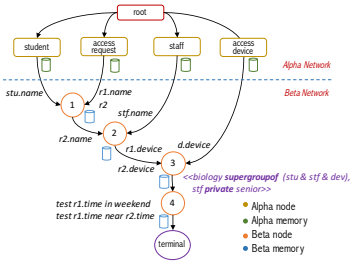


Figure 11: The scoped Rete graph for policy 3 – The expensive tests are replaced with scope tests performed by efficient encoding operations.

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

```
  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.

To decide *who to notify*, i.e., which group of which tenant should receive the notification, Serena rules expose a *notify* construct that specifies notifications once the rule is fired. The **notification scopes** invoke similar binary operations as in Section 4.2 to determine the groups to notify when performing a scope check during matching. In the case of the rule in Listing 5 the `notify` construct will notify members of `administrative` group and its subgroups.

5 EXPERIMENTAL EVALUATION

We evaluate our approach with the University Services Access Control scenario detailed in Section 2.1. We focus on investigating whether the scoping metadata architecture has significant computational benefits over traditional techniques in current rule engines.

a) Setup: The example scenario was implemented as a set of simulations that connect to a multitenant web server. The final application has a total of 61 groups in hierarchies, 39 access rules, and 73 concurrent clients across 3 sample universities. All clients are connected to the multitenant server concurrently through websocket connections managed by the framework. The server runs Node.js and has an AMD Opteron Processor 6272 at 2.1Ghz. The maximum RAM allocated to the entire experiment was 20GB.

b) Methodology: We categorised the general experiment setup into two categories: one with traditional rules using test expressions to enforce data discrimination (unscoped), and another with Serena's scoped rules using Serena. We randomly generated access requests from clients in both categories throttled in ranges of between 1-5 seconds (simulating real-world access request intervals), with one session consisting of 12 hours of runtime. The requests model students and staff from different departments or personnel levels *randomly* accessing various university locations. We ran 35 iterations for each category, making a total of approximately 70 sessions and 840 hours runtime. During the simulations we logged the number of beta joins and tests, memory (RSS) used and activations by the server.

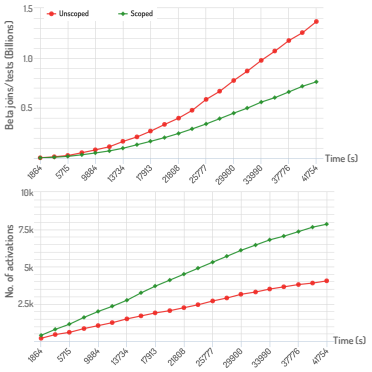


Figure 12: Results in one simulation session – We compare the cumulative results of one run of 12 hours.

c) Results & Discussion: Figure 12 shows the cumulative number of beta computations performed and rule activations from a single session of both scoped and unscoped engines, and in Figure 13 we calculate and chart box plots showing the distribution of observed results of all the 70 randomised sets of simulations.

From the graphs in Figure 12 we observe that the traditional unscoped Rete graph built from manually programming data discrimination within the rules suffers a marked increase in the number beta computations compared to our scoped graph. The unscoped approach spent more time processing the expensive join operations and beta tests in the engine. The scoped engine's scope checks use the encoding, leading to better performance, and consequently to a higher number of activations recorded (by approximately 31%) within the same session.

The aggregated results in Figure 13 show evidence of a better overall performance of the scoped engine. Compared to a traditional approach, Serena on average improves the computation of scope tests and total memory consumption, increasing the average number of rule activations of all randomised sessions of the experiment. The reduced memory consumption is as a result of space optimisations of scoping metadata

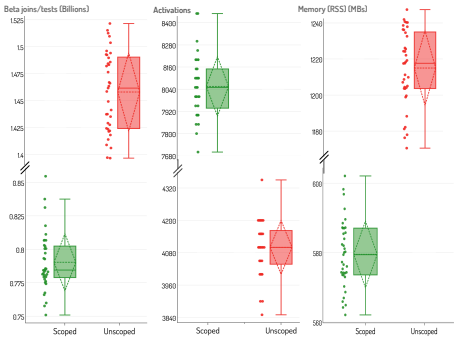


Figure 13: Aggregated results over all randomised simulations – The results were collected from 70 sessions of approx. 840hrs.

internally in the framework as opposed to using fact attributes in the traditional approach.

From the implementation and the results we observe that introducing metadata-driven reentrancy in the engine results in simpler rule design and efficient computation in the framework's rule engine. This in effect means that the security monitoring system can process a larger number of access requests at a faster rate than the traditional rule engine approach. The requirement is that the tenants need to specify their group structures to fully take advantage of the Serena framework: the groups to be encoded in the university hierarchy have to be defined and added to the engine so that the benefits provided by the encoding can be fully realised.

6 RELATED WORK

We describe similar techniques that support the development of multitenant applications focusing on preventing unnecessary duplication of processes and resources at the application level.

Decomposition in Rule-based Systems – Modern rule engines such as Drools (Proctor, 2012) and Jess (Hill, 2003) are based on the Rete algorithm as well and optionally provide Web server extensions. Techniques that they use to decompose larger rule bases into groups of rules in other engines all embrace the concept of *rulebooks* that consist of isolated sets of rules with no relationships between them. Examples are *modules* in Jess and *ruleflows* or *event sources* in Drools. Essentially these give each tenant their own Rete graphs, making them fundamentally independent. Serena in contrast facilitates and encodes scoping within a single heterogeneous Rete graph thus taking full advantage of structural similarity and temporal redundancy.

Schema Sharing in Databases – A common technique to support multitenancy is by mapping the context of clients into the existing patterns of conventional databases and similar systems, since most have limited out-of-the-box support for handling the metadata needed for multitenancy (Jacobs et al., 2007). Advanced schema-based techniques such as Sparse Columns (Chu et al., 2007), Extension Tables (Copeland and Khoshafian, 1985) and Multitenant Shared Tables (Grund et al., 2008) exist, but these have static and complex configurations that degrade in performance when ported to reactive rule engines with eager incremental processing as in our approach.

Multitenant Middleware – Most dedicated middleware for multitenant architectures aim to support multiple tenants at the application level using various techniques. The research in (Yaish et al., 2011) and (Fiaidhi et al., 2012) achieves this through variations of the aforementioned schema-based techniques. The SaaSMT approach (Pal et al., 2015) supports process-based tenant shareability based on architectural layers, which is limiting when requiring incremental, reactive processing. Support for application-level middleware through platforms like the Google App Engine/AppScale (Zahariev, 2009) and GigaSpaces (Cohen, 2004) use approaches similar to namespaces that partition application data across tenants but do not intrinsically support the flexibility and expressiveness of our formalised scopes. Nevertheless, with some effort they can be utilised as foundations of its runtime.

Distributed Event-based Systems – Distributed Event-based Systems exchange loosely-coupled data asynchronously between producers and consumers with notifications. Work in (Lim and Conan, 2014) and (Fiege et al., 2006) provides custom routing of event notifications from producers to subscribed consumers. Most of existing research, however, focuses on the existence of an overlay of brokers that filter notifications before reaching the respective consumers. In contrast, scoping in Serena is primarily for improving reentrancy in the inference engine during the matching process. Furthermore, Serena provides filtering of rule notifications at the event source to connected clients, which does not require the use of a broker architecture.

7 CONCLUSIONS AND FUTURE WORK

We have described Serena, a framework for reasoning in multitenant architectures via a Rete-based rule engine. Our technique is useful in a number of multi-

tenant applications to deal with the problem that much of the heterogeneous knowledge significant when performing reasoning and deductions can be structured hierarchically within a multitenant setup. The technique uses groups and common relationships between them to build an internal representation that captures the scopes present in many multitenant domains by using a hierarchy of groups. The model precisely controls the amount of deduction or computation performed automatically by the framework as information from tenants flows into the system in a both expressive and computationally effective manner.

As future work we would like to measure service shareability to estimate cost models. We would also like to investigate support for dynamic scopes that can be defined by the tenants during execution of the engine, thus affecting the encoding and the state of the inference engine's intermediate memories.

REFERENCES

Agrawal, R., Borgida, A., and Jagadish, H. V. (1989). Efficient management of transitive relationships in large data and knowledge bases. *SIGMOD Rec.*, 18(2):253–262.

Aït-Kaci, H., Boyer, R., Lincoln, P., and Nasr, R. (1989). Efficient implementation of lattice operations. *ACM Trans. Program. Lang. Syst.*, 11(1):115–146.

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. (2010). A view of cloud computing. *Communications of the ACM*, 53(4):50–58.

Brooks, Jr., F. P. (1987). No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19.

Chu, E., Beckmann, J., and Naughton, J. (2007). The case for a wide-table approach to manage sparse relational data sets. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 821–832, New York, NY, USA. ACM.

Cohen, U. (2004). Inside GigaSpaces XAP-Technical overview and value proposition. *New York, NY: GigaSpace Technologies Ltd*.

Copeland, G. P. and Khoshafian, S. N. (1985). A decomposition storage model. *SIGMOD Rec.*, 14(4):268–279.

Fiaidhi, J., Bojanova, I., Zhang, J., and Zhang, L. J. (2012). Enforcing multitenancy for cloud computing environments. *IT Professional*, 14(1):16–18.

Fiege, L., Cilia, M., Muhl, G., and Buchmann, A. (2006). Publish-subscribe grows up: support for management, visibility control, and heterogeneity. *IEEE Internet Computing*, 10(1):48–55.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37.

Giurca, A. and Pascalau, E. (2008). JSON rules. *Proceedings of the of 4th Workshop on Knowledge Engineering and Software Engineering, KESE*, 425:7–18.

Grund, M., Schapranow, M., Krueger, J., Schaffner, J., and Bog, A. (2008). Shared table access pattern analysis for multi-tenant applications. In *Advanced Management of Information for Globalized Enterprises, 2008. AMIGE 2008. IEEE Symposium on*, pages 1–5.

Guo, C. J., Sun, W., Huang, Y., Wang, Z. H., and Gao, B. (2007). A framework for native multi-tenancy application development and management. In *The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007)*, pages 551–558.

Habib, M. and Nourine, L. (1994). Bit-vector encoding for partially ordered sets. In *Orders, Algorithms, and Applications*, pages 1–12. Springer.

Hay, D., Healy, K. A., Hall, J., et al. (2000). Defining business rules – What are they really? *Final Report*.

Hill, E. F. (2003). *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA.

Jacobs, D., Aulbach, S., et al. (2007). Ruminations on multi-tenant databases. In *BTW*, volume 103, pages 514–521.

Lim, L. and Conan, D. (2014). Distributed event-based system with multiscoping for multiscalability. In *Proceedings of the 9th Workshop on Middleware for Next Generation Internet Computing*, MW4NG '14, pages 3:1–3:6, New York, NY, USA. ACM.

Nayak, P., Gupta, A., and Rosenbloom, P. S. (1993). The Soar Papers. chapter Comparison of the RETE and TREAT Production Matchers for Soar (a Summary), pages 621–626. MIT Press, Cambridge, MA, USA.

Newell, A. (1973). Production systems: Models of control structures. Technical report, DTIC Document.

Pal, S., Mandal, A. K., and Sarkar, A. (2015). Application multi-tenancy for software as a service. *SIGSOFT Softw. Eng. Notes*, 40(2):1–8.

Pathirage, M., Perera, S., Kumara, I., and Weerawarana, S. (2011). A multi-tenant architecture for business process executions. In *2011 IEEE International Conference on Web Services (ICWS)*, pages 121–128.

Proctor, M. (2012). Drools: A rule engine for complex event processing. In *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*, AGTIVE'11, Berlin, Heidelberg. Springer-Verlag.

Yaish, H., Goyal, M., and Feuerlicht, G. (2011). An elastic multi-tenant database schema for software as a service. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 737–743.

Zahariev, A. (2009). The Google App Engine. *Helsinki University of Technology*.