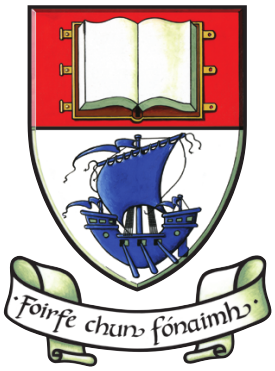
Access Control System Specification and its Implications for Performance



Bernard Butler, MSc.

School of Science and Computing

Waterford Institute of Technology

This dissertation is submitted for the degree of

Doctor of Philosophy

Supervisors: Dr. Brendan Jennings and Dr. Dmitri Botvich

May 2016

I would like to dedicate this thesis to my wife Mary and children Mairéad and Liam, and to my late parents Liam and Siobhán.

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy, is entirely my own work and has not been taken from the work of others save to the extent that such work has been cited and acknowledged within the text of my work.

Student ID 20021204

Signed .....

Bernard Butler, MSc.
May 2016

Acknowledgements

I acknowledge and thank my supervisory team of Dr Brendan Jennings and Dr Dmitri Botvich, who provided excellent support in my research. You asked the right questions, hauled me back on track when my research went down blind alleys, and generally encouraged me when I encountered any setbacks. Brendan, in particular, worked with me close to paper submission deadlines, to help get the papers over the line. . . He was always ready to share the benefit of his experience and, notably, to help me understand when the work was ready for submission!

I am also grateful to Miguel Ponce de Leon for his support and encouragement early in the journey, making it easier for me to juggle my responsibilities on projects in his team with the need to devote concerted time and effort to my PhD research. After I moved to my present unit, Dr Alan Davy supported me in every way possible, particularly by giving me space in the last few months to complete writing the dissertation.

I am grateful to colleagues in TSSG for many stimulating discussions about technical topics related to my research and to computing and life more generally. There are too many to mention here, so I have thanked you individually instead. I also wish to thank Dr Willie Donnelly and Eamonn de Leastar, who founded TSSG as a research group in which I found a home and a collegiate environment in which to follow my research interests. I am grateful to the PostGrad Support Unit in WIT, especially Angela who helped resolve any issues as they arose.

I am grateful to my thesis examiners Prof Rolf Stadler (KTH Stockholm) and Prof Stefan Decker (RWTH Aachen) for their careful consideration of my dissertation, stimulating discussion at the *Viva* and for the helpful suggestions which greatly enhanced the final product.

I quickly learned that pursuing a PhD is an all-encompassing ambition. Therefore I am deeply grateful to my family for their love, patience and support, especially when it seemed that I was obsessed with PhD research work. For a long time, they could not see my progress, or know when I would be ready to finish. Even though it was difficult at the time, they continued to believe in me, to look for "the light at the end of the tunnel". My in-laws also lent a hand, especially Ger, who regularly took the children on excursions when my wife was working on weekends. My siblings and wider family were also supportive. Mary, Mairéad and Liam: you gave me space to pursue the dream: I could not have done it without you, I love you all deeply, and am looking forward to spending more time with you now that the long PhD journey has ended!

Access Control System Specification and its Implications for Performance

Bernard Butler, MSc.

Supervisors: Dr. Brendan Jennings and Dr. Dmitri Botvich

Abstract

Access control evaluation performance is a challenge in modern enterprises. Such enterprises are characterized by workflows involving extensive communication events, as information is shared within and between groups in that enterprise. Security administrators are tasked with enabling communication events that help the business achieve its objectives, and of preventing the rest. They develop policies and deploy them in ever more complex access control infrastructures, and it is not always clear how to ensure the deployments have adequate performance.

In response, we propose a performance testbed (`STACS`), a means of generating policies and requests in bulk (`DomainManager`) for that testbed and a system for analyzing the performance measurements obtained from the testbed (`PARPACS`). `STACS` provides a means of performing reproducible, controlled experiments, so researchers can compare different performance improvement proposals on standard test infrastructure. `DomainManager` is built upon a flexible domain model that can be used as a foundation for a) generating large numbers of consistent, scenario-specific policies and requests and b) generating variants of those artifacts for performance comparison in `STACS`. `PARPACS` enables robust statistical models of performance to be built, so that researchers can *predict* performance and not just perform limited comparisons. Indeed, the three components are part of a larger `ATLAS` framework for diagnosing performance problems in an existing deployment and/or dimensioning a new deployment.

Using these research contributions, we conducted extensive experiments to evaluate `ATLAS` and generated more research contributions in the form of findings. These findings relate to the effects on performance of domain size, policy authoring patterns, policy optimisations, request complexity, system resource (e.g., memory) availability etc. While many of the main effects might be expected, there are significant (and often surprising) interaction effects that need to be considered in any access control deployment.

Although the motivating application concerned access control, `ATLAS` was designed so that it could be extended to other client-server performance studies, such as those concerning database query performance.

Contents

List of Figures

List of Tables

List of Algorithms

List of Source Examples

Publications

Butler, B. and Jennings, B. (2013). How soon can you decide whether Alice is permitted to communicate or share resources with Bob? *TinyToCS*, 2.

Butler, B. and Jennings, B. (2015). Measurement and Prediction of Access Control Policy Evaluation Performance. *Network and Service Management, IEEE Transactions on*, 12(4):526–539.

Butler, B., Jennings, B., and Botvich, D. (2010). XACML Policy Performance Evaluation Using a Flexible Load Testing Framework. In *Proc. 17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 648–650. ACM. Short paper.

Butler, B., Jennings, B., and Botvich, D. (2011). An experimental testbed to predict the performance of XACML Policy Decision Points. In *Proc. IM 2011 - TechSessions*.

Davy, S., Barron, J., Shi, L., Butler, B., Jennings, B., Griffin, K., and Collins, K. (2013). A Language Driven Approach to Multi-System Access Control. In *Proc. IM 2013 - AppSessions*, Ghent, Belgium.

Griffin, L., Butler, B., de Leastar, E., Jennings, B., and Botvich, D. (2012). On the performance of access control policy evaluation. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2012)*, pages 25–32. IEEE.

Chapter 1

Introduction

The motivation for the research presented in this dissertation was a discussion at a meeting attended by Cisco Inc. staff, notably Dr Keith Griffin and Kevin Collins. They raised several research challenges. One of those challenges related to access control evaluation performance and scalability in enterprise communications deployments.

This chapter introduces the research work done in response to this challenge. § 1.1 describes the problem in more detail, particularly in the context of enterprise communication management. Management of enterprise communications is not limited to facilitating desired interactions. It also includes preventing unwanted communication events, by applying access controls. The focus of this dissertation is on the *performance* of these access controls. Predominantly, it draws upon research in two domains: *client-server performance analysis* and *access control system specification*. Its two main contributions reflect this twin approach: 1) a framework for analysing the performance of client-server systems, of which access control policy evaluation is the example used in this dissertation, and 2) findings relating to how to specify such an access control system with the objective of increasing its performance.

§ 1.2 provides an overview of the dissertation. It indicates how access control and policy-based management are linked. § 1.2.2 outlines the Main Contributions of the research conducted in response to the problem statement described in § 1.1. Finally § 1.2.3 describes the structure of the rest of the dissertation and highlights some of the more significant sections and their contributions.

1.1 Problem Statement

Communications are the lifeblood of modern enterprises. They include voice communications, either face-to-face or mediated by communication services such as telephony and video conferencing. They also include written communications, from traditional pen and paper to electronic media such as email, IM, social network

updates and group chats. Indeed, many of the electronic media support multiple media types: plain text, audio, video and documents with rich text formats like spreadsheets and web pages, so many combinations of sender(s), receiver(s), communication channel and media type are possible.

Communication services support enterprise workflows all along the enterprise value chain. Typically each function manages its own set of workflows, comprising its own communication entities, but the underlying "data" often crosses functional boundaries. For example, the Finance function collects data on past and proposed transactions so as to manage the cashflow and ensure that the enterprise is using its assets efficiently and can meet its obligations. Meanwhile other functions in the enterprise produce and consume some of the same data, sharing it with other functions (such as Sales) as required.

Some of this data is expensive to produce or perhaps impossible to reproduce after the fact. Therefore it makes sense to reuse such multi-purpose data as much as possible and to reduce any "friction" caused by the need to transform it to meet new uses. Consequently many enterprise communication services also need to store the metadata (e.g., who called whom at what time) at the very least. In some cases such metadata is needed by auditors, or may be required by legislation. Other examples of "second use" of such data include business intelligence applications, particularly in support of operations management objectives.

Unified communications (UC) is a set of technologies designed to make the whole ecosystem of enterprise communications easier to manage. Regardless of communication mode (voice, text, video. . .), data can be shared and re-purposed. Furthermore, individuals, groups and responsible authorities can manage these communications. For example, individuals can manage their *presence* information and manage their own data-sharing obligations, such as "Send report X to person Y by Friday". The group manager can report on the group's activities, many of which are represented as data sharing events, such as "Sent purchase orders X,Y,Z to Person A last week".

As we have seen above, unified communications facilitate both data sharing and reporting on that sharing. They also have another role, that of limiting sharing events to those that serve the objectives of the enterprise and/or preventing sharing events that do not. This third role is given many names: depending on the context/vendor, it

may be called entitlement management, privilege management or access control. In this dissertation, we use the term *access control* as it is the most general, since it has meaning outside the world of enterprise communications, and many of the concepts presented in this dissertation have wider applicability to access control systems in general.

Enterprise access controls are often required by external agencies to ensure good corporate governance, to protect subscriber privacy, to prevent fraud and generally to act in the best interests of customers and other stakeholders. Other access rules are required by the enterprise itself to ensure the nondisclosure of sensitive material such as corporate strategy documents. Generally, the number and complexity of rules governing access to data (and to communication services generating that data) increases over time.

The source of the problem addressed in this dissertation is that the third (controlling and limiting) role is in conflict with the first (enabling) role. If it is not configured correctly, the controlling role can hinder the enabling role to the extent that it damages the ability of the enterprise to conduct its operations. One type of misconfiguration occurs when the access control rules have errors, so the access decisions could be too strict or too lenient in certain situations. This dissertation is not concerned with misconfiguration of this sort. Another type of misconfiguration can reduce the *performance* of the access control system to the point where it delays legitimate data sharing unnecessarily. In extreme cases, the unified communications system as a whole becomes unusable.

Systems administrators might argue against the loaded term “misconfiguration”, saying that the reason for performance deficiencies is that, over time, the business users mandate ever more complicated access rules, thereby increasing the difficulty of making access decisions in a timely fashion. Furthermore, it is also not always obvious to the system vendor whether the access control system has been dimensioned correctly prior to deployment, and has sufficient capacity for the inevitable growth in demand.

Therefore, given the market need indicated by Cisco Inc, together with the challenge of ensuring adequate performance of an access control system where the causes of performance problems are often less than obvious, we sought to prove the following research hypothesis:

> By building a domain model, measurement testbed and predictive model, it is possible to conduct experiments to answer questions relating to access control policy evaluation performance, and thereby to gain insight into the best way to setup and maintain a performant access control system for enterprise communications.

This research hypothesis is consistent with the twin approach described in §1 in that it has *constructive* elements (building models and infrastructures to support them), *abstructive* elements (explaining and predicting behaviour in the testbed with statistical predictive models) and *inductive* reasoning (extending the statistical predictions obtained in the controlled environment of the testbed to actual access control deployments).

Table 1.1 High-level Research Questions addressed in this dissertation

ID	Question
RQ1	How can access control evaluation performance be measured for use in performance experiments?
RQ2	How can domain models be specified and used to express enterprise access control scenarios?
RQ3	How can the data from performance experiments be used to understand and predict access control evaluation performance?
RQ4	What are the main factors affecting access control evaluation performance?

The Research Questions are consistent with this hypothesis, and are outlined in Table 1.1. They use the Research Hypothesis as a foundation and are intended to address some of the research gaps identified in Chapter 2.

1.2 Overview of the Dissertation

1.2.1 Introduction to access control evaluation performance

As mentioned in § 1.1, a typical enterprise uses *access control* procedures to control what users (both inside and outside the enterprise) can do with the digital assets managed by that enterprise. In practice, access control procedures are implemented

using policy based management. An access control *policy* $P = \{c, R_i, \{F_j\}\}$ is a finite set of rules R_i whose *consequences* are combined according to *combining algorithm* c and which evaluates to either `Permit` or `Deny`, optionally supplemented with *follow-up actions* F_j. One such follow-up action might be an obligation to log the decision that was made in a given context. *Policy authoring* is the process of capturing the access control objectives of the enterprise, refining them and encoding them in a format suitable for evaluation. Policy-based management is fundamentally a request-response protocol, in which an entity submits an access request to the policy evaluation system, the characteristics of the request are considered when searching for the relevant rules in the policy, which rules are then evaluated to provide a decision that is returned to the access requester. There is also an enforcement mechanism that ensures that access decisions are honoured: permitted access requests proceed without further hindrance; denied access requests do not proceed.

An example policy might take the form: "R_1: If the requester has the role of Implementer, he/she can set up the specified Group Chat. R_2: if the requester is in Marketing or Finance, he/she can participate in the specified Group Chat. R_3: the requester is not allowed to have anything to do with the specified Group Chat. c: Permit decisions override Deny decisions.". Given this policy, and the following request contexts

1. C_1: { Alice (role = Implementer), setup, weekly Group Chat };
2. C_2: { Bob (department = Marketing, role = Assistant), join, weekly Group Chat };
3. C_3: { Carol ((department = Operations, role = Manager), join, weekly Group Chat }.

the access control decisions would be D_1 = `Permit`, D_2 = `Permit` and D_3 = `Deny`, respectively. This is equivalent to saying that

1. A_1: Alice (who has the role of Implementer) can set up this weekly Group Chat;
2. A_2: Bob (a Marketing Assistant) is allowed to join this weekly Group Chat;
3. A_3: Carol (an Operations Manager) is not allowed to join this weekly Group Chat.

Each decision arises from matching the context of the given request against the conditions of the Rules in the Policy. In this instance,

1. C_1 matches the antecedent of R_1 which evaluates to `Permit`; it does not match R_2 but it does match R_3. However, because of c (`Permit` overrides `Deny`), the policy evaluates to $D_1 =$ `Permit`;
2. C_2 does not match the antecedent of R_1 but it does match the antecedent of R_2 which evaluates to `Permit`. It also matches R_3 which evaluates to `Deny`, but as with C_1, the rule combing algorithm c ensures that policy evaluates to $D_2 =$ `Permit`.
3. $D_2 =$ `Deny` because C_3 fails to match the antecedent of either R_1 or R_2 although it does match R_3 which evaluates to `Deny`. Since there are no `Permit` decisions, the overall (policy) decision is $D_3 =$ `Deny`, as stated.

These rules form part of the larger policy used as an example throughout this dissertation, and introduced in more detail in § 4.4 on page 143.

A common pattern for writing such policies is to have lists of rules that evaluate to `Permit`, followed by a rule that evaluates to `Deny` for all access requests, in much the same way as (depending on programming language) a `switch` or `case` statement "falls through" to a default value.

Note that each rule can have an arbitrarily complex matching condition, and the number of such rules can be very large; the rules themselves are often deeply nested. Indeed, as policies become larger and more complex (i.e., as their specifications grow more fine-grained) and policies need to evaluated more frequently (because their rules increasingly depend on dynamic context), policy evaluation can add significantly to the latency experienced by users of the system subject to such access controls. Furthermore, usage control, where data is shared and subject to ongoing checks to ensure that the conditions under which it was shared still apply, also results in more access control policy evaluations, see § 2.1.2.1 on page 28. Therefore understanding, modelling and predicting access control policy evaluation performance, which is the subject of this dissertation, is a topic of considerable importance in enterprise communications and data sharing.

1.2.2 Main Contributions

1.2.2.1 Contributions specific to understanding access control performance

The primary objective is to model access control performance with a view to comparing the performance of different access control settings and predicting performance of new access control system configurations.

The control settings include:

- choice of Policy Decision Point;
- memory and other computing resources assigned to the PDP server;
- domain size (e.g., small, medium and large);
- request complexity;
- how the rules are expressed: structure, language, encoding, etc.

We assume that these settings are associated with different performance characteristics. The effects of such a settings change need to be estimated using a valid statistical model so that a case can be made for changing the settings to new values. Let the latency added by the access control system be $x_1 >$ tol seconds, where "tol" is the maximum additional latency that is acceptable to the users. The justification is that, following a change to the access control system, the latency added by the access control system decreases to $x_2 <$ tol seconds. Of course any change in settings is usually associated with a cost, and different interventions might be associated with different costs, so it is necessary to weigh up the expected benefits of making the change against the cost of implementing that change.

Table 1.2 highlights the research contributions and findings that we believe will apply to actual deployments. Generally, the measurement-based statistical evidence in support of each finding in the table is very strong, and the interpretation of the cause of the effect is based on features of the testbed that we believe are consistent with actual deployments.

The PDP is that part of the access control infrastructure that has the task of consulting the policy set for each incoming request, and returning a response containing an access control decision (typically `Permit` or `Deny`). It is supported by

ID	Research Contribution
1	The choice of Policy Decision Point can have a very large bearing on access control performance.
2	Modifying a PDP to improve its software engineering, e.g., to use standard libraries rather than developing its own, can reduce its performance
3	Adding extra computing resources (e.g., memory) does not always improve performance
4	The Java Virtual Machine "caches" results for free (no user action is required) provided there is a) good similarity between requests, b) the request arrival rate is high and c) adequate memory is available
5	Removing duplicate rule clauses can *increase* service times
6	Increasing the size of a policy does not always reduce performance; it also depends on the size of the request
7	Small changes to a policy, e.g., adding or removing guard clauses (which might be suggested by simple tools) have very little effect on performance

Table 1.2 Research contributions (inductive/abductive) applicable from testbed measurements to actual deployments

other functions but is potentially the bottleneck in the access control system, particularly if it is not replicated and needs to handle all the incoming requests. We show that different PDPs can have very different service time response distributions, which makes their performance more or less sensitive to admission control strategies (§ 3.4.5 on page 79) and/or changes in the mix of incoming requests (§ 3.4.6 on page 84). Furthermore, PDPs using very different technologies can have very different mean levels of performance, usage of resources, etc. (§ 3.5 on page 91).

We compared version 1.4 and 2.0 of a PDP, where each version is functionally identical to the other (i.e., they give the same response for the same input), but which we found have dramatically different performance. Version 2.0 was developed from version 1.4 to remove specific parsing and other code—this was replaced with calls to standard libraries from the Spring Framework[1] and similar respected sources. While this effort removed lots of bespoke code and thus should make the PDP code easier to maintain, it comes at the expense of worse performance, particularly as policy and request sizes grow (§ 5.4 on page 201).

When looking for better performance, it is often tempting to *scale up* a server instance, by adding extra memory, processing cores, etc. However, we found that additional

[1] https://spring.io/

memory has limited benefit in its own right (§ 5.4 on page 201). However, for some PDPs, there is an interaction between additional memory and larger problem sizes, so additional memory can partially offset the adverse effects of larger/more complex requests in particular. Therefore *scaling out* (replicating the PDP function) might be more beneficial in general.

Since PDPs are essentially state machines, it is worth considering whether some form of caching is beneficial. That is, if a request arrives that is "similar" to a previous request arrival, it might be quicker to lookup the response cache than to evaluate it from scratch. While caching is attractive, it can also be complicated. However, we found that, if the request arrival rate is high enough, it seems that the Java Virtual Machine itself acts as an object cache (§ 5.4 on page 201). We believe this is because its garbage collection frequency can be low enough that the required objects are still in memory when the next request (of that type) arrives. Note that this occurs without the need for external caching mechanisms or effort by the PDP developer.

There is a widespread belief that redundant rules in a policy set necessarily result in worse performance. We discovered this was not always the case (§ 6.5.2 on page 223). We believe this is because matching processes can terminate without needing to check *all* possibilities. The procedure can end if, at an early stage, a match is found, or it becomes clear that a match cannot be found. Therefore, judiciously placed "redundant" rules can enable such early completion criteria to hold, even though, viewed in totality, rules elsewhere in the policy have the same effect. This suggests that, when removing redundant rules, care should be taken to leave any rule in place that short-cuts policy evaluation.

Another belief is that service times increase when policy sets increase in size. To some extent, this is true; however, it is only part of the story. The size and complexity of the request set being matched to the policy also plays a major role in access control evaluation performance. For small requests, service times can actually *decrease* as policy set size increases (§ 6.5.3 on page 225). We believe this is due to the same feature of early match procedure completion.

Lastly, small-scale changes to the structure of a policy set appear to bring little benefit (§ 6.5.1 on page 221). It appears that any policy set refactoring needs to take into account the benefits of matching the requests that are met in practice. Again, this emphasises the fact that requests play an important and largely overlooked role in

ID	Research Contribution
1	An extensible domain model for access control evaluation
2	Tooling to manage that domain model, e.g., generating suites of policies and requests; visualising policy sets
3	An extensible testbed for measuring service times, labelling them with full contextual information, and managing the process
4	An extensible statistical performance model, with analyses to validate the model and predict service times based on fitted models to the service time distribution
5	Model-based simulation, to compare the effects of interventions such as adminssion control and effects of changing request

Table 1.3 Research contributions (constructive) that generalise the scenarios and facilitate future research

access control evaluation performance. It also shows that static analysis of the policies alone is insufficient for access control performance prediction.

1.2.2.2 Contributions to understanding request-response system performance

The secondary objective was to build a testbed to perform controlled experiments on a request-response system (the policy management infrastructure, in the case of the scenario addressed in this dissertation), with the intention of modelling and predicting the performance of that system. The main contributions are listed in Table 1.3.

Table 1.3 highlights the research contributions that are transferable to other access control performance evaluation scenarios. The research contributions in Table 1.2 on page 8 arose from answering research questions using artifacts (notably policies and requests) motivated by enterprise communications management. To answer those questions, significant infrastructure elements were built: `Domain model Manager (DomainManager)`, `STACS` and `Performance Analysis, Reporting and Prediction of Access Control Systems (PARPACS)`; see Figure 1.1 on the following page. This infrastructure represents man-years of effort and is, to the best of our knowledge, the most comprehensive available for answering such research questions. Apart from `DomainManager`, which is used to generate policies and requests for access control system specification and so is specific to access control performance experiments, the other components are adaptable to experiments designed to understand the

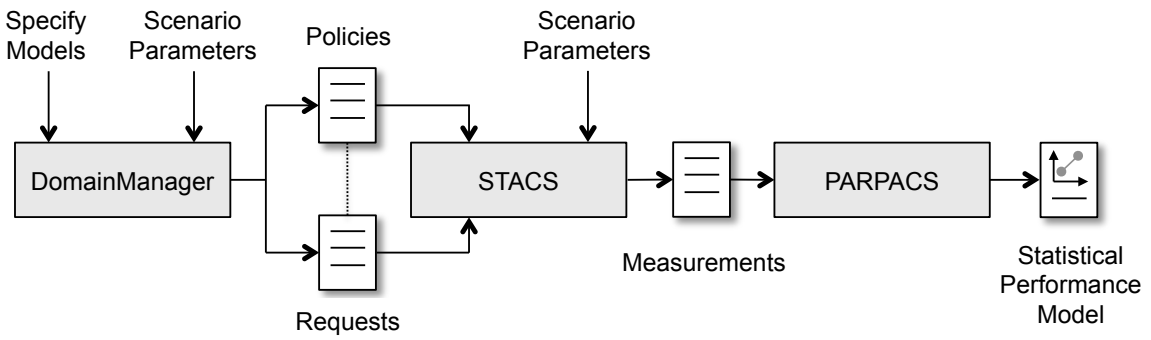


Fig. 1.1 Overview of the performance testbed (labeled `STACS`), the (policy and request) data generator (labeled `DomainManager`) and the analytics module (labeled `PARPACS`) developed to answer the Research Questions posed in this dissertation.

performance of any client-server system. In such systems, clients issue requests to servers that handle those requests according to some queueing discipline (First In, First Out (FIFO) is assumed in this dissertation) and issue responses to the clients by return. Other examples of client-server systems include

Web servers (which receive HyperText Transfer Protocol (HTTP) requests and either change some internal state (if the HTTP verb is `POST` or `DELETE`) or reply to the client with a response whose body which might contain a web page or equivalent content that was hosted on the server (if the HTTP verb is `GET`), and

Database servers which receive Create, Read, Update, Delete (CRUD) requests from clients and either change the state in the underlying knowledge base, e.g., by adding or removing data, or search for the answer to a query, which is then returned to the client.

In each case, the request-response protocol of access control policy evaluation aligns exactly with `GET` requests issued to a web server, and queries (e.g., in the format of a SQL `SELECT` statement) issued to a database server. Consequently, the A TooL for dimensioning Access control Systems (ATLAS) policy evaluation framework outlined in Figure 1.1 could be used, in principle, when modelling and improving the performance of web and of database servers; see §2.1.9 for some background.

One of the difficulties with measuring policy evaluation performance is that access control policies themselves are commercially sensitive. Thus there is a dearth of publicly available access control policies for the researcher. It is even more difficult to find requests that are related to those policies. Access Control Policy Tool

(ACPT) (Hu et al., 2011) is an excellent tool for model checking and combinatorial testing of policy sets and includes a GUI for generating access control test suites. However, since its focus is on testing functional (correctness and completeness) properties rather than performance, it lacks many of the features of `DomainManager` (§ 4.3 on page 126) and it does not have anything comparable to the other major components (`STACS` and `PARPACS`). The domain model we developed is sufficiently generic that it can be adapted to many other access control scenarios, and the realisation of that model in `DomainManager` enables powerful analysis and visualisation (§ 4.4 on page 143 and § 4.5 on page 159) of the underlying model semantics.

The primary purpose of `DomainManager` is to enable the production of suites of policies (§ 4.4 on page 143) and requests (§ 4.5) suitable for access control performance investigations. The input is a set of rules specified at attribute level (e.g., `Member.function` = `Finance`), rather than instance level (e.g., `Member.name` = `Alice`...). Many configuration options are available, so different variants of policies and requests can be generated together, supporting comparative experiments.

ACPT is designed to work with model checkers and with Automated Combinatorial Testing for Software, so combinatorial (mutation and similar schemes) testing can be conducted and the results collected for analysis. While `STACS` (§ 3) does not support the extensive combinatorial testing facilities of Automated Combinatorial Testing for Software (ACTS), it offers the ability to check that the responses are consistent across the suite of policies and requests, and uniquely, it collects `PDP` service time measurements, labelled with the full context of each measurement. Therefore it works in concert with `DomainManager` to enable researchers to conduct multi-factorial performance experiments.

Having collected the measurements, it is necessary to analyze them: look for unusual features, fit a statistical model, predict the performance under new conditions and estimate the uncertainty in those predictions. `PARPACS` is introduced in Chapter 5 and plays a major role in answering the extended scenario research questions in Chapter 6. Using `PARPACS`, researchers can follow a guided exploration path leading to recommendations that are based upon modern statistical procedures. `PARPACS` uses powerful statistical modelling procedures to build robust representations of the processes which are used in a PDP to make an access decision. So even though the PDP, and more generally the response-generating system, is treated as a black box, it is still possible to use `PARPACS` to predict its performance in new situations, assuming

the measurements from `STACS` are reliable and representative samples of its performance behaviour. Those performance predictions from `PARPACS` can be as general (e.g., "`SunXACML PDP` v1.4 has greater performance than `SunXACML PDP` 2.0.") or as specific (e.g., "Given PDP W, policy configuration X, request profile Y, and CPU configuration Z, increasing the available memory from 2GB to 4GB has no significant effect on performance.") as needed. The most important advantage of having a reliable underlying model is that it is possible to estimate the statistical uncertainty and hence judge whether a performance difference is statistically significant (i.e., unlikely to arise from chance) or not. Because the models are specified as formulae at run time, it is possible to configure them to include particular terms. These terms need to be consistent with the attributes of the data, but are otherwise unconstrained. Consequently, `PARPACS` can be used to model a whole class of performance models relating to request-response systems, which include database management systems, where the request is to perform some CRUD operation and the response is either 1) a status code to indicate whether the operation succeeded or 2) the results of a query.

The measurement testbed currently collects access control service times without considering arrival rates and patterns. That is, the PDP instance participating in the test works serially: it accepts the next request as soon as it returns the response for the previous request, and the service time covers the PDP processing time only; any time spent queueing for service is ignored. In a more realistic scenario, that queueing time would also be considered. However, if queueing time were included, it could easily dominate the PDP-only service time, in which case it would be difficult to focus on the PDP-specific effects. Therefore we chose to employ a measurement-based simulation approach:

1. collect the PDP-only service times;
2. derive the (multivariate) service time distribution and
3. use that distribution to configure discrete event simulations that take account of the total latency of the access control system

Note that this total delay is dominated by the PDP-only service time and any time spent waiting in the queue for service. By decoupling the two main latency components in this way, we can investigate the effects of scaling outwards (of the PDP), using different queueing disciplines, bunched arrivals, etc., all the while reusing the results of a single measurement run.

The approach is consistent with the methodology for experiment-based scientific research presented in Algorithm 3.1 on page 61.

1.2.3 Dissertation Organisation

The rest of the dissertation is organised as follows. Chapter 2 on page 19 provides a longer introduction to the problem than the relatively brief treatment in § 1.1 above. § 2.1 describes the background of the problems relating to access control performance. The topic has many facets, as it includes

- organisational behaviour, regulation and security (§ 2.1.1 on page 19);
- the design of rules-based access control systems (§ 2.1.2 on page 26);
- the industry standard for such systems: eXtensible Access Control Markup Language (XACML) (§ 2.1.3 on page 29);
- what performance means, in relation to access control, the relationship between performance and scalability, and what role caching might play (§ 2.1.4 on page 31);
- why an access control performance testbed would be a valuable contribution (§ 2.1.5 on page 34);
- how an access control performance testbed could learn from and complement existing work on access control correctness testing (§ 2.1.6 on page 34)
- policy authoring and conflict detection, which may have a role to play in performance analysis (§ 2.1.7 on page 35);
- statistical performance models, which enable us to predict performance in real-life deployments based on the analysis of service times measured under controlled conditions in the testbed (§ 2.2.11 on page 54);
- taking a wider view, considering recent advances in web service performance and scalability, and what they might offer (§ 2.1.8 on page 36).

Of course, the research community has also been active, and some of the relevant publications are reviewed in § 2.2 on page 37. The literature is classified under the following headings:

- the metamodels that policy authors use to structure and express their policies and that affect performance in interesting ways (§ 2.2.1 on page 38);
- the formal languages, often based on logic programming and semantic web technologies, that researchers use to understand the meaning and complexity of policies (§ 2.2.2 on page 40);
- with these foundations, researchers have proposed their own approaches to reduce the time taken to evaluate the policies for a *single* request (§ 2.2.3 on page 42, summarised in Table 2.1 on page 44);
- one of our main contributions is to develop a means of generating policies in bulk, so we review some of the relevant literature on policy authoring in general (§ 2.2.4 on page 45);
- our bulk policy generation technique is related to policy refinement, so we review some relevant papers (§ 2.2.5 on page 46);
- it also makes sense to consider ways of breaking the policy evaluation down to a set of smaller tasks, suitable for distribution to more than one PDP (§ 2.2.6 on page 47);
- other researchers have performed PDP intercomparisons, etc., so we survey their efforts (§ 2.1.5 on page 34);
- policy testing, as least for correctness has a longer history and researchers have faced similar challenges to those we faced (§ 2.2.8 on page 50);
- the characteristics of a "good" policy set for testing, and how to generate such a set, were considered by other researchers (§ 2.2.9 on page 51);
- similarly, bulk generation of requests is also needed for testing; other researchers have generated requests mainly for correctness testing (§ 2.2.10 on page 52)

Given this analysis, the knowledge gaps became more apparent and are presented in § 2.3 on page 55. Chapters 3–6 each take a subset of the research questions presented in Table 2.3 on page 58. and use that subset to motivate their contributions

Chapter 3 on page 59 addresses research question RQ1: "How can access control evaluation performance be measured for use in performance experiments?". Algorithm 3.1 on page 61 outlines the methodology used throughout the research, which is based on the use of the analysis of performance measurements to answer

specific questions, which will often motivate further experiments. The first concern is with the scope of the performance model, particularly what *service time* is and why there should be a distinction between infrequent and frequent arrivals. Having scoped the performance model, it is then time to describe STACS, its architecture and usage for both comparative and predictive experiments, and the crucial finding that requests can be clustered by service time for a given PDP, so the service time distribution can be quite complex. § 3.4 on page 72 describes how STACS can be used with a simulation package like OPNET™ to explore the effects of different requests mixes (§ 3.4.6 on page 84), or to compare different load control algorithms when the request arrival rate increases (§ 3.4.5 on page 79). We also show that closed form expressions exist for the queue length and the average time spent in the queue, assuming the service time follows a hyperexponential distribution with estimated centres (§ 3.4.1 on page 73). In these experiments, the publicly available `continue` policies and associated requests set were used and some interesting findings were made regarding the PDPs used. § 3.5 on page 91 introduces a prototype PDP using new technologies with high performance (§ 3.5.1 on page 93) which, despite its limitations, can be compared with more traditional PDPs (§ 3.5.2 on page 96) using our analysis framework. The initial (screening) experiment was a direct comparison between the new PDP named `njsrpdp` and the reference `SunXACML PDP`. The comparison indicated that the performance of the the new PDP was significantly better, so a supplementary experiment was undertaken to determine whether the reduced request size associated with `njsrpdp` contributed to its better performance; it does.

One of the problems with Chapter 3 on page 59 is that it is limited to the `continue` policy set and hence is not truly representative of enterprise access control. Thus Chapter 4 on page 111 describes how to generate suitable policies and requests for the domain of interest. Indeed, this is also the topic of research question RQ2: "How can domain models be specified and used to express enterprise access control scenarios?". § 4.1.1 on page 113 describes the challenge of creating policies that are sufficiently restrictive, but not overly so. § 4.1.3 on page 116 outlines how requests can be related to policies. § 4.2 on page 117 describes the domain model, particularly its three component submodels: the static, policy and request models. § 4.3 on page 126 describes the property graph metamodel that underlies all three domain model components. § 4.4 on page 143 describes one of our main contributions, which is that many policies can be generated from a seed *template* policy by means of a special refinement procedure, described in more detail in Appendix A on page 259. An

algorithm for generating requests consistent with those policies is described in § 4.5 on page 159; more details are provided in Appendix B on page 263. However, to understand such algorithms, it is necessary to understand the way that policies are represented in the property graph model, notably the rule and target hierarchies (§ 4.3.2.1 on page 136). Of course, being a graph, it is possible to derive some measures that summarise some of its properties (§ 4.6.1 on page 177). It is necessary to evaluate whether these models, and the software component that manages them (`DomainManager`) work well with `STACS`. Therefore § 4.6 on page 177 does some comparisons using representative enterprise access control policies and a static model representing a bank and a consulting company that need to work together closely.

Now that there are many more parameters available for performance experiments, so there is a need for greatly enhanced visualisation and performance analysis. Therefore, Chapter 5 on page 182 describes another major research contribution: `PARPACS`. The topic addressed in the chapter is RQ3: "How can the data from performance experiments be used to understand and predict access control evaluation performance?". § 5.1 on page 183 introduces the concept of *advanced* factors which are made possible by the introduction of `DomainManager` and the enhanced `STACS`. The analysis of such factors requires the development of a new analysis component: `PARPACS`. § 5.2 on page 185 provides an overview of this component, particularly its statistical provenance, mapping access control performance experiments into procedures that can benefit from the extensive capabilities of R (R Core Team, 2014). The first step in this mapping is to build the statistical model (§ 5.2 on page 185) and the easiest way to understand the process is to work through a scenario (§ 5.3.1 on page 187) where security consultants are attempting to troubleshoot a problematic access control deployment. This is an integrated scenario, in that it draws upon all the major software components: `STACS`, `DomainManager` and `PARPACS`. The policies and requests used are described in § 4.4 on page 143 and § 4.5 on page 159 § 5.3.3 on page 191 indicates the scope of the measurements collected by `STACS`. The statistical model needs to be tuned to fit the measured data (§ 5.3.4 on page 192) through a series of visualisations (mostly of the behaviour of the residuals) and statistical tests, such as those outlined by Butler et al. (1999). The statistical model is then sufficiently reliable to be used for prediction (§ 5.4 on page 201) where the predicted effects of certain factors and their interactions are used to answer the security consultants' questions.

The analysis in Chapter 5 on page 182 used data that had been summarised and labelled with composite model factors. This analysis answered some of the consultants' questions definitively. However the consultants decided to perform a more detailed follow-up study, mostly relating to the interaction of domain size and request complexity (Chapter 6 on page 207). The topic addressed in the chapter is mostly RQ4 "What are the main factors affecting access control evaluation performance?". § 6.1 on page 208 introduces the extended evaluation procedure and § 6.2 on page 209 describes how we check that the decisions are correct (where the expected responses are known) or consistent across factor settings otherwise. § 6.3 on page 211 describes the extended scenario questions and § 6.4 on page 213 describes the measurement runs, model refinements and analysis steps we used. § 6.5.1 on page 221 considers the performance effects of different policy authoring choices, § 6.5.2 on page 223 considers the performance effects of different policy optimisations, § 6.5.3 on page 225 considers the performance effect of different levels of request complexity and § 6.5.4 on page 228 considers, in detailed interaction terms, the effects of domain size on performance.

Finally, Chapter 7 on page 231 draws together the general principles of access control performance (§ 7.4 on page 247), abstracted mainly from Chapters 3 on page 59, 5 on page 182 and 6 on page 207. The general principles include our main research contributions, which are also collected in Tables 1.2 and 1.3. The findings from all the scenario experiments are assembled in § 7.2 on page 238 for convenience. Although significant progress has been made, there is still more research to do (§ 7.3 on page 241) and we hope to continue this research journey.

Chapter 2

Background and Literature Review

2.1 Background

2.1.1 Enterprise communications

Information and communication technologies (ICT) enable people to communicate with each other and to create and share content more easily than ever before. The technological and cost barriers to creating and sharing content are much lower than in previous decades. Direct communication between people is also easier, particularly when participants share their *presence* information.

With this greater power comes greater responsibility. For example, individuals need to protect their privacy from intrusion by unwanted parties, and corporate users need to protect the confidentiality and integrity of sensitive data, business plans and other digital resources. That is, some communication events, with or without intermediate content, are "considered harmful"[1].

In an enterprise scenario, the number of possible communication events is very large. Communication events include sharing a document with a colleague, sending an email to the team or joining a group chat. Some of these events would not serve the goals of the enterprise. Indeed, in some companies, particular groups of employees are prevented from conversing directly for regulatory and/or business governance reasons. Of course, there are other communication events that are considered valuable and are facilitated by the enterprise, e.g., events where project teams form groups to work together and share digital assets, including persistent entities such as computer files and transient entities such as group chats. The access control of group chats was an initial motivation for our research and is the canonical use case in this Chapter.

The enterprise network architect and administrators capture this knowledge of desirable and undesirable communication events and encode it in the form of policies

[1] With apologies to Edsger Dijskstra...

that are used by the enterprise's access control system to achieve the business goals of the enterprise.

In the case of enterprises that wish to enforce business rules regarding desirable communication events, every communication event should be vetted first. Thus, behind the scenes, Principals issue a request before instigating any communication event. Each request triggers a policy evaluation and the policy sets themselves become larger as they need to cover ever more communication events. The resulting policy evaluation overhead increases network traffic (reducing available bandwidth for other, more directly useful network communications) and also introduces latency (delay) at the start of *each* access attempt. Thus policy evaluation has the potential of becoming a bottleneck in modern communications. For example, policy control of instant messaging communications in enterprises necessitates large numbers of policy evaluations, when many users seek to access a common resource (the group chat) at approximately the same time and there might be complex rules in place governing a) who can participate and b) what privileges they have once they have joined the chat.

Requests to initiate communication events require simple but robust and performant access control *procedures* that are informed by access control *policies*. Indeed, enterprises can purchase products to achieve their access control objectives and corporate integrated communication solutions (Cisco, 2012; IBM, 2012) have widespread adoption within industry. The tools have revolutionised communication by both enabling and controlling it, in near real time. As such, any access control mechanism used to protect corporation communications must provide a decision in near real time in order to preserve usability of the underlying communication medium.

However when vendors deploy such systems, they find that meeting the agreed performance service level can prove challenging; the reasons include:

Compliance An example business policy might be to enforce the principle of *Separation of Duties (SoD)* to ensure that credit decisions are made only by an authorised officer and that all such decisions must be approved by a separate review committee before they can be issued. Compliance also mandates that all access requests and responses should be auditable, to ensure that they meet minimum standards of business practice—this increases the overhead further. Other sources of regulatory requirements include the Sarbanes-Oxley Act[2] and

[2]From Wikipedia: The Sarbanes–Oxley Act of 2002 (Pub.L. 107–204, 116 Stat. 745, enacted July 30, 2002), also known as the "Public Company Accounting Reform and Investor Protection Act" (in

the FDA/21CFR[3] programmes in the food/medical sector. Such regulations specify mandatory, auditable information management procedures;

BYOD Bring-Your-Own-Device (BYOD) is a growing trend and represents both an opportunity (lower procurement and training costs) and a threat (less direct control over client devices). Compared to the days when all devices accessing the network and its assets were managed centrally and hence implicitly trusted, there are many more heterogeneous devices and the *trust boundary* between client devices and the protected resources they wish to access is much more complicated. Consequently, it becomes essential to check each access request to ensure it should be permitted;

Insider threats Recent news reports have highlighted cases such as that of Edward Snowden (Esposito and Cole, 2013) where employees and/or contractors with unnecessarily elevated privileges were able to obtain and disclose sensitive data. The response in many organisations has been to review their privilege management systems and to introduce additional and even more extensive controls;

zero-day attacks Pervasive access control can help to limit the risks even of undiscovered vulnerabilities (Peterson and Nair, 2015). If outside attackers break through perimeter defences such as firewalls and SSL-based authentication, rigorous, dynamic access control provides greater defence in depth by making it harder for attackers to exploit any security breach;

internal requirements Each organisation needs to ensure the confidentiality of its own sensitive information, such as strategic plans or trade secrets (like the specifications of products or services). Many organisations also have a duty of care to protect the privacy (in the form of personally identifiable information) of stakeholders including their employees, suppliers and customers.

---

the Senate) and "Corporate and Auditing Accountability and Responsibility Act" (in the House) and more commonly called Sarbanes–Oxley, Sarbox or SOX, is a United States federal law that set new or expanded requirements for all U.S. public company boards, management and public accounting firms for corporate governance and robust accounting procedures.

[3]From Wikipedia: Title 21 CFR Part 11 is the part of Title 21 of the Code of Federal Regulations that establishes the United States Food and Drug Administration (FDA) regulations on electronic records and electronic signatures (ERES). Part 11, as it is commonly called, defines the criteria under which electronic records and electronic signatures are considered trustworthy, reliable, and equivalent to paper records (Title 21 CFR Part 11 Section 11.1 (a)).

As mentioned previously, one of the main areas where (near) real-time access control can be difficult is when managing large scale group chats. Discussions with Cisco representatives highlighted the access control overheads of managing large scale group chats that happen within or between complex enterprises. In such organisations, the organisational structures and roles impose constraints on *who* can join a chat. Often the policies are dynamic (context-sensitive) such as when an Ethical Wall (Brewer and Nash, 1989) security property needs to be enforced. Dynamic policies often lead to more onerous policy evaluation. For example, when n staff need to join a group chat, at least $\sum_{i=1}^{n} i = O(n^2)$ policy evaluations are needed where n is the number of participants. Moreover, the constraints do not apply just to potential participants. For example, to protect confidential information, some media exchanges (such as voice conference calls) may be permitted but sharing certain media types (such as spreadsheets) between specific (groups of) participants may be prohibited.

Achieving compliance requires specialist knowledge and tools, so enterprises often purchase solutions and/or expertise from suppliers in the *Identity and Access Management (IAM)* sector. Such regulations help to meet societal goals but can be difficult and expensive to implement, and should not hinder legitimate communication events which are necessary for enterprise operations.

Clearly, if policy evaluation performance requirements are not met, the overhead of ensuring conformance with fine-grained access control policies can prove detrimental to other system requirements such as timeliness of response and general system usability. That is, policy-based access control systems should ensure that *safety* objectives are achieved, without restricting the expected/normal functioning of the organisation. Therefore the challenge is to maintain a satisfactory user experience while ensuring that system security goals are met at a reasonable cost and in a flexible and easily maintained way.

Performance considerations require careful analysis of the full privilege management system. The potential causes of performance problems are many and varied, from policy formulation to server configuration. Hence a *comprehensive* domain model is needed in which to evaluate different policy-based access control proposals before deployment. A policy-based management system for access control comprises, at a minimum, the following entities

1. the policies that encode the business constraints;

2. the requests from *Principals* (or their agents) to access (potentially) sensitive resources;
3. a (possibly replicated) decision point, which receives requests, checks the policies to decide whether access should be granted and then formulates the response;
4. the physical infrastructure (servers, etc.) that is specified, configured and maintained by administrators acting on behalf of the organisation.
5. the responses from the management system indicating whether access is permitted or not;
6. the actions triggered by these responses, such as a) maintaining a log of access requests and responses, b) creating and using an authorisation token if access is permitted, and c) falling back to an alternative if access is denied;

In this dissertation we consider entities 1) to 5) above.

2.1.1.1 Policy evaluation architecture

The industry standard for policy-based access control is XACML (OASIS XACML-TC, 2014). The XACML standard also describes an architecture with server roles etc., and describes how server participate in the overall data flow—see Figure 2.1. Each flow has a label l_i where $l_i < l_j$ implies a temporal ordering such that flow i happens "before" flow j; the overall flow is consistent with the description of the main features of policy-based access control systems in §2.1.

It is clear from Figure 2.1 that the architecture ensures, owing to its design principles, that policies can be managed separately from requests. By necessity, they interact at the Policy Decision Point (PDP), but otherwise the system architect has the freedom to make implementation choices to ensure that *functional* safety objectives are met, while maximising policy evaluation performance and other *non-functional* utility measures.

2.1.1.2 Fine-grained policies

For optimal flexibility, *fine-grained* access rules are needed to ensure that

- access is *permitted* only where necessary (so security requirements are met), and

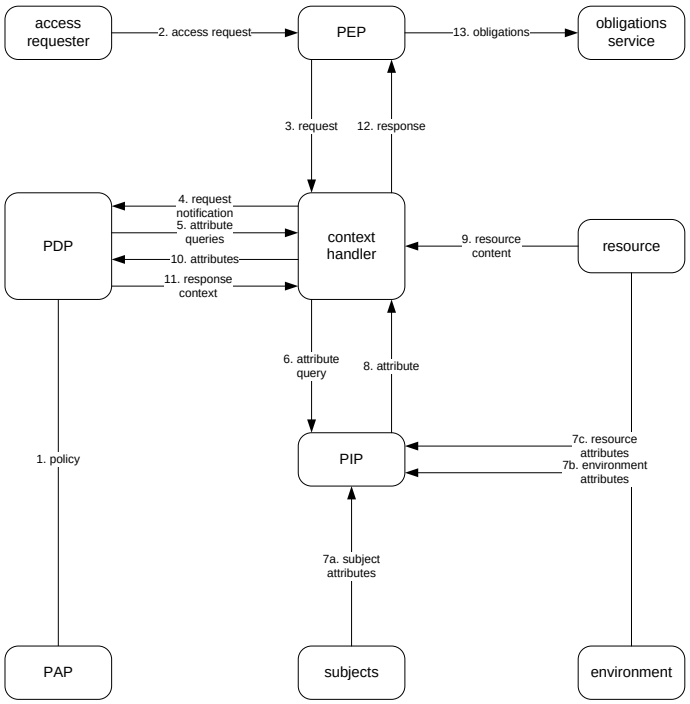


Fig. 2.1 Data flows in policy-based access control—taken from XACML 2.0 standard (OASIS XACML-TC, 2014)

- access is *denied* only where necessary (so service functionality is maintained).

Such fine-grained access control enables system administrators to implement security policies with complex boundaries between what is permitted and what is denied but also leads, in general, to more complex policy sets, resulting in longer PDP processing times. Fine-grained access control also means that more types of behaviour need to be approved so the PDP has more checking to do within a session. As an example, two Subjects may be permitted to exchange Resources with media type 1 (e.g., voice or plain email messages) but not Resources with media type 2 (e.g., email or IM file attachments). As with any security deployment, it is necessary to respond rapidly as new threats arise, so *dynamic* updates to rules specifying that decision boundary are necessary, e.g., in the case of Bring Your Own Device (BYOD). This requirement to support policy sets that evolve over time makes it more difficult to use caching and similar strategies to improve PDP performance and scalability.

Organisations are deploying ever more complex communications and content management systems to control entitlements to read, write and share protected digital resources. The resulting access control policy infrastructure defines and resolves the

(a) Before: simple boundary, few rules (b) Now: complex boundary, many rules

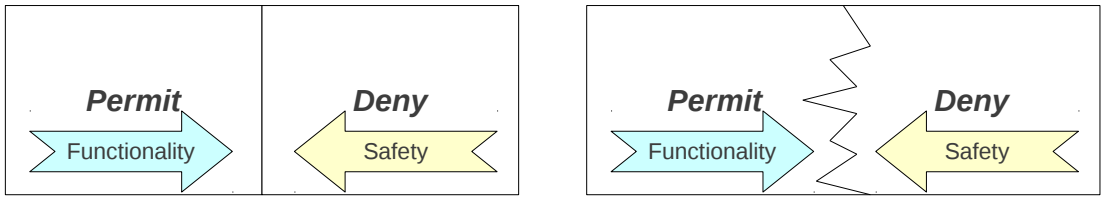


Fig. 2.2 The contrast between access control then (coarse-grained) and now (fine-grained). More context and rules are needed to decide whether to Permit/Deny a given request in the new regime so the PDP has significantly more work to do.

complex decision boundary between *safety* and *openness*, see Figure 2.2. This complexity poses major usability concerns in real-time corporate communication scenarios (due to high complexity) and in online social networks (due to large scale).

2.1.1.3 Dynamic policies

Some security properties (such as the maintenance of ethical walls to prevent conflicts of interest) require *dynamic* policy enforcement. In the case of ethical walls, where past access decisions affect present access decisions, the policies are highly dependent on context. In some workflows, such as loan approval, a SoD security property applies and sometimes an individual could have a specific role for that workflow instance and a different role for the next workflow. Since the context changes, this means that each step in the workflow requires a specific access request to be permitted; it is not possible to infer whether access should be permitted based just on static information or previous access decisions outside this workflow.

Generally, resource authorisation decisions are made by considering the request context, looking up the relevant attributes in policy sets and organisational databases as necessary. Thus the (setup) time required to respond to an access request can be a significant overhead when completing a communication event. In many cases, this overhead is independent of improvements in other factors (such as bandwidth) that are intended to improve communication performance.

Returning to our motivating example, policy control of enterprise group chats causes many access requests to be issued, because the access control system must decide which participant pairs can communicate. Such policy control is needed in

organisations where ethical walls (Brewer and Nash, 1989) must be maintained between groups for business reasons.

2.1.2 Introduction to access control

We have already seen that access control *mechanisms* are used to prevent "harmful" communication events. For example:

- a parent might wish to share family photographs with a grandparent, but not with the wider user community of a photograph-sharing site;
- in an organisation, corporate governance procedures impose Binding of Duties (BoD) controls between those who use its resources and those who approve this use.

Such scenarios require simple but robust and performant access control *procedures* that are informed by access control *policies*. Much of the public literature on access control policies focuses on techniques to ensure that sets of deployed policies are consistent with high level security requirements. This dissertation addresses the *performance* of the access control evaluation, on the assumption that the policies are correct.

In the enterprise communication events considered in this dissertation, communication events are modeled as `Subject`s applying `Action`s to `Resource`s. Access control is a system which enables an *Authority* to limit these interactions. Access control constraints are *binary*-valued decisions, being either `Permit` or `Deny`. Each decision is made by searching business `Rule`s to find one or more matches and evaluating the rules relating to the given request. Rules are combined into business policies.

It should be noted that the functional objective of an access control system is to apply policies to ensure that Subjects can access Resources *if and only if* they are entitled to do so. That is, the access control system should:

- meet its safety objectives: no unauthorised access is granted;
- meet its usability objectives: no authorised access is denied;
- meet its nonfunctional objectives such as performance; to paraphrase the legal maxim: "access delayed is access denied";

- is maintainable: it can support and adapt to changes in requirements and the computing environment.

As we have seen, access control policies often need to be *fine-grained* (in the sense of having a large number of low-level (hence highly specific) policies) to take full account of this context. Attribute-Based Access Control (ABAC) (Hu et al., 2013) was designed to meet the need for such flexible, fine-grained access control policies, and the model is sufficiently flexible to be able to support dynamic policies too.

Access control policies appear to be relatively simple, since the policy *action* is generally binary-valued: either permit or deny a request. However this is only part of the story, because policy rules come in (at least) three different forms

1. specific conditions that apply *directly* to business entities such as employee roles, resource types and user groups. These direct conditions define constraints on permitted combinations of Subject, Resource and Environmental attributes;

2. general conditions that apply *indirectly* to these business entities. They encode structural relationships between these entities, e.g., a `Person` can have 0, 1 or more `Role`s. While they are encoded as rules, they generally apply to Subjects, or to Resources, or to Environmental attributes, rather than to combinations of such entities;

3. general conditions that apply to higher-level abstractions and serve as security properties such as (static or dynamic) SoD or ethical wall policies.

Thus incoming access requests need to be matched against complex knowledge bases comprising the three rule types above. The PDP can match the request context directly against type 1. policies above. The structural relationships supporting Type 2. policies are often stored in external databases, so the PDP calls on the Policy Information Point (PIP) to provide this information, possibly resulting in other type 1 policy context matches. Of course, the higher level entities might also be subject to Type 3. policies, in which case even more policies may become relevant.

Good PDP *performance* is an important access control system requirement. Indeed, if policy evaluation performance is poor, the access control system intrudes more and more on legitimate user activity, so the user experience suffers. Anecdotally, each policy evaluation takes longer as policy sets grow larger; we see in Chapters 4– 6 how we can measure this effect.

Access control evaluation performance has been getting some interest at the OASIS XACML TC. In private correspondence with us, David Brossard (Axiomatics VP Product Management and XACML TC member) said:

> Now that XACML 3.0 has become a standard, we can focus on usability and adoption. XACML will only succeed with mass adoption. And mass adoption will only happen if we give end-users and developers the tools and techniques they want. [David Brossard (Axiomatics), 22 Mar 2013].
>
> My personal impression is that the key to XACML's future success is adoption. That will only happen through the adequate tooling and environments - *the output of your proposed research* is one such environment." [David Brossard (Axiomatics) 27 Mar 2013].

2.1.2.1 Extending access control: delegation and usage control

In the enterprise communication scenarios considered in this dissertation, access control is considered a once-off operation. That is, the requester asks permission to perform some action on a protected resource, and the responsibility of the access control system is to provide a correct and timely answer to that specific request. Subsequent *usage* of that resource is subject to any existing rules; the requester is trusted to operate responsibly within the limits of what access has been granted. However, there are two scenarios where this narrow interpretation of access control needs to be reviewed:

Delegation is the procedure in which access rights can, with suitable safeguards, be shared with other entities in response to specific circumstances, such as "break-glass" policies in emergencies. Often this is done by adding extra rules to the existing access control policies.

Usage control is where the data is subject to ongoing access checks as its context changes, e.g., as it passes from one entity to another. *Sticky policies* bind these access rules to the resource. Usage control can help to secure the *privacy* of data shared by users who wish to maintain control of what happens to their data (Kelbert and Pretschner, 2012).

Delegation is related to the administration of the access control system itself and so is largely outside the scope of this dissertation. In practice it can lead to more complex

policies. Rissanen and Lockhart (2014) provides advice for policy authors in this regard. Usage control is more relevant here, because it is associated with repeated evaluation of access control policies and hence more load on the access control system and on the PDP in particular.

2.1.3 Importance of XACML

The technical challenge of pervasive access control is being addressed by the growing adoption of externalised authorisation systems, in which access control rules are specified declaratively in an industry-standard language such as eXtensible Access Control Markup Language (XACML), and where a *reference monitor* checks every request for a protected resource.

Indeed, many enterprise-level access control systems encode access controls as XACML (Moses, 2005) hence researchers focus on XACML policies and requests and their use in Policy Execution Point (PEP) servers and XACML-based PDPs.

XACML is an industry-standard (OASIS) XML dialect specifying access control rules. The XACML standard also defines an architecture for access control enforcement. XACML policies are hierarchical: rules roll up into policies and thence into policy sets, which can roll up to higher-level policy sets. Listing 2.1 is an example XACML 2.0 policy showing how elements of the policy are nested inside each other. Such policies can be nested (to arbitrary depth) inside `PolicySet` elements, which can be nested inside other `PolicySet` elements. A full specification is available from the OASIS website (Moses, 2005).

One of the key features of the XACML architecture is that it externalises access control so that access policies are collected in one location (the Policy Retrieval Point (PRP)) and access control can be offered as a service to other applications. The architecture also provides functional separation between access control and other network uses, in terms of resources such as servers and bandwidth (OASIS XACML-TC, 2014). Whilst the flexibility XACML provides is highly valuable, the manner in which the architecture is implemented can be significant for performance.

Listing 2.1 Example policy: IIA001 from XACML 2.0 conformance test suite (Kuketayev, 2005).

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy
  xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "urn:oasis:names:tc:xacml:2.0:policy:schema:os access_control-xacml-2.0-policy-schema-os.xsd"
  PolicyId="urn:oasis:names:tc:xacml:2.0:conformance-test:IIA1:policy"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
  <Description>Policy for Conformance Test IIA001.</Description>
  <Target/>
  <Rule RuleId="urn:oasis:names:tc:xacml:2.0:conformance-test:IIA1:rule" Effect="Permit">
    <Description>Julius Hibbert can read or write Bart Simpson's medical record.</Description>
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                        Julius Hibbert</AttributeValue>
            <SubjectAttributeDesignator
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
              DataType="http://www.w3.org/2001/XMLSchema#string"/>
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <Resource>
          <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:anyURI-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI">
                        http://medico.com/record/patient/BartSimpson</AttributeValue>
            <ResourceAttributeDesignator
              AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
              DataType="http://www.w3.org/2001/XMLSchema#anyURI"/>
          </ResourceMatch>
        </Resource>
      </Resources>
      <Actions>
        <Action>
          <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                            read</AttributeValue>
            <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
              DataType="http://www.w3.org/2001/XMLSchema#string"/>
          </ActionMatch>
        </Action>
        <Action>
          <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                        write</AttributeValue>
            <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
                DataType="http://www.w3.org/2001/XMLSchema#string"/>
          </ActionMatch>
        </Action>
      </Actions>
    </Target>
  </Rule>
</Policy>
```

Two major research gaps exist:

1. XACML policies are difficult to specify correctly and it is even more difficult to ensure that they remain correct as security needs evolve;
2. The Access reference monitor function is a complex decision system, whose performance is not easily predicted because it depends on many factors. Other factors being equal, the number of rules is a function of the number of elements in the access control matrix (and hence its complexity scales as $\#(\mathtt{S}ubject) \times \#(\mathtt{R}esource) \times \#(\mathtt{A}ction)$). Therefore explicit access control matrices become unwieldy as the domain size increases and so a more scalable specification of access control requirements is as a set of rules rather than a large number of Permit/Deny-valued cells in an enormous access control matrix. However, the rule set could still be very complex and require relatively long execution times.

The focus of this dissertation is on the latter problem, namely that of predicting the performance of the system, dimensioning it accordingly, and managing it so that it continues to meet its performance objectives.

2.1.4 Access Control Performance

In large organizations, enterprise communication and content management systems provide the environment in which policy based access control systems mediate access to resources. Note that safety requirements are addressed directly when:

- ensuring that all likely access requests are covered by the available policies;
- ensuring that all access requests are intercepted by the policy system so that no resource is released for use unless access has been granted.

By contrast, performance considerations require careful analysis of the full policy-based access system. Performance problems could arise anywhere, from policy formulation to server configuration. Hence a *comprehensive* domain model is needed in which to evaluate different policy-based access control proposals before deployment.

In the standard architecture (Moses, 2005), access requests are sent to Policy Execution Point (PEP)s, which hand off the access decision itself to one or more PDPs. The PEP is largely stateless and so scales outwards easily. However, the PDP

needs to consult a policy set for each request and hence can become a performance bottleneck. Thus PDP performance is an important characteristic of access control requirements in deployed ICT systems. When addressing PDP and hence system performance, it is timely to consider what that means in respect of a request-response system such as an access control reference monitor.

The PDP service time is defined as the time taken from the point that the access request arrived at the PDP to the time when the response was issued by the same PDP. As seen by the access-requesting client, it is the *latency* added by the PDP to the time taken from requesting access to gaining that access (assuming the response was `Permit`).

While latency is the main concern of an individual client, the average number of requests handled per second has similar importance for overall service performance. Generally, by reducing service time (which is equivalent to latency for infrequent arrivals), the (potential) mean service rate is increased. However, the (supply-side) service rate should always be compared against the (demand-side) arrival rate: if the latter exceeds the former, even for a short time, queues will develop. If the queue length is managed carefully, the impact on clients can be minimised. In this dissertation, most of the focus is on measuring the service time, though queueing is considered in the simulations in Chapter 3.

2.1.4.1 Access Control Scalability

Scalability is a measure of the ability of a system to continue to respond well as the system load increases and the available resources increase in proportion with that load. Some of our experiments consider the effect on service time as domain size increases, for different resource (memory and CPU) allocations. If the service times grow at a faster rate than most measures of domain size, the PDP in question does not scale well in respect of performance. However, scalability is a derived quantity (it is not measured directly) and so specific performance experiments are needed to answer specific scalability research questions.

Ensuring the scalability of individual components within that architecture may not be enough. For example, it is relatively easy to distribute (in the sense of scaling out) the architecture's PEP function, but much more challenging to distribute the PDP engine. When minimising the total latency, some tradeoffs may be needed (e.g., adding more

resources for Component X means that Component Y needs more resources to benefit), or there might be a net benefit from adding extra components (such as caching) to the existing mix.

Scalability of the policies themselves can also be a challenge, if the policy set grows too quickly as the domain size increases. When formulating the policies, one problem is that policy authors prefer to define policies at a higher level of abstraction than the request handling system. For example, a policy might reference *groups* of subjects, but the corresponding requests generally reference *individuals*. The resulting "impedance mismatch" needs to be considered when generating policies and requests that together represent a domain. A mechanism for doing this constitutes one of the main contributions of this dissertation.

2.1.4.2 Caching

One of the standard techniques used to improve the performance of stateful request-response system is to *cache* "hot" data near where it is needed. If clients can get the response they need from the cache, this can reduce the load on the policy server. However, the policy cache proxy itself can become swamped with lookup queries, in which case it makes sense to create multiple caches. This results in two problems that need to be addressed

addressing to which cache should the incoming query be directed?

consistency how do the caches remain synchronised with the master copy of the policies, when that master copy is subject to updates in the policy server?

In such a scenario, *consistent hashing* can be used to provide a scalable means of implementing a distributed policy cache. If the policy cache is distributed in this way, there is no need for individual caches to communicate all the time as they apply cache coherency protocols, and more policy caches can be added on the fly without invalidating existing policy caches.

Use of the consistent caching algorithm enables high scalability of busy internet sites but is challenging to implement with access control policies implemented in XACML. The main reason is that administrators need to update policies dynamically, but it is far from trivial to understand how to update the policy caches. Indeed, XACML is verbose (difficult for humans to interpret any but small policy sets) and has complex

rule combining algorithms (so tools struggle to manage that complexity) and so it lacks support for change impact analysis.

This dissertation does not address caching any further, although some of the analysis could be used to help identify the "hot data" that might be suitable for caching.

2.1.5 Use of a testbed

As can be seen in §2.2.3, many researchers have suggested ways in which access control performance (in the sense of service time per request) can be improved. Generally, the evidence presented by those researchers is based on comparisons with the SunXACML PDP reference implementation (Proctor, 2004), often using unpublished policies and requests. Therefore, while there is a conformance test suite to check the *correctness* of a given PDP, there is no common test suite of policies and requests for performance evaluation. Since the experimental conditions differ, and are generally not reproducible by other researchers, it is difficult to compare improvements with each other, or to determine what tradeoffs occur either in respect of functional requirements (like policy expressiveness) or performance.

2.1.6 Links with policy testing

One of the considerations in correctness testing is to define the *minimal covering set* of requests, as this set is designed to exercise the policies in the PDP thoroughly. For performance evaluation this minimal set of requests should be augmented with additional requests to perform a more thorough sampling of the rule space, thereby reducing the risk of missing unexpectedly large policy evaluation times. Augmenting the minimal set also reduces the stochastic risk associated with the fact that PDP performance also depends on stochastic parameters such as the distribution of policy request arrivals. The goal is to achieve both the sampling and the volume requirements of the PDP performance test. Static analysis of the policy set will indicate the longest path (by duration) between the root node and all terminal nodes. However, this is insufficient, because actual performance will depend both on the average path length for a typical policy request mix and on stochastic quantities such as the request arrival rate.

2.1.7 Links with policy authoring

Anecdotally, each policy evaluation takes longer as policy sets grow larger. Therefore, many researchers recommend that policies should be as efficient as possible. One way of achieving this is to search for (and remove) any redundant policies.

Because it is "easy" to add policies to an existing set, and many Policy Administration Point (PAP)s do not offer any means of analysing the effects of adding such policies, it is also easy to introduce policy conflicts inadvertently. XACML *expects* multiple, possibly conflicting, policies to become active at once for a given policy evaluation, and relies on policy- and/or rule-combining algorithms to resolve any discrepancies. These combining algorithms are generally simple aggregations of rules without richer semantics such as prioritisation. As an example, assume a policy has four rules $\{R_1, R_2, R_3, R_4\}$, which evaluate to $\{\texttt{Permit}, \texttt{Deny}, \texttt{Permit}, \texttt{Deny}\}$, respectively. Then the 3 most common XACML 2.0 rule combining algorithms $\{\texttt{*-first-applicable}, \texttt{*-permit-overrides}, \texttt{*-deny-overrides}\}$ evaluate to $\{\texttt{Permit}, \texttt{Permit}, \texttt{Deny}\}$, respectively.

Although these rule-combining algorithms (and their policy- and PolicySet-combining counterparts) are designed to ensure that a single Decision (Permit or Deny) is given for every request, they can hide problems arising from inconsistent semantics. For example, consider the following example of policy conflict:

Rule 1 Person A cannot read Document X

Rule 2a Role B can edit Document X

Rule 2b Person A has role B

Rule 2c Read action is a subset of the Edit action group

Note that the inference of Rules 2a, 2b and 2c is that Person A *can* read Document X, which is in direct conflict with Rule 1. Often this will result in an `Indeterminate` decision, but, particularly in XACML 3.0, rules elsewhere in the policy set might be sufficient to resolve the conflict (Brossard, 2014b). Given the hierarchical rules in XACML, combining algorithms are essential but it could also be argued that such algorithms *hide* policy conflicts and also make the access control decision itself less easily traceable to the relevant policy or policies.

From a performance perspective, a policy conflict might be expected to increase the service time with no obvious benefit to the policy semantics that were intended by the policy author.

Perhaps the best way to proceed is either

- to use policy representations that can identify conflicts easily, using logical operations, or
- to query the underlying policy representation and look for them explicitly.

This dissertation does not consider policy conflicts as such, although it would be straightforward to generate policies with and without known policy conflicts and to compare their service times, thereby predicting the effects of those conflicting policies on measured service times.

2.1.8 Learning from recent web service performance advice

It is also instructive to consider access control systems as part of the larger class of web services. Indeed, most ICT resources requiring protection are web services and need to manage their own performance carefully. Thus it is instructive to see what web service standards and practices might offer to access control deployments. In particular, cloud computing generates new and highly challenging requirements for scalability and elasticity. Non-blocking I/O, where interrupt handling is achieved through the extensive use of callbacks, has re-emerged as a way of maximising web server utilisation. The need for blocking is removed by passing a callback parameter that is invoked on completion of the deferred task. Javascript (Crockford, 2011) was designed as a language to run client-side in the browser and hence was designed to use asynchronous I/O via callbacks as one of its programming idioms. It is being re-evaluated as a more general-purpose language with the community promoting Node.js (Dahl, 2011) as a container for server-side Javascript applications. Many software engineers claim the result is better scalability and more efficient use of resources (Lerner, 2011; Tilkov and Vinoski, 2010). More generally, software architects are exploring the use of the non-blocking I/O approach to re-appraise well-established software engineering best practices based on threads.

Outside of the Node.js community, similar patterns and frameworks are being used in the JVM (notably `Vert.x`[4] and the `Play Framework`[5] (which both take advantage of the `Actor` model (Agha and Kim, 1999; Haller and Odersky, 2009)).

Another area where work is under way, this time by the XACML community itself, is in the area of request encoding. Traditionally, XACML has been expressed in XML format, but there is growing interest in the use of JavaScript Object Notation (JSON) as a means of encoding access requests. This is because many client applications are moving away from SOAP/XML to lighter weight REST/JSON (Crockford, 2006) protocols. If compliant XACML PDPs are unable to accept JSON-encoded requests and can emit only XML-encoded responses, this would be another area where latency is added. Consequently, the OASIS XACML TC has published the REST/JSON profile (Brossard, 2014a) of XACML as an addition to the XACML 3.0 standard.

2.1.9 Performance analysis of database operations

Database performance analysis is of ongoing concern. Data volumes are increasing rapidly and scaling upwards is not able to keep pace, resulting in the phenomenon of "Big Data" and growing interest in non-traditional techniques for managing data at scale. In some scenarios, such as when database management occurs on virtualised infrastructures in the cloud, this leads to new optimisation objectives (Florescu and Kossmann, 2009). The recent *Beckman report* (Abadi et al., 2016) identified "Scalable big/fast data infrastructures" as the first (of five) grand challenges of database research at this time. In justifying its conclusion, the authors indicate some of the complexity arising from many, possibly interacting, factors, all of which makes performance prediction, hence management, more difficult. It should be noted that some of these factors, notably "query processing and optimisation" are shared with access control policy evaluation.

2.2 Literature Review

The background to the research topic has many facets, as indicated in § 2.1. The facets include enterprise communications management, security modelling, policy

[4] http://vertx.io/
[5] https://www.playframework.com/

management and performance modelling. Each has a long established and active research community, publishing papers that are relevant to our research topic, which draws upon all of these facets. Therefore we need to survey the literature in disparate areas, and we consider the most relevant of these below.

2.2.1 Policy metamodels

The two main policy metamodels in use in enterprises are Role-Based Access Control (RBAC) and ABAC. The former was a great advance over older models which failed to take account of the flexibility of staff in modern enterprises, who can often change role according to context; see §2.2.1.1. The latter generalises RBAC by allowing additional attributes (not just `Subject` role) to be used when specifying policies, see §2.2.1.2. Generally, it is possible to represent RBAC policies in ABAC form but not vice-versa. XACML was designed to represent ABAC policies so, in principle, it should be able to represent any enterprise access control policy. The converse is that any domain model for such policies should follow the "spirit" of XACML (as in its metamodel) to ensure that they are sufficiently representative of the domain. As seen in Chapter 4, the XACML metamodel provides the scaffolding of the domain model presented in this dissertation.

2.2.1.1 RBAC

Ferraiolo and Kuhn (1992) is the seminal paper on role-based access controls. The authors propose that RBAC is a good fit for non-military organisations because the three basic rules of the RBAC model are sufficiently general to be applied to many practical scenarios. All that is needed is to configure what is meant by *Subject*, *Role* and *Transaction* in a given context. Thus objectives requiring the *Principle of Least Privilege* and/or BoD can be recast as RBAC policies in a suitable language. Such protection objectives are also within the scope of the domain model presented here. The model grew in importance in the early 2000s, largely due to the influence of (Ferraiolo and Kuhn, 1992). In the context of a corporate intranet, Ferraiolo et al. (1999) was particularly influential.

Rizvi et al. (2004) is one of the first papers to use the term *fine grained access control* (in relation to database queries). However, RBAC was not really suited to this

concept, because of its focus on "coarser" entities such as roles. Practical problems arise when the number of roles and similar entities become very large.

2.2.1.2 ABAC

Indeed, the need for fine-grained access control was one of the primary motivations for the ABAC model. The ABAC model was proposed as a generalisation of the the RBAC (Ferraiolo and Kuhn, 1992) model. ABAC models can represent many of the dominant models (including DAC, MAC and RBAC) (Jin et al., 2012). ABAC provides the native access control model used in XACML. The key idea is that policies can be tuned carefully (by adding and removing selected attributes) to meet a specific access control objective. For example, a *role* is just a convenient *persistent grouping* of attribute values that may be shared by more than one Subject in a given context. In many organisations, attributes are maintained as hierarchies in key-value stores (such as LDAP directories) or in relational databases (such as those used in ERP systems). The policies themselves are sufficiently complex (e.g., the schema is dynamic (Russello et al., 2008) and the relationships between entities might even be recursive (in the form of *part-whole* hierarchies)) so they do not fit easily into a relational representation.

Even with the RBAC profile, XACML 2.0 does not support the RBAC model with constraints that would be common in a corporate setting such as SoD, so Ferrini and Bertino (2009) introduced the XACML+OWL framework where XACML policies manage authorisation at a role level, but the role hierarchies and the constraints affecting individuals in roles are managed and decided using an ontology (specified in OWL). This paper shows that, in a practical implementation, it is necessary to distinguish between relatively static information (such as the assignment of persons to roles), policies authorising those roles (commonly specified in XACML format) and security properties such as the *principle of least privilege*, which are overarching concerns that are not easy to specify as static rules (such as might be found in a XACML policy set). Some of these ideas, around having separate but consistent models of the static and policy model, form the foundation of the domain model presented in Chapter 4

2.2.2 Formal policy languages

Zhang et al. (2004) is one of the earliest efforts to write *verified* policies in a way that can be translated to XACML and hence incorporated in industry-standard access control systems. The authors introduce a domain specific language (DSL) they call *RW* which offers a terse encoding of access control statements. The compiler and its underlying database require significant configuration and so their system is not well-suited to developing large policy sets, but the generated policies have the benefit of sound logical foundations. The other major contribution is the idea that policies can be written in one language and then compiled to XACML for use in standard PDPs.

Dougherty et al. (2006) uses another language to represent polices: Datalog. Unlike the DSLs proposed by many other authors, Datalog is a general-purpose declarative language, with similar syntax to Prolog. As with RW that was proposed by (Zhang et al., 2004), it can be used for logic programming and so supports reasoning about policies. Thus it is more suited to writing policies with large semantic complexity rather than large size.

Ramli et al. (2014) derive the formal syntax of XACML 3.0, stripped down to its essentials (excluding the XML details) and uses logical reasoning to derive its formal semantics, particularly its combining algorithms. With this formal representation in place, Ramli et al. (2013) show how Answer Set Programming can be used to look for gaps in the policy coverage and to check whether particular security properties hold. As with (Dougherty et al., 2006), logic programming provides a means of checking the semantics of a policy set. Ramli et al. (2014) claim that their treatment of XACML 3.0 is almost complete. Therefore, given suitable tooling to convert a policy set to their representation, it should be possible to handle most policies that might be encountered in practice, although they do not say much about the time needed to conduct such an analysis.

Other researchers are concerned with "quality assurance" of policies, namely whether and how well a given policy set meets the *protection* objectives of an organisation. Sometimes this can be achieved by formal verification techniques, e.g., if the policies have been specified in (or translated to) representations that are based on Description Logic (DL). Thus properties such as *safety* (equivalently: the non-leakage of privileges) can be recast as constraint satisfaction problems and checked using DL reasoners such as Pellet (Kolovski et al., 2007; Sirin et al., 2007). The benefit for performance is that

reasoners are designed to search a large rulebase efficiently. If the reasoner's algorithm or its implementation is updated in a way that improves its performance, that improvement is available for policy evaluation too, without the need to address XACML policy evaluation explicitly. Kolovski et al. (2007) also point out that their DL representation fits well with OWL-DL ontologies describing the policy domain. In principle, the domain model could have a common OWL-DL representation for the static domain data *and* the constraints that apply to sharing in that scenario. As will be seen in Chapter 4, it is essential to ensure that the static domain data is consistent with the constraint model.

Fisler et al. (2005) chose to represent access control policies as propositional logic statements. The authors model such policy statements as Multi-Terminal Binary Decision Diagram (MTBDD)s, which encode the rules in a notationally efficient and scalable fashion as weighted directed graphs. This paper is interesting because it links the logical treatment of policies (with scope for applying logical operations such as inferencing) with Binary Decision Diagram (BDD)s (which makes explicit the most efficient graph-based representation of the policy set).

Policy similarity measures (Lin et al., 2007) can form the basis of techniques to quantify policy complexity. If there are many similar policies (with overlapping rule effects), the same decision can be made by following different routes to a decision node. Thus, in the case of rule combining algorithms such as *first-applicable* or even *permit-overrides* (in the latter case when the similar rules yield a *Permit* decision), the size (measured as the number of rules) of the policy set might overestimate the policy set complexity if there is a large number of similar policies. A good estimate of the *effective* policy set size helps to estimate the policy evaluation service time, so "policy set size" is an important parameter.

In summary, treating XACML policies as logic programs has much potential. Chapter 4 describes ways of generating policies taking their semantics as given, but it would be interesting to investigate whether greater knowledge of the policy semantics, and how they might change as policies are added or as the domain size increases, could be used to gain insight into access control performance.

2.2.3 Proposals to improve access control performance

The problem of access control evaluation performance has received attention in the past ten years or so; the level of interest has increased steadily over that time. XACML is the industry-leading access control architecture and XACML-based PDPs are common in enterprise communication environments sold by the major vendors. One option is to look at alternatives to XACML, mostly originating from academia rather than industry. Proposed languages include PERMIS (Chadwick et al., 2008) and SecureUML (Lodderstedt et al., 2002). However systems using these languages do not appear to have any specific performance advantages over XACML and their adoption is low. Therefore, because of its ubiquity, many researchers have chosen to address XACML PDP performance in particular.

Liu et al. (2008) was seminal in that it showed that it was possible to evaluate policies with service times that were typically hundreds of times less than those of the reference SunXACML PDP (Proctor, 2004) implementation. The authors describe Xengine PDP that

1. converts the XACML textual policy to numerical form;
2. standardises the numericalised policy by flattening its structure;
3. of the seven standard combining rules, uses only the *first-applicable* combining rule;
4. encodes the standardised policies in data structures that are optimised for search operations.

Liu et al. (2008) used some "real-life" policy sets to which they had access, as well as some generated policies, together with randomly-generated requests using the correctness testing request generator described in (Martin et al., 2006). This mix of policies and requests was used when checking that policy evaluation performance improved compared to the SunXACML PDP. While the performance results are impressive, it is not clear that the results will apply to actual deployments, because the characteristics of the *actual* policies, requests and infrastructure used are also likely to affect the performance, and not just the choice of PDP. Liu et al. (2008) also acknowledge that removing inconsistent and redundant rules, as described by (Kolovski et al., 2007), might improve performance but note that this was not proven at the time of writing their paper. Also, while the authors claim (and demonstrate)

dramatic performance increases, this is at the cost of an opaque, non-symbolic policy representation.

Various researchers made specific policy evaluation improvement proposals by trying to optimise the policies themselves, independently of the PDP, so that there is a performance uplift even for the classic SunXACML PDP. These proposals include:

- deriving *policy similarity measures*, with the potential to short-circuit policy evaluation (Lin et al., 2007). Policies that are similar do not need to be checked if a similar policy has already been checked, so this technique would help to evaluate policies that have large size but moderate complexity (owing to subsets of policies whose member policies are similar to each other);
- similarly, *policy integration*, in the context of business federations is considered in (Mazzoleni et al., 2006)—of course integration could also be seen as a means of simplifying a policy set and hence improving its evaluation performance. The key idea is that, by grouping similar policies together, particularly when the similarity derives from structural properties of the domain, it is possible to reduce the policies to evaluate;
- *policy reconfiguration* (Miseldine, 2008), that is changing the structure of a policy set, by identifying the evaluation graph for a given request and estimating the evaluation cost for each node in the graph, together with its dependencies. Depending on the combining algorithms in force, and the partial results of the evaluation graph, it might be possible to ignore parts of that graph whose results do not affect the overall result. Generally, an entity (such as a rule or a policy) is evaluated in the order given by its position in its enclosing entity (such as a policy or policy set, respectively). If the partial evaluation costs are known, it might be beneficial to change that evaluation order (so Rule 1 is evaluated before Rule 2, say) such that the policy semantics do not change, but the evaluation path has fewer unnecessary evaluations.
- *policy reordering* (Marouf et al., 2011), exploiting the fact that many combining rules of the `*-overrides` form do not require full evaluation of all rules—so, if a counter-example is found, evaluation of that branch of the policy tree can stop. The problem is that the rules might not be ordered in a way that allows such short cuts. Thus Marouf et al. (2011) estimates the statistical distribution of the requests, clusters the policy rules and reorders the rules based on the estimated

Table 2.1 Summary of techniques for improving XACML evaluation performance

Authors	Change policies	Change PDP	Limitations
Marouf et al. (2011)	categorisation, reordering, clustering	No	None
Liu et al. (2008)	Numericalisation; Tree structures	Extensive	First-Applicable policy combining rule only
Miseldine (2008)	reconfiguration (limited scope)	No	None
Mazzoleni et al. (2006)	policy integration (flattening the tree)	No	None
Ngo et al. (2013); Pina Ros et al. (2012)	graph/decision diagrams	Yes	Depends on implementation
Griffin et al. (2012)	Node.js PDP	Yes	Partial implementation
Kolovski et al. (2007)	Translate into DL (Pellet)	Use DL Reasoner	Excludes Ordered-* combination rules

distribution of the requests so that policy evaluation time *for a specific request distribution* is minimised.

All of these techniques have the advantage, compared to (Liu et al., 2008), that they could conceivably be applied to any XACML PDP, including SunXACML PDP.

Pina Ros et al. (2012) recast XACML policies as BDD. This format is the most efficient graph-based representation of Boolean functions (such as policy sets) by design. As a combinatorial optimisation problem (to find the optimum ordering of the variables in the Boolean expression), deriving the BDD from the input policy specification is NP-hard, but once the policies have been transformed into a BDD, evaluation should be fast. Compared to (Liu et al., 2008), their implementation does not numericalise the policies and keeps more of the expressiveness of XACML. Thus

comparison functions do not need to be equalities, more combining algorithms are supported, etc., but their implementation does not support multi-valued attributes.

Ngo et al. (2013) describe a PDP that overcomes most of the restrictions of Xengine PDP while keeping many of its performance advantages. In particular, they show that the policies that are not convertible to Xengine PDP's restricted format can be handled by their PDP prototype, and yet benefit from the MTBDD formulation of Xengine PDP. They also extend the coverage of Pina Ros et al. (2012): they handle all the XACML 3.0 `Indeterminate` rule combinations, as well as multi-valued attributes. The theoretical justification is strong and the performance improvement, relative to SunXACML PDP appears significant, though not as good as Xengine PDP. However, it is not possible to be more precise, because the performance experiments suffer from the same problems as those of Xengine PDP and the various "policy modification" algorithms in that, while it is entirely plausible that significant performance improvements occur, there is no common experimental setup with which to compare them against other techniques or indeed to estimate how much performance will improve in a given policy deployment.

Other proposals include re-engineering the PDP and using lightweight data formats like JSON that are commonly utilised in highly scalable web systems. In that regard, Griffin et al. (2012) describes such an implementation.

Table 2.1 summarises the policy evaluation improvement proposals described above.

2.2.4 Policy authoring

Policy authoring is a well-studied topic, with many researchers focusing on the difficulty of authoring policies that are semantically consistent with higher-level business objectives (Davy et al., 2007) and with each other (Davy et al., 2008; Jajodia et al., 1997; NIST/NSA, 2010).

There is general agreement that writing semantically correct, maintainable XACML policies is difficult, particularly in statutory domains. Abou-Tair et al. (2007) describe a framework in which German privacy laws are encoded in an ontology from which XACML policies can be generated. Healthcare record management is another domain where such approaches show promise (Rahmouni et al., 2009). Such law-based ontologies provide sound foundations, but might not generalise to other domains.

A less formal (and perhaps more common) starting point would be business process specifications such as Business Process Execution Language (BPEL) and Business Process Model and Notation (BPMN) (Wolter et al., 2007). This is attractive because such specifications decompose security properties such as SoD and BoD into elements that have the required granularity to be translated directly to XACML.

Using either an ontology or a set of business specifications as a starting point for policy authoring seems entirely reasonable, and is compatible with the policy authoring procedures of Chapter 4.

2.2.5 Policy refinement

Stepien et al. (2011) describe ABAC as a generalisation of RBAC, solving some of the problems encountered with RBAC. For example, ABAC is as suited to Deny rules as it is to Permit rules. It supports fine-grained policies without the "role-explosion" that occurs in RBAC when the roles themselves are made too fine-grained, resulting in inefficient policy specification. They also note the difficulty of writing rules with complex constraints such as SoD directly as XACML, owing to XACML's verbosity and its non-local nature (where related rules can be "scattered" through the policy hierarchy). The intention of the policy can often be lost in all the detail, and the large number of ways in which to write policies to achieve the same objective can quickly result in unmanageable and probably sub-optimal (in performance terms) policy sets. Therefore they propose a non-technical notation that can be compiled to XACML or even to Prolog for evaluation, while retaining its conceptual simplicity for the benefit of less technical policy authors. With regard to policy evaluation performance, they show that policy authors using their notation sometimes use a more efficient formulation (fewer, more complex, but also more cohesive rules) than would be the case if the rules had been written directly in XACML format. They claim this is a result of reducing the "noise" in the language, and curbing some of its excessive flexibility. Indeed such cohesive policies should also be more manageable and more suited to caching.

Craven et al. (2011) describe how policy refinement principles can be used to ease the task of authoring policies in rapidly changing domains. This is because the policies are written at a more abstract level. As part of the evaluation process, they use logic programming to refine the policies, getting the missing details from an UML model.

To some extent, this procedure happens in the XACML architecture when a PIP is consulted at policy evaluation time. However, the more formal formulation in Craven et al. (2011) helps because policy refinement has been well studied in the wider policy community (not just relating to access control) and there is much advice in the broader literature that might be relevant to the access control domain. Also, with a careful decomposition/refactoring of the policies, it is possible to use their approach in the case where multiple policy suites are required.

If a full model is not available, Rochaeli (2009) describes how it may be possible to write the policies at a higher level of abstraction and then to use DL-based refinement techniques to derive the missing specific rules (Rochaeli, 2009).

2.2.6 Policy integration and decomposition

One of the drawbacks of the ABAC metamodel is that its flexibility can also be daunting, in that it is up to the policy author to design the structure of the policies. Indeed, the OASIS XACML TC have recognised this, and have published non-normative *profiles* (analogous to the *design patterns* used in object-oriented analysis and design) for typical use cases. Examples include the RBAC (Anderson, 2005) and hierarchical resources (Rissanen et al., 2010) profiles for XACML. Another form of structuring is concerned with creating policies by combining smaller policy sets such as when two groups in an enterprise wish to federate their services and the access control system needs to be updated accordingly.

Mazzoleni et al. (2006) introduce the problem of *policy integration*, in which separate "organisations" (possibly within the same overall enterprise) need to combine separately-administered policy sets. In particular, they introduce algorithms for combining rules based on their similarity, particularly whether rules *converge*, *diverge*, *restrict and extend* or *shuffle*.

Rao et al. (2009) look at a similar problem of "policy composition", providing an algebra with which to combine policies represented by MTBDDs.

Decat et al. (2012) describe the inverse problem of how to decompose policies, particularly where business federations and/or cloud computing are involved, so that they can be distributed effectively. With a suitable decomposition strategy, it should be possible to use distributed computing techniques to improve policy evaluation

performance. Clearly, the policy decomposition should be semantically equivalent to the original policies. Decomposition brings potential advantages in respect of both managing and evaluating the policies. In a cloud computing scenario, since policies have their own context, a particular set of rules should be managed and evaluated by the tenant and a different set of rules is the responsibility of the cloud service provider. Regarding performance optimisation, it is necessary to estimate the policy evaluation cost of the sub-policies and this is where performance experiments should help when decomposing a large policy set into smaller sets to be distributed across multiple PDPs for better performance.

El Kateb et al. (2012) and the related `PolicySplitter` tool (Mouelhi, 2015) describe seven different policy splitting (refactoring) criteria. Policy splitting is related to policy decomposition in that they both subdivide a policy set into smaller parts that are more suited to separate evaluation, but they differ in terms of the motivation and the algorithms used. In the case of policy decomposition, the focus is often on decomposing sets of policies that were combined for external reasons, e.g., the amalgamation of business units. In the case of policy splitting, the goal is to use "algebraic" operations to refactor the policies. In each case, the goal is divide-and-conquer: to break a large policy set into smaller parts that are better suited to distributed policy evaluation. The authors show the benefits that accrue from policy refactoring. Both SunXACML PDP and Xengine PDPs can exhibit substantial performance improvements when they, and the policies they use, are distributed. Interestingly, the choice of policy splitting criterion should be matched carefully to the characteristics of the policy set, of which three were available to the authors: Library Management, Virtual Meeting and Auction Sales Management. The number of rules per policy domain is fixed, but the nature of the rules differ, which is why certain splitting criteria are more suited to a given domain than others. Their analysis is static, so perhaps careful performance analysis (such as that presented in this dissertation) would help to make the best choice of policy splitting criterion.

Deng et al. (2014) provide a detailed set of algorithms for distributing policies with the objective of maximising policy evaluation performance. They decompose the policies based on structural aspects, such as their Subjects, Resources and Actions, rather than focusing on the rules themselves. However, the main contribution is that they also consider requests when distributing the policies: the Policy Enforcement Point (PEP) hands over requests to a request distribution module which attempts to

match the request to the most suitable PDP, given characteristics of the request. They propose a greedy algorithm to distribute the policies in an "optimal" way, again based on the time costs of the sub-policy evaluation times. Their algorithm attempts to estimate those time costs and their evaluation shows that a distributed SunXACML PDP deployment using their policy decomposition and request distribution algorithm outperforms an equivalent naïve distributed SunXACML PDP deployment. However, a possible weakness in their approach is the use of heuristics for estimating the time cost; perhaps measurement would help in this regard. Also, while the evaluation shows the (encouraging) effect of adding PDP threads, they are limited in their ability to consider other factors such as policy set size and complexity, since they are limited to three unrelated policy sets. Nevertheless, by placing distributed policy evaluation in an optimisation setting, it is a very important contribution.

This dissertation does not consider either policy integration or policy decomposition/splitting any further, but such techniques are consistent with what is presented later, and could be the source of new performance experiments. We consider this later in Chapter 7.

2.2.7 Use of testbeds

Other authors have performed experiments comparing PDPs. Turkmen and Crispo (2008) compare 3 open source PDPs: SunXACML PDP (Proctor, 2004), EnterpriseXACML PDP (Wang, 2010) and XACML Light (Gryb, 2008) using generated data, focusing on the client-side concern of access requests processed per unit time.

Kohler and Brucker (2010) created a flexible proxy between the PEP and PDP to compare different caching strategies over a single SAP R/3 implementation, using data derived from scenarios (business processes) supplied by SAP as templates. This is a testbed to evaluate caching strategies in particular, but its domain is limited to SAP systems.

Our STACS testbed (Butler et al., 2010, 2011) is, to the best of our knowledge, still the state of the art in access control performance testbeds.

2.2.8 Policy testing

The problem of generating a large and representative set of policy requests for performance evaluation is related to that of generating a test set that covers as many of the policy conditions as possible. By ensuring full coverage, *all* policy conditions are checked and so there is a path to each terminal node in the decision tree inferred from the policy set (Martin, 2006). Martin (2006) also describes how Margrave (Fisler et al., 2005) can be used to determine redundant rules in a complex policy set, which can safely be removed for the purposes of correctness testing.

Martin et al. (2006) describe how policy mutation testing (Geist et al., 1992) may be used to determine how well a given test set of XACML requests discovers faults (deliberately injected as *mutations*) in policy sets. The goal is to estimate the effective coverage of the test set of XACML requests. Typical mutations are CRE (Change Rule Effect), in which the result of a rule is inverted, e.g., a `Permit` is changed to a `Deny` and vice-versa. Each mutation is generated and the test set of XACML requests is applied to the mutated policies. If at least one of the XACML requests results in a different response compared to the response obtained from original (unmutated) policies, that mutation is said to be *killed* and the associated XACML request is marked as essential for inclusion in the test set. Mutation testing can be used to measure the quality of a test set (of XACML requests in this instance) and hence to compare two test sets. (Strong) mutation testing may also be used to determine which XACML requests are essential to achieve each of the required policy, rule and condition coverage metrics.

Ammann et al. (2003) describe a theoretical model for coverage criteria for logical expansions. The authors distinguish between Clause Coverage (CC) and Predicate Coverage (PC) where a predicate comprises one or more logical clauses). Interestingly, neither coverage criterion subsumes the other, i.e., $PC \not\subseteq CC$ and $CC \not\subseteq PC$. Combinatorial Coverage (CoC) is guaranteed to subsume all other forms of logical coverage, but is unnecessarily onerous, since it has 2^n tests where n is the number of predicates in the policy set. To that extent, Ammann et al. (2003) introduce Determination (of predicates by clauses) and hence Active Clause Coverage (ACC) and the concept of *active* and *inactive* clauses. Ammann et al. (2003) presents the subsumption relations between the various flavours of ACC (and its inverse, Inactive

Clause Coverage). They recommend using the Akers derivative (Akers, 1959) to make a clause determine a predicate.

We note that, while correctness and performance testing share some features, they also differ in relation to the weight given to finding a minimal testing set to find errors, versus comparing different mixes of requests to look for performance differences. Therefore the approach taken in this dissertation *does not* use mutation testing or equivalent, and derives its own policies and requests instead.

2.2.9 Generating policies

If the ultimate output is intended to be XACML policies, and the focus is less on verification than on creation of policies, it could be argued that it is sufficient to use a XACML policy editor directly. However this is problematic because it is widely accepted (see for example, (Lang et al., 2008)) that manual editing of XACML policies is tedious and error-prone. Lang et al. (2008) describe a policy editor that is based on what they call "policy views", which derive from a classification of XACML policy elements into either attribute definitions or policy definitions (which in turn comprise combinations of restrictions on Subjects, Resources, Actions and Environment). Lang et al. (2008) describe a prototype that inserts such elements into a XACML policy template. This idea is developed further in Chapter 4 of this dissertation.

Axiomatics (a leading vendor of ABAC systems) offers *Axiomatics Language For Authorization (ALFA)*, which is a Domain-Specific Language (DSL) for ABAC policies that is easier to write than XACML because ALFA omits many of the less important technical details. Axiomatics provide tools to support ALFA in the form of a syntax-aware editor based on `eclipse` that also "compiles" the ALFA source instances to standards-compliant XACML policy instances. ALFA encapsulates many of the "policy views" ideas of (Lang et al., 2008) and, by making the policy representation more explicit, it might also be used for other purposes, such as communicating policies to other stakeholders.

Coincidentally, and independently of ALFA developments in Axiomatics, we contributed the `BPOL` and `SPOL` languages to (Davy et al., 2013), which showed how policy refinement techniques, combined with the `eclipse` language modelling tools (notably, the Eclipse Modelling Framework (`EMF`) plugin), could be used to write

policies for federated access control. SPOL is syntactically similar to ALFA but its policy refinement origins mean that, as an intermediate language between BPOL and XACML, the language refinement transformations are explicit. While these languages and the associated tooling help the security administrator to write a *single* suite of policies, they are not suited to writing *multiple* suites of policies, as needed in performance experiments. However many aspects of their design have motivated the design choices described in Chapter 4.

As we recall from §2.2.5, Stepien et al. (2011) presented a "non-technical" notation for access control policies. Since their notation can be transformed to XACML, it could be used in principle to generate policies in bulk, although that is not its intention.

As an alternative to these approaches which are based on the domain semantics, it is possible to build policies from their structural elements. Thus it is possible to create very large sets of *synthetic* policies with minimal semantic considerations. In that regard, Turkmen and Crispo (2008) compares the *performance* of three PDP implementations, using synthetic policies with differing structures and difficulty rather than domain semantics.

Summarising, there are good approaches for writing policies with specific semantics (Davy et al., 2013; Lang et al., 2008; Stepien et al., 2011) and ALFA, or even for bulk generation of policies based on technical criteria (rather than domain semantics) (Turkmen and Crispo, 2008), but there is a research gap relating to the bulk generation of policies based on domain semantics. This research gap is addressed in Chapter 4.

2.2.10 Generating requests

To date, the focus in the literature has been on means of generating sets of requests to send to a PDP in order to exercise the PDP's associated policy set in a manner that ensures coverage of all possible policy conditions. Thus, the emphasis has been on *functional testing*, not performance evaluation. However, the two tasks are related, since the the problem of generating a large and representative set of policy requests for performance evaluation is related to that of generating a test set that covers as many of the policy conditions as possible. By ensuring full coverage, we ensure that all policy conditions are checked and hence there is a path to each terminal node in the

decision tree inferred from the policy set (Martin, 2006). This is a necessary but not sufficient condition for any performance test set to capture worst case performance for a given conditional system under test; it is not sufficient as there could be many paths from the root of the policy set "tree" to its leaves which represent terminal decisions such as Permit, Deny, Indeterminate and Not Applicable. Each path potentially has a different time cost and there is no guarantee that a covering set includes the paths with the highest cost.

There is already substantial research on applying functional testing of decision systems to policy-based systems. Martin (2006) describes how Margrave (Fisler et al., 2005) can be used to determine which rules in a complex policy are redundant. Let policy A be the policy under test, and policy B be the same, except the decision of the specified rule is inverted. If that rule is redundant, Margrave will identify no change impact between policies A and B, otherwise it will generate counter examples as policy requests. Thus Margrave can be used to generate policy requests to exercise the rules which are deemed to be essential.

The ALFA system is an attractive policy editor but the NIST ACPT (Hu, 2008) has a wider scope because it also offers a means to generate access requests. The ACPT policy editor provides a GUI by which authors can assemble policies using text controls such as dropdown menus; ACPT creates (necessarily verbose) XACML behind the scenes. In that regard it is reminiscent of the prototype policy editor of (Lang et al., 2008). However, it is also designed to integrate with the NIST combinatorial testing tool ACTS (Hu et al., 2011; Kuhn et al., 2010) as described in (Hu, 2008). The focus is on creating policy sets and generating *adequate* access requests with which to test those policies, particularly their correctness. Such testing can be done combinatorially and/or by *mutation testing* (Martin et al., 2008). Thus the policy requests are generated to test access control policies and not for their own sake. Therefore, the requests to be generated are selected based on their ability to "debug" a policy set and are typically thrown away unless they identify an incorrect policy or conflicts between policies. This is in contrast to the aim of generating events (arising from typical usage scenarios) to provide a *representative* set of requests for these scenarios.

Bertolino et al. (2012) describe some new techniques for generating requests, notably the "Simple Combinatorial" and "Incremental XPT" algorithms. They claim that their incremental algorithms improve on the methods in (Martin et al., 2006) in the sense that they achieve the same (or better) levels of coverage for smaller (or the

same) numbers of requests. The focus remains on coverage rather than on making the requests as similar as possible to the types of request encountered in practice.

Turkmen and Crispo (2008) generate *synthetic* requests to match their synthetic policies. Note that a request set with good policy coverage might also be suitable for identifying worst case performance. However, for more general performance testing, more control would be needed, e.g., to generate requests *representative* of actual deployments and/or requests parameterised with particular performance predictors. The difficulty with synthetic policies and requests is that an actual deployment has a performance profile that is a weighted combination of the performance of synthetic policies and requests. In practice, those (unknown) weights play an important role in ensuring the external validity of any performance investigation.

Li et al. (2008) describe a similar experiment to that of (Turkmen and Crispo, 2008), using a different set of PDPs, but with more details concerning how the policies and requests are generated. In particular, their policy synthesiser uses policy templates with the missing details being added so as to generate policies with specific technical properties (such as attribute type) rather than domain semantics. The request generator creates requests as exhaustive combinations of attribute `id` and `value` pairs (Martin et al., 2006) (it ignores the semantics of the policies and was originally developed for policy coverage testing), which is unlikely to be representative of real-world requests, either in type or distribution.

In summary, there are many proposals for generating requests for the combinatorial testing of policies, with the emphasis being on policy coverage, not fidelity to the typical profile of requests in that domain, which is the objective of the request generation algorithm presented in Chapter 4.

2.2.11 Performance models

As mentioned above, the industry standard language for expressing access control policies is XACML (OASIS XACML-TC, 2005a). The XACML architecture supports highly scalable PEPs but the performance of the PDPs is a concern. Generally the response of other researchers is not to develop performance models, but to perform direct comparisons of the performance of a particular policy evaluation scheme against a known reference.

The PDP has to solve a complex search problem, depending on many factors such as the number of rules sharing a `Target` that matches the request, how deeply nested the policy set is and what rule- and/or policy-combining algorithm is in force. Therefore an explicit performance model would be complex and might be difficult to find.

Rather than trying to build a (fragile) explicit model for service times, it is possible to collect timing observations and build a simpler statistical model based on these observations. This experimental approach operates at the level of request *ensembles* rather than individual requests. While some detailed insight is lost, we potentially gain a flexible model that can be updated easily (e.g., in respect of clusters of observed requests, arrival rates, etc). Our previous research in access control performance (Butler et al., 2010, 2011) developed relatively limited statistical predictive models of performance and form the basis of the extended performance models presented in this dissertation.

2.3 Research questions

Table 2.2 classifies the main requirements for a policy performance investigation study, as identified from the review of the state of the art (§ 2.1 and § 2.2). The state of the art is evolving particularly quickly in relation to the manipulation of policies (integration, decomposition, splitting, etc.) but advances in experimental evaluation of policies appear to have slowed, perhaps given the availability of tools such as ACPT, which is intended for correctness and completion testing, not performance.

Another feature is the fact that policy analysis tends to get much more attention than request analysis—most researchers see requests as just a means to test policy properties. A select few—Marouf et al. (2011) and Deng et al. (2014) are notable exceptions in this regard—have addressed the question of whether it is better to consider both policies and requests as predictors of access control system performance. This is a theme that pays a major part in this dissertation.

Apart from our own papers, the most recent influential experimentally-based access control policy performance analysis is Turkmen and Crispo (2008) and we have concerns about its external validity. Our papers have begun to address this gap, notably Butler and Jennings (2015), but it is noteworthy that other researchers have

Table 2.2 Summary of Requirements and Methods

	Requirements	Methods
1	Support multiple access control models: DAC, MAC, RBAC and their variants (Ferraiolo and Kuhn, 1992; Jin et al., 2012) and dynamic policies (Brewer and Nash, 1989; Russello et al., 2008)	Offer ABAC model with support for various access model "design patterns"
2	Support policy analysis (Davy et al., 2008; Jajodia et al., 1997; Lin et al., 2007; NIST/NSA, 2010)	Flexible representation can support inference
3	Support policy combination (Mazzoleni et al., 2006; Rao et al., 2009)	Attributes can be compared and rule similarity can be computed
4	Separation of concerns between attribute and rule definitions (Hu, 2008; Kuhn et al., 2010; Lang et al., 2008)	Model has two or more linked components
5	Consider how to generate policies and requests (Bertolino et al., 2012; Martin et al., 2006)	Identify options for generating semantically consistent policies and requests
6	Measure the benefits from policy improvements such as refactoring and decomposition (Decat et al., 2012; Deng et al., 2014; El Kateb et al., 2012)	Allow different variants of the same policy: check that the same decisions are made and compare their performance

generally not tried to use performance measurements gathered under controlled conditions to predict access control performance in real deployments.

Table 2.2 also indicates how these requirements can be realised in this dissertation. They appear as general themes, e.g., the representation of policies and requests and its role in performance experiments plays a major role in Chapter 4 and is the main subject of the evaluation in Chapter 6.

We believe a testing framework for XACML policy evaluation is needed, to facilitate research into the performance and scalability problems facing XACML-based access control. The aim of our work is to provide a *flexible* (in the sense of being easily configured) framework, enabling researchers to perform quantitative *experiments* (hence under controlled and repeatable conditions). Indeed, performance testing needs to respond to (transient) environmental conditions by replicating the request set in a manner which is similar to the typical load on a PDP. This dissertation describes such a performance testbed, which forms one of the main research contributions.

Table 2.3 presents the research questions that are addressed in this dissertation. As can be seen, they are motivated by considering the research background and related literature earlier in this chapter.

Subsequent chapters in this dissertation consider these questions in turn.

Table 2.3 Research questions addressed in this dissertation

ID	Question
RQ1	How can access control evaluation performance be measured for use in performance experiments? – What form does the service time distribution take? – What simulations can be performed to explore the effect of different request arrival patterns? – What analysis can be performed when the systems under test use different languages, frameworks and encodings?
RQ2	How can domain models be specified and used to express enterprise access control scenarios? – How can different variants of domain models be specified in a flexible and easy to use way? – How can access control evaluation performance be compared at different domain sizes?
RQ3	How can the data from performance experiments be used to understand and predict access control evaluation performance? – What types of exploratory data analysis are suitable for the performance experiments? – What are the steps needed to build statistical models predicting access control performance?
RQ4	What are the main factors affecting access control evaluation performance? – What are the effects of PDP choice, domain size and resources? – What are the effects of domain size, policy and request characteristics?

Chapter 3

STACS: a testbed to explore access control performance

Table 3.1 Research questions addressed in Chapter 3

ID	Question
RQ1	How can access control evaluation performance be measured for use in performance experiments? – What form does the service time distribution take? – What simulations can be performed to explore the effect of different request arrival patterns? – What analysis can be performed when the systems under test use different languages, frameworks and encodings?
RQ2	How can domain models be specified and used to express enterprise access control scenarios? – How can different variants of domain models be specified in a flexible and easy to use way? – How can access control evaluation performance be compared at different domain sizes?
RQ3	How can the data from performance experiments be used to understand and predict access control evaluation performance? – What types of exploratory data analysis are suitable for the performance experiments? – What are the steps needed to build statistical models predicting access control performance?
RQ4	What are the main factors affecting access control evaluation performance? – What are the effects of PDP choice, domain size and resources? – What are the effects of domain size, policy and request characteristics?

This chapter introduces the policy evaluation service time measurement testbed that is featured in this dissertation. The first step in managing access control system performance is to *measure* the time taken by the `PDP` when evaluating the policies for a set of requests. We developed a measurement testbed for this purpose and this chapter reviews the scope of the performance model that underpins the measurement testbed (§ 3.2 on page 62), with that model having different characteristics, depending on the frequency of request arrivals. Having set the scene, § 3.3 on page 65 introduces the testbed, indicating how it can be used for different measurement scenarios. Two major usages of the testbed are then presented. The first uses *measurement-based simulation* (§ 3.4 on page 72) to collect service time measurements, characterise them and configure two types of simulation to generate predictions where the request *ensemble* needs to be considered as a unit, i.e., any performance predictions are based on request ensembles having a given range of characteristics. The second usage (§ 3.5 on page 91) treats the requests as just a means to exercise the policy set and deployed `PDP`s, and is more concerned with measuring the effects of different resource choices. The first theme is not developed further in this dissertation, but we intend to return to it in future research. The second theme is extended greatly in Chapter 4 on page 111 (which addresses the problem of generating artifacts that more closely represent the enterprise access control domain), Chapter 5 on page 182 (which expands upon the statistical model, to take account of the vastly richer domain model introduced in the previous chapter) and Chapter 6 on page 207 drills down into the enhanced "domain + statistical" model to identify some representative performance predictions.

Of course the research investigation in this, and subsequent chapters, needs to be placed on a sound philosophical and methodical foundation. In particular, the objectives and limitations of the testbed need to be considered, as well as the way that experiments conducted in the testbed can relate to performance of access control systems in enterprise deployments. § 3.1 below attempts to place the methodology in context and to address some of the concerns above, and provides some justification for the work described in later chapters.

3.1 Methodology

The research methodology used in this dissertation has the pattern in Algorithm 3.1.

Algorithm 3.1 Dissertation methodology

1: Model the domain, leaving some free parameters
2: **repeat**
3: Assign values to the free parameters in the model
4: Configure performance experiments
5: Measure service times
6: Estimate parameters/predict performance
7: Recommend improvements
8: **until** (Performance is acceptable) or (No significant improvement is possible)

The procedure in Algorithm 3.1 is typical of engineering and experimental computer science. It is based on *positivist* philosophy[1], in which the scientific method is used to set up a falsifiable hypothesis such as "The performance model f with parameters $x = \{x_i\}$ has better performance and scalability than model f' with parameters $x' = \{x'_i\}$". However, the complexity of the underlying domain is such that the model can provide only a partial explanation of access control performance in enterprises. Indeed, its uncertainty is sufficiently large that a post-positivist (more specifically: *critical realist*) perspective is more appropriate: a) all observations have error (and even our estimates of that error are uncertain) and b) there is an objective (but effectively unknowable) model to predict access control evaluation performance. Consequently, it is unreasonable to expect, *in a single study*, to build a faithful model of access control system performance, except for very specific, constrained scenarios.

§3.2 presents the scope of our performance model, indicating that the focus of our study is on the time taken by the PDP to make access control decisions. In addition to the model predicting the service time of a *single* request, it is also necessary to consider access control performance of an *ensemble* of requests arriving with a specified temporal distribution. Requests are generated by a stochastic process, so queueing will occur except in (uninteresting) cases where request inter-arrival times are much greater than request service times. Queueing is considered in §3.2.2.2 and in more detail in §3.4.

[1]To quote Lord Kelvin: "To measure is to know." and "If you can not measure it, you can not improve it."

3.2 Scope of the performance model

3.2.1 Overview of the model used in this dissertation

In a request-response system, requests arrive at one or more servers, which compute responses to these requests. Usually, as is the case here, a response is generated by the server for each incoming request. Therefore it is meaningful to consider the service time t as being the time taken by the server to compute a response for a given request. Each request is generated by some external process, which affects the rate and type of requests that arrive at the server(s). The observed request arrival distribution is $a(t; p_a)$, where t represents arrival time and p_a is a set of (unknown) parameters that affect how the requests are generated. Similarly, the observed service times belong to a distribution $b(t; p_c, p_s, a(t)^-)$, where p_c represents the relevant parameters of the computational procedure that generates the response from the request, p_s represents the performance-influencing parameters of the server and $a(t)^-$ represents the (historical) set of arrivals from time $t = 0$ until the time $t = T$ when the server starts to process the specific request for which the service time is captured.

The complete model has many stochastic elements and interdependencies which make both controlled experiments and prediction more difficult. Therefore, in this dissertation, the following simplifications are applied

Isolated servers. In practice, each server will have many running processes as it manages a workload that includes serving the requests but also includes other tasks. However, to reduce the scope for nuisance factors yielding spurious conclusions, the service times are assumed to be independent of all other server processes and `STACS` is configured accordingly, so that other processes, even data collection, require negligible resources compared to the main process(es) on each server that serves the requests.

Decoupled arrival and service processes. When the arrival rate increases relative to the service rate, a queue forms at the server. This introduces some queueing overheads, as the server needs to manage the queue in addition to computing the response to each request. However most servers have some multiprocessing ability: processes can run in parallel either at the operating system or other levels (e.g., as threads in the Java Virtual Machine (JVM)). If multiple CPU cores are available, some pipelining/parallel processing is possible,

and the average service time is reduced accordingly. The presence of both queueing overhead *and* multiprocessing speedup means that the time taken to service a specific request depends on what was previously in the queue; these two factors introduce a stochastic element to the estimate of service time. If there is a way to remove this dependence, the service time becomes easier to predict.

Single servers only. A single queue with multiple servers often exhibits less extreme variation in response time than if each server had its own queue. However this "smoothing" effect applies to ensembles of requests and introduces another stochastic factor that influences service time, again making performance prediction more difficult.

All requests are served with the same priority. There are many queueing disciplines, of which the most common is probably FIFO. Furthermore, when the queue size exceeds a given limit, some requests might be dropped and need to be resubmitted at a later time. Again these features introduce stochastic components into the predictive model for service times.

These simplifications/assumptions are designed to minimise the stochastic elements of request evaluation. However such simplifications are not common in practice. Therefore it could be argued that predictions based on the simplified model are unrealistic. However, the removal of many of these sources of uncertainty has the following benefits:

- by reducing the sources of uncertainty, predictive models can focus on fewer but more easily controlled factors, such as hardware capabilities (RAM size, CPU clock speed), server configuration and request formulation;
- notably, if the hardware capabilities and server configuration are maintained at constant values, it is possible to focus on the characteristics of the requests to predict performance. This would enable more intelligent load-balancing based on inspection of the incoming requests.

As will be seen in § 3.4.5 on page 79 and § 3.4.6 on page 84, it is possible to reintroduce stochastic effects in a controlled fashion through a procedure known as *measurement-based simulation*; see § 3.4 on page 72. Therefore this hybrid approach helps to derive insights into possible specific interventions by system administrators, while leaving open the possibility of synthesising stochastic and non-stochastic models to generate performance predictions for more realistic scenarios.

It should be noted that the models, predictions and analysis in Chapter 4 to Chapter 6, inclusive, are all based on the simplifications described above. Indeed, further work in the form of simulations would be needed to map such results to a less controlled, but more realistic deployment. The present chapter describes some initial efforts in this regard, see § 3.4.5 on page 79 and § 3.4.6 on page 84. The main objective of such proposed hybrid (measurement, modelling and simulation) experiments would be to determine the extent to which the measurement and modelling findings are still significant when stochastic factors are reintroduced. While the addition of stochastic factors is not expected to change findings such as the ranking of PDPs by performance, the differences might become less dramatic because stochastic factors such as arrival distributions are likely to introduce greater performance variability and hence reduce the power of many statistical procedures.

3.2.2 Access control arrival analysis

3.2.2.1 Intermittent request arrivals

If requests arrive infrequently, so the PDP has already responded to the previous request, there is no time for a queue to develop. In such a case, the performance model can be used to analyse the factors that are expected to affect the service time per request. Typical factors might include the PDP, the policy set size, the way the policies are encoded, the memory available to the PDP, etc.

In practice, the model will have a statistical component, enabling the effects of such factors to be estimated and visualised. Reducing the service time has a direct effect on throughput and latency.

3.2.2.2 Frequent request arrivals

If the request arrival rate increases, and/or the request service time increases, the performance model described in §3.2.2.1 is no longer adequate. It becomes necessary for the performance model to consider the effects of a build-up of requests at the PDP. The delay experienced by the entity that submitted the access request is no longer a simple function of the service time of that specific request.

Using Kendall's notation from queueing theory (Kleinrock, 1975), we expect policy evaluation to take the form of a queueing system of type $M/G/k/q$. We assume that each policy server has the same characteristics and that

- the arrival process is Memoryless (M): the number of arrivals before time t does not affect the number of arrivals after t. This is generally true, except where requests are generated in bunches, e.g., to prevent conflicts of interest;
- the service process follows a General distribution: we do not yet know the form of the distribution, in particular whether service time can be written as a function of requests[2];
- there are k policy servers, possibly physically distributed, each with its own (logical) queue;
- there are q service waiting places. Assuming n is the number of policy requests that can be held in each server's buffer, $q \equiv kn$.

Using this model we can estimate average queue lengths, capacity and expected throughput for given queue parameters. Alternatively, given practical constraints on server numbers (k), memory size (hence n) and per-server queue lengths (hence a possible need for load balancing between `PDP`s), and targets for throughput, etc., we can "solve" for the estimated service rates. If the estimated service rate cannot be achieved, neither can the throughput. In that case we can suggest what additional servers, cache sizes, etc., are needed.

3.3 Introduction to STACS

Many enterprise-level access control systems encode access controls as XACML (Moses, 2005) hence researchers focus on XACML policies and requests and their use in PEPs and XACML-based PDPs.

One of the key steps in Algorithm 3.1 is to measure PDP service time performance t_p. Most PDP implementations are not instrumented to capture service times, so obtaining the measurements would be difficult (forcing in new features to an existing

[2]As will be seen later in this chapter, in many deployments, the requests can be clustered by service time (resulting in peaks at specific times) but the relative size of each peak depends on the request mix.

system) and apt to introduce severe measurement errors. Even if it were possible to collect such measurements in this way with sufficient accuracy, it is not clear whether it would be productive. Implementing a data collection strategy in a production environment has both advantages and disadvantages. The main advantage is that the measurements apply to the PDP configuration under test, and so their *external validity* is assured (because the test and production environments are identical). However, there are serious disadvantages, including the fact that the measuring system itself places extra load on the production access control system, when the latter can be assumed to be suffering from performance difficulties already. Furthermore, it is difficult to see how to apply Algorithm 3.1 in a production environment. Firstly, the researcher has limited control over the infrastructure and so any study is *observational*, which severely limits its statistical power. Secondly, the overall optimisation loop pre-supposes that infrastructure managers would permit changes to a production environment, where those changes, though predicted to improve performance, might have unforeseen consequences.

Given the significant drawbacks described above, `STACS` (Butler, 2015c) provides a "safe" environment in which to measure access control performance and to validate performance improvement interventions before applying them in a production environment. Note that `STACS` makes Algorithm 3.1 a practical proposition: implementing Algorithm 3.1 on `STACS` solves the problems identified above with conducting performance experiments with a production access control system. However, the downside is that it is necessary to limit any threats to external validity owing to the fact that a testbed is used in place of the real system.

`STACS` needs to be able to do more than collect measurements. For example, typical performance measures of a PDP set include *latency* and *throughput*, so it needs to be able to compute both. Moreover, it is also necessary for `STACS` to supply data to other analysis components for purposes such as visualisation and performance prediction.

One of the key benefits of `STACS` is that it provides a common standard for comparing different PDPs and different access control performance improvement proposals, without favouring any of the competing PDPs and/or proposals (Butler et al., 2010). In that regard, the evidence presented by access control performance researchers has traditionally been based on comparisons with the `SunXACML PDP` reference implementation (Proctor, 2004), often using unpublished policies and requests. Hence it is difficult to compare one approach with another, or to determine what tradeoffs

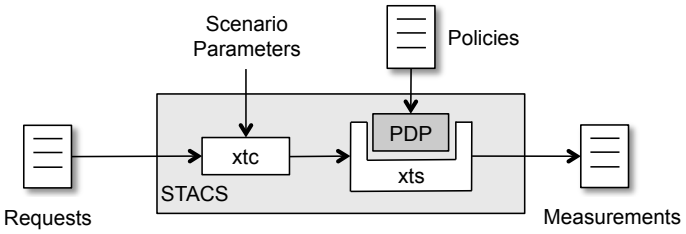


Fig. 3.1 XACML Load Testing System Architecture (STACS).

occur. The aim of STACS is to provide a *flexible* (easily configured) framework, enabling researchers to perform quantitative *experiments* (not just observational studies) under representative, controlled and repeatable conditions.

3.3.1 STACS Overview

The architecture of the STACS load testing framework is presented in Figure 3.1, and is a development of that presented in Butler et al. (2010). The XTS (server) comprises a PDP, a simplified "universal" PEP and specific PDP *adapter*. Each PDP implementation needs an adapter to wrap calls from the universal PEP. The adapter also brackets each PDP call with timing calls to compute the elapsed time *at the PDP*.

> If t_{in} is the time at which the request leaves the adapter for the PDP, and t_{out} is the time at which the response arrives at the adapter from the PDP, the measured service time $t_p \equiv t_{\text{out}} - t_{\text{in}}$.

The XTC (client) submits requests to XTS. XTA collects the results and writes them either to a text file in the file system or, more commonly, a table in a database. Figure 3.1 presents the architecture of STACS.

3.3.2 Access Control Service Time distribution

The continue-a policy set described by Fisler et al. (2005) was loaded into the SunXACML PDP policy repository. An instrumented PEP (XACML Testing Server (XTS)) was developed within STACS to call the SunXACML PDP, with timing hooks inserted in XTS to capture the total time spent per request a) converting the

	cluster									
decision	1	2	3	4	5	6	7	8	9	Total
Deny	34	12	18	26	11	6	5	10	22	144
NA	9	5	5	5		1			6	31
Permit	6	4	2	5	1		1	3	3	25
Total	49	21	25	36	12	7	6	13	31	200

Table 3.2 Contingency table relating *observed* decisions to *inferred* request clusters

XACML-encoded request into the PDP's internal representation in memory, b) searching the policy set for matching policies and c) returning the decision as a XACML-encoded response. Two hundred representative requests (the `single` set from Fisler et al. (2005)) were issued against the server and the timings were recorded in a text file. This process was repeated 100 times, in random order, on a laptop that was otherwise idle, to minimise the effect of anomalous timings (if a background process started, say). The algorithm used to process this raw data to provide *clustered* measurement data for simulation purposes is presented below. More context is needed to interpret the clusters—as can be seen from Table 3.2, there is no direct relationship between policy decisions and cluster membership.

Let $t = t(S, P; R, q) \in \mathcal{R}^{u \times q}$ be the set of PDP service times, where S represents (characteristics of) the PDP server, P represents the policy set to search, R is the set of requests, $r = |R|$, U is the combination of $S \times P \times R$ experimental conditions, $u = |U|$ and q is the number of replicate measurements of t, holding conditions S, P, R fixed. A kernel smoother (Wand and Jones, 1994) provides a finer discretisation of the domain, easing the task of locating the peaks.

Algorithm 3.2 lines 1–2 removes anomalous service times by choosing the minimum of the replicate service times for each $S \times P \times R$ combination of experimental conditions. Lines 3–4 compute the (probability) density of service times for each $S \times P$ combination, based on the r available service times for that combination. Lines 5–6 inspect the service time density function for each $S \times P$ combination and estimates the number n of request clusters. Lines 7–12 compute a function of each service time distribution such that the minima of this function are candidate cluster centres. Lines 13–14 label requests according to their membership of the service time clusters, for each of the $S \times P$ service time distributions. Lines 15-16 estimate the mean and variance of the Gaussian distribution fitted to service times of requests in the $|S| \times |P| \times n$ clusters. In Line 6, the user needs to intervene to judge the number of

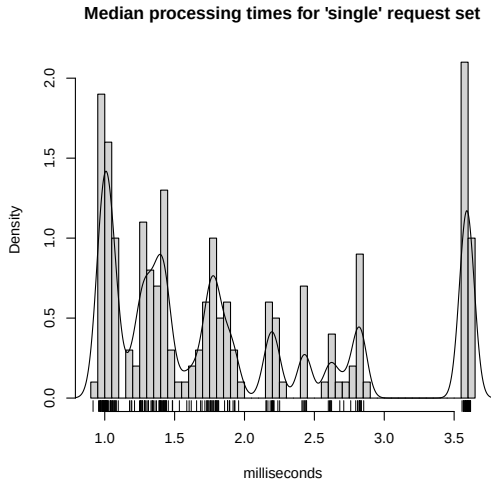
Algorithm 3.2 Algorithm to derive the Request-cluster contingency table

1: **for** i=1 to r **do**
2: Calculate the median $\tilde{t} = \tilde{t}(S, P; R)$.
3: Apply a Gaussian kernel smoother $d = d(t)$, with window size 0.05 (milliseconds), generating $m = 1000$ density points. /* Visually inspect the density function and decide the number of peaks. */
4: Compute $\dot{d}(t)$ and $\ddot{d}(t)$ by finite difference approximation.
5: Sort $|\dot{d}|$.
6: Count the peaks, say n
7: **for** i=1 TO n **do**
8: **if** $|\dot{d}_i| <$ tol and $\ddot{d}_i < 0$ and $d_i > 0$ **then**
9: Store the location of peak[i] as $\hat{p}_i \equiv (\hat{t}_i, \hat{d}_i)$
10: Set $\lambda_1 = t_{\min}$;$\lambda_{n+1} = t_{\min}$
11: **for** i=2 TO n **do**
12: Compute the cluster interval breakpoint $\lambda_i = \alpha\hat{t}_i + \beta\hat{t}_{i-1}$ where $\alpha = \frac{\hat{y}_{i-1}}{\hat{y}_{i-1}+\hat{y}_i}$ and $\beta = \frac{\hat{y}_i}{\hat{y}_{i-1}+\hat{y}_i}$
13: **for** i=1 to r **do**
14: Find $\max_j(\lambda_j)$ such that $\lambda_j <= t_i$. Then cluster indices $C_j = C_j \cup \{i\}$.
15: **for** i=1 TO n **do**
16: Fit a Gaussian distribution $N_i(\mu_i, \sigma_i^2)$ and store μ_i and σ_i^2.
17: Create a Decision $\times$ Request-cluster contingency table.

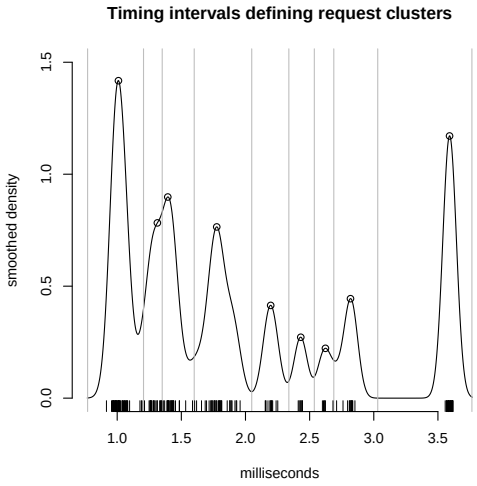
peaks n by visually inspecting a plot of the smoother from Lines 3–4 above; otherwise the algorithm is fully automatic. The algorithm was implemented in R (Hornik, 2009).

The `continue-a` policies with the associated `single` requests (Krishnamurthi, 2003) were used in (Butler et al., 2010). An example run using these policies and requests is shown in Figure 3.2. Figure 3.2a shows service time measurements as a histogram overlaid with a fitted density curve; this is the starting point for the algorithm. Figure 3.2a indicates how the cluster end-points are derived from the service time density curve.

Two PDP implementations (SunXACML PDP (Moses, 2005) and EnterpriseXACML PDP (Wang, 2010)) were each given the same policies and requests and each gave the same response for every policy-request combination. However the service time distributions were different. The most interesting feature in Figure 3.3 is the evidence that service times are clustered around a small number (less than 10) of times. Consequently the statistical distribution of the service time sample is not a classical distribution; rather, it is a *mixture* of simple unimodal distributions. Indeed, both PDP implementations in the experiment show some evidence of service time clustering, but at different cluster centres. Since policy evaluation is essentially a search operation, and the search can terminate once the decision is known beyond doubt,

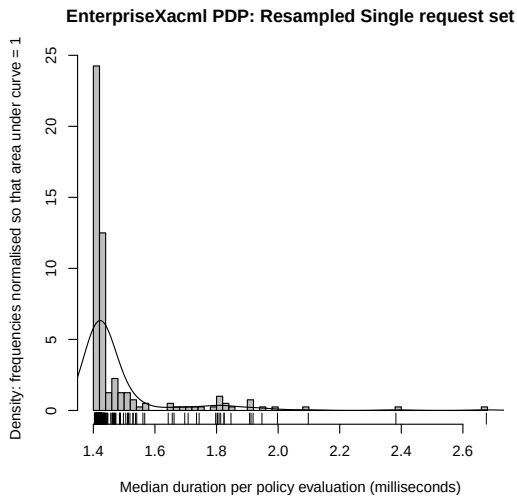


(a) Distribution of measured request processing times.

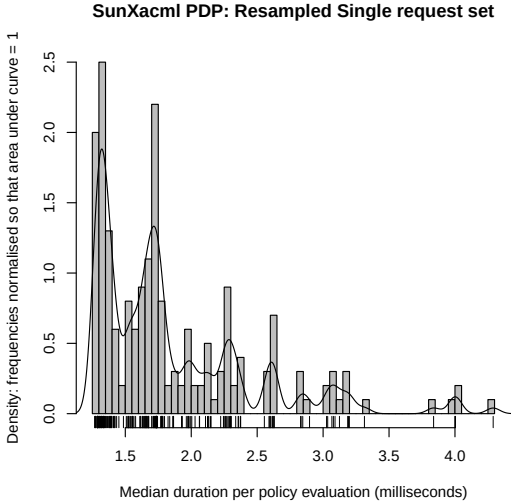


(b) Assigning requests to clusters.

Fig. 3.2 Clustered service times for `continue-a` policies and requests on `SunXACML PDP`.



(a) `EnterpriseXACML PDP` evaluation duration frequencies.



(b) `SunXACML PDP` evaluation duration frequencies.

Fig. 3.3 Comparison of the performance profiles of two XACML PDPs on the same policy and request sets.

different cluster centres reflect differences in how that search proceeds in each PDP. For example, if the *PDP* indexes the policies in an effective manner, it might be possible to locate the policies that match a request in "constant" time; otherwise the PDP needs to search the policy tree and the service time is a function of the paths used to search the tree.

> The findings are valid *for that combination of policies and requests* since we employ a randomised block experimental design and control for other known factors.

3.3.3 Uses of STACS

The STACS testbed can be used for two purposes:

Prediction of dynamic conditions To estimate a performance metric given a new set of conditions, e.g., a change in the access request mix, or rapid changes in request arrival rates. Here time plays an important role because future access control performance depends on current load and queue length. In this case, the measurement testbed of STACS needs to be augmented with simulation, as described in § 3.4.

Comparison of static parameters To estimate the effects of an experimental treatment under controlled conditions, by comparing cases with or without that treatment. Treatments might include projected PDP improvements, increased policy set size, etc. No temporal ordering is implied: the effects would be the same even if the order in which the treatments were applied was shuffled.

§3.4 describes the former prediction scenario, where STACS collected service times which were then processed to configure a discrete event simulation to estimate the effectiveness of a proportional thinning admission control algorithm in cases where the PDP was subject to very high request arrival rates. Butler et al. (2011) provides more information on how simulation based on service time measurements obtained with STACS can be used to predict the latency *experienced* by users when the request arrival rate is high enough to be considered "frequent".

§ 3.5 describes the latter comparison scenario, where STACS was used to analyse the effects of switching to a dramatically different PDP implementations, and even to

estimate how much of the performance difference was due to different secondary factors. Griffin et al. (2012) presents a new PDP design and shows, using experiments involving `STACS`, how its performance exceeds that of more conventional PDPs like `SunXACML PDP` and `EnterpriseXACML PDP`.

3.4 Measurement based simulation

By instrumenting the open source `SunXACML PDP` implementation, we noticed that the execution time of many of the steps taken by the PDP do not depend on the data (i.e., the policy set and incoming request). The major exception to this observation is the policy search step, where the PDP seeks to match the request against the policy set. This is a complex search problem, depending on many factors such as the number of rules sharing a Target that matches the request, how deeply nested the policy set is and what rule- and/or policy-combining algorithm is in force. Rather than trying to build a (fragile) explicit model, we collect timing observations and build a simpler implicit model and predict its behaviour by simulation. This experimental approach operates at the level of request *ensembles* rather than individual requests. While we lose some detailed insight, we gain a flexible model that can be updated easily (e.g., in respect of clusters of observed request). We can also model an *admission control* scenario, where the PEP rations admission to the PDP to prevent PDP utilisation from exceeding an operational threshold.

The simulation tool we use (OPNET™) has extensive support for queueing experiments, enabling powerful analysis of "what if?" scenarios using simulated data. However, our simulation experiments are grounded in actual measurements from real PDPs, thereby reducing threats to their *external validity* (Thakkar et al., 2008). By taking measurements regularly, we can also monitor the performance impact of policy changes, by analogy with the use of Margrave to do (logical) policy change impact analysis (Fisler et al., 2005).

Measurement-based simulation for performance modelling and enhancement has a long history (Jamin et al., 1997). Sometimes it offers the only practical approach for modelling the behaviour of a complex system in difficult conditions. However, for a given performance improvement technique, such as those outlined in Chapter 2, it is difficult to decide whether that improvement technique brings *material* benefits in

PDP performance. Our initial studies suggest that Measurement Based Simulation provides a way of answering that question.

While the explicit analytical model presented in § 3.4.1 is attractive and convenient for sensitivity analysis and other uses, it is not sufficient:

- explicit formulae are unknown for quantities such as the queue length variance
- known formulae evaluate mean values only, reflecting long-term queue evolution not transient effects
- if an explicit service time distribution model is not available, explicit formulae will not exist.

To overcome these limitations, we developed a simulation model. Following the explicit model, we model the XACML PDP as a single processor, serving a single queue and employing a FIFO queueing discipline. The justification is that both `PDP`s used in the study are single-threaded, deployed on a single server and the `XTC` component of `STACS` ensures that each request waits until the `PDP` has issued a response for the previous request that was submitted. Arriving XACML requests are placed at the tail of the queue and served in order of arrival. For the M/H_k/1 queue corresponding to our analytic model, we view the simulated PDP as comprising k disjoint components, each associated with a single request cluster. Request tokens are produced by k Markov processes representing the k clusters of requests. The token generation rate of each Markov process becomes the inter-arrival rate of the queue for the appropriate PDP cluster-specific component. By this device, we decouple request token generation and consumption into separate per-cluster streams; see Figure 3.4.

3.4.1 Mean Value Analysis of an Analytical Queueing Model

A PDP can be modelled as a queue: requests arrive with mean arrival rate λ and exit with rate $\mu^{(s)}$. For the queue length to be bounded, we require $\rho < 1$, where $\rho = \lambda\bar{x}$, where $\bar{x}$ is the mean service time. In typical deployments, the arrival process may be nonstationary, e.g., request arrival rates are greater during working hours. However, in the simplest case, the arrival process is memoryless and hence the inter-arrival times have an exponential distribution, as assumed in this dissertation. We consider nonstationary entensions to the model in §3.4.3. The simplest queueing model is $M/M/1$ with FIFO scheduling. Since the measured service times are known to be

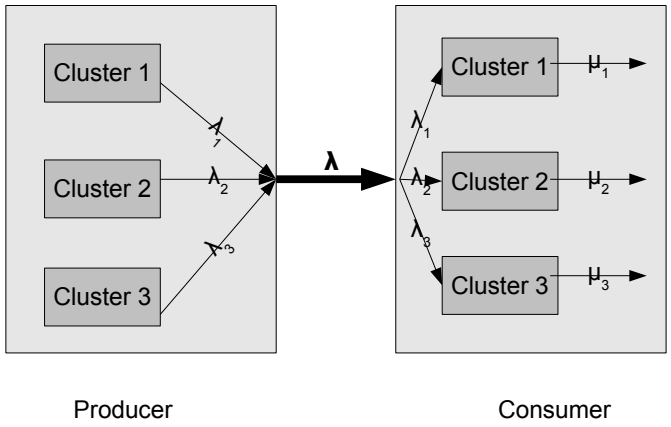


Fig. 3.4 Decomposing the simulation request token producers and consumers into cluster-specific components.

clustered, the queue does not satisfy the assumptions of this simple model. Instead, we model the queue as $M/G/1$, i.e., access requests are generated by a Markov process (hence arrivals are memoryless), but the service times are drawn from a "General" distribution. Because of the presence of request clustering, we choose to model PDP service times as being drawn from a distribution, hyperexponential; see Equation 3.5. Each exponential term is associated with a single measured request cluster. The weights combining the exponential distributions depend on the arrival rates of requests belonging to the different request clusters.

The PDP utilisation, (equivalently: mean load on the server at equilibrium) is

$$\rho = \lambda \bar{x}, \text{ where } \rho < 1 \text{ for the queue to remain bounded} \tag{3.1}$$

By definition, the coefficient of variation C_b of the service time distribution with density function $b(x)$ is defined by

$$C_b^2 \stackrel{\text{def}}{=} \frac{\sigma_b^2}{\bar{x}^2} \tag{3.2}$$

where

$$\begin{aligned} \bar{x} &\equiv E\{X\} = \int_0^\infty x b(x)\, dx \\ \sigma_b^2 &\equiv E\{X^2\} - (E\{X\})^2 = \int_0^\infty x^2 b(x)\, dx - \bar{x}^2. \end{aligned} \tag{3.3}$$

The Pollaczek-Khinchin mean-value formula for mean queue length, denoted $\bar{q}$ at departure instants (Kleinrock, 1975, Eq 5.63) is

$$\bar{q} = \rho + \rho^2 \frac{(1 + C_b^2)}{2(1 - \rho)}, \tag{3.4}$$

which is an explicit formula in terms of the quantities defined in Equations 3.1 and 3.2.

For hyperexponentially-distributed service times, the service density function is

$$b(x) \stackrel{\text{def}}{=} \sum_{i=1}^{p} \alpha_i \, \mu_i^{(\text{s})} e^{-\mu_i^{(\text{s})} x}, \tag{3.5}$$

where

$$\sum_{i=1}^{p} \alpha_i \equiv 1 \equiv \int_0^\infty b(x) dx$$

Substituting Equation 3.5 into Equation 3.3 gives

$$\begin{aligned} \bar{x} &= \sum_{i=1}^{p} \frac{\alpha_i}{\mu_i^{(\text{s})}} \\ \sigma_b^2 &= \sum_{i=1}^{p} \frac{2\alpha_i}{(\mu_i^{(\text{s})})^2}. \end{aligned} \tag{3.6}$$

Note that $\mu_i^{(\text{s})}$ and $\frac{1}{\mu_i^{(\text{s})}}$ are the mean service *rate* and mean service *time*, respectively for cluster i. We can substitute Equation 3.6 in Equation 3.2 and hence in Equation 3.4 to obtain $\bar{q}$.

Therefore, given p request clusters, with measurements of the mean service time per request cluster $\mu_i^{(\text{s})}$), we can compute expected queue lengths $\bar{q}$ for different request cluster mixes $\alpha_i, i = 1, 2, \ldots, p$.

We can also compute the mean queue waiting time using (Kleinrock, 1975, 5.70)

$$W = \rho \frac{(1 + C_b^2)}{2(1 - \rho)} \bar{x} \tag{3.7}$$

Note that mean value analysis yields the mean and variance of the service time distribution as closed form expressions involving known quantities, namely the service time cluster centres and the relative frequencies of the service time clusters. Furthermore, the mean waiting time is also a closed form expression, depending on the same quantities, together with the mean arrival rate of the requests taken as a whole.

3.4.1.1 Control objectives

As described above, an $M/G/1$ queue is assumed to be a good model for access control policy evaluation at a single PDP. Therefore, it is possible to interpret control objectives such as

- Ensure the mean queue length $\bar{q} < \bar{N}_{\text{allowed}}$;
- Ensure the mean queue waiting time $W < \bar{t}_{\text{allowed}}$;

in terms of the parameters of Equation 3.4 and Equation 3.7, respectively. The goal of the system administrator is to (re)configure the access control system so that these targets ($\bar{N}_{\text{allowed}}$ and $\bar{t}_{\text{allowed}}$) are achieved, at least in a mean-value sense. The controllable parameters include a) the number of servers (PDPs) available; b) the request arrival rate; c) the service time distribution of the combination of PDP, policy set and request.

Replicating the PDP might be expected to generate a significant performance improvement, assuming the requests can be routed efficiently to servers that are not particularly busy. However, such load balancing considerations are outside the scope of this dissertation.

The system administrator is unlikely to have much control over the rate at which requests arrive, as they arise from the normal business processes of the enterprise. However, one of the findings in this dissertation is that requests can be clustered according to their service times. Knowledge of this clustering could be used to ensure that as many as possible of the incoming requests belong to clusters with short service times.

Service time distributions depend on the choice of PDP so, other factors being equal, the system administrator should choose the PDP with the most favourable service time profile. The service time distribution might also depend on the way the policies are formulated. Often there are many ways to specify policies to achieve the same semantic objectives (i.e., the same decisions (output) are made for the same requests (input). The `STACS` testbed can be used to compare different policy formulations, hence determine the service time profile for each.

Therefore, given a relatively small set of parameters (notably including a model of the service time distribution ($b(t)$)), Equation 3.4 and Equation 3.7 can be used to

estimate the mean queue length and waiting time respectively, which in turn can be compared against their control limits.

Of course, the closed form expressions above represent only the expected values of the queue-length and time-in-queue random variables. *Observed* queue-length and time-in-queue will, in many cases, exceed their expected (mean) values. Indeed, control objectives expressed in terms of mean values might be achieved, even though worst-case objectives might not. Therefore, to address such concerns, discrete event simulation approaches and numerical experiments are described in §3.4.5 and §3.4.6.

3.4.2 Service times and arrival rates

We note that the arrival rate of each cluster-serving component is the product $\alpha_i\lambda$ of

- the relative frequency of requests belonging to that cluster: α_i
- the global arrival rate, ignoring cluster membership: λ

Because of the way α_i is defined,

$$\lambda \equiv \sum_{i=1}^{k} \alpha_i\lambda = \sum_{i=1}^{k} \lambda_i. \tag{3.8}$$

The user needs to specify the (per-cluster) mean inter-arrival times $\frac{1}{\lambda_i}$ and the measurement-derived mean service times $\frac{1}{\bar{x}_i}$ of the discrete event simulation. The mean service time is estimated by computing the weighted mean of the individual cluster service means. In practice, α_i would be found by classifying actual access requests (labelling them by their service time cluster) obtaining the empirical distribution of $\{\alpha_i\}$.

Using the cluster assignments $C(r_i) = j$ (r_i being the i^{th} request type and C being the function mapping r_i into cluster index j) from the measurements above, we can

1. Compute the mean service time $\bar{x}$ using Equation 3.6
2. Estimate the capacity (the maximum arrival rate λ such that the queue length remains acceptable ($\rho < R$ where $R < 1$) of the PDP server used to generate the measurement data above for a *given mean service time*.

3.4.3 Extending the model: steady state plus overload

Because the request arrivals (both baseline and overload) are generated by a (memoryless) Markov process, overload requests can be modelled separately from baseline requests. That is,

$$\rho = \rho^{(\text{base})} + \rho^{(\text{overload})} \tag{3.9}$$

where, in general terms, the utilisation

$$\rho^{(\odot)} = \lambda^{(\odot)} \bar{x}^{(\odot)} \tag{3.10}$$

and the general service mean

$$\bar{x}^{(\odot)} = \sum_{j=1}^{n} \alpha_j^{(\odot)} \bar{x}_j \tag{3.11}$$

Let $\lambda^{(\text{overload})} = \gamma \lambda^{(\text{base})}$ where γ is the overload factor; then

$$\begin{aligned} \bar{x}^{(\text{base})} &= \sum_{j=1}^{n} \alpha_j^{(\text{base})} \bar{x}_j \\ \bar{x}^{(\text{overload})} &= \sum_{j=1}^{n} \alpha_j^{(\text{overload})} \bar{x}_j \end{aligned} \tag{3.12}$$

So

$$\begin{aligned} \rho^{(\text{base})} &= \lambda^{(\text{base})} \sum_{j=1}^{n} \alpha^{(\text{base})} \bar{x}_j \text{ as before;} \\ \rho^{(\text{overload})} &= \gamma \lambda^{(\text{base})} \sum_{j=1}^{n} \alpha_j^{(\text{overload})} \bar{x}_j \end{aligned} \tag{3.13}$$

The base arrival rate can be computed from the base utilisation and base service times:

$$\lambda^{(\text{base})} = \frac{\rho^{(\text{base})}}{\sum_{j=1}^{n} \alpha^{(\text{base})} \bar{x}_j} \tag{3.14}$$

Note that the free parameters in Equation 3.13 are γ and $\{\alpha_j^{(\text{overload})}, j = 1, \ldots, n\}$; all other parameters are either measured directly or computable from measurements.

3.4.4 Policies and requests used

The policy set used in the following trials is the same `continue-a` set referenced in (Fisler et al., 2005) and obtained as part of the Xengine PDP source distribution. The policy set was loaded into each PDP policy repository. As with § 3.3.2, two hundred requests from the `single` set were used, but augmented here with requests from the `multi22` (Fisler et al., 2005). This made four hundred requests in all, separated into two hundred in each group.

As in §3.3.2, these requests were issued against the server and the timings were recorded in a text file. This process was repeated 100 times (with the order of the requests being randomised in each replication) on a server instance (hosting the XTS component) that was otherwise idle, to minimise the effect of anomalous timings (if a background process started, say).

3.4.5 Scenario 1: Load Control

When a server is under heavy load, typically when the arrival rate exceeds the service rate, it is sometimes better to apply an admission control procedure to meet quality of service requirements. If some requests have higher priority than others, the high priority requests should be served first. Even if requests have equal priority, but the request arrivals are "bunched together", which is typical of a bursty temporal profile of request arrivals, other forms of admission control may be beneficial. Indeed, the management of large queues of requests is itself a drain on already stretched PDP system resources. Consequently, if the PEP clients take responsibility for re-sending the requests if they have not received a response within a reasonable time, this can help to improve overall throughput because management of the queue is delegated to the PEPs. In such situations, a simple admission control algorithm like proportional thinning (Jennings, 2001) becomes attractive. The algorithm is described in § 3.4.5.1. § 3.4.5.2 presents a simulation study of how `STACS` can be used to explore the benefits of this algorithm for controlling admission to heavily loaded PDPs.

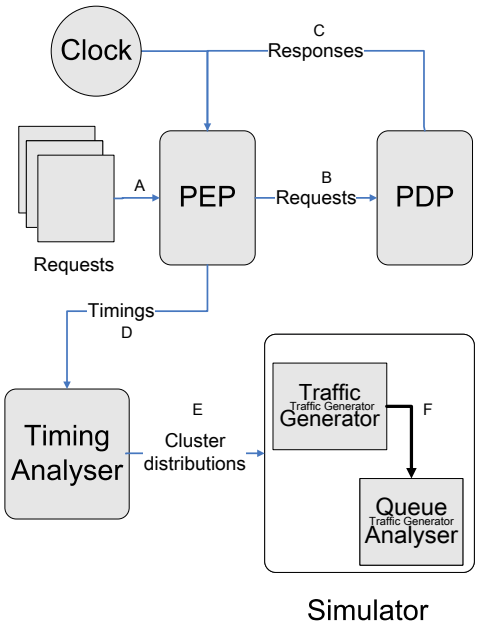


Fig. 3.5 The main measurement system and simulator components.

3.4.5.1 Load Control Algorithm Specification

We studied XACML PDP performance by considering its worst-case behaviour, i.e., when the request arrival rate approaches the capacity of the PDP processor. We reuse established methods from network management, notably measurement-based admission control (Jennings et al., 2001; Thakkar et al., 2008). The architecture is shown in Figure 3.5. Note that the PEP collects timing measurements, which are then analysed and used in the simulator.

We specified a load control algorithm that aims to ensure that the number of XACML requests forwarded to the PDP by the PEP is such that the processor of the PDP does not overload. In practice this means that, during periods of high demand, the number of requests sent to the PDP is sufficient to keep its processor utilisation (which we refer to as "Carried Load") at its maximum capacity, or some utilisation close to that capacity – in our experiments we parametrise the load control to aim for 90% processor utilisation.

We argue that the load control should be placed at the PEP, as placing it at the PDP would mean utilising PDP processor capacity itself to perform load control actions during high load conditions. Furthermore, since the PEP and PDP form a closed

system, we assume that it is possible for the load control algorithm at the PEP to be configured with measurements of the mean processing times for the identified XACML request clusters, plus information allowing the mapping of individual XACML requests to their corresponding identified request cluster.

Our proposed load control operates as follows. Every time a request arrives at the PEP a Bernoulli trial is performed to ascertain whether the request is admitted, that is, forwarded to the PDP. The trial uses a percentage thinning coefficient, denoted p_a, where $0 \leq p_a \leq 1$ and generates a random number r in the range $(0, 1)$. It uses these values to make a decision as follows:

$$Decision = \begin{cases} \text{ADMIT} & r \leq p_a \\ \text{REJECT} & r > p_a \end{cases}$$

The coefficient p_a (which is effectively a probability of acceptance) is calculated at the start of each control interval, $(t, t+T)$, and remains constant during that interval. To calculate the value of p_a the PEP maintains counters for the number of arrivals of requests of each cluster during each control interval. The counter for requests of cluster i for control interval $(t, t+T)$ is denoted $c_i(t, t+T)$. As XACML requests are typically generated by a large number of independent actors we argue that it is reasonable to model request arrivals at the PEP as a Poisson arrival process. The properties of the Poisson process mean that the number of arrivals in an interval $(t-T, t)$ is as good an estimate of the number of arrivals that will happen in $(t, t+T)$ as any. Therefore, we set the estimated number of arrivals of cluster i for control interval $(t, t+T)$, denoted $\hat{c}_i(t, t+T)$, as:

$$\hat{c}_i(t, t+T) = c_i(t-T, t)$$

If we let $\bar{x}_i$ denote the mean processing time of a cluster i request at the PDP then we can estimate the expected offered load for control interval $(t, t+T)$, denoted $\hat{O}(t, t+T)$, as:

$$\hat{O}(t, t+T) = \sum_i \hat{c}_i(t, t+T)\bar{s}_i$$

Given this, we can calculate the percentage thinning coefficient, where C_{target} denotes the target PDP processor load as a fraction of its total capacity, as:

$$p_a = \begin{cases} \frac{C_{target}T}{\hat{O}(t,t+T)} & \hat{O}(t,t+T) > C_{target}T \\ 1 & \hat{O}(t,t+T) \leq C_{target}T \end{cases}$$

Thus, p_a will be set to 1 – so that all requests will be accepted – if the expected offered load is less than the PDP processor target load. Otherwise, it is set to accept the percentage of the offered load that should result in achieving the target PDP processor load. It should be noted that, for brevity, we formulated this algorithm under the assumption of a single PEP, it is straightforward to generalise to the multiple PEP case. For the multiple PEP scenario each PEP would act independently to control utilisation of its predefined shared of the PDP processor.

3.4.5.2 Simulation Model and Experimental Analysis

We model the PDP as a single processor, served by a single FIFO queue of infinite length. Arriving requests are placed at the tail of the queue and served in order. Processing times for individual requests are randomly selected using an exponential distribution for the request cluster in question. Crucially, the mean values used to create these distributions are those values obtained using the measurement testbed `STACS`, see § 3.3.2.

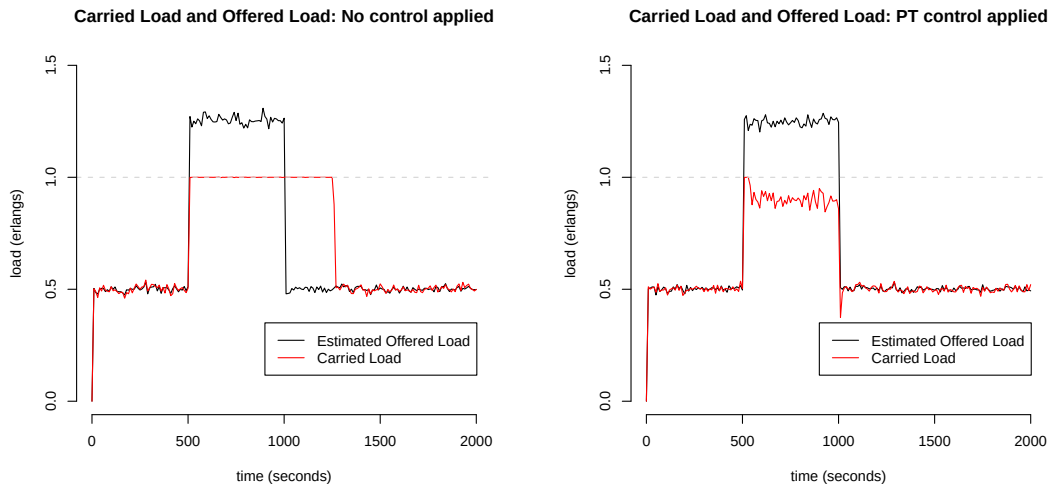
We model a single PEP, which receives XACML requests from nine independent sources, each generating requests for one of the nine identified XACML request clusters. The sources generate requests with exponentially distributed inter-arrival times (a Poisson process). The mean inter-arrival rates for each request cluster are set such that the steady State load on the PDP corresponds to 50% processor utilisation. Furthermore, the proportion of arrivals of the various request clusters in steady state corresponds to the proportion of requests falling into the nine clusters when measurements were taken using the `continue-a` policies and `single` and `multi22` requests (200 of each). As we assume the PDP processor is the bottleneck in the system we model neither the PEP processor, nor the network connecting the PEP and PDP.

We run simulations for a duration of 2000s, with a steady State load of 50% PDP processor capacity and an overload period between 500s and 1000s, during which the offered load is 125% of PDP processor capacity. We assume that if the PEP rejects a XACML request this is interpreted as a Deny result by the requester; thus there is no need to model request reattempts. We model the overload traffic as being equally caused by requests for 3 of the 9 identified request clusters (namely clusters 6, 8 and 9). Our load control algorithm is parameterised to aim to achieve a maximum 90% utilisation of the PDP processor.

We now present simulation results for two cases: a) where no load control is applied and b) where our percentage thinning load control is applied. Figure 3.6a shows the offered and carried loads for both cases. We see that when no load control is applied the PDP processor becomes quickly saturated and this saturation lasts well beyond the duration of the actual overload event. This prolonged saturation is due to the buildup of a very large PDP processor queue, which takes a long time to service. Figure 3.7a shows the queuing delay experienced by requests for the no load control case. We see that the queuing delay quickly builds up to unacceptable levels, such that even when requests are eventually processed the delay has reached unacceptable levels from the point-of-view of the actor requesting access to a resource.

On the other hand, Figure 3.6b and Figure 3.7b show that when our load control is activated the situation is greatly improved. Figure 3.6b shows that the load control algorithm broadly succeeds in maintaining the PDP processor utilisation at an average 90% during the overload period. Figure 3.7b shows that the queue size does not build up to the degree that the queuing delay experienced by requests reaches unacceptable levels.

Whilst these results do confirm that our load control algorithm succeeds in fulfilling its objectives we note that it has two main limitations. Firstly, it will operate successfully only if the profile of request processing times at the PDP remains roughly as expected. Thus, it would need to be reconfigured every time the deployed policy set changes or the profile of policy requests against this policy set changes. Ideally, the load control would dynamically adapt to such changes but that is not the case with this version of the algorithm. Secondly, when overload occurs the load control algorithm treats all request clusters similarly. All are throttled at the same rate, regardless of which clusters are responsible for creating the overload. Furthermore, the algorithm is unaware of the fact that certain request clusters might be more important from an



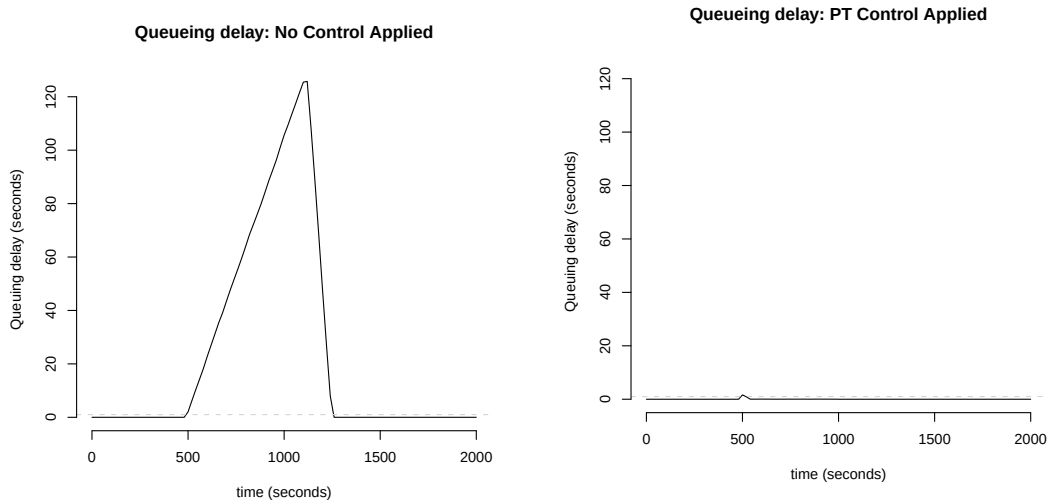
(a) Without Control. (b) With PT Control.

Fig. 3.6 Effect of PT (Percentage Thinning) Control on *Carried Load* when Offered Load exceeds PDP processor capacity.

access control perspective. Arguably, an ideal XACML load control algorithm would have the ability to prioritise the processing of particular request types over others during overload conditions.

3.4.6 Scenario 2: Exploring the effects of different mixes of requests

As seen in §3.3.2, the requests form clusters based on their service time. One interpretation is that particular groups of requests might look different to each other but from the perspective of the PDP with a given set of policies, they could be equivalent. That is, the amount of computation required by the PDP is much the same for all requests in that cluster. However, it is difficult to determine cluster membership just by inspecting the requests. The service time density is defined both by the peak locations and their widths. The peak locations depend on the (PDP × Policy × Request) combination but the request mix affects the (relative) peak heights. Therefore, when modelling the performance of PDP with frequent requests, it is necessary for any simulation to take account of the request mix. The following experiment used `STACS` to estimate the service time density and then used OPNET to simulate "favourable" and "unfavourable" request mixes.



(a) Without Control.

(b) With PT Control—Note max delay is 1.5 seconds compared to Figure 3.7a.

Fig. 3.7 Effect of PT (Percentage Thinning) Control on *Queueing Delay* when Offered Load exceeds PDP processor capacity.

3.4.6.1 Scenario 2 setup

Balanced full factorial trials were run as indicated in Table 3.3. Each host used the Ubuntu 10.04 LTS operating system. They had identical versions of applications such as Java (JDK 7). The same testbed *source* code was deployed on each. Both used dual-core 64-bit Intel processors. They differed in that `bear` had a 32-bit operating system rather than a 64-bit operating system as on `inisherk`. They also had different motherboards and memory configuration. Indeed, `inisherk` was about two years

host	pdp	Request Group
bear	SunXACML	single multi22
	EnterpriseXACML	single multi22
inisherk	SunXACML	single multi22
	EnterpriseXACML	single multi22

Table 3.3 Main experimental conditions for the trials

newer than `bear` and hence might be expected to have generally lower service times, however we cannot assume that all requests will be subject to the same speedup factor. It is also inadvisable to assume that request cluster membership is the same for different hosts: some requests could "migrate" to nearby clusters owing to differences in aspects such as memory configuration between hosts.

The two PDP implementations are representative of different design goals. `SunXACML PDP` was designed as a reference implementation, `EnterpriseXACML PDP` as an implementation with more focus on performance (Wang, 2010). They were developed independently and hence might be expected to exhibit different service time clustering behaviour. The XACML structural differences between the `single` and `multi22` *request groups* are not the focus here, rather the fact that their service times were expected to cluster differently.

The 16 cases in Table 3.3 summarise a more detailed experiment in which there are 100 replicate measurements on each of the 200 request types in the specified request group. Each arrival weight α_j depends on the arrival rate of requests in cluster j relative to requests from all clusters. Ideally α_j would be computed by observing the frequency of requests in an *actual* deployment. For the purpose of this scenario, we assume request types have identical arrival rates, in which case α_j is the relative size of cluster j.

The simulation model uses the OPNET™ simulation environment. OPNET simulations are time-based, so the user needs to specify the mean request inter-arrival time ($\frac{1}{\lambda}$, the mean service time per request ($\frac{1}{\mu}$) and the simulation duration T. OPNET's Discrete Event Simulator produces and consumes "requests" (more correctly, standard tokens) and records the queueing statistics requested by the user. OPNET request tokens are tagged by cluster ID and directed to a simulated server that handles one request at a time with the mean service time depending on the cluster ID tag, consistent with Figure 3.4.

3.4.6.2 Measured service times and clustering

Figure 3.8 shows how service times are distributed for a given PDP-data combination. Plots like this alerted us to the presence of service time clustering. Referring to Figure 3.8, visual inspection suggests the number of clusters n is 8 and the relative spacing tolerance is 0.05.

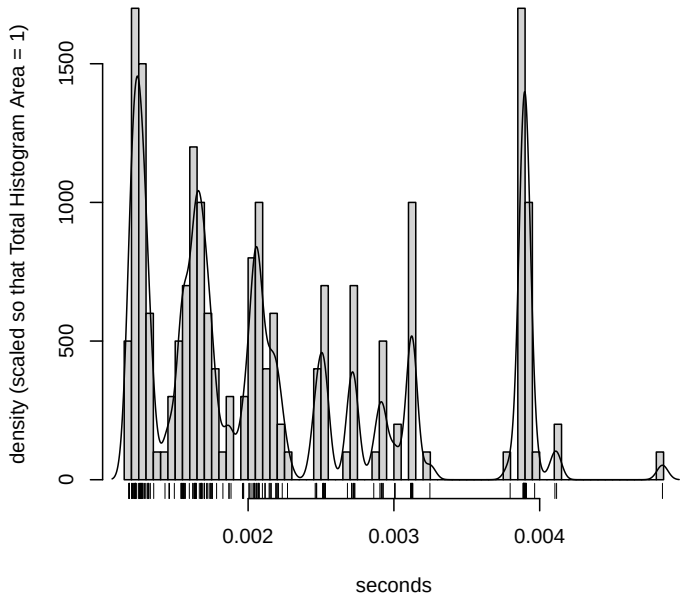


Fig. 3.8 Distribution of measured request service times on `bear` using `SunXACML PDP`.

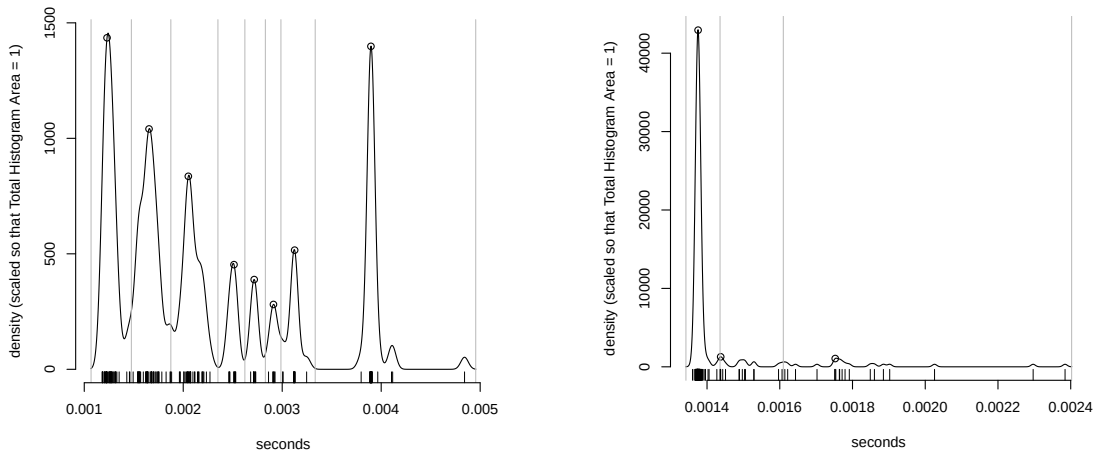
Figure 3.9a shows the clusters found by Algorithm 3.2 when applied to the data in Figure 3.8. Clearly the clustering algorithm finds the main features in the service time data, though the cluster boundaries are not easily defined. For comparison, Figure 3.9b shows the equivalent clusters when `EnterpriseXACML PDP` is used instead of `SunXACML PDP`. In this case there are only 3 clusters, with most requests being assigned to the first cluster.

The plots for cases using `inisherk` instead of `bear` and `multi22` instead of `single` are qualitatively similar to the Figures shown, indicating that the gross features (e.g., number of clusters and their sizes) of the service time distribution are determined by the PDP implementation.

3.4.6.3 Case study 1: Comparison

Given the experimental setup from Section 3.4.6.1 and corresponding measurements from the testbed, namely

- the decision made by the PDP (`decision`)
- request type (1 to 200, `ind`)
- the service time



(a) Using `SunXACML PDP` (b) Using `EnterpriseXACML PDP`

Fig. 3.9 Clustering `SunXACML PDP` and `EnterpriseXACML PDP` service times. Service times for `SunXACML PDP` have more clusters (9) compared to `EnterpriseXACML PDP` (3). Note that the host (`bear`), policies and requests are the same, so the differences in request service time distribution are wholly attributable to the choice of `PDP`.

- the cluster index

we performed an Analysis of Variance to determine the contributions of factors and their interactions to the overall variance, see Table 3.4. We note that all the identified factors, and their interactions, are very significant, except for the interaction between `host` and `reqGrp`.

Given this evidence that the ANOVA model appears significant, we proceeded to an Analysis of Means. Reviewing Tables 3.5 and 3.6, `inisherk` outperforms `bear` and `EnterpriseXACML PDP` outperforms `SunXACML PDP`, respectively.

Table 3.7 indicates that service times for `multi22` requests are slightly less than those for `single` requests, but more detailed study (i.e., white box testing) would be needed to discover why this might be true.

Interestingly, Table 3.8 indicates that service mean times differ greatly by decision, with `NotApplicable` decisions taking longer to make. This suggests that (some) PDPs might "fall through" to that decision only if other decisions are not available. It also suggests that there is a strong case for keeping policy sets up to date to avoid such (long service time) edge cases.

We present the 2-level interaction results in Tables 3.9, 3.10 and 3.11. Generally, they confirm the overall main effects analysis above, but there is one anomalous result in

	Mean Sq	F value	Pr(>F)	Code
host	2.8e-04	1.91e+04	0	***
pdp	4.2e-05	2.83e+02	0	***
reqGrp	1.9e-06	1.32e+01	2.99e-04	***
decision	4.6e-05	3.14e+02	0	***
ind	5.5e-07	3.74e+00	0	***
host:pdp	5.9e-06	4.03e+01	3.00e-10	***
host:reqGrp	5.3e-08	3.60e-01	0.54e+00	
pdp:reqGrp	2.9e-05	1.95e+02	0	***
host:pdp:reqGrp	2.7e-06	1.85e+01	1.78e-05	***
Residuals	1.5e-07			

Table 3.4 Analysis of variance relating (measured) Service Times to experimental factors `host`, `pdp`, `reqGrp`, `decision`, `ind`. A blank Code implies $P > 0.05$ (not significant) and Code = '***' implies $P < 0.001$ (very significant).

	bear	inisherk
	1.8e-03	9.5e-04
rep	800	800

Table 3.5 Comparison of service times for Hosts `bear` and `inisherk`.

	SunXacmlPDP	EnterpriseXacmlPDP
	1.5e-03	1.2e-03
rep	800	800

Table 3.6 Comparison of service times for PDPs `SunXacmlPDP` and `EnterpriseXacmlPDP`.

	single	multi22
	1.4e-03	1.3e-03
rep	800	800

Table 3.7 Comparison of service times for Request Groups `single` and `multi22`.

	Deny	NotApplicable	Permit
	1.3e-03	2.1e-03	1.1e-03
rep	1244	136	220

Table 3.8 Comparison of service times for Decisions `Deny`, `NotApplicable` and `Permit`.

		PDP	
		SunXacmlPDP	EnterpriseXacmlPDP
host	bear	2.01e-03	1.56e-03
	inisherk	1.05e-03	0.840e-03

Table 3.9 Comparison of service times for pdp:host interactions.

		Request Group	
		single	multi22
host	bear	1.83e-03	1.75e-03
	inisherk	0.970e-03	0.920e-03

Table 3.10 Comparison of service times for request Group:host interactions.

that the mean service time for EnterpriseXACML PDP on `inisherk` is greater than it is on `bear`. Again, further study would be needed to discover why this is the case.

Summarising, collecting measurements from a balanced full factorial design such as this can provide insight into PDP performance because the researcher is able to control experimental conditions in `STACS`.

3.4.6.4 Case study 2: Prediction

In this scenario, we model the case where the PDP has reached a steady state ($\rho = 0.5$ is a constant), then 25% of request types suddenly have triple ($3\times$) their arrival rate, which is maintained over a prolonged period and then returns to its previous $\rho = 0.5$ level. Thus $\lambda^{(\text{overload})} = 0.25(3-1)\lambda^{(\text{base})} = 0.5\lambda^{(\text{base})}$, so the overload factor is $\gamma = 0.5$. While this is an idealised scenario, it might represent a situation where there is a sudden rise in access control requests on the hour as project groups attempt to initiate group chat sessions across a matrix-structured organisation.

		Request Group	
		single	multi22
host	SunXacmlPDP	1.70e-03	1.36e-03
	EnterpriseXacmlPDP	1.10e-03	1.30e-03

Table 3.11 Comparison of service times for request Group:pdp interactions.

To make the scenario more concrete, we need to choose how the additional requests are distributed across the clusters. We consider two such request distributions: *low* where the extra requests are skewed towards lower service times hence the lower clusters, and *high* where they are skewed in the opposite direction. For the free parameters in the model, we choose

$$\begin{aligned} \alpha_j^{(\text{overload:lo})} &= \frac{n-j+1}{\sum_{i=1}^{n} i} \\ \alpha_j^{(\text{overload:hi})} &= \frac{j}{\sum_{i=1}^{n} i} \end{aligned} \tag{3.15}$$

Substituting Equations 3.14 and 3.15 in Equation 3.13 gives the required explicit expression for the overload process contributions $\rho^{(\text{overload:lo})}$ and $\rho^{(\text{overload:hi})}$.

The OPNET simulation model can also be extended to include overload arrival profiles equivalent to Equations 3.13. Note that the simulation results (indicated by points) and explicit results (indicated by lines) in Figure 3.10 agree well and that the distribution of overload requests affects the overall load experienced by the PDP. Equivalent plots for `EnterpriseXACML PDP` showed smaller differences between the favourable and unfavourable overload request profiles, due to that PDP's different clustering behaviour.

3.5 Measuring performance and resource usage

If the focus changes to prediction of the access control system performance for single requests, simulation of large numbers of request arrivals is no longer necessary. The emphasis changes to designing experiments where measurements are taken and statistical procedures (ANOVA and related analysis techniques) are used to analyse those measurements to understand how the service time changes in response to changes in factors such as resource capability, PDP choice, request type, etc.

As an example of such an analysis, we consider a comparison between the established `SunXACML PDP` and a prototype (incomplete and far from production quality) PDP developed by a colleague to see whether a PDP using a non-blocking I/O approach (Griffin et al., 2011) could have higher performance than the existing reference `PDP`. This comparison is described in this section.

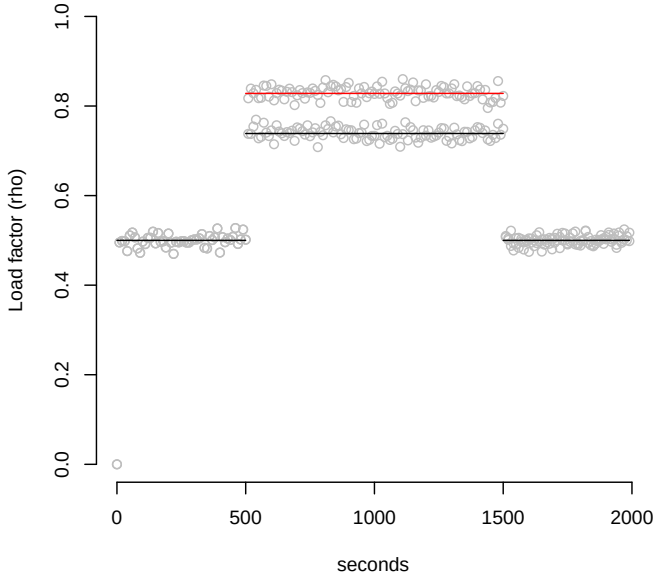


Fig. 3.10 Comparing server utilisation for 2 different overload request profiles. Unfavourable (mostly high service time) overload requests: $\max(\rho) > 0.8$ are represented by a red line. Favourable (mostly low service time) overload requests: $\max(\rho) < 0.8$ are represented by the black line. The difference is due to the difference in the request mix.

feature	Traditional (JEE)	New Generation (Node.js)
concurrency	single-threaded but with non-blocking I/O implemented using callbacks	multi-threaded but with locks and transactions to manage shared state
container	JVM (established; well-tested; stable)	Node.js (new; becoming more stable)
development language	originally Java but now many languages compile to JVM bytecode	Javascript (or related languages like CoffeeScript and TypeScript that compile to Javascript)
serialisation format	traditionally binary or XML; more recently JSON, YAML etc. added	Native support for JSON; support for other formats using Node packages

Table 3.12 Traditional versus more lightweight modern approaches for building request handling systems (such as web services)

3.5.1 The Case for a PDP using newer technology

Griffin et al. (2012) describes how applications using new combinations of architectures, languages, concurrency models and frameworks can perform very well compared to more traditional approaches. The work contrasts the traditional Java EE approach with one of the newer alternatives, as summarised in Table 3.12.

At least for "simple" web services, Node.js (Tilkov and Vinoski, 2010) provides an environment to execute server-side Javascript. Griffin et al. (2011) shows that this architecture-language-environment combination is capable of very impressive performance. However, it was not clear whether such performance gains were still achievable when the request triggers policy evaluation at the PDP. Therefore, a colleague (Dr. Leigh Griffin) developed a prototype PDP which he named `njsrpdp` with the following characteristics:

- the PDP, PEP and PRP were written in Javascript;
- he translated the `continue-a` policies and the associated `single` requests to JSON;
- he uploaded and saved the policies in a `redis` key-value store and optimised it for the fast retrieval of policies;

- the PEP served many of the functions of XTS and XACML Testing Client (XTC) of `STACS`.

Our hypothesis was that the following features lead to improved PDP evaluation performance:

- Policies and requests should be encoded more efficiently
  - policies and requests should be relatively terse, to reduce the string handling overhead per request;
  - policies and requests should be encoded in a way that minimizes the parsing overhead;
  - policies should be directly implementable.
- PDP implementations should be more efficient
  - policies and requests should be stored in ways that make retrieval more flexible and efficient;
  - the PDP should scale outwards, to enable more efficient use of available resources.

Before proceeding to describe the performance experiments, it is necessary to state the following caveats concerning `njsrpdp`:

- it does not support all the features of XACML 2.0 policies, just those that were essential for the `continue-a` policies and associated `single` requests. As an example, it ignores any `<Condition>` elements.
- it is concerned only with making the correct decision for those policies with those requests: other features of a PDP (such as error handling or even working with policies that have been nested to an arbitrary depth) are not considered.

One of the consequences of these restrictions is that the fit between the Javascript code and the JSON policies and requests is almost seamless. Indeed, Griffin et al. (2012) describe the combination as being "friction-free": the parsing of policies and requests is trivial and policy decisions are made by looking up keys in a tree.

While some of the performance gains would be lost if the PDP had all the features of a XACML PDP, we believe it would still be highly competitive with existing PDPs. In that regard, work has completed on the development of a Javascript/Node.js/JSON

PDP that passes all the XACML 2.0 conformance tests (Kuketayev, 2005). We have rerun the experiment described below with the full PDP and the results are broadly similar, although there are some features, e.g., relating to `PDP` memory usage and synchronisation, that are new. We intend to investigate these and other issues by extending the study using the greatly enhanced framework presented in Chapter 4.

Listing 3.1 JSONPL (JSON Policy Language) Policy Excerpt. The original XACML-encoded policy fragment had 1473 characters versus 454 characters (including generous whitespace, to aid readability) for the equivalent policy in JSONPL encoding. The whitespace to the left of the policy fragment is significant: this degree of indentation indicates the highly nested structure of the original policy.

```
                        "Policy":{
                            "id":"RPSlist.7.0.1",
                            "target":{
                                "subjects":{
                                    "subject":{
                                        "role":"admin"
                                    }
                                },
                                "resources":{
                                    "resource":{
                                        "isPending":"false"
                                    }
                                },
                                "actions":{
                                    "action":{
                                        "action-type":"write"
                                    }
                                }
                            },
                            "rule":{
                                "id":"RPSlist.7.0.1.r.1",
                                "effect":"permit"
                            }
                        }
```

Listing 3.1 is a fragment of the `continue-a` policy set after it was converted manually from XACML to JSON. Similarly, Listing 3.2 is the result of manually converting the the first request in the `single` request set from XACML to JSON.

Listing 3.2 JSON Request example, converted manually from `1-req.xml` from the `single` requests associated with the `continue-a` policy set.

```
{
    "subject" : {
        "category" : "access-subject"
       ,"role" : "pc-chair"
    }
   ,"resource" : {
        "isPending" : "false"
       ,"resource-id" : "DEFAULT RESOURCE"
    }
   ,"action" : {
        "action-type" : "write"
    }
}
```

Two features are apparent: the structure and conditions of the `continue` policy set and `single` request set are maintained, and the JSON policy encoding is much less verbose than the corresponding XML encoding. Thus the two policy encodings express the same rules when interpreted by the relevant PDP. Thus providing these policies and requests to `njsrpdp` is semantically equivalent to submitting their XACML equivalents to more traditional PDPs like `SunXACML PDP` and `EnterpriseXACML PDP`.

3.5.2 Comparison of PDPs

Experiments were performed to compare the JSON/Node.js/Redis implementation described above with more traditional XACML/Java implementations of `SunXACML PDP` and `EnterpriseXACML PDP`. A set of XACML policies and their related requests was chosen and were translated manually to their JSONPL equivalents. The two Java-based PDP implementations were placed in `STACS` (Butler et al., 2011) so that service times per request could be recorded in a repeatable fashion. The prototype Node.js implementation was instrumented in the same way, taking advantage of the Node.js eventing model to collect service times based on the same triggering events that were used in `STACS`:

- PDP Policy Read *start*;
- PDP Policy Read *end*;

Table 3.13 Service time measurements and their context.

Name	Type	Possible values
policy	Common	`continue`
reqGrp	Common	`single`
host	Factor	`bear`, `inisherk`
pdp	Factor	`SunXACML PDP`, `EnterpriseXACML PDP`, `njsrpdp`
duration	Response	Numeric

- Request *arrives at* PDP;
- Response *leaves* PDP.

A simplified queueing discipline was employed, namely, when response n from the PDP arrived at the PEP, it triggered the submission of request $n+1$ from the PEP to the PDP. This sequential processing was easily achieved in `STACS` using loops and in the `njsrpdp` harness using callbacks. The entire experiment was replicated $N_{\text{rep}} = 100$ times, in random order, for each set of `host` $\times$ `pdp` conditions.

The measured service time data was standardized to use the same labels and time units to ensure that data features were consistent between `STACS` and non-`STACS` sources.

The factors considered in our main experiment are shown in Table 3.13. The `continue` policy and `single` request group are published (in XACML form) as part of the test suite for XEngine (Liu et al., 2008) and were translated to JSON format as described earlier. This policy set and associated requests was used in the experiments and models access control rules and requests for a Conference Paper Management System. While that domain does not require microsecond evaluation times, the policy set contains reasonably complex business rules such as separation of duties constraints and other features representative of real-time corporate communications. The two `host` instances were Intel 64-bit dual-core machines, each with 2GB RAM but differing in other computing resources, running Ubuntu 11.04.

The primary experiment compares `njsrpdp` with two existing XACML PDP implementations. The secondary experiment examines *how* `njsrpdp` achieves increased performance.

Table 3.14 Analysis of Variance: host, pdp, host:pdp effects are very significant—α probability underflows machine epsilon ε.

	Df[a]	SumSq	MeanSq[b]	F value[c]	Pr(>F)
host	1	1.97e-05	1.97e-05	2.24e+04	$< \varepsilon$
pdp	2	1.15e-04	5.75e-05	6.55e+05	$< \varepsilon$
decision	2	1.00e-09	5.00e-10	5.14e-01	0.60
requestIndex	190	1.61e-07	1.00e-09	9.67e-01	0.61
host:pdp	2	7.50e-06	3.75e-06	4.27e+03	$< \varepsilon$
Residuals[d]	954	8.37e-07	1.00e-09		

[a](Number of) degrees of freedom
[b]SumSq/Df
[c]F ratio: MeanSq/MeanSq_Residuals
[d]Other, unspecified factors

Table 3.15 Analysis of Means: host `inisherk` has better performance than `bear`.

host	bear	inisherk
time	6.3e-04	3.7e-04
#replicates	576	576

3.5.3 Comparison experiment 1: njsrpdp vs. its peers

Figure 3.11 shows histograms of the service times for `SunXACML PDP`, a *reference* Java-based XACML PDP, compared with the service times for `njsrpdp`, the implementation introduced in §3.5.1. The influence of the `host` and `pdp` factors can be seen clearly. Indeed, `njsrpdp` has noticeably better performance when other factors are equal. The `EnterpriseXACML PDP` has service times that are generally higher than those in the reference `SunXACML PDP`; see Table 3.16. One possible explanation is that the `EnterpriseXACML PDP` reads the schema file and verifies each request against the schema before attempting to parse the request. By contrast, the `SunXACML PDP` does not check each request beforehand: if the request is faulty, an exception occurs.

Table 3.16 Analysis of Means: PDP `njsrpdp` has better performance than the other PDPs.

pdp	`SunXACML PDP`	`EnterpriseXACML PDP`	`njsrpdp`
pdp	5.56e-04	8.61e-04	9.3e-05
#replicates	384	384	384

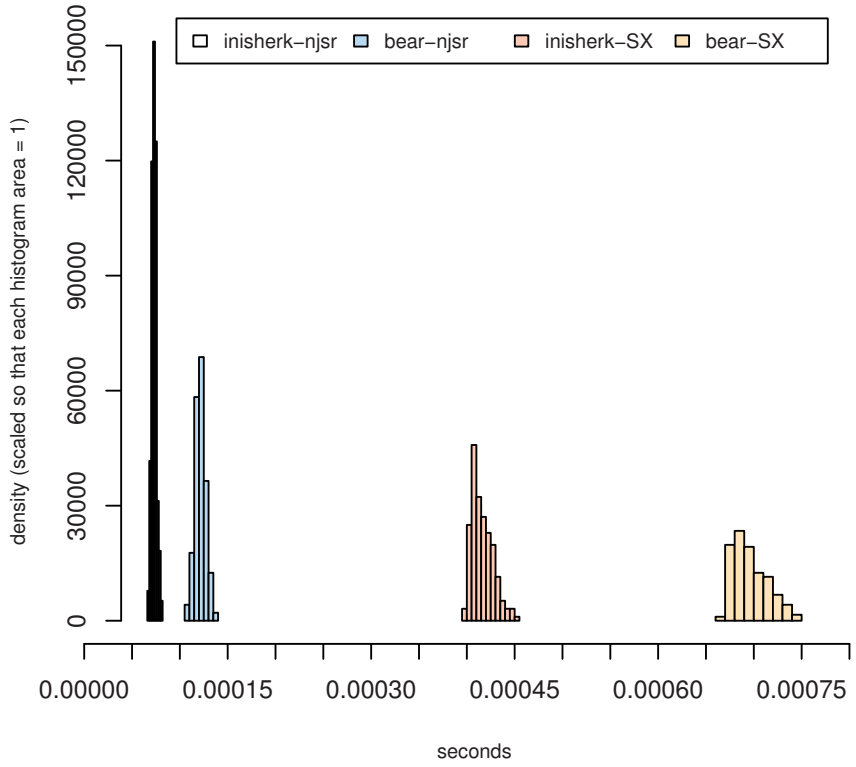


Fig. 3.11 Comparative service time histograms for hosts `bear` and `inisherk` and PDP implementations `SunXACML PDP` and `njsrpdp`, for Scenario 1A.

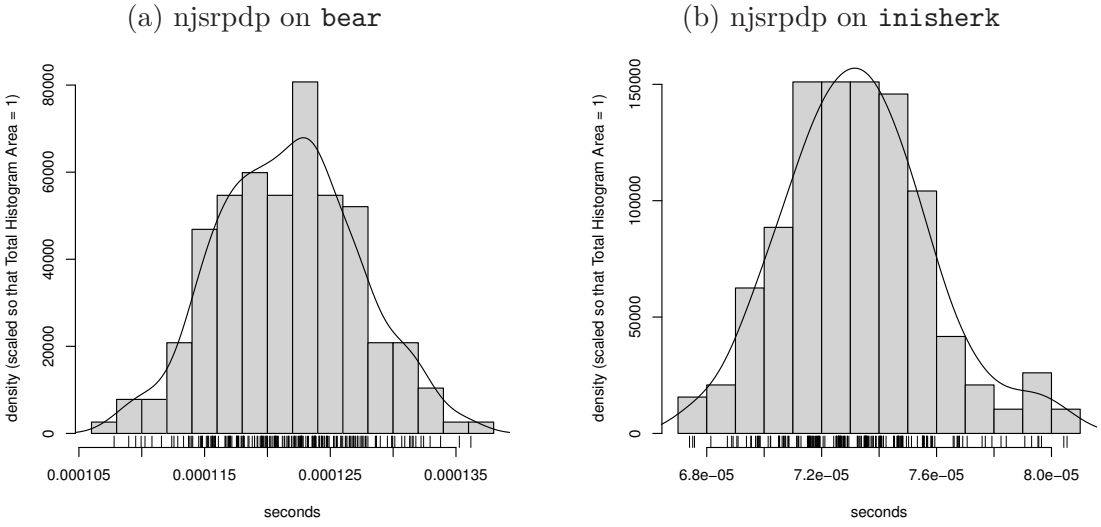


Fig. 3.12 njsrpdp request service times on hosts `bear` and `inisherk`.

The Node.js/Redis prototype PDP implementation, labeled `njsrpdp` in Figure 3.12 has the following performance features:

1. The mean service time per request is much less (one sixth that of `SunXACML PDP`, one eighth that of `EnterpriseXACML PDP`), see Table 3.16.
2. The performance profile for `njsrpdp` is bell-shaped; for `EnterpriseXACML PDP` it is approximately uniformly distributed; for `SunXACML PDP` it is a skewed mixed distribution.
3. The implementation on the two hosts shows a similar profile (see Figure 3.12) though with different performance levels, see Table 3.15 because `inisherk` has a faster CPU and more L1 cache. This suggests that performance scales vertically on a single host and also that the performance profile and observations are *reproducible*.

It should be noted from Table 3.14 that these differences are statistically significant (Hothorn and Everitt, 2009) and hence are highly unlikely to arise by chance. The challenge is to show how the design principles outlined in §3.5.1 and implemented in the `njsrpdp` prototype contribute to the statistically significant performance improvements summarized in Table 3.16.

The system resources used by the JSON and XACML implementations were captured using `dstat`, which collects resource statistics (cpu, memory, disk usage, etc) on a timed basis while the experiments run in the testbed. Figure 3.13 shows that `njsrpdp` uses far less CPU (10% versus 60%, say). The cpu `wait` time is generally low, suggesting that both Node.js and the JVM are quite efficient. However, the `user` cpu cycles are much greater for the Java/XACML implementations. The CPU has to work much harder to evaluate policies in Java/XACML PDP implementations. This is consistent with observations elsewhere in building scalable web applications (Griffin et al., 2011). Furthermore, the `idle` cpu usage is much higher for `njsrpdp`, suggesting there is much more capacity available for increased throughput.

The memory usage was also recorded and shown in Figure 3.14, supporting the contention that `njsrpdp` makes particularly efficient use of computing resources, including 35% less memory.

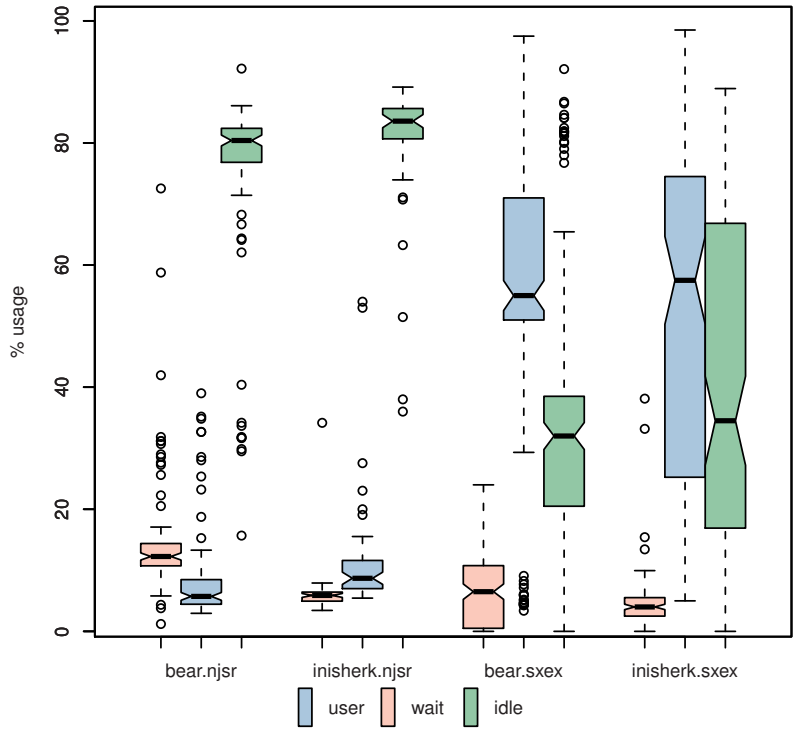


Fig. 3.13 CPU usage for selected host × pdp combinations. Hosts are `bear` and `inisherk` and PDPs are `SunXACML PDP`, `EnterpriseXACML PDP` (collectively labeled `sxex`) and `njsrpdp` (labeled `njsr`). If memory is labelled 'user', it is assigned to a user process; if it is labelled 'idle' it is available for other processes; if it is labelled 'wait' it is neither assigned to a user process, nor is it available for use by other processes. Generally: more wait = less efficient, more user = more busy on user tasks, more idle = more scope for scalability if those resources can be used.

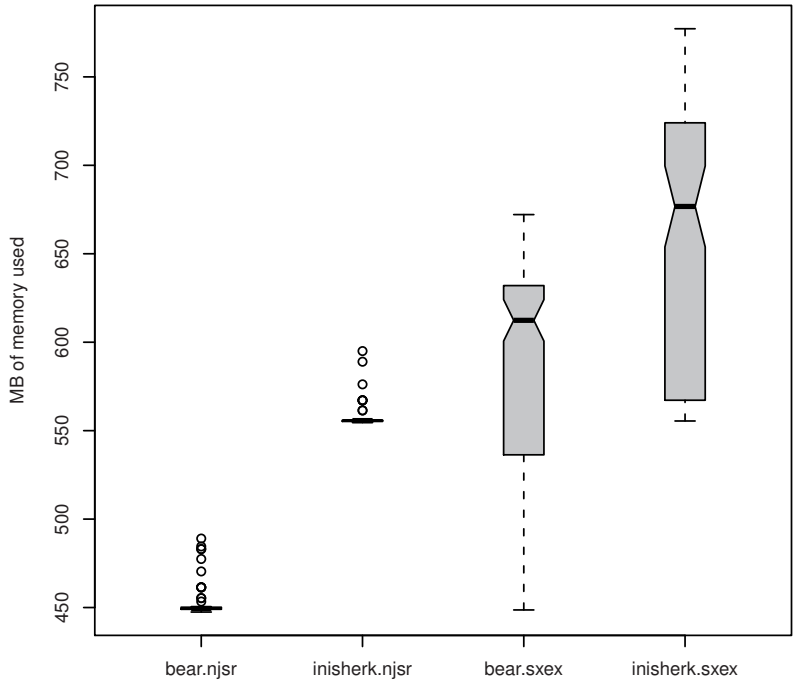


Fig. 3.14 Memory usage for different host × pdp combinations. It can be seen that `SunXACML PDP` hods onto much more memory than `njsrpdp`. However, preliminary results from our evaluation of a full XACML 2.0-conformant PDP implemented in Node.js (contrasted with the incomplete `njsrpdp` prototype presented in this Figure) suggests that low memory usage is not typical of Node.js applications in general.

Table 3.17 Scenario conditions

	Manually generated Policies	Auto generated policies (bloated)
Manually generated Requests	Scenario 1A	Scenario 1B
Auto generated requests (bloated, prepared)	Scenario 2A	Scenario 2B
Auto generated requests (bloated, on the fly)	Scenario 3A	Scenario 3B

3.5.4 Comparison experiment 2: What are the benefits of terse policies and/or requests?

The results in §3.5.2 indicate that `njsrpdp` performance is significantly better than either of the JVM-based PDPs. There are at least two possible reasons for this improvement:

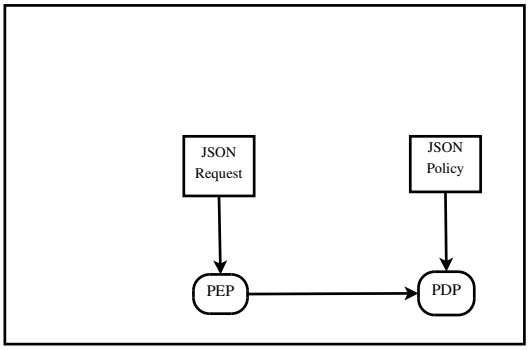
1. the Node.js-based PDP could be more efficient;
2. the JSON-encoded policies and requests are shorter than their XML counterparts

A further set of experiments was conducted to determine how much of that improvement is due to each of the two possible causes above. Six experimental scenarios are considered as described in Table 3.17 and were used to compare the effects of different policy and request formulations for a given PDP (in this case, the `njsrpdp` prototype).

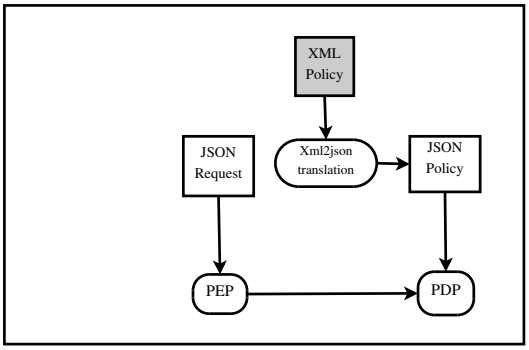
Referring to Figure 3.15, we see the main features of the scenarios to be compared with each other. Summarising,

- the A scenarios formulate the *policies* as JSON in ways that are "optimized" for evaluation performance, while the B scenarios use the policies as translated by a generic XML to JSON converter.
- the "1" scenarios formulate the *requests* as JSON in ways that are "optimized" for evaluation performance

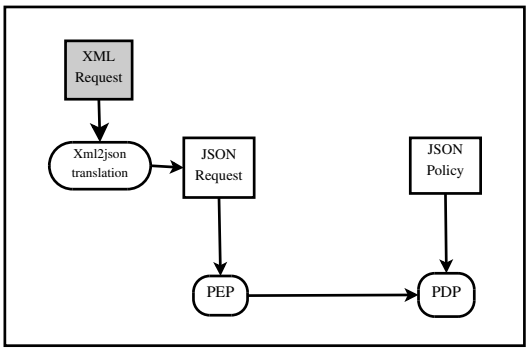
(a) Scenario 1A: Manual×Manual



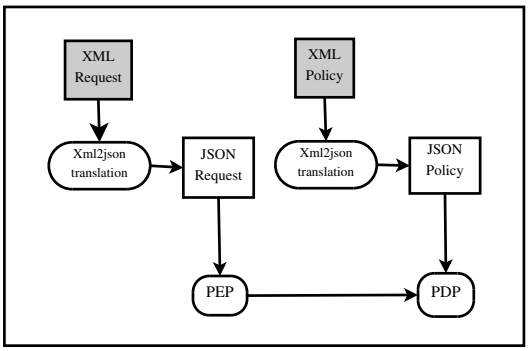
(b) Scenario 1B: Manual×Auto



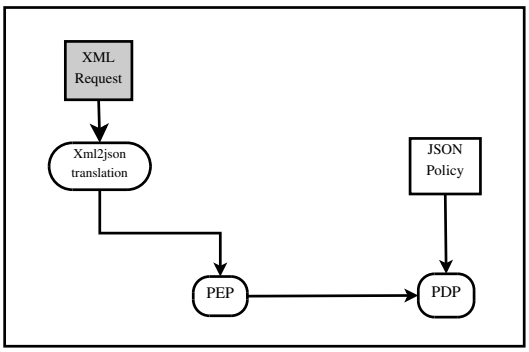
(c) Scenario 2A: Prepared-Auto×Manual



(d) Scenario 2B: Prepared-Auto×Auto



(e) Scenario 3A: Inflight-Auto×Manual



(f) Scenario 3B: Inflight-Auto×Auto

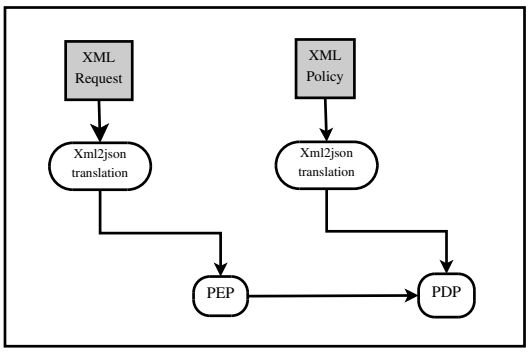


Fig. 3.15 `njsrpdp` policy×request scenarios; scenario conditions are defined in Table 3.17.

- the “2” and “3” scenarios use the requests as translated by a generic XML to JSON converter. They differ in respect of when the translation occurs: *before* reaching the PEP for “2”, or within the PEP itself for “3”.

Note that the experimental conditions in Scenario 1A are those used when comparing `njsrpdp` with the `SunXACML PDP` and `EnterpriseXACML PDP`, see §3.5.2.

In summary, all of the policies and request artifacts were originally encoded as XACML and so they benefit from the XACML ecosystem. However the artifacts are converted to JSON by different methods and at different stages of policy evaluation. The performance improvements arising from each research contribution can be estimated by comparing the timing results.

Scenario 3 incurs serious overheads. Firstly, the converter requires 83% of the total time needed to make the access decision. Secondly, translation introduces a large object that needs to be maintained at the top of the callback chain. When the PDP evaluates and wishes to pass its decision to the PEP it must “walk” back up the callback chain. The top level callback needs to retain a link including the context of the request and its arguments throughout the whole chain. By placing such a large object in the top callback, translation imposes greater overheads down the callback chain, so the computation time is increased. Therefore the next step is to investigate how the system would perform if the penalty for translation, which increases evaluation time in two ways, were removed.

By pre-translating the requests the overhead incurred in translating the requests at run time is removed as well as the added overheads in the callback chain. The challenge becomes that of guaranteeing safe and accurate policy evaluation. One complication is that, depending on the XML schema, the ordering of some child elements may be unspecified. Consequently the position of sibling child elements in policies within the same policy set can be different. A XACML PDP’s XML parser has no difficulty in this regard but the translating program makes no allowance for consistency in the generated JSON. Thus `njsrpdp` has to account for this, handling all ordering permutations so as to operate correctly. While these problems also occur in Scenario 3, the additional translation overhead masked this feature. A minor performance gain was identified when using a combination of pre-translated JSON requests and optimized (JSONPL-formatted) policies, as there is less overall bloat.

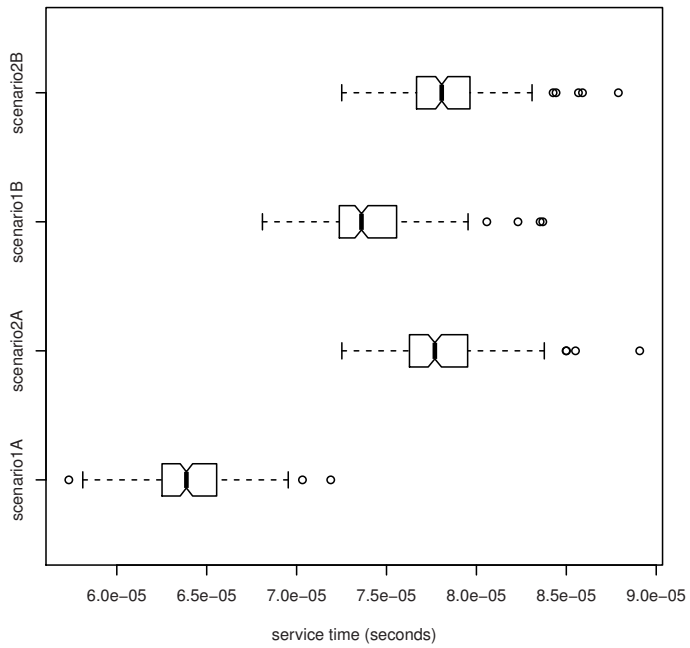


Fig. 3.16 Service times for Scenarios 1A, 1B, 2A, 2B. The Scenarios are defined in Table 3.17 and shown schematically in Figure 3.15. Scenario 1A (manual policies, manual requests) gives the best performance, with lower performance when the policies are auto-generated. Scenario 2A and Scenario 2B show very similar performance to each other, so if requests are auto-generated, there is little to be gained from manually generating the policies.

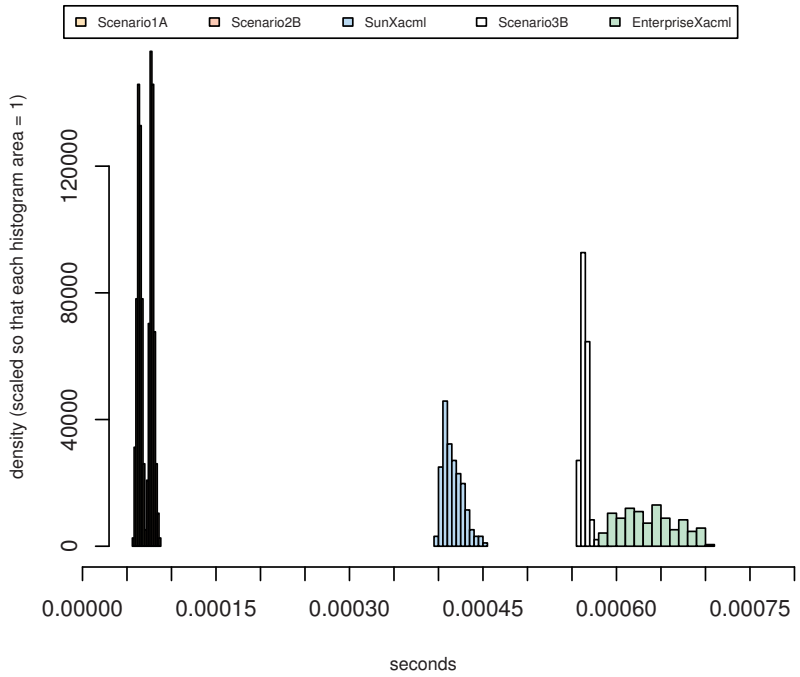


Fig. 3.17 Service time comparison. Ranked in decreasing order of performance (left to right in the figure above), they are: `njsrpdp` Scenario 1A, 2B; `SunXACML PDP`; `njsrpdp` Scenario 3B, `EnterpriseXACML PDP`.

The boxplot in Figure 3.16 indicates that the best performance is obtained when optimized (JSONPL) policies and requests are used (Scenario 1A) and that performance degrades as bloated/more complex automatically translated JSON is used to represent polices and requests (Scenario 2B).

The service time histograms in Figure 3.17 show how different `njsrpdp` scenarios compare with "traditional" PDP implementations. Clearly there is no net performance benefit of the JSON implementation when requests are translated on the fly: mean services times for `njsrpdp` Scenario 3A are greater than those for `SunXACML PDP`, but `njsrpdp` has much greater performance potential (see Scenario 1A).

Table 3.18 confirms that factors such as scenario type, decision and request type are significant when modelling service times. The comparison of mean service times for each Scenario in Table 3.19 shows how different Scenario 3 is to the others, and how much request format optimization affects performance (compare Scenario 1A and 2A).

Table 3.18 Analysis of Variance for Scenario service times

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
scenario	5	6.4e-05	1.27e-05	7.43e+05	$< \varepsilon$
decision	2	1.0e-09	4.00e-10	2.54e+01	1.8e-11
requestIndex	190	8.0e-09	0.00e+00	2.32e+00	$< \varepsilon$
Residuals	954	1.6e-08	0.00e+00		

Table 3.19 Mean service times for each of the Scenarios

1A	6.4e-05	*1B*	7.4e-05
2A	7.8e-05	*2B*	7.8e-05
3A	57.8e-05	*3B*	56.5e-05

3.6 Extending STACS

`STACS` has enabled us to undertake rich performance experiments, as described earlier in this Chapter and published in (Butler et al., 2010, 2011; Griffin et al., 2012). However, the comparison of `SunXACML PDP` with `njsrpdp` highlighted the assumption that any PDP under test could be deployed in the same JVM as the XTS universal PEP component of `STACS`. This restriction is being removed as work has started on distributing `STACS` across more than one host in a client-server architecture. This will enable researchers to measure service times of distributed access control systems directly, in addition to the current procedure where service times are measured on a single PDP and the performance of a distributed access control system is estimated by means of measurement-based simulation. Other extensions have already been implemented, including:

- All factors defining an experiment, such as choice of PDP, policy encoding format etc., are now modeled as *properties* and externalised in text files. `STACS` reads those configuration files and the same software can thus handle a huge variety of different measurement scenarios without changing `STACS` code.
- `STACS` v1.0 stored the performance results in text files on the file system, with the context of a result depending on both a label in a given results file, as well as the full path to that results file. While that was convenient for small experiments, it soon became cumbersome as experiments grew more complex. It also made comparison of JVM-based and non JVM-based PDP implementations more

difficult because of a lack of standards. Therefore, the measured service times are now stored in a relational database, and the chosen table schema makes it easy to handle different experimental scenarios with minimal change to STACS.

The two enhancements above greatly ease the task of a researcher who wishes to apply STACS to measure access control service times in new scenarios, such as those that will be discussed in Chapter 4.

3.7 Summary

This chapter describes STACS, our response to the perceived need for a testbed in which to perform access control performance experiments. We have seen how request service times collected using STACS have been used to compare the performance of different PDPs, even when the underlying technology and policy formats are dramatically different. One of the most interesting features of the service time measurements is the fact that the service time distribution for a given PDP appears to be a mixture of simple, overlapping distributions associated with emergent request clusters. This behaviour is indicative of the internal working of that PDP. Looking beyond the analysis of infrequent request arrivals, to cases where some request queueing occurs, we have also seen how service time measurements from STACS can be used as the basis of measurement based simulation. One experiment considered the queueing effects of keeping the request arrival rate constant and varying the request mix, the other considered the effect of keeping the request mix constant but increasing the arrival rate, and using techniques such as `proportional thinning` to manage the size of the queue. All of these experiments and simulations serve to validate STACS as an access control performance testbed.

However, although STACS provides a useful basis for performance experimentation, the analysis in this chapter is not sufficient to prove the *external validity* of STACS because it is not based on policies and requests from our domain of interest, which is enterprise communication control. This is because suitable policy and request sets have proven difficult to obtain. Therefore, the primary contribution of Chapter 4 is a set of algorithms for generating representative policies and requests consistent with each other and with a specified domain model. The secondary contribution is that the generated policies and requests have free parameters that enable them to match

different sizes of domain. Consequently, it is possible to perform experiments in which uninteresting factors are controlled (set to a constant value) and the domain semantics are maintained, while allowing the researcher to vary interesting factors such as the domain size.

Note that the `STACS` framework can be extended to other client-server performance experiments, by adding suitable adapter and scenario run classes. The biggest challenge would be to map the factors of that scenario in that domain to properties that can be used by the adapter and run classes in `STACS`.

Chapter 4

DomainManager: A domain model and tools to configure STACS

Table 4.1 Research questions addressed in Chapter 4

ID	Question
RQ1	How can access control evaluation performance be measured for use in performance experiments? – What form does the service time distribution take? – What simulations can be performed to explore the effect of different request arrival patterns? – What analysis can be performed when the systems under test use different languages, frameworks and encodings?
RQ2	How can domain models be specified and used to express enterprise access control scenarios? – How can different variants of domain models be specified in a flexible and easy to use way? – How can access control evaluation performance be compared at different domain sizes?
RQ3	How can the data from performance experiments be used to understand and predict access control evaluation performance? – What types of exploratory data analysis are suitable for the performance experiments? – What are the steps needed to build statistical models predicting access control performance?
RQ4	What are the main factors affecting access control evaluation performance? – What are the effects of PDP choice, domain size and resources? – What are the effects of domain size, policy and request characteristics?

4.1 Introduction

In Chapter 3 and in (Butler et al., 2010, 2011; Griffin et al., 2012), we described `STACS`, a software platform to investigate the performance of a PDP implementation by measuring the service times per request. While the experiments proved that `STACS` provided a principled means of evaluating access control performance under controlled conditions, the public domain policies and requests that were used were not representative of a typical real-world deployment. Hence, only limited claims could be made in relation to access control performance.

Policies and requests relating to enterprise access control have not been made publicly available, so there was a need to generate such artifacts. Also, access control performance is commonly believed to depend to some extent on the characteristics of the policy sets used to make those access decisions, so the policies (and their associated requests) became a subject for study in their own right.

An enterprise access control policy is a means of specifying the security properties that need to hold in an enterprise. Such security properties do not arise in isolation: they are in response to requirements that need to be captured as rules and expressed in terms of the enterprise itself. The rules are stated in terms of the enterprise stakeholders, the resources being protected and the activities that occur. Thus what is needed is a comprehensive domain model for enterprise access control, extensions to `STACS` to run more the more extensive experiments facilitated by this domain model, and an analysis component that has the ability to derive statistical insights from the new, more comprehensive measurement experiments.

The enterprise access control domain model (see §4.2) enables policy and request generation but an application is needed to manage that model by populating it with data and using the data in the model to generate policies and requests. In that regard we developed `DomainManager` to manage the enterprise access control domain model and to make the authoring of large policy sets easier. The key insight is that enterprise policies may often be very large, having structure in the form of implicit entity hierarchies, but their very size makes this structure hard to identify and exploit. Indeed, there are often patterns governing how policies are formulated by policy authors but the structure is often much less obvious when the policies are cast in a form suitable for deployment in a PDP. `DomainManager` provides a way to use these patterns, enabling its users to create and edit a smaller set of *template policies* and

then to automatically derive the full policy set and its corresponding example requests from the template set.

Secondly, we enhanced `STACS`, primarily to be able to handle the much richer scenarios supported by `DomainManager` and hence the much larger and more context-rich measurement sets that arise in each experiment (see §3.6).

One of the main advantages of the greatly enhanced domain model and scenario investigations supported by `DomainManager` and `STACS` is that the scope for statistical analysis and insight increases dramatically. First steps towards such an analysis component were described in (Butler et al., 2011; Griffin et al., 2012) and discussed in Chapter 3. However, after `DomainManager` was developed, more factors became available to explain the performance of the system under test, and this potential for richer statistical models offers greater opportunities and challenges than before. Given this requirement, we developed `PARPACS` to interpret the measurements (see Chapter 5). However, in the present Chapter, §4.6 describes a preliminary evaluation of `DomainManager` using the types of analysis (design plots, service time density plots and ANOVA) used in Chapter 3.

4.1.1 Policy authoring

The first step is to identify access control policies (modelled as *constraints* on behaviour) that meet whatever safety objectives are specific to the organization. This challenge is addressed in research on *policy authoring*, see Davy et al. (2013). Some of the issues to consider include:

- ensuring that the constraints are both necessary and sufficient. That is, they protect sensitive resources in all realistic contexts (Lampson, 1974) (and so the protection is *adequate*) but they do not prevent legitimate business operations and processes (and so the protection is *minimal*) (Egelman et al., 2010; Johnson, 2012);
- ensuring that the constraint set is *efficient* in the sense that constraint conflicts are few and preferably nonexistent. Some conflicts affect constraint coverage (described above). Other conflicts increase the size of the policy set because they result in *redundant* constraints, or inefficient representation of constraints as

policies, such as when n policies are used to represent $k, k \ll n$ independent constraints.

Since most policy authoring occurs offline, its effect on the time needed for *policy evaluation* is indirect. Some policy formulations have better evaluation performance characteristics than others, but knowledge of the effects on performance is generally not available when a change is made to a policy set.

4.1.2 Policy Generation approaches

There are (at least) three basic techniques that can be used to generate policies

1. **Overloading the XACML schema document**. XACML policies are specified in a dialect of XML. OASIS has published the XACML policy schema document (OASIS XACML-TC, 2005b). This schema document describes the main components of the language, namely policy sets, policies, targets, rules and conditions, and how they relate to each other. However XML schema documents can also define the "vocabulary" that can be used, e.g., lists of user names, shared resources, etc. Hierarchies can be modelled easily in the tree structure of the XML document. Using an XML editor, a local copy of the XACML schema document can be created, which can be augmented with lists of possible values (subjects and resources) maintained as suitable hierarchies and `TAXI` (Bertolino et al., 2007) can be used to generate policies from the augmented XML schema document.

2. **Using a domain-specific language optimised for specifying large policy sets**. A domain specific language (DSL) can be designed to specify a set of policies based on *hierarchies* of subjects and resources, with inbuilt iteration to facilitate creation of large policy sets. An editor for this domain specific language could be built and a converter/loader developed to store the policies in memory. The `SunXACML PDP` policy server implementation (Proctor, 2004) already provides classes to serialize in-memory policies in XACML format. Defining DSLs and building tools to convert between the DSL and the target language is a feature of Model/Language Driven Architectures. A DSL such as ALFA (see Chapter 2) could be used as a basis for such a "large policy set" DSL.

3. **Editing a property graph representation** Policy authoring and the related topic of policy conflict analysis have been studied by researchers over many years, and this is likely to continue. Bulk policy generation adds problems of scale since it becomes very difficult to understand a large set of policies, or even to specify them in a convenient and transparent way. The procedure introduced in § 4.4 uses linked graph models to generate policies.

Technique 2 requires more development effort than Technique 1 but is cleaner, since Technique 1 does not support the same degree of semantic control, in particular it does not model groups explicitly. Thus Technique 1 makes the creation of large group-based policy sets a more manual process. Technique 2 is attractive, given that significant tooling is available in `eclipse`, building upon the underlying language metamodel of eclipse to simplify the *manual* editing of DSLs. The difficulty is that semi-automated editing is required, and this requires much more complex manipulation of largely opaque eclipse metamodel objects. Technique 3 exploits the fact that policies connect rules to attributes, and the rules themselves are statements connecting attributes. Hence a graph representation is appropriate for representing policies, and graph operations (such as path finding, traversals, etc.) can be used to generate new policy statements. As with Technique 2, the working representation of the domain (i.e., the eclipse language metamodel for Technique 2 and the property graph for Technique 3) needs to be converted to a standard policy language such as XACML for evaluation purposes. However, eclipse (particularly its `Xtext` distribution) provides many tools to help convert from one language representation to another, usually when the EBNF representation of the grammar of both the source and target languages are specified. In the case of Technique 3, the source representation (a schema-free graph but with rigorously-enforced conventions regarding what data can and must be stored in each entity) does not have a formal grammar, but the target language (`XACML`) has an XML schema document that serves to define its syntax. Both Technique 2 and 3 require an explicit step that can be viewed as *deserialising* the internal representation to one that can be consumed by a PDP.

Technique 3) (i.e., inferring a large set of policies from a smaller set of template policies) is the approach that is featured in this dissertation.

4.1.3 Request Generation approaches

During normal operations within an organization, Principals issue access requests that the access control system must review against the constraints. For this to happen in a reliable fashion, policies "cover" a scenario by protecting some of the resources participating in that scenario.

To achieve both safety and performance objectives, it is essential that the generated requests should be consistent with the policies (so that policy coverage is adequate) *and* with the underlying processes for which access is requested.

For better performance, it is also necessary to consider the time required to assemble and encode the access request, particularly if it requires significant processing, e.g., if it needs to issue a database or web service call to provide the necessary details for matching to proceed. In that regard, to achieve better performance, many `PIP` implementations cache the attribute lookups and /or store them in memory, so that the lookup data is available to the PDP when evaluating each request.

It is essential to ensure that the policies and requests share the same basic concept vocabulary and structure, otherwise matching operations are ill-posed. In the case of incompatible `XACML` policies and requests, the PDP will typically return decisions such as `Not Applicable` or `Indeterminate`. The former is often caused by gaps in rule coverage; the latter by more complex exceptions that are caught by the PDP. With XACML 3.0, `Indeterminate` is split into three kinds that participate in different ways in combining algorithms, which facilitates finer control when handling edge cases and easier debugging of exceptions. In general, such non `Permit-Deny` decisions can be viewed as *warnings* concerning the "semantic quality" (domain coverage) of the policy set used by that PDP. Optionally, `PEP`s may be configured to map such exceptional cases to more acceptable decisions such as `Permit` or `Deny`. This makes life easier for the access-requesting clients, but care should be taken not to "hide" indefinitely the deficiencies in the policy set.

A further requirement is that the number of requests to be generated should be "large enough" that the generated requests provide a *representative* sample of requests to explore both the semantics (decision mix) and the performance characteristics of a given policy-PDP combination.

Two approaches were considered. Each can be considered complementary to the other:

Generating requests from first principles Weighted sampling of the people, resources and actions in the domain, generating combinations of these entities that can be interpreted as access requests.

Reverse engineering from the policies analysing the policies and inferring the types of requests that the policies are designed to permit and to deny.

The first approach has the potential to create much larger sets of requests than the second, but the challenge is to make them more representative of actual and expected behaviour. Indeed, our experiments showed that the reverse engineering approach generally gave more satisfactory results than the forward generation approach:

1. more `Permit` and `Deny` decisions relative to `Not Applicable` and `Indeterminate` decisions;
2. a more representative balance of `Permit` decisions compared to `Deny` decisions;
3. an adequate number of requests generated without the need to devise an arbitrary sampling strategy to make the set more manageable.

Approach 2) (i.e., reverse engineering requests from policies) is the approach that is featured in this dissertation.

4.2 Components of the domain model

The scope and complexity of the domain (policy-based access control in organizations) arises from the difficulty of representing resource sharing and the typical constraints on that sharing. Our response to this complexity is to create a set of linked representations to make the domain model more tractable.

The representations and their relationships comprising the domain model are outlined in Figure 4.1. The model is divided into subcomponents according to use case:

1. the *static* model acts as a foundation: it defines the fundamental vocabulary and relationships underlying the operations of a given organization;
2. the *policy* model defines the constraints and authorization rules enforcing the security model and objectives of the organization;

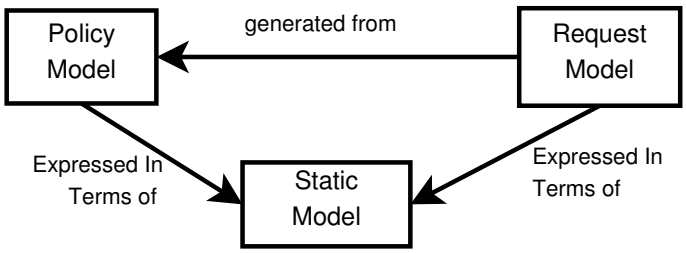


Fig. 4.1 Domain model overview: concepts and interactions. Note that the static, policy and request models are merged into a common model, with the entities retaining a `modelType` attribute which can be `Static`, `Policy` or `Request`.

3. the *request* model defines the instances of activity in the organization for which authorization is required.

An application is needed to maintain consistency between such loosely-coupled model components and to support operations like adding and updating data, querying and visualising the overall model. `DomainManager` fulfills that role.

`DomainManager` can be used to generate (suites of) access control policies. As such it has some features in common with more general purpose policy authoring tools. However, its inputs include simpler policies, which it uses to generate policies and requests according to parameter settings that control various characteristics of the resulting policies and requests. As such it complements other policy authoring tools. Work is under way to build a set of tools to streamline the production of its inputs, thereby bypassing the need for an external policy authoring system.

The first representation of the domain model that we used was *relational* in the sense that the entities are decomposed into simpler concepts and stored in tables, so that they can be reassembled using primary-foreign key relationships. The policy and request relational submodels are designed to capture knowledge about the domain with the same basic representation as is used in the XACML 2.0 metamodel. That is, the relational schema vocabulary and the structure of both the policy and context models is a transformation of the vocabulary and structures found in the XACML2 policy (OASIS XACML-TC, 2005b) and context (OASIS XACML-TC, 2005c) XML schema documents into an equivalent Third Normal Form (3NF) relational model. The static model (comprising Assets, Agents, Actions and their groups) is designed to act as a foundation for modelling business processes to which access control needs to apply.

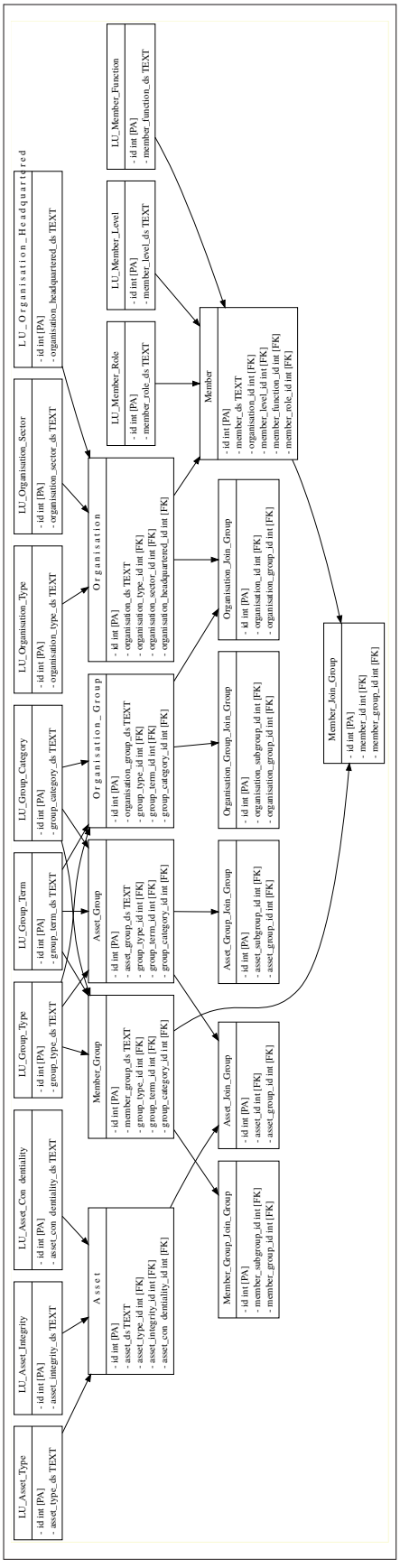


Fig. 4.2 Graphical representation of the *static* relational schema defining sharing operations in organization(s).

Table 4.2 Static model: Entities and Attributes.

Entity type	Entity name	Attribute

Agent	`organisation`	`headquartered`
		`sector`
		`type`

	`Member`	`function`
		`level`
		`role`

Asset	`Asset`	`confidentiality`
		`integrity`
		`type`

Action	`Action`	`type`

Group	`Group`	`category`
		`term`
		`type`

The lookup `LU_*` tables in Figure 4.2 are used to supply lookup data for the domain model, but the remaining static tables and also the policy and context (request and response) tables are not used to support the domain model—the property graph model described in §4.3 instead in the latest version of `DomainManager`.

4.2.1 The static model

The starting point for the *static* model is to define three hierarchies: for `Agent` (comprising `Member`s and `organisation`s), `Asset` and `Action`. Each hierarchy has a rich set of attributes, such as might be encountered in typical instances of resource sharing within and between organizations. Because the model represents static (or possibly slowly changing) entities only, instances of *events* (where an `Agent` performs an `Action` on `Asset`) are not represented in this model component.

Apart from `Action`, these entities can be combined into Groups. Furthermore, `Member` instances can be associated with `organisation`s, which differ from `MemberGroup` entities in that they are persistent rather than transient. For example, a `MemberGroup` could contain members of a team tasked with delivering a new product, but the `Member`s could belong to one or more `organisation`s.

Table 4.3 Static model: Attributes and example values

Entity name	Attribute	Example values
`organisation`	`headquartered`	'National', 'Same bloc', 'Other'
	`sector`	'Banking', 'Consulting'
	`type`	'Public', 'Private'
`Member`	`function`	'Finance', 'Technical'
	`level`	'Senior', 'Support'
	`role`	'Manager', 'Analyst'
`Asset`	`confidentiality`	'Low', 'Medium'
	`integrity`	'Low', 'Medium'
	`type`	'Webpage', 'Chat'
`Action`	`type`	'Document', 'Communication', 'Task'
`Group`	`category`	'Project', 'Supply chain'
	`term`	'Short', 'Medium'
	`type`	'Member', 'Asset'

Table 4.2 outlines the entities in the static model, together with their descriptive attributes. Of course, an instance of such an entity would have an associated `name` in addition to the attributes listed in the table. Table 4.3 indicates typical values for each attribute of each entity in the static model. The static model can be extended easily: it is trivial to add new values for existing attributes and new attributes can be added also with slightly more effort in `DomainManager`.

This static model is indicative of the enterprise communications domain, and is motivated by the original discussions with Cisco staff. This static metamodel is used by all `DomainManager` example policies and requests used in this dissertation. For other domains, such as access control in an online social network, the static model would contain different data and possibly attributes that are more pertinent to that domain. However the basic concepts remain the same: `Agent`s, `Asset`s and `Action`s have attributes that are relevant to access control policy specification and evaluation. The relational schema is inflexible in this regard, so an alternative representation is presented in §4.3.

4.2.2 The policy model

The most noticeable feature of the XACML 2 policy schema in Figure 4.3 is the presence of hierarchies of `Subject`, `Resource`, `Action` and `Environment` that are combined in `Target` relations. This is consistent with traditional access control concepts, where the reference monitor needs to check whether particular combinations of Subjects (often termed *Principals* in the security literature) are permitted to perform specific Actions on specific Resources, when the decision possibly depends on some contextual (Environment) conditions.

While this relational model is able to express policies that are consistent with XACML 2, it is perhaps too closely coupled to XACML 2 policy specifications. Indeed, the corresponding schema for XACML 3 is quite different, because Subject, Resource, Action and Environment entities are implicit, the distinction between them being captured in the data stored in generic entities. Therefore, different model representations would be needed for each language dialect, so a more fundamental representation would be more valuable.

4.2.3 The context model

The XACML 2 context schema in Figure 4.4 also shows that Subjects, Resources, Actions and Environments play a key role in specifying requests. Again, for XACML 3, these "hard" schema distinctions are replaced by "soft" data distinctions. Furthermore, as a consequence of the equal treatment of these entities in XACML 3, the distinction between Subject and Resource, compared to Action and Environment that is visible in Figure 4.4 no longer applies.

Therefore, the relational context schema suffers from the same practical difficulties as the relational policy schema. We therefore propose an alternative domain model representation in § 4.3. As will be seen, the alternative representation is more dynamic and flexible and so is also more suitable for the operations we require, such as policy and request generation.

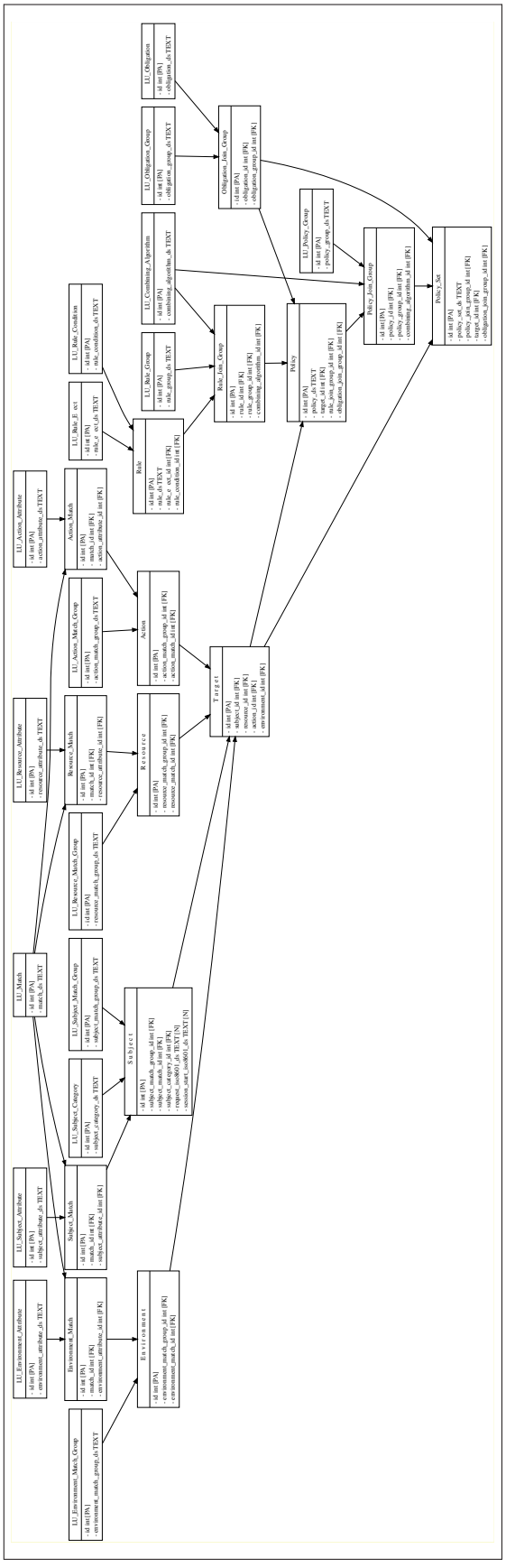


Fig. 4.3 Graphical representation of the *policy* relational schema defining constraints on sharing operations in organization(s).

The lookup tables in the static schema continue to be used as a means of collecting lookup information for the improved model, but the rest of that schema, together with the policy and request schemas, is no longer used.

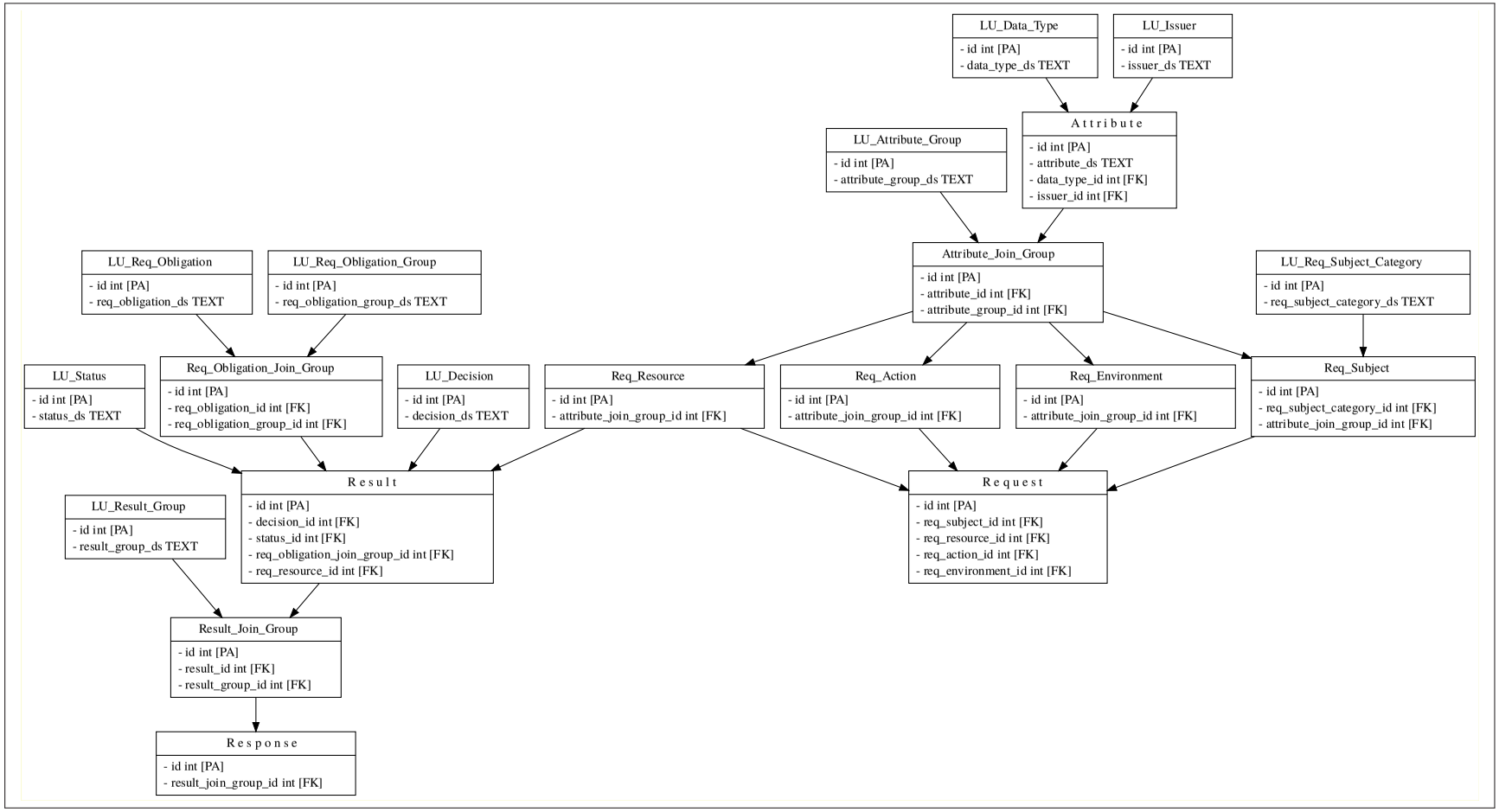


Fig. 4.4 Graphical representation of the *context* relational schema defining access *requests* and corresponding *responses* relating to sharing operations in organization(s).

4.3 A graph representation of the domain model

Given the difficulties associated with the relational models for domain components, (static, policy *and* context), we propose a new representation. The domain is to be modelled as a property graph (Rodriguez and Neubauer, 2010) as this provides a flexible and expressive formalism for domain modelling. A *property graph* is a *directed*, *labeled*, *attributed* multi-graph. More formally, a property graph $G = G(V, E)$ is a function of vertices $V = \{v_i\}, i = 0, \ldots, n_v$ and edges $E = \{e_k\}, k = 1, \ldots, n_k$, where each edge $e_{ij} = v_i \rightarrow v_j$ and where k has a 1-1 correspondence with ij. We say that the edge e_{ij} defines the adjacency relationship linking two vertices v_i and v_j. In a domain model, the edge e_{ij} is the k^{th} relationship in the model, relating v_i and v_j. Some or all of the vertices and edges may have one or more attributes (*typed* scalar data) associated with them. Indeed, one such attribute might be the name (label) of the vertex or edge, but, for modelling purposes, we choose to consider the name as being distinct in character from other attributes.

Robinson et al. (2015) position graph databases (i.e., database management systems that support property graph models directly) as one of the NoSQL quadrants (Robinson et al., 2015, Figure 2-1). The other quadrants are *Document stores*, *Key-Value stores* and *Big-Table/Column stores*. The property graph model appeared to be a good fit to the needs of modelling a complex domain, for the following reasons:

1. a relational model would be difficult to modify to handle new scenarios because, even with mature DBMS tooling, schema changes can result in relatively expensive database refactoring. However, relational models remain important sources of data (in the form of lookup tables, etc.) for the domain model;

2. the other NoSQL models lack expressivity: key-value stores lack direct support for relationships between entities; document stores have richer composite "values" so high-level relationships are explicit but lower-level relationships are baked into the document objects; column stores capture hierarchical relationships well but do not enable *any* entity to be linked with *any other* entity, dynamically. By contrast, apart from atomic entities such as vertices and edges, graph databases are *schema-free*, although an informal schema can be imposed dynamically in response to the needs of a given scenario;

3. the property graph model includes semantic graphs as a special case, so it is possible to map from a property graph instance to a semantic graph instance and many features will be shared. Conversely, it is also possible to perform an inverse mapping—indeed, it is somewhat easier. The main challenges with the forward mapping are to convert taxonomies into OWL `SubClassOf` axioms and the vertex and edge attributes into OWL (Object-, DataType-, Annotation-) property axioms (Motik et al., 2012). Within the exported OWL ontology, there is *direct support* for semantic restrictions such as quantification but these need to be added to the property graph model *explicitly* and implemented in the application that manages the property graph database;

4. parts of a graph are themselves graphs, so at the level of syntax, structures can be broken down and reassembled easily. This feature is used heavily when deriving rules and generating new policies and requests;

5. many of the operations required when using the domain model are concerned with following relationships and traversing (sub)graphs. Such operations are directly supported in the property graph model.

Property graphs (Robinson et al., 2015) offer a flexible metamodel for many forms of domain model. The nodes and edges in the graph can each represent domain entities, with each entity comprising a set of *properties*, realised as attribute name-value pairs. Of course, other types of graph (notably semantic graphs comprising RDF triples) can be used to model domains. Unlike semantic graphs, property graphs store knowledge in domain entities in a form that needs to be interpreted by the application. Thus, in contrast to ontologies, there are no explicit axioms in property graph models. Instead such knowledge is implicit in the graph itself and needs to be inferred by the application (`DomainManager` in this case).

In the case of an enterprise, it is possible to model the static domain in terms of `Member`, `organisation`, `Asset` and `Action` nodes and their associated subnodes and edges. This *static* model acts as a foundation for the *policy* model that encapsulates the access control rules in that enterprise. The static model also supports the request-generating event model, in the sense that interactions between static model entities need to be checked against the policies. Note that the three models (static, policy and request) take the form of property graphs; indeed, the combined domain

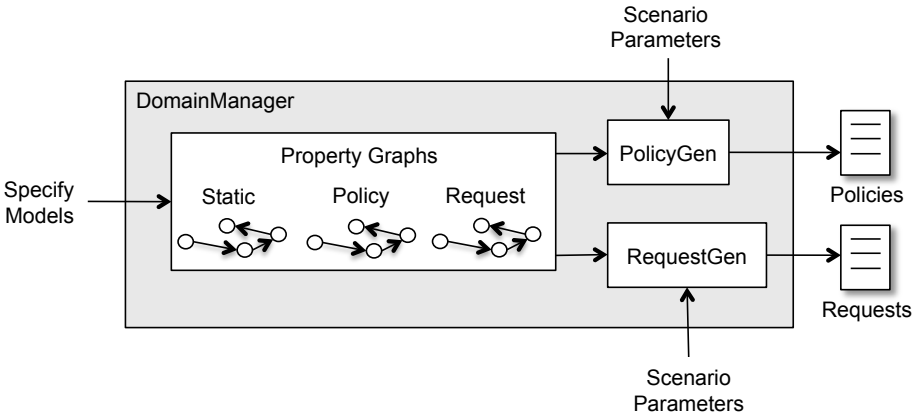


Fig. 4.5 Generation of policy and request sets. The models are specified as property graphs, stored in a graph database. Scenario parameters mandate the characteristics and size of the policy and request sets generated by the PolicyGen and RequestGen processes.

model is still a property graph, comprising the three component models with additional edges to capture their relationships.

Given a domain model in property graph format, it is relatively easy to analyse, modify and derive new data. In practice it is also necessary to extract the policies and requests from the model in text format, notably in XACML 2.0 syntax, for deployment in existing access control systems. This is what the `PolicyGen` and `RequestGen` components (depicted in Figure 4.5) do, based on scenario-specific parameters that control the characteristics and number of generated policies and requests.

The property graph domain model is based on the premise that there exist both prototypical and derived (specialised) entities. The prototypical entities for the domain model are `AccessNode` and `AccessRelationship`. These contain the attributes that are common to all derived entities, such as the entity `name`, `manager` (derived type), `modelGroup`, etc. Specialised variants of these prototype entities are derived to contain entities such as `Policy` and `MemberJoinGroup` respectively. The collection of attributes of each extended `AccessNode` (equivalently, `AccessRelationship`) define an *informal* type. Even though it is not a requirement of property graphs, `DomainManager` ensures that domain model entities of the same type share the same attributes (but not the same attribute values). This use of "typed" entities is akin to a schema in a relational database. Unlike a relational database, a new entity (node or relationship) is created for each instance (analogous to a row in a relational table). Generally, the

combination of (a subset of the) label and attribute values in an entity (node or relationship) is unique across all entities in the database, by analogy with the *key* in a relational table. Consequently a node may be retrieved either:

- directly, by looking up its attribute values (effectively, its *key*), using a separate data structure (an index over the nodes). This is how non-graph storage technologies (relational, key-value, document- or column-) stores perform queries; or
- indirectly, by navigating to it by following a path from another node. This form of querying is unique to graph databases; notably, it is supported by RDF triple stores as a special instance of a graph database.

Each index is a *global* data structure to enable the application to find nodes (or edges) satisfying criteria specified as logical relationships involving the properties of a set of nodes (or edges). As an example, the `Member` index can be queried to return (a set of references to) static nodes where a member (person) is `Senior` and works in the `Finance` department. Although the index adds complexity to graph operations such as insertion and deletion of nodes (or edges), it has the great benefit that such graph-global queries can be answered without the need to visit every node (or edge) in the graph.

Having found a set of nodes (or edges) using an index query, the next step is often to follow paths beginning (or ending) at these nodes (or edges). An index query is not needed to progress along every step of the path. Instead, graph database implementations are optimised to make such in-graph (local) operations very efficient.

One of the great benefits of storing a property graph model in a graph database such as `neo4j` Eifrem et al. (2015) is that the logical and physical models are identical. The person modelling the domain does not need to maintain separate logical and physical views of the domain model. This also means that the choice of graph database implementation is not critical at an early stage, although implementation choices can have significant effects on query performance and similar issues. Apart from the obvious API differences, the two most important features from a (physical) modelling point of view are how the graph database manages indexes, and how to interact with the graph database (e.g., by issuing in-process calls to the graph database engine via its API, or by sending requests to a TCP socket or to a web service over HTTP, etc.).

4.3.1 The static model with semantic enhancements

Listing 4.1 Example specification of how to generate 10 Document Assets and 3 variants of Chat Assets for the `small` domain.

```
seed = 1234
nGroups = 4
0.fixed.group = Document
0.fixed.type = Marketing Plan, Corporate Strategy
0.omit.confidentiality = Unspecified, Unknown
0.omit.integrity = Unspecified, Unknown
0.namePrefixes = Part, Section, Chapter, Webpage
0.attributeCombinationCount = 6
1.fixed.group = Communication
1.fixed.type = Chat Room
1.fixed.confidentiality = High
1.fixed.integrity = Unspecified
1.namePrefixes = All Bank Staff Chat
1.attributeCombinationCount = 1
2.fixed.group = Communication
2.fixed.type = Chat Room
2.fixed.confidentiality = Medium
2.fixed.integrity = High
2.namePrefixes = All Project Staff Chat
2.attributeCombinationCount = 1
3.fixed.group = Communication
3.fixed.type = Chat Room
3.fixed.confidentiality = Medium
3.fixed.integrity = Medium
3.namePrefixes = All Bank Finance Staff Chat
3.attributeCombinationCount = 1
small.instanceCount = 10, 1, 1, 1
medium.instanceCount = 60, 4, 3, 3
large.instanceCount = 110, 7, 5, 5
```

The descriptive attributes of the static domain are defined and managed in an external relational model. The *instances* of static model entities such as `Asset` are created in the graph database by combining the relevant properties (e.g., `Asset.confidentiality`, `Asset.integrity` and `Asset.type` in the case of `Asset`) and assigning each instance combination a name. The instances are generated by a process of *sampling with replacement*.

For example, Listing 4.1 specifies that, when bulk generating `Asset` instances for the `small` domain:

- Sampling with replacement is used; the seed is specified so that the generation procedure is repeatable (but not predictable) across runs.
- `small.instanceCount = 10, 1, 1, 1` Generate 10 instances of group 0 (`Document`) and one each of groups 1,2,3 (`Chat`).

- `0.fixed.type = Marketing Plan, Corporate Strategy` The Asset type property of each document is drawn with replacement from `Marketing Plan, Corporate Strategy`.
- `0.omit.confidentiality = Unspecified, Unknown` The Asset confidentiality property of each Document is selected with replacement from all values of the `Lu_Asset_Confidentiality` table *excluding* `Unspecified, Unknown`.
- `0.omit.integrity = Unspecified, Unknown` The Asset integrity property of each Document is selected with replacement from all values of the `Lu_Asset_Integrity` table *excluding* `Unspecified, Unknown`.
- Each document is assigned a generated name starting with one of `Part, Section, Chapter, Webpage`.

Note that this procedure is trivially scalable to larger numbers of instances and, with a little more effort, to additional properties of `Asset`. The `DSL`s used to specify other base entities of the STATIC model are similar.

The static model is stored as a property graph and hence can be visualised as such. We colour the nodes according to their "type" and use a radial layout to make it easier to understand. Comparing Figures 4.6 and 4.7, it is clear that the graph has structures that become even more apparent as the domain size increases. The rapid increase in the number of nodes and edges is also noteworthy. The corresponding *large* static model has similar features.

Although the static model is populated with generated data, we believe that real organisations with content management system share many of the same features as those evident in our generated data.

4.3.1.1 Semantic enhancements

The procedure described above is insufficient to capture the rich semantics of the static domain. This is because each Entity instance is generated in isolation and does not take account of links and semantic constraints between such entities. Note that these static constraints should not be confused with policy rules:

- Static constraints are intrinsic to a well-specified static domain: they are definitional and do not need to be checked and/or enforced by the policy system.

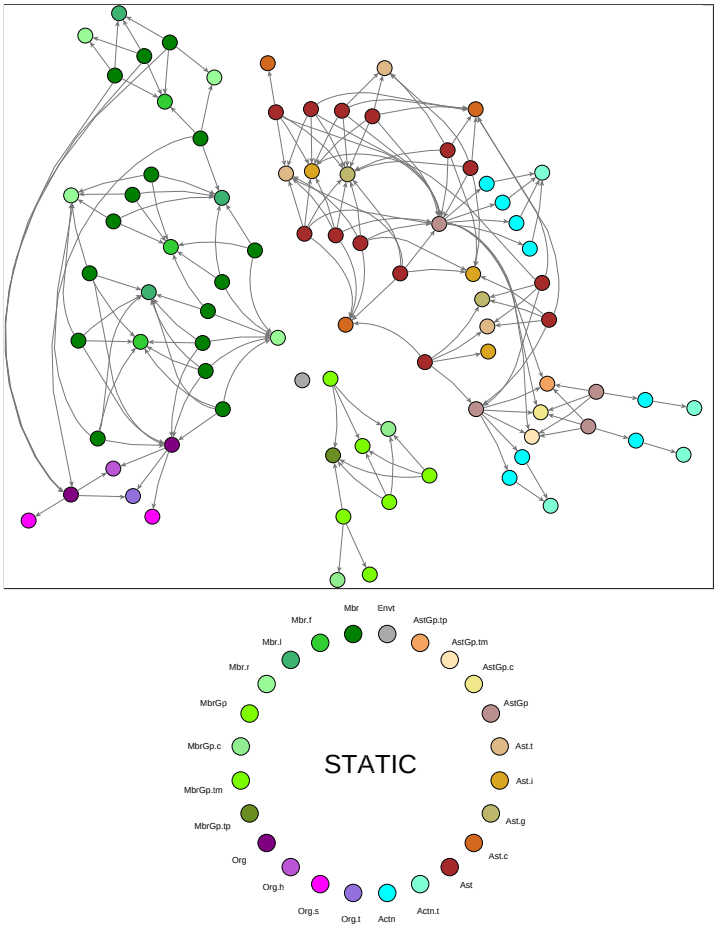


Fig. 4.6 The *small* static model, represented as a property graph, in a radial layout. The legend uses a similar layout and it is clear that most of the static model consists of Asset and Member nodes and attributes. Each of the static nodes corresponds to a row of a look up table in the static relational model (Figure 4.2). As an example, the node labelled `Ast.i` corresponds to the `asset_integrity_ds` field in the `Lu_Asset_Integrity` table, with values described in Table 4.2.

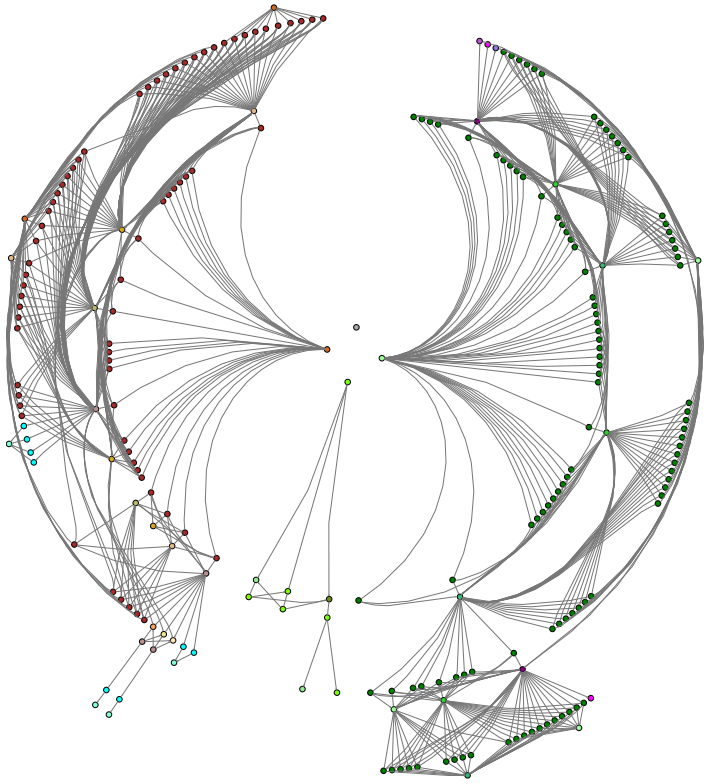


Fig. 4.7 The *medium* static model, represented as a property graph, in a radial layout. It is clear that the graph size grows more rapidly than any of its component entities, e.g., Assets.

Table 4.4 Category mapping from static model entities to policy model entities

Static	Category	Policy
Member Organisation MemberGroup OrganisationGroup	Subject	Subject
Asset AssetGroup	Resource	Resource
Action ActionType	Action	Action

- Policy constraints define the boundary between desired and undesired behaviour between static model entities. Based on knowledge in the static model only, there is no reason why a particular relationship should or should not exist. However, it might need to checked and/or enforced as a business policy.

Static constraints are analogous to the axioms in an ontology.

Given a static entity E_{static}, its `category` is defined by its relationship to a policy model entity S, R, A (Subject, Resource or Action). A given E_{static} has a category relationship with one of S, R, A. The category relationships are shown in Table 4.4.

Two types of static constraint are considered:

1. Within-category, such as `Member`:`MemberGroup` (Many:Many) and `Member`:`organisation` (1:Many)
2. Between-category, such as `Asset`:`Action`, which encodes the fact that some actions on some assets are meaningless or infeasible.

Listing 4.2 Example specification of how to assign existing Members to two existing Organisations

```
nGroups = 2
0.Organisation.name = National Bank
0.Member.function = Marketing, Finance
0.Member.role = Manager, Decision Maker, Officer
1.Organisation.name = Can-Do Consultants
1.Member.function = Technical
1.Member.role = Analyst, Implementer
```

For the Within-category case, Listing 4.2 states that

1. Members with Member.function ∈ {Marketing, Finance} and Member.role ∈ {Manager, Decision Maker, Officer} belong to an organisation named National Bank.
2. Members with Member.function ∈ {Technical} and Member.role ∈ {Analyst, Implementer} belong to an organisation named Can-Do Consultants.

Listing 4.3 Example specification of how to align AssetGroups with ActionTypes

```
join.AssetGroup.name = Action.type
```

For the Between-category case, Listing 4.3 states that Asset and ActionType share the same domain, which is defined in the relational model as {Communication, Document, Person, Task}. If this were a relational and not a graph model, the two underlying columns would be related with a foreign key constraint. However there is more than just data equivalence: Asset and ActionType are semantically linked in the sense that

> Actions with type a = ActionType are associated only with Assets with group a = Asset. All other combinations of Actions and Assets are invalid. Moreover, this principle applies to all model types: static, policy and request.

Static entities can have many properties and participate in hierarchies (taxonomies). However, it is also necessary to apply within-category definitional constraints such as "Member X belongs to MemberGroup Y and MemberGroup Z" and "Member X belongs to Organisation Y" and between-category constraints such as "ActionType X can be applied to AssetGroup X".

These semantic constraints are not stored in the graph database. Instead they are built into the PolicyGen and RequestGen procedures and are enforced when populating the policy and request model entities, as described in § 4.4 and § 4.5 respectively.

By contrast with the relational model, the property graph model above can be modified easily to match any reasonable (Agent, Asset, Action)-based static domain, just by adding data to the graph.

4.3.2 The policy and context models

4.3.2.1 Rules and (target) hierarchies

XACML policy evaluation starts with *Target Matching*. Indeed, if a rule is to apply, the attributes of the request must:

1. satisfy the `Target` condition of the rule;
2. satisfy any additional requirements in the optional *general* `Condition` element,

in that order.

According to the XACML 3.0 standard Rissanen (2013, §3.3.1.3)

> **Condition** represents a Boolean expression that refines the applicability of the rule beyond the predicates implied by its target. Therefore, it may be absent.

The `Condition` element can have arbitrarily complicated logic expressions encoded in combinations of `Function` elements. Indeed, for a `PDP` to conform to the `XACML` standard, it needs to support the large set of `Function`s listed in Rissanen (2013, Appendix A.3). Many PDP implementations also enable their users to add user-specified `Function`s that are typically Boolean-valued functions of attributes.

However, each `Condition` is invoked only when the `Target` of that rule and the corresponding request attributes match. In a large policy set, efficient Target matching is necessary, but not sufficient, for good policy evaluation performance. Even if Target matching is efficient, evaluation of the general `Condition`s arising from the matched targets could limit policy evaluation performance. However, because it is not evaluated as frequently as the `Target` conditions, we assume that the `Condition` filter has a second-order influence on policy evaluation performance. By omitting `Condition`, the expressiveness of the policies in the domain model is reduced, but the remaining expressiveness is adequate for the scenarios considered in this dissertation. The main advantage of omitting `Condition` processing is to make it easier to analyse policies, because policy evaluation reduces to asking whether logical statements with a constrained structure (of the form: "Is the Target of a given rule *satisfied* by a given request?") are true or false.

Other authors also separate Target matching from Condition evaluation. For example, Tschantz and Krishnamurthi (2006) introduce *Core XACML* which is a proper subset of `XACML` (and hence is accepted by all conforming `PDP`s). They prove that Core XACML has certain semantic properties such as determinism and lack of monotonicity, so it is a non-trivial subset of `XACML`. Core XACML and the XACML subset supported by `DomainManager` are similar, e.g., the absence of general `Condition` elements, but there are some aspects where Core XACML is more restrictive than "`DomainManager` XACML", e.g., in respect of combining algorithms. (Masi et al., 2012) also distinguishes between XACML-excluding-conditions and full XACML, by defining separate parsers for conditions and the remainder of `XACML`.

Assumption 4.1. The `Condition` element of XACML policies is ignored in this dissertation and rules are evaluated based on their `Target` conditions only.

Apart from `Condition`s, XACML policies can specify other security requirements, apart from access control decisions. The most important of these include `Obligation`s. Typically an `Obligation` is triggered after a decision is made by the `PDP`. For example, if the decision was to permit access to a controlled resource, there might be an obligation to log the fact that access was granted, and to add the *Subject* to a controlled list for audit purposes. If the decision was to deny access, there are also scenarios where that decision should trigger further action, e.g., to permit reduced access to the resource or full access to a less sensitive substitute resource.

The `PDP` generally hands `Obligation`s to the `PEP` for further action. `Obligation`s are often enacted in an asynchronous fashion and hence do not contribute to latency at the `PDP` and might not even be noticeable to users. Of course, each `Obligation` adds to the load on the overall security infrastructure. `Obligation`s can themselves be combined during policy evaluation, so that multiple obligation actions are triggered after the access decision is made. However, it is probably more common that a single `Obligation` action, if any, is triggered. The focus in this dissertation is on access control decision performance. A study that was more inclusive of the entire security infrastructure would be needed to estimate the performance effects of Obligations.

Assumption 4.2. The `Obligation` element of XACML policies is out of scope in this dissertation because the focus is on access control *decision* performance only and

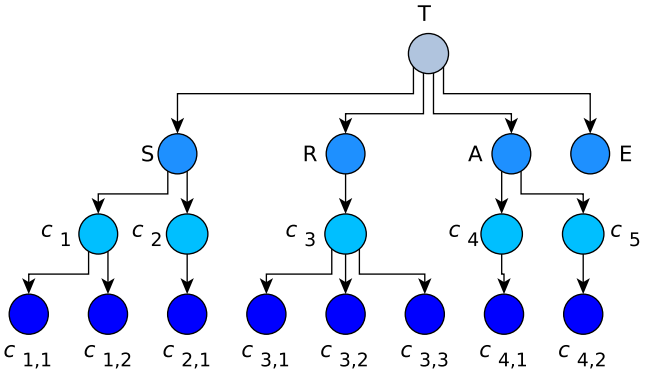


Fig. 4.8 Example Target Hierarchy: $T = S \wedge R \wedge A \wedge E$, where $C^{\texttt{Subject}} \doteq S = ((c_{1,1} \wedge c_{1,2}) \vee c_{2,1})$, $C^{\texttt{Resource}} \doteq R = (c_{3,1} \wedge c_{3,2} \wedge c_{3,3})$, $C^{\texttt{Action}} \doteq A = (c_{4,1} \vee c_{5,1})$ and $C^{\texttt{Environment}} \doteq E = \top$.

`Obligations` are enacted outside of the `PDP` and depend on factors that are external to the access control system *per se*.

Generally, a `Target` consists of `Subject`, `Resource`, `Action` and `Environment` entities. According to access control conventions, an Agent (`Subject`) wishes to perform an `Action` on an Asset (`Resource`), subject to external conditions captured in an `Environment` clause. Each of these entities comprises a set of sub-conditions that need to be composed to form the `Target` condition. More formally, let C represent a `Subject` (S), `Resource` (R), `Action` (A) or `Environment` (E) target entity. Then C can be decomposed into subentities c_i and $c_{i,j}$ as follows:

$$C = \bigvee_i c_i \tag{4.1}$$

$$\text{where } c_i = \bigwedge_j c_{i,j}, \tag{4.2}$$

where $\vee$ represents disjunction (logical *or*) and $\wedge$ represents conjunction (logical *and*).

For convenience, C is named a `Collected Target Component (CTC)`, c_i is named a `Target Component (TC)` and $c_{i,j}$ is a `Target SubComponent (TSC)`, a *logical clause* of the target condition. An example target hierarchy using this notation is shown

graphically in Figure 4.8. For illustration, let

$$
\begin{aligned}
c_{1,1} &= \texttt{Member.level} = \texttt{Senior} \\
c_{1,2} &= \texttt{Member.role} = \texttt{Finance} \\
c_{2,1} &= \texttt{Organisation.name} = \texttt{Acme Bank} \\
c_{3,1} &= \texttt{Asset.confidentiality} = \texttt{High} \\
c_{3,2} &= \texttt{Asset.integrity} = \texttt{High} \\
c_{3,3} &= \texttt{Asset.type} = \texttt{Financial Plan} \\
c_{4,1} &= \texttt{Action.name} = \texttt{Read} \\
c_{5,1} &= \texttt{Action.write} = \texttt{Write} \\
c_1 &= c_{1,1} \wedge c_{1,2} \\
c_2 &= c_{2,1} \\
c_3 &= c_{3,1} \wedge c_{3,2} \wedge c_{3,3} \\
c_4 &= c_{4,1} \\
c_5 &= c_{5,1} \\
S &= c_1 \vee c_2 \\
R &= c_3 \\
A &= c_4 \vee c_5 \\
E &= \top \\
T &= S \wedge R \wedge A \wedge E.
\end{aligned}
$$

As can be seen, the Target comprises a hierarchy of Boolean-valued expressions. There is no limit to the number of `TSC`s and `TC`s in each expression. Furthermore, Targets can be nested within Targets, in which case the Rule-, Policy- or PolicySet-combining algorithm can be used to evaluate the overall match value. The entity hierarchy in Figure 4.8 is an example of the target conditions to be applied when matching access requests against policy rules.

4.3.2.2 Canonical representation

We make the further assumption that the vocabulary of the domain model is limited to a finite symbol alphabet for each entity. As an example, `Asset.confidentiality`

can be either `low`, `medium` or `high`; *Time.hourOfDay* can be one of `09.00`, `10.00`, etc. Thus it is possible (*without loss of expressiveness*) to reformulate as necessary so that each clause c_{ij} asserts *identity* between concepts. More formally: if it is assumed that the domain of each entity (such as `Asset.confidentiality`) is both discrete and finite, clauses with other verbs ($\neq$, $\subseteq$, or even $\leq$ (if the symbols satisfy an ordering relation)) can be reformulated as combinations of *equality* constraints.

Thus a clause asserting inequality can be expressed by enumerating the complement of the clause with the inequality verb replaced with equality. As an example, let

$$c_{i,j} := \texttt{Asset.confidentiality} \neq \texttt{high}$$

. Then we can restate $c_{i,j}$ as

$$\begin{aligned} c_{i,j} :=& (\texttt{Asset.confidentiality} = \texttt{low}) \vee \\ & (\texttt{Asset.confidentiality} = \texttt{medium}). \end{aligned}$$

Other non-equality clauses can be reformulated as equality clauses using similar constructive procedures.

The revised target hierarchy is constructed as follows. For each derived equality clause, a new target component ($c_{\hat{\imath}}$) is created as follows:

$$c_{\hat{\imath}} = \bigwedge_{k;k\neq j} c_{i,k} \wedge c_{i\tilde{\jmath}}, \tag{4.3}$$

where $c_{i\tilde{\jmath}}$ is a clause of the same form as c_{ij} but using one of the complement attribute values $\tilde{\jmath}$ of attribute value j.

Expanding on the example above, letting

$$\begin{aligned} c_i :=& (\texttt{Asset.type} = \texttt{document}) \wedge \\ & (\texttt{Asset.confidentiality} \neq \texttt{high}), \end{aligned}$$

and using De Morgan's Laws, it is clear that this is equivalent to

$$\begin{aligned} c_i :=& ((\texttt{Asset.type} = \texttt{document}) \wedge \\ & (\texttt{Asset.confidentiality} = \texttt{low})) \\ \vee & ((\texttt{Asset.type} = \texttt{document}) \wedge \\ & (\texttt{Asset.confidentiality} = \texttt{medium})). \end{aligned}$$

Set containment, such as

$$c_{i,j} := (\texttt{Asset.confidentiality} \subseteq \{\texttt{low},\texttt{medium}\})$$

can be rewritten as

$$\begin{aligned} c_{i,j} :=& ((\texttt{Asset.confidentiality} = \texttt{low}) \vee \\ & (\texttt{Asset.confidentiality} = \texttt{medium})) \end{aligned}$$

and the same transformation to c_i is possible, as described above.

Summarising, there are many alternative ways of formulating the same *collected target component*. The "canonical formulation" of a `Collected Target Component (CTC)` is a disjunction of conjunctions of equality clauses. While this formulation does not generally minimise the number of logical clauses needed to specify the collected target component, it has the benefit of being easier to analyse, as will be seen in §4.4.

Assumption 4.3. The static entities are drawn from an enumerable set: $S = \{s_1, s_2, \ldots, s_n\}$, where S is known *a priori* for each entity type (members, assets, etc.) Consequently, the canonical representation of any `Target` is a conjunction of `CTC` entities, each of which is a disjunction of `TC` entities, each of which is a conjunction of `TSC` entities, each of which is an *equality* clause. This representation can express any qualifying Target, though it is not guaranteed to be the most *efficient* representation (measured by number of logical clauses).

4.3.2.3 Policy and request clauses

One of the key features of the XACML metamodel that is replicated in the domain model described in this dissertation is the structural similarity between target conditions and access requests. Indeed, the target hierarchy is practically identical to the request hierarchy. A target (equivalently, request) has `Subject`, `Resource`, `Action` and `Environment` `CTC`s. Each `CTC` is a disjunction of `TC`s, although in the case of a request, the disjunction is degenerate for `Action` and `Environment` `CTC`s, because only one `TC` is permitted in each case. Each `TC` comprises a conjunction of `TSC`s. However, the *interpretation* of the `TSC`s differs between policies and requests. In the case of policies, each `TSC` is a logical condition clause, evaluating to true or false as part of a rule. In the case of requests, each `TSC` is a *definitional* clause, specifying the *context* of the request. The purpose of policy evaluation is to find the targets that match the context definitions and to evaluate the matching target hierarchies, applying the relevant rule and/or policy combining algorithms to derive the overall policy decision.

In XACML 3, the structural hierarchy of Policy `Target` entities is the same as that of `Request` entities, but in earlier versions of XACML, the Request structure is slightly simpler than a Policy `Target` because the `Action` and `Environment` target component types for a `Request` are `TC`s, not `CTC`s as they are for `Subject`s and `Resource`s.

> Policy targets and Requests are structurally equivalent, differing mainly in intent. This makes policy evaluation easier, but also has potential benefits when generating policies and requests.

4.3.2.4 Clause restrictiveness

As more `TSC`s $c_{i,j}$ are added, the target entity generally becomes more restrictive; the least restrictive is the "empty" `CTC`, such as E in Figure 4.8 that evaluates to a tautology ($\top$). Furthermore, an instance-based `TSC` is satisfied by a single instance and so is generally more restrictive than an attribute-based `TSC`, which is often satisfied by multiple instances.

For example, requiring that the *name* attribute of a request sub component should *equal* a specified name (e.g., $c_{i,j}$:= `Asset.name` = `marketing-plan-2014` is more restrictive than requiring that a *descriptive* attribute (which could be shared by many

instances) should equal a specified descriptive value (e.g.,
$c_{i,j}$:= `Asset.confidentiality = high`).

4.3.2.5 Policy matching

By restricting target components to equality constraints only, they are strictly additive: $c_{i,j} \cap c_{i,k}, j \neq k$ is more restrictive than either $c_{i,j}$ or $c_{i,k}$ alone, for all $j \neq k$. Thus a request clause of the form $c_{i,j} \cap c_{i,k}, j \neq k$ is more restricted than a policy target clause $c_{i,j}$ and hence is "matched" by it. The converse is not true. In other words, more specific request clauses match less specific policy targets but not vice-versa. If a request is to match a policy, *all* of its target components must "match" (be semantically contained in) *all* the target components of a given rule CTC. The overall access decision is computed by semantically aggregating the target matches according to the rule and policy combining algorithms in force at each target in the policy tree.

4.4 Generating policies—PolicyGen

Policy authoring and the related topic of policy conflict analysis have been studied by researchers over many years (Davy et al., 2008). Bulk policy generation adds problems of scale since it becomes very difficult to understand a large set of policies, or even to specify them in a convenient and transparent way. One way to proceed, which we adopt, is to start with a well-specified smaller set of "template" policies and to add extra policies until the desired scale of policy set is achieved. With this approach, the problem becomes one of "scaling up" from a relatively small number of semantically consistent policies. However, it is necessary to constrain the rules being added to the smaller policy set so that the decisions remain predictable.

Template policies specify rules in terms of statements involving attributes. A property graph can represent such template policies, and graph operations (such as path finding and traversals) can be used to generate "full" policy statements.

Bulk policy generation can be "primed" by writing *template* policies in a convenient Domain Specific Language (DSL). These high level, attribute-based policies deliberately avoid reference to instances of Subjects and Resources. Instead, they are specified as (combinations of) Subject and/or Resource attributes. Such high-level

policies have the following advantages over more concrete low-level (instance-based) policies

- they are easier to interpret because the "business" intention of each policy is more explicit;
- they can be specified by business users who do not need to concern themselves with implementation details;
- even for relatively complex access control requirements, the number of policy rules depends upon the size of the domain vocabulary, which is generally much smaller than the number of instances (of Subjects and Resources, particularly) in the domain;
- the template policies are generally more stable in the face of change, e.g., as staff join and leave an organisation, the same template policies still apply as they did before any such change at instance level. They would still need to change if larger scale changes occurred, say as new groups/teams were formed, attributes were added or removed, etc.

The main disadvantage is that there is an "impedance mismatch" with the incoming requests, which are generally defined at instance level because they are necessarily more concrete and explicit. Therefore, either the policies or the requests need to be transformed to the specification level expected by the other.

> Using the template policies described above, it is possible to infer instance-based policies by linking the policy targets to the instances defined in the static domain model.

The example template policy in Listing 4.4 has a syntax that is based on Java property files and can be parsed in `DomainManager`. The policy contains typical business rules controlling interactions between a Bank and a service provider. As with XACML, rules can be combined in a policy via a rule combination strategy (permit_overrides in this example) There is one `Deny` guard clause. Even a short policy set can have complex conditions and the intended logic is expressed in a terse, more explicit form than in other policy representations like XML-encoded XACML, where is far more verbose.

Note that the policy DSL assigns settings using the same basic syntax as Java properties files, but some terms in the DSL, such as `policyRef`, are keywords in the

DSL and hence are interpreted by the parser as having special meaning. The following features are noteworthy

- There is one policy set (G3) with three policies (G0, G1, G2) and eight rules (R0, R1, R2, R3, R4, R6, R7, R9).
- One rule (R9) is to deny everything. The other rules permit access in specific scenarios.
- Policy G0 has rules defining situations where `Subject`s are permitted to `Read` or `Write` documents.
- Policy G1 has rules permitting `Subject`s to `Setup` or `Join` group chats.
- Policy G2 has a default rule to deny all access.
- PolicySet G3 uses a *First Applicable* algorithm to combine policies G0, G1 and G2. Note that if any of the `Target` conditions in G0 or G1 hold, a Permit rule applies. Otherwise the decision "falls through" to the G2 rules, so the default Deny rule applies.
- `TSC`s (target clauses) define the conditions for each rule.
- `TSC` labels have the form `RuleId.TargetComponentType.Id`.
- If there are multiple instances of `TSC`s with the same label, the corresponding logical clauses are *AND*ed together. Otherwise, `TSC`s having the same `RuleId` and `TargetComponentType` but different `Id` belong to different target components and are *OR*ed together.
- The most complex `CTC` is `R4.Subjct`, where `S` has two `TC`s each with two `TSC`s, hence the structure (`SUBJECT.3` $\wedge$ `SUBJECT.1`) $\vee$ (`SUBJECT.3` $\wedge$ `SUBJECT.0`).

In addition to the high-level policies described above, it is necessary to infer instance-based policies by linking the policy targets to the instances defined in the static domain model. For every high-level `TC`, we lookup its attributes in the property graph and identify the Subject, Resource and Action *instances* sharing these attributes, resulting in the corresponding instance-based formulation. Typically, a single high-level `TC` corresponds to many low-level target conditions.

Butler and Jennings (2015) and § A describe the algorithm used in `DomainManager` to generate instance policies from high-level *template* policy specifications.

Summarising, based on the user-supplied policy DSL, both an attribute- and an instance-based policy formulation can be created and represented in the property graph model. Given specifications of both the static domain and the template policies to be applied on that domain, it is possible to derive instance-level policies that can be deployed in a PDP. In property graph form, it is instructive to derive some graph measures (see § 4.6.1) and to visualise the generated policies (see §4.4.3). The `DomainManager` application prototype (Butler, 2015a) collects the policy template specifications, generates the instance-based policies in property graph format and also enables the user to export XACML 2.0-format policies encoded as XML or JSON for use in performance experiments. We now outline the steps involved in the policy generation process.

Listing 4.4 Listing of the *base* template access control policy used in this dissertation

```
ref =  PermitOverrides-OneDeny
staticRef =  large
policyCategory = PO-OD
policyVersion =  0.1
runId =  1

baseGroups=G3

SUBJECT.0=Member.level,equals,Chief
SUBJECT.1=Member.level,equals,Senior
SUBJECT.2=Member.function,equals,Finance
SUBJECT.3=Member.function,equals,Marketing
SUBJECT.4=Organisation.name,equals,National Bank
SUBJECT.5=Member.role,equals,Implementer
RESOURCE.0=Asset.confidentiality,equals,High
RESOURCE.1=Asset.integrity,equals,High
RESOURCE.2=Asset.integrity,equals,Medium
RESOURCE.3=Asset.type,equals,Corporate Strategy
RESOURCE.4=Asset.type,equals,Marketing Plan
RESOURCE.5=Asset.type,equals,Chat Room
ACTION.0=Action.name,equals,Read
ACTION.1=Action.name,equals,Write
ACTION.2=Action.name,equals,Setup
ACTION.3=Action.name,equals,Participate In

Groups=G0,G1,G2,G3
Rules=R0,R1,R2,R3,R4,R6,R7,R9

R0.Desc=Member.level = Chief can Read Asset.confidentiality = High
R0.Subjct.0=SUBJECT.0
R0.Resrce.0=RESOURCE.0
R0.Action.0=ACTION.0
R0.Decisn=Permit

R1.Desc=Member.level = Senior can Read any permitted Asset
R1.Subjct.0=SUBJECT.1
R1.Action.0=ACTION.0
R1.Decisn=Permit

R2.Desc=Member.level = Senior can Write Asset.integrity = High \
OR Asset.integrity = Medium
R2.Subjct.0=SUBJECT.1
R2.Resrce.0=RESOURCE.1
R2.Resrce.1=RESOURCE.2
R2.Action.0=ACTION.1
R2.Decisn=Permit

R3.Desc=Member.level = Chief AND Member.function = Finance can Read \
OR Write any permitted asset
R3.Subjct.0=SUBJECT.2
R3.Subjct.0=SUBJECT.0
R3.Action.0=ACTION.0
R3.Action.1=ACTION.1
R3.Decisn=Permit

R4.Desc=Member.function=Marketing AND Member.level = (Senior or Chief) \
can Read any permitted asset
R4.Subjct.0=SUBJECT.3
R4.Subjct.0=SUBJECT.1
R4.Subjct.1=SUBJECT.3
R4.Subjct.1=SUBJECT.0
R4.Action.0=ACTION.0
R4.Decisn=Permit

G0.Desc=Combine rules R0-R4 with permit_overrides\, for Organisation.name = \
National Bank and Asset.type = Corporate Strategy or Marketing Plan
G0.Cntains=R0,R1,R2,R3,R4
G0.CmbnAlg=permit_overrides
G0.Subjct.0=SUBJECT.4
G0.Resrce.0=RESOURCE.3
G0.Resrce.1=RESOURCE.4

R6.Desc=Member.role=Implementer can Setup any permitted asset
R6.Subjct.0=SUBJECT.5
R6.Action.0=ACTION.2
R6.Decisn=Permit

R7.Desc=Member.function = Marketing OR Member.function = Finance can \
Participate In any permitted asset
R7.Subjct.0=SUBJECT.2
R7.Subjct.1=SUBJECT.3
R7.Action.0=ACTION.3
R7.Decisn=Permit

G1.Desc=Combine rules R6-R7 with permit_overrides\, for Asset.type = Chat Room
G1.Cntains=R6,R7
G1.CmbnAlg=permit_overrides
G1.Resrce.0=RESOURCE.5

R9.Desc=Default decision is to deny access
R9.Decisn=Deny

G2.Desc=Safety default: deny everything
G2.Cntains=R9
G2.CmbnAlg=permit_overrides

G3.Desc=Combine groups G0\,G1 and G2 with permit_overrides
G3.Cntains=G0,G1,G2
G3.CmbnAlg=permit_overrides
```

4.4.1 Step 1—populate the template policy facade

At present, the template policies are specified in a DSL, of which Listing 4.4 is an example. However, this DSL might change in future, particularly if it became possible to export template policies from an existing policy deployment. Therefore the template policies are loaded into an in-memory *facade* which is a slightly simplified version (e.g., `RuleGroupFacade` eventually maps to both `Policy` and `PolicySet` of the property graph model). This use of an intermediate facade representation has two advantages:

1. If new template policy specification formats are added, new software is needed to transform it to the facade model, but both the core property graph model and the complex algorithms that depend on it can be left untouched.
2. It is possible to begin the data transformation process when populating the property graph from the facade objects. In particular, it is convenient to supplement the template policy with additional conditions to ensure that within-category static model semantic conditions are applied when generating instance policies and requests from the template policies.

In the case of the latter advantage, these between-category conditions are difficult to apply unless we have easy access to the two relevant entity categories (`Action` and `Resource` in this case) *before* they are committed to the database.

4.4.2 Step 2—instantiate the template policy entities in the property graph

The facade object graph is a `Directed Acyclic Graph (DAG)` with a finite set of `RuleGroupFacade` root nodes. Typically there is only one such root node. Starting from each root node, it is possible to recursively descend through the tree induced from that node, populating the property graph nodes from the bottom up. That is, each `TargetSubCompFacade` becomes a `TSC`, and the set of `TargetSubCompFacade`s referenced in a `TargetCompFacade` become a set of `HAS_TARGET_SUBCOMP` relationships between a `TC` node and the `TSC` nodes that it references. Thus we have concordance between policy semantics and its graph representation: each `TC` *entity* entity is a conjunction of `TSC` entities, or equivalently each `TC` *node* has a `HAS_TARGET_SUBCOMP`

relationship to a set of `TSC` nodes, which is derived from each `TargetCompFacade` *object* containing a set of references to its `TargetSubCompFacade`s.

Similar considerations recurse back up the tree from the `TSC` leaves to the root `Policy` or `PolicySet` node, via `CTC HAS_TARGET_COMPONENT` relationships to `TC` nodes, `Target` {`HAS_SUBJECT`, `HAS_RESOURCE`, `HAS_ACTION`, `HAS_ENVIRONMENT`} relationships to `CTC` nodes, `Rule HAS_TARGET` relationships to `Target` nodes, etc.

Along the way, `DomainManager` collects the user-specified properties from the facade objects and populates the relevant graph nodes and edges with this data. In addition, it derives, for the `TSC`, `TC`, `CTC` and `Target` nodes, the index queries that can be used to derive the corresponding instance-based `TSC`, `TC`, `CTC` and `Target` nodes. Each of these index queries is termed an *instance query* and is derived by aggregating its sub instance queries using the same composition operator that is used by the template entity which contains that instance query. Indeed, since the composition operator can be taken outside the instance query, it is possible to define a set of recurrence relations to create the instance queries, starting from the instance queries for the `TSC`s.

Let $Q(e)$ be the instance query associated with template (attribute-based) policy element e, where $e \in \{\texttt{TSC}, \texttt{TC}, \texttt{CTC}, \texttt{Target}\}$. Then $Q(e)$ satisfies the following:

$$Q(c_{i,j}) \doteq \{T_{i,j} \in t_k\}. \tag{4.4}$$

$$Q(c_i) \doteq Q(\bigwedge_j \{c_{i,j}\}) = \bigwedge_j \{Q(c_{i,j})\}. \tag{4.5}$$

$$Q(c) \doteq Q(\bigvee_i \{c_i\}) = \bigvee_i \{Q(c_i)\}. \tag{4.6}$$

$$Q(T) \doteq Q(c^{(S)} \wedge c^{(R)} \wedge c^{(A)} \wedge c^{(E)}) = Q(c^{(S)}) \wedge Q(c^{(R)}) \wedge \ldots \tag{4.7}$$

where T is any `Target`, $c^{(\circ)}$ is a `CTC` where T `HAS_SUBJECT` $c^{(S)}$, etc. Also c is any `CTC`, c_i is a `TC` where c `HAS_TARGET_COMPONENT` c_i and $c_{i,j}$ is a `TSC` where c_i `HAS_TARGET_SUBCOMP` $c_{i,j}$.

Figure 4.9 indicates that the hierarchical structure is almost identical to that of the XACML metamodel. The main difference is that the relationships in the graph capture the links between entities in an explicit manner, and thereby play a vital role in specifying the template policy as a property graph. Although it is not obvious, each node and relationship contains a rich set of properties so that, by writing suitable

Listing 4.5 Listing of the cutdown *base* template access control policy used to illustrate policy generation. It is derived from Listing 4.4 but with some rules and rule groups removed to reduce the visual clutter in the graph visualisations.

```
baseGroups=G0

SUBJECT.0=Member.level,string-equal,Chief
SUBJECT.1=Member.level,string-equal,Senior
SUBJECT.2=Member.function,string-equal,Finance
SUBJECT.4=Organisation.name,string-equal,National Bank
RESOURCE.1=Asset.integrity,string-equal,High
RESOURCE.2=Asset.integrity,string-equal,Medium
RESOURCE.3=Asset.type,string-equal,Corporate Strategy
RESOURCE.4=Asset.type,string-equal,Marketing Plan
ACTION.0=Action.name,string-equal,Read
ACTION.1=Action.name,string-equal,Write

Groups=G0
Rules=R2,R3,R5

R2.Desc=Member.level = Senior can Write Asset.integrity = High \
OR Asset.integrity = Medium
R2.Subjct.0=SUBJECT.1
R2.Resrce.0=RESOURCE.1
R2.Resrce.1=RESOURCE.2
R2.Action.0=ACTION.1
R2.Decisn=Permit

R3.Desc=Member.level = Chief AND Member.function = Finance can Read \
or Write any permitted asset
R3.Subjct.0=SUBJECT.2
R3.Subjct.0=SUBJECT.0
R3.Action.0=ACTION.0
R3.Action.1=ACTION.1
R3.Decisn=Permit

R5.Desc=Default decision is to deny access
R5.Decisn=Deny

G0.Desc=Combine rules R0-R5 with first_applicable\, for Organisation.name = \
National Bank and Asset.type = Corporate Strategy or Marketing Plan
G0.Cntains=R2,R3,R5
G0.CmbnAlg=first_applicable
G0.Subjct.0=SUBJECT.4
G0.Resrce.0=RESOURCE.3
G0.Resrce.1=RESOURCE.4
```

queries against the graph, it is possible to derive new information such as the policy graph measures introduced in § 4.6.1.

Figure 4.9 shows the template policy, as specified by the policy author. It lacks many features of a full policy; these missing features are presented in § 4.4.2.1 (semantic constraints), and § 4.4.3 (instance-based policies).

4.4.2.1 Enforcing the static semantic constraints

It is timely to consider how the static semantic constraints impact upon policy generation. Generally, within-category constraints are enforced when deriving the `TSC` instance queries and have no effect on the template (attribute-based) policy model. In effect, infeasible `TSC` nodes are omitted from the instance-based policy, with effects that propagate up the policy tree to the `Target` nodes.

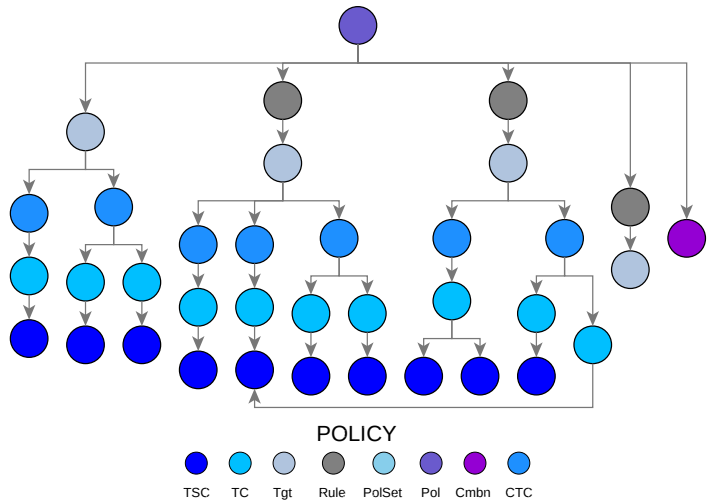


Fig. 4.9 Cutdown template policy model derived from Listing 4.5, represented as a property graph. The hierarchical structure of the policy, its 3 rules and their components is maintained in the graph.

However, between-category constraints are more difficult to apply by filtering the `TSC` nodes in this way. Two approaches were considered:

use more complex instance queries With this solution, the template policy represented as a property graph closely matches the structure of the template policy specified by the user in the DSL. However, the instance queries are more complex and the instance-based policies look different in kind to the attribute-based policies from which they were derived.

augment the template policies With this solution, it might become necessary to add attribute-based `TC` nodes so that the resulting attribute-based policy takes account of the static semantic constraints, in which case the instance queries can be derived as normal, without taking special care to ensure that the static semantic constraints are satisfied.

`DomainManager` version 1.x used the first approach, but `DomainManager` version 2.x uses the second. Despite some reservations, mostly related to the blurring of the distinction between static model and policy model considerations, we now believe the second approach is better.

Firstly, it appears that (human) policy authors are not purist about such distinctions and may add extra `TC`s anyway, though perhaps not in all cases where they would be needed for this purpose. Conversely, in some cases, policy authors add such `TC`s even

when they are *not strictly necessary*, i.e., they *over-specify* the policies. Functionally, these unnecessary conditions do not affect the decisions made during policy evaluation, though they might affect policy evaluation performance. So if extra template policy `TC`s are already in place, why not use them?

Secondly, it should be noted that instance requests also need to satisfy between-category semantic constraints. If the second approach is used, these constraints can be applied with less effort than would be needed for the first approach. For this reason, Occam's Razor suggests that the second approach has more merit.

Thirdly, if a variety of different semantic constraints were needed, it is easier to add them explicitly in the form of extra `TC`s rather than more implicitly by altering the queries used to derive the instance policies.

Lastly, it is possible to parametrise the second approach by introducing the factor `extraTcType`, which can take any of the following values:

- `none`, where the semantic constraints are ignored: this choice is not recommended
- `minimal`, where the only `TC`s that are added are those that are necessary for the semantic constraints
- `full`, where additional `TC`s of a similar kind to the `minimal` set are added, but that do not change the policy semantics.

This is an added bonus in the sense that, when considering access control evaluation performance, one of the features of interest relates to the "quality" of the policy set, and this factor provides a way of measuring one aspect of this quality.

To implement the second approach, we need to apply the `augmentAsset` and `augmentAction` operations described in Algorithm 4.1.

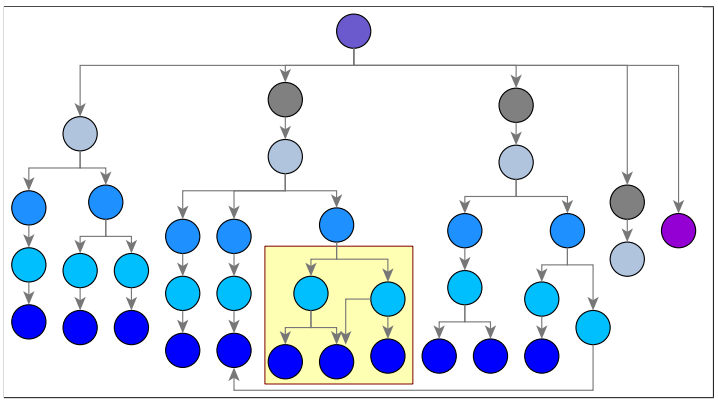
Figures 4.10a and 4.10b can be compared with Figure 4.9 as that represents the cutdown policy specified in the DSL (Listing 4.5), or equivalently `extraTcType` = `none`. As can be seen, the *minimal* policy has two extra `TC`s to ensure that rule `R2` applies to `Asset.type = Document` because the associated Action is `Read` which has `Action.type = Document`. The *full* policy has these additional `TC`s, plus a few more. It adds an `Action.type = Document` condition to the policy `G0` target, because even though there are no other Action `TC`s in that target, the associated Resource `TC`s resolve to `Asset.type = Document` and so this induces an `Action.type = Document TC` to ensure

Algorithm 4.1 The `augmentAsset` operation that is used when `extraTcType` $\neq$ `none` to ensure that the between-category (Asset-Action) semantic constraint is applied. Note that, when `extraTcType` = `full`, additional resource `TC`s are added even if there was no $\{c_i^{(R)}\}$ there in the first place. Otherwise (i.e., `extraTcType` = `minimal`), such empty elements are left unchanged. The `augmentAction` operation is similar but with the roles of `Asset` and `Action` exchanged.

Require: Policy `CTC`facade c; `extraTcType`
Ensure: Add extra `TC`facade c_i and corresponding changes to `CTC`facade

procedure AUGMENTASSET(c, `extraTcType`)
  $\{c_{i,j}^{(A)}\} \leftarrow$ FINDACTIONATTRIBUTETSCIN(c)
  if $\{c_{i,j}^{(A)}\} = \varnothing$ **then**
    STOP
  else
    for all $\{c_{i,j}^{(A)}\}$ **do**
      if attribute($c_{i,j}^{(A)}$) = `Action.type` **then**
        `actionType` $\leftarrow$ value($c_{i,j}^{(A)}$)
      else
        `action` $\leftarrow$ value($c_{i,j}^{(A)}$)
        Lookup `action` = `a` among the `Action` nodes in the property graph
        `actionType` $\leftarrow$ type(`a`)
  $\{c_{i,j}^{(R)}\} \leftarrow$ FINDRESOURCEATTRIBUTETSCIN(c)
  if $\{c_{i,j}^{(R)}\} = \varnothing$ **then**
    if `extraTcType` = `full` **then**
      Create $\{c_{i,j}^{(R)}\}$ based on `Action.type` = `actionType`
      Create $\{c_i^{(R)}\}$ to reference $\{c_{i,j}^{(R)}\}$
      Create $\{c^{(R)}\}$ to reference $\{c_i^{(R)}\}$
  else
    Create $\{c_{i,j}^{(R)}\}$ based on `Action.type` = `actionType`
    Create $\{c_i^{(R)}\}$ to reference $\{c_{i,j}^{(R)}\}$
    Add $\{c_i^{(R)}\}$ to the existing set of `TC`s referenced by $c^{(R)}$

(a) Cutdown COARSE policy with `extraTcType` = `minimal`.



(b) Cutdown COARSE policy with `extraTcType` = `full`.

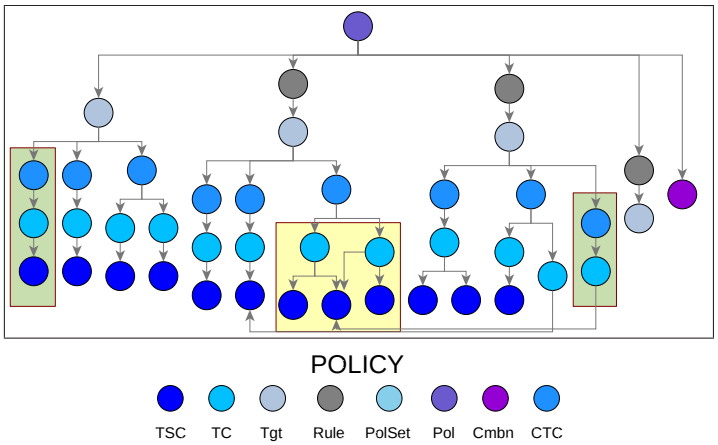


Fig. 4.10 Cutdown COARSE policy model (template policy plus extra `TC`s, represented as a property graph, comparing `extraTcType` = `minimal` with `extraTcType` = `full`. The `minimal` extra `TC`s (compared to the base template policy model (`extraTcType` = `none`) is highlighted with a yellow background. The `TC`s added specifically for the `full` constraint type are highlighted with a green background.

consistency is maintained. Elsewhere, Rule `R3` gains an `Asset.type = Document` condition for similar reasons: the associated Action `TC`s relate to `Read` and `Write`, which have `Action.type = Document` and so `Asset.type = Document` is added to Resource `TC`.

Thus, even in the case of a relatively small policy set, these semantic constraints can add multiple `TC`s to the template policy. Also, the *full* constraints can occur in real policy sets, perhaps accidentally as policies are added over time. Since they do not affect the correctness of the policy decision, it is often difficult to identify such redundant rules. Figures 4.10a and 4.9 indicate how they can sometimes be discovered by visualising the underlying policy graph.

> The template policies are represented in the property graph and hence enjoy the benefits of that representation: ease of manipulation, ease of analysis and the ready availability of many options for visualising the graph and hence the rules governing a domain.

4.4.3 Step 3—derive instance policies and instantiate in the property graph

Each instance policy tree differ from its source template policy tree only in relation to the `Target` elements and their descendants. That is, nodes such as `Policy` and `Rule` are unchanged but each `Target` node should have a relationship with instance-based `CTC`s in addition to the existing relationships with template `CTC`s. `DomainManager` uses the `Granularity` property to distinguish between template (`Granularity` = `COARSE`) and instance-based (`Granularity` = `FINE`) policy model nodes and edges. Furthermore `DomainManager` also distinguishes between `extraTcType` = `minimal` and `extraTcType` = `full` for each `Granularity`.

Starting from each `Target` t, its instance query is used to identify the set of matching static model nodes. Each of the static model nodes in that set is used to derive the `FINE TSC` node $c_{i,j}^{(\odot)}$ that is satisfied by that static model node and no other static model node. That is, if the static model node is `N` and its `name` key has a value of `valu`, the corresponding `FINE TSC` is `N.name = valu`.

A FINE TC $c_i^{(\odot)}$ is created for each of these FINE TSC $c_{i,j}^{(\odot)}$ nodes, where again each of these FINE TC $c_i^{(\odot)}$ nodes has a single HAS_TARGETSUBCOMP relationship with one of the set of FINE TSC $c_{i,j}^{(\odot)}$ nodes. Thus the number of FINE TC nodes is the same as the number of FINE TSC nodes.

The FINE CTC $c^{(\odot)}$ has a HAS_TARGETCOMPONENT relationship with each of the FINE TC $c_i^{(\odot)}$ nodes.

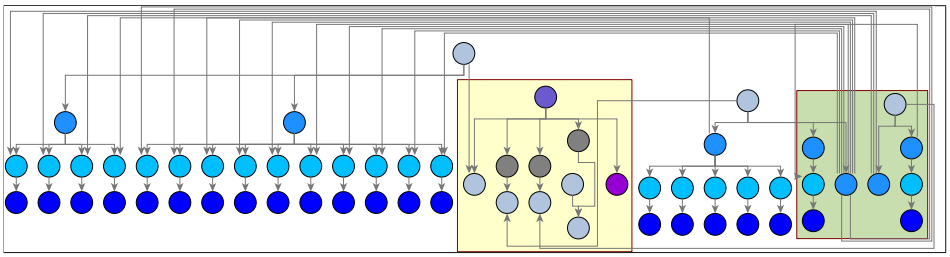
Note that $\odot$ is a placeholder and represents any one of subject (S), resource (R), action (A) or environment (E). Thus the Target node can have a FINE HAS_SUBJECT relationship with the Granularity = FINE CTC $c^{(S)}$. For convenience, the COARSE Target is a different node to the FINE node. Even though they are separate nodes, it is possible to look one up from the other because the FINE Target node has an IS_REFINED_FROM relationship to the COARSE Target node[1].

More details of the algorithm used to refine COARSE policies can be found in Appendix A. In particular it shows that *enumeration* and *aggregation* operations form the core of the algorithm used by PolicyGen.

Referring to Figures 4.11a and 4.11b, it is clear that each Granularity = FINE Target is matched with a Granularity = COARSE Target via an IS_REFINED_FROM relationship, and that all nodes "above" Target in the hierarchy are shared with the COARSE policy. Also, there is a significant increase in the number of TSC and TC components compared to the template (Granularity = COARSE) policy. The differences between the *minimal* and *full* policies are also magnified, as seen by the nodes that are highlighted with a green background. It should be emphasised that these diagrams relate to the *small* domain. As might be expected from the dramatic growth in the static model when the domain size increases (see Figures 4.6 and 4.7) the size of the FINE policy set depends superlinearly upon the size of the domain. However, as can be seen the size of the policy set is decoupled from the complexity of the semantics in the template policy from which it was derived.

[1](Moffett and Sloman, 1993) introduced the concept of policy hierarchies and the refinement operations that convert a policy into a less abstract (equivalently: more concrete) form. In our formulation, there are only two levels in the "policy hierarchy": attribute-based (template) polices with COARSE granularity, and instance-based policies with FINE granularity. Consequently, policy refinement in our formulation relates to deriving the instance-based policies from the attribute-based policy specification.

(a) Cutdown FINE policy with `extraTcType` = `minimal`.



(b) Cutdown FINE policy with `extraTcType` = `full`.

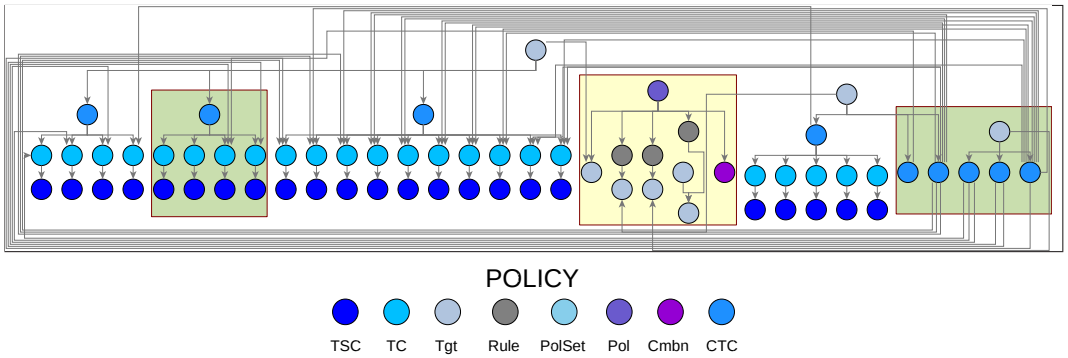


Fig. 4.11 Cutdown template policy model, with `extraTcType` = `minimal` and `Granularity` = `FINE` and `extraTcType` = `full` displayed together for ease of comparison. The highlighted nodes in the middle of each graph are the nodes that are shared with the `COARSE` Policy and its Rules and Targets. The nodes comprising the extra `TC`s that were added to support the semantic constraints are highlighted with a green background.

Because each set of instance policies is *generated* from a common set of template policies and a specific static domain, it is relatively easy to generate instance policies for different domain sizes. By measuring the performance of sets of instance policies that differ by domain size, it is possible to compare the effect of domain size on access control evaluation performance.

4.4.4 Step 4—export policies

The property graph model contains the policy model graph in all its "flavours": granularity, choice of `extraTcType`, choice of linked static model, etc. However, for performance experiments, the corresponding textual language representation needs to be exported.

The first step is that the specification of the required export language needs to be transformed into a (Java) class model. In the case of XACML 2 and XACML 3 policies and requests, this can be achieved using suitable language tools because the language specifications have each been published in Xml Schema Document (XSD) format. Thus, using a Java And Xml Binding (JAXB) provider such as `eclipselink`, it is possible to configure the provider's Xml to Java Compiler (xjc) tool to derive the set of Java classes that are equivalent to the XACML 2 (or XACML 3) XSD model.

Other languages might use different specification formats: Extended Backus-Naur form (EBNF) grammars, etc. For many such formats, tooling such as `antlr` can often be used to help users to create class models to represent language instances.

`DomainManager` follows the path from each source `Policy` and/or `PolicySet` policy model root node to its `TSC` policy model leaf nodes. `DomainManager` maps each node in that path into its corresponding `xjc`-generated class so the resulting object contains the relevant data and fits into the appropriate slot in the object model graph.

The JAXB provider can then marshal this object graph to text, which can then be written to the file system as XACML 2 policy files (say).

The overall data binding process is outlined in Figure 4.12. Note that the generation of the language-specific class model (the upper path labeled "1. XML Schema Document" → "2. Derived Java classes") is a once-off operation.

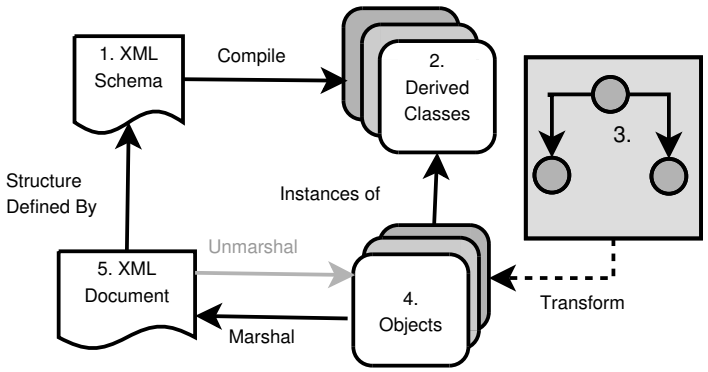


Fig. 4.12 JAXB: Binding Java classes to XML schema documents, transforming the graph policy model nodes to populate the Java objects that are instances of these classes, and marshalling the resulting Java objects to XML documents (policies in this instance).

However, the generation of policy files requires the following steps in `DomainManager`:

1. recursive descent through the property graph policy model *for each* policy model root node;
2. transformation of the policy model nodes on the path to Java objects that are instances of the language-specific class model ("4. Objects" in Figure 4.12);
3. marshalling the Java objects to textual policy files ("5. XML Document" in Figure 4.12).

4.5 Generating requests—RequestGen

The PolicyGen component of `DomainManager`, described in § 4.4, receives template policies and derives COARSE policies (by adding `TC`s to enforce the semantic constraints) and refines these COARSE policies to FINE policies (by evaluating the instance queries and merging static domain data into the COARSE policies). This is sufficient for *static analysis* of the policies. However, a representative set of policies is necessary but not sufficient for domain-aware access control performance experiments. Such experiments require other artifacts, notably a set of requests that is *representative* of the domain, *consistent* with the policies used to control access and *substantial* to capture as many exceptional cases as is reasonable.

§ 4.1.3 describes several approaches and recommends one of these (Approach 2) for the generation of representative and consistent requests. We now present the RequestGen component of `DomainManager`, which implements request generation Approach 2 and which also strives to ensure that the generated request set is substantial.

For Approach 2, discussed in § 4.1.3, where requests are to be generated from the policies, it is necessary to derive the requests from the bottom up, i.e., starting with the `TSC`s. Stage 1 is described in § 4.5.1.

Stage 2 (see § 4.5.2) is to derive the request `TC`s from the policy `TC`s and the request `TSC`s. Although the request `TC`s share many of the properties of the policy `TC`s from which they were derived, their `HAS_TARGET_SUBCOMP` relationships with request `TSC`s are typically more varied than the equivalent policy `HAS_TARGET_SUBCOMP` relationships. This is particularly the case when `useTscReduction` is True. Also, generally speaking, there are *more* request `TC`s than policy `TC`s.

Stage 3 is to derive the request `CTC`s from the policy `CTC`s and the request `TC`s, as described in § 4.5.3.

§ 4.5.4 describes Stage 4, where the `COARSE` Requests are created by generating all possible combinations of request `CTC`s and `TC`s so that each combination is related to a single request.

Stage 5 (presented in § 4.5.5) is where instance-based (`FINE`) Requests are derived from the attribute-based (`COARSE`) Requests that were created in Step 4.

The five stages described above build the requests in a "layer-by-layer" fashion. This is in contrast to policy generation, where the hierarchical structure of the policy set is known *a priori* from the template policies and so PolicyGen can proceed in a top-down manner. Indeed, as a consequence, PolicyGen is always aware of its "place" in the policy graph. However, RequestGen does not have this "structure map" to hand as it derives the request set. Instead, the structure of that request set emerges from the bottom-up. The layer-by-layer evolution of requests also enables consistent treatment of all entities in that layer, so RequestGen enforces semantic constraints at each layer right up to Request.

An alternative specification of Steps 1–4 above can be found in § B.1, where the algorithms used by RequestGen are presented.

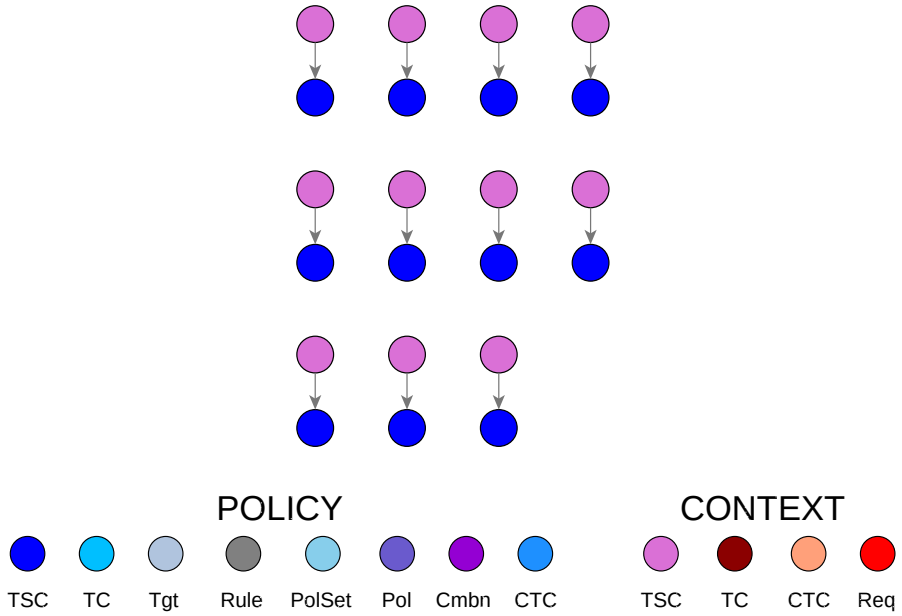


Fig. 4.13 One-to-one correspondence with `IS_DERIVED_FROM` relationship between policy `TSC`s and request `TSC`s.

4.5.1 Step 1—derive the request TargetSubComponents

The context `TSC`s are functionally the same as the policy `TSC`s. `DomainManager` performs a graph-global search for the `Granularity` = `COARSE` `TSC`s ($c_{i,j}^{(P)}$) with the specified `extraTcType`. It then a) makes a copy of each of the nodes, b) changes each copied node's `modelGroup` and c) creates an `IS_DERIVED_FROM` relationship from each request `TSC` to its source policy `TSC`, see Figure 4.13. At the end of Stage 1, `DomainManager` has persisted the set of request `TSC`s $\{c_{i,j}^{R}\}$ in the database.

4.5.2 Step 2—derive the request TargetComponents

The first step is to perform a graph-global search to find all the `COARSE` policy `TC` nodes ($\{c_i^{(P)}\}$). Note that each policy `TC` node is treated the same as all the others: its position in the policy hierarchy is not of interest at this stage. Next `DomainManager` performs a graph-local search on each policy `TC` ($c_I^{(P)}$) to find its set of `COARSE` policy `TSC`s ($\{c_{I,j}^{(P)}\}$).

At this point there is a major choice to be made, depending on the value of `useTscReduction`. In the simplest case, `useTscReduction` is FALSE and the set of

policy TSCs $\{c_{I,j}^{(P)}\}$ is used, without alteration, as the source of the request TSCs $\{c_{I,j}^{(R)}\}$ by looking them up in the graph using the IS_DERIVED_FROM request TSC to policy TSC relationship created in Stage 1. Otherwise, useTscReduction is TRUE and the power set of the set of policy TSC nodes $\mathcal{P}_{I,j}^{(P)} = \mathcal{P}(\{c_{I,j}^{(P)}\})$ is created. By definition, $\varnothing \subseteq \mathcal{P}_{I,j}^{(P)}$ but this is discarded, because the intention is that there should be no *degenerate* coarse TC, where a degenerate TC has no HAS_TARGET_SUBCOMP relationship with a TSC. Consequently, if $|\{c_{I,j}^{(P)}\}| = 1$, it does not matter what the value of useTscReduction is. However, in more interesting cases where the number of policy TSCs for a given policy TC exceeds 1, the power set introduces more variation in the request TCs that can be derived.

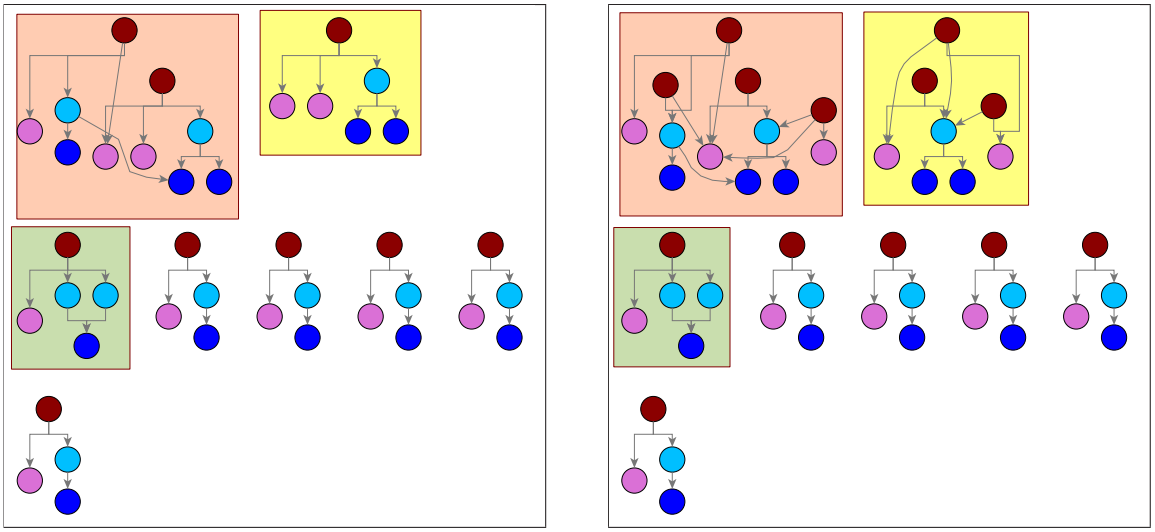
Figure 4.14 is a comparison of COARSE Request TCs for 2×2 combinations of extraTcType (minimal and full) and useTscReduction (with and without). The first observation is that, across all four cases, there are some (unhighlighted) COARSE request TCs that are derived directly from the equivalent policy TC, without change to its structure, typically where the policy TC has a single TSC. Also, it is possible to see the two extra policy TC when extraTcType = full compared to extraTcType = minimal. Interesting subgraphs in Figure 4.14 are highlighted with yellow, pale green and pale orange backgrounds.

The "yellow subgraph" is an instance of the case where a COARSE policy TC (R3_SUBJECT_0) has two TCs (Member.function = Finance and Member.level = Chief). We recall that there is an implicit AND operation between the two TSCs. When useTscReduction is False (Figures 4.14a and 4.14c), the derived request TC has the same structure. However, when useTscReduction is False (Figures 4.14b and 4.14d), RequestGen derives two extra request TCs, one for each of the two individual request TCs.

The "pale green subgraph" shows that the policy TCs R2_ACTION_0 and R3_ACTION_1 are functionally equivalent because they both have a single policy TSC Action.name = Write. They are labeled differently because they have different locations in the template graph. Only one request TC is derived, because the position in the template policy is immaterial for context nodes. However, it is still possible to lookup the two source policy TC nodes by following the two IS_DERIVED_FROM edges. Also, because there is only one TSC involved, the setting of useTscReduction has no effect in this subgraph.

(a) Minimal *Without* TSC reduction.

(b) Minimal *With* TSC reduction.



(c) Full *Without* TSC reduction.

(d) Full *With* TSC reduction.

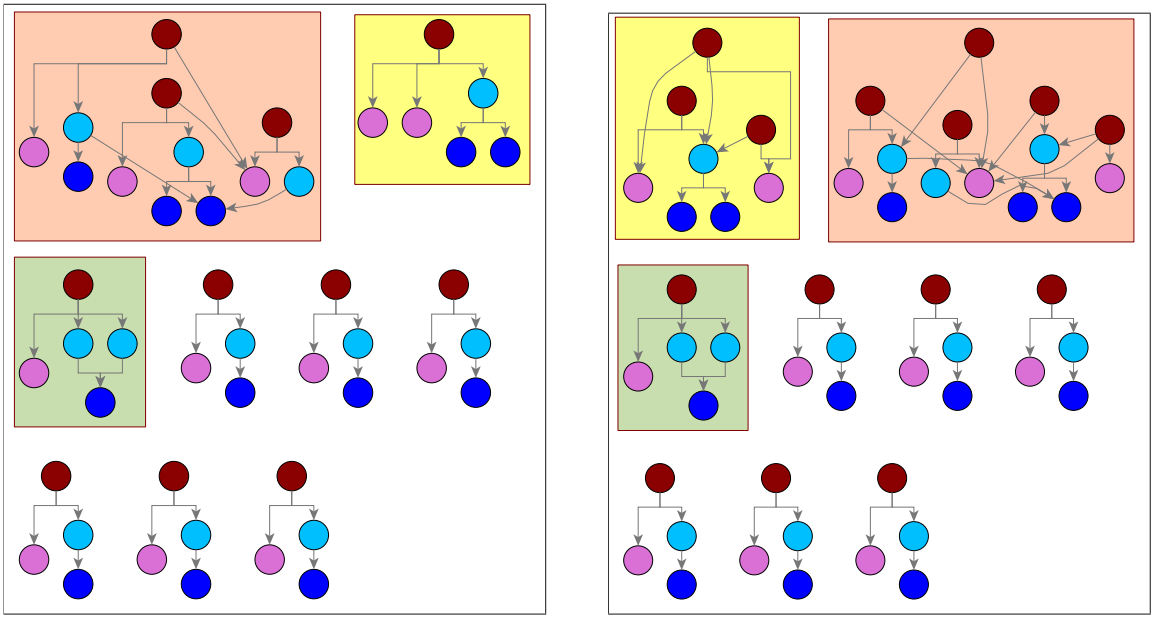


Fig. 4.14 Comparison of COARSE Request TCs, showing their relationship with the request TSCs and the policy TCs.

The "pale orange subgraph" is concerned with the policy TCs R2_RESOURCE_0 and R2_RESOURCE_1 from the template policy and R3_RESOURCE_0 added for extraTcType = minimal. When extraTcType = minimal, policy TC G0_RESOURCE_0 is included in this group, but when extraTcType = minimal, it is treated as one of the unremarkable set of TCs. In most cases, the number of request TCs exceeds the number of policy TCs from which they were derived, particularly when useTscReduction is True (as might be expected).

RequestGen is able to derive *new* request TCs based on the policy TSCs and TCs. The degree to which new TCs are derived depends on various settings that can be controlled by the DomainManager user. The new TCs are available in the graph for use when deriving other request entities.

4.5.3 Step 3—derive the request CollectedTargetComponents

The procedure to derive request CTCs is analogous to that used to derive request TCs. Step 1 performs a graph-global search to find all the COARSE policy CTC nodes ($\{c_k^{(P)}\}$). In Step 2, DomainManager performs a graph-local search on each policy CTC ($c_k^{(P)}$) to find its set of COARSE policy TCs ($\{c_{k,I}^{(P)}\}$).

It is possible to use the policy TCs directly (Step 3a) or, if useTcReduction is True, to use the power set of $\{c_{k,I}^{(P)}\}$ (with the empty set removed) to derive new CTCs (Step 3b). Each element of the reduced power set gives rise to a new request CTC. One of those elements has all the request TCs derived from the source policy CTC, but in the case where $\{c_{k,I}^{(P)}\}$ has more than one element, there will be other elements of the reduced power set with fewer request TCs, resulting in *new* request CTCs. All of this is equivalent to the case when deriving request TCs from policy TCs, with useTscReduction for generating request TCs playing a similar role to useTcReduction for generating request CTCs.

However, there is an additional step when deriving the COARSE request CTCs. Given each set of policy TCs, we can (Step 4) lookup the request TCs for that set, by reversing the direction of the IS_DERIVED_FROM edges for each policy TC. Consequently the set of request CTCs can benefit from the enlarged set of request TCs arising from the case

when useTscReduction is True. Also, by using the set of policy TCs $\{c_{k,I}^{(P)}\}$, rather than the set of request TCs $\{c_{k,I}^{(R)}\}$ when generating the power set, fewer but better CTCs can be generated. Therefore, the set of request CTCs can be made quite comprehensive while still being derived from the policy CTCs.

In Step 5, each RESOURCE request CTC is checked to ensure that it is associated with one and only one Asset.type, as it should be because all RESOURCE policy CTCs are associated with only one Asset.type and each request CTC is derived by disassembling and reassembling related policy CTCs. Any RESOURCE request CTC that does not satisfy this condition is removed from the set. The check proceeds by looking at each request TC associated with each RESOURCE request CTC, descending to its TSC node(s), converting each TSC to an Asset lookup checking the AssetGroup associated with the Assets returned by that Asset lookup.

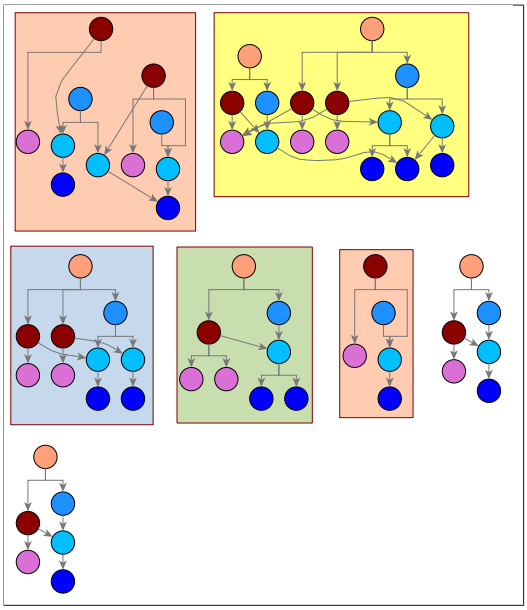
Figure 4.15 compares the generation of request CTCs, when extraTcType = full. It shows the effect of the choice of useTscReduction and useTcReduction. The subgraphs with the pale orange background are COARSE Action TCs and are included for completeness because Actions participate in Requests at the TC level, not the CTC level used by Subjects and Resources. The smaller subgraph relates to the Action.type = Document condition; the larger relates to Action.name = Read and Write.

The two unmarked Request CTCs in each subplot relate to two conditions derived from G0_SUBJECT (Organisation.name = National Bank) and R2_SUBJECT (Member.level = Senior). Since they are isolated single conditions, they are not affected by different settings of useTscReduction and useTcReduction.

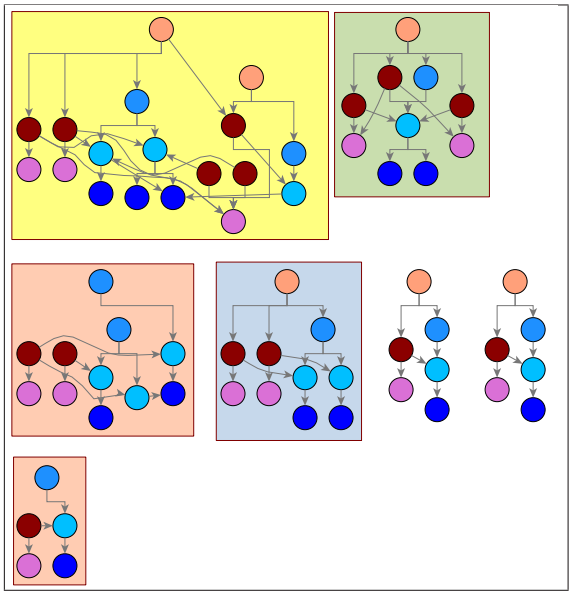
The subgraph with the pale blue background is derived from G0_RESOURCE, which has two conditions Asset.type = Marketing Plan and Asset.type = Corporate Strategy. useTscReduction has no effect because the two policy TC each have a single TSC. However, because of the double condition represented by the two policy TCs, when useTcReduction is True, three CTCs can be formed from the three member set defined by the power set of the TCs less the empty set.

The subgraph with the pale green background is derived from R3_SUBJECT, which has two TSCs Member.function = Finance and Member.level = Chief. In this case, when useTscReduction = True (Figures 4.15b and 4.15d), three possible request TCs are derived from the power set of the set containing the two TSCs. Since there is only one policy TC, the setting of useTcReduction has no effect.

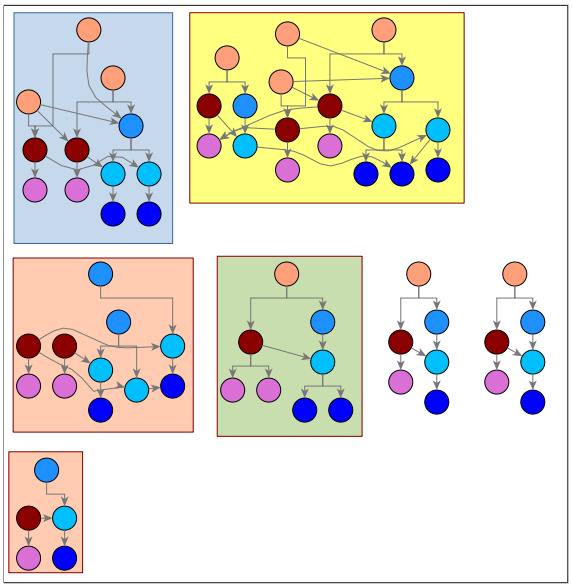
(a) Full *Without* TSC and *Without* TC reduction.



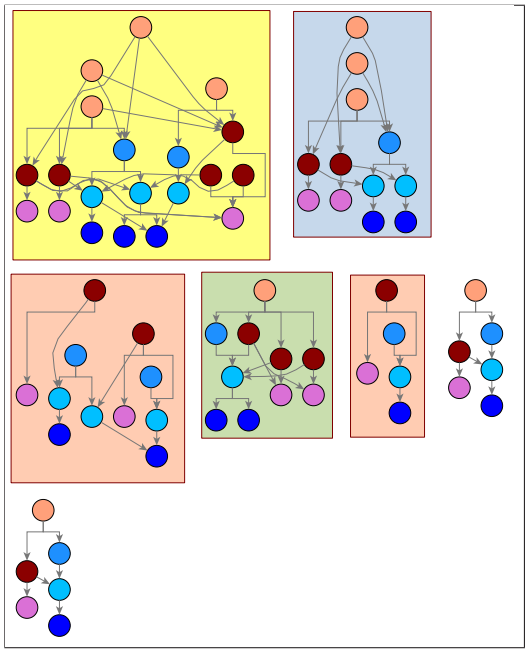
(b) Full *With* TSC and *Without* TC reduction.



(c) Full *Without* TSC and *With* TC reduction.



(d) Full *With* TSC and *With* TC reduction.



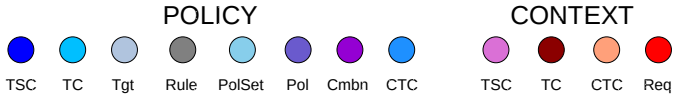


Fig. 4.15 Comparison of COARSE Request CTCs, showing their relationship with the request TCs and the policy CTCs. Also shown is the COARSE Action Request TCs and their relationships (subgraphs with pale orange background).

By far the most complex situation arises in the subgraph with the yellow background. The policy CTCs are R3_RESOURCE and R2_RESOURCE which is associated with two policy TCs each of which is associated with two TSCs. Consequently the R2_RESOURCE policy CTC is the source of seven request TCs (from three policy TCs) when useTscReduction is True and four request CTCs (from two policy CTCs) when useTcReduction is True.

The corresponding figures for request CTCs when extraTcType is minimal is not shown. However, it has the same characteristics the differences are wholly due to the fact that full and minimal policy CTCs are different.

> RequestGen is able to derive *new* request CTCs based on the policy CTCs and any request TCs. Settings such as useTscReduction and useTscReduction can be used together and their effect is cumulative, but controlled, so that a) the request CTCs can differ but not too much from the source policy CTCs and TCs and b) there are few if any logically-redundant clauses of the form $A \wedge A$ and $A \vee A$.

4.5.4 Step 4—create COARSE (attribute-based) requests

Unlike XACML 3, XACML 2 treats SUBJECTs and RESOURCEs differently from ACTIONs and ENVIRONMENTs Moses (2005). In terms of our property graph model, for each XACML 2 Request, the SUBJECT and RESOURCE are each CTCs and the ACTIONs and ENVIRONMENTs are each TCs.

In Step 1, DomainManager performs a graph-global search for each of the following:

- SUBJECT request CTCs
- RESOURCE request CTCs
- ACTION request TCs
- ENVIRONMENT request TCs

Step 2 is to classify the RESOURCE CTCs by AssetGroup. By classify, we mean that a set of CTC is associated with a value of Asset.

Step 3 is to classify the ACTION TCs by ActionType. Note that the values of AssetGroup and ActionType are drawn from the same set: Document, Communication, etc.

Step 4 is to derive the expanded resource CTCs. Each (expanded) CTC can be associated with a single AssetGroup. If a CTC does not satisfy this condition, it is expanded by ANDing it with a CTC with this property. We note that the condition is satisfied by construction for policies when extraTcType is either minimal or full. The condition continues to be satisfied unless either TSC or TC reduction is applied. In such circumstances, a subset of request TCs might not be sufficient to associate the derived CTC with a single AssetGroup.

Step 5 makes 3 passes through the augmented resource CTCs. The first pass "pivots" the labelled resource CTCs so that instead of assigning an AssetGroup label to each resource CTC, resource CTCs sharing the same AssetGroup are collected together in a set, and the entire set (and not its individual elements) is labelled with AssetGroup. The second pass considers the set of resource CTCs that has not yet been assigned an AssetGroup. Let $c^{\text{unclassified}}$ be an unclassified resource CTC. DomainManager can consider each of the "active" AssetGroups, and each of the classified resource CTCs associated with that AssetGroup. Let $c^{\text{classified}}$ represent any of these resource CTCs. Then, by definition, $c^{\text{unclassified}} \wedge c^{\text{classified}}$ is associated with the same AssetGroup as $c^{\text{classified}}$ and so can be added to the set of such classified resource CTCs. Note that *conjunction* (ANDing) of two CTCs is achieved by recursively ANDing their TCs by assembling the set union of their TSCs.

Symbolically, DomainManager computes

$$
\begin{aligned}
c^{\text{augmented}} &= c^{\text{unclassified}} \wedge c^{\text{classified}} && (4.8)\\
&= \{c_i^{\text{unclassified}}, \forall i\} \wedge \{c_j^{\text{unclassified}} i, \forall j\} && (4.9)\\
&= \{c_i^{\text{unclassified}} \wedge c_j^{\text{unclassified}}, \forall i, j\} && (4.10)\\
&= \{\{c_{i,l}^{\text{unclassified}}, \forall l\} \wedge i\{c_{j,m}^{\text{unclassified}}, \forall m\}, \forall i, j\} && (4.11)\\
&= \{\{c_{i,l}^{\text{unclassified}} \cup c_{j,m}^{\text{unclassified}}, \forall l, m\}, \forall i, j\}. && (4.12)
\end{aligned}
$$

A side effect of the second pass is that more resource CTCs can be generated by this CTC conjunction process. Each combined CTC is traceable back to its two source request resource CTCs using AUGMENTED_FROM_UNCLASSIFIED_RESOURCE and

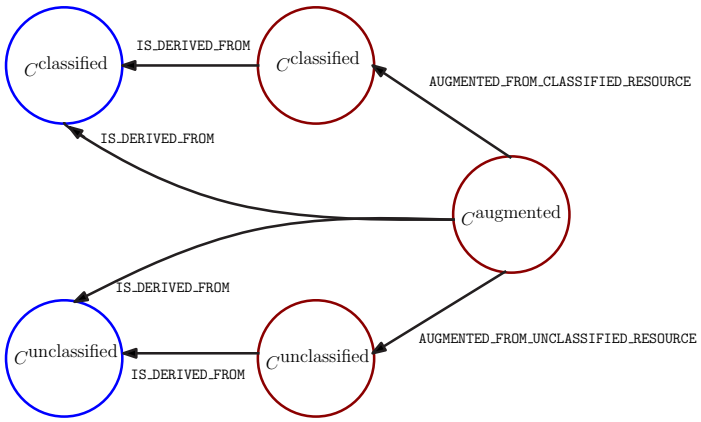


Fig. 4.16 The augmented *request* resource CTC $c^{\text{augmented}}$ is derived from an unclassified *request* resource CTC $c^{\text{unclassified}}$ and a classified *request* CTC $c^{\text{classified}}$, each of which is derived from a *policy* resource CTC. Note that, by transitive closure, $c^{\text{augmented}}$ has an IS_DERIVED_FROM relationship back to each of the source policy CTCs.

AUGMENTED_FROM_CLASSIFIED_RESOURCE relationships. It is then possible to compute the transitive closure of these relationships, together with the IS_REFINED_FROM relationships, as shown in Figure 4.16.

In the third pass, the unclassified RESOURCE request CTCs are removed from the data structure containing the output of the first pass, and the augmented RESOURCE request CTCs arising from the second pass are distributed in that data structure, depending on their associated AssetGroup.

> The semantic constraints are checked and reapplied if necessary to ensure that each entity that is available for combination as an access request is internally consistent and has metadata to ensure that it can be combined with other entities to form a semantically valid request.

Now that the components of a request are in place, it is necessary to generate *all valid combinations* as requests. DomainManager uses the following nested loop structure

```
For all Action TC: a
  Lookup the ActionGroup of a: aG
  For all Resource CTC with AssetGroup aG: r
    For all Subject CTC: s
      For all Environment TC: e
        generateRequest(s, r, a, e)
```

As can be seen, the semantic constraints are built into the loop structure and ensure that the generated Action-Resource combinations are valid. If other cross-entity constraints are needed, they can be applied in a similar way.

Figure 4.17 shows the requests that are generated by RequestGen, depending on factors that also affects policies (`extraTcType`) and that affect only request generation (`useTscReduction` and `useTcReduction`).

Although the plots look complex, the structure is repeated for each request. That is, each request has a relationship to a SUBJECT `CTC`, a RESOURCE `CTC` and an ACTION `TC`, similar to the XACML 2.0 request hierarchy in Figure 4.18. Note that there is no ENVIRONMENT `TC`, because it does not appear in the template policy (see Listing 4.5). The `CTC`s and `TC`s are shared amongst the set of COARSE requests, which is designed to contain all semantically valid combinations. Clearly the settings affect the number of requests, but this is because they affect the number of `CTC`s and `TC`s.

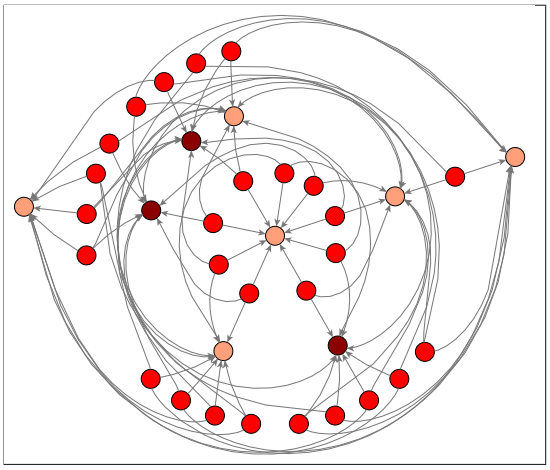
As soon as the generated requests have been added to the property graph, they are available for analysis (see §4.6.1) and for use in performance experiments. As was the case with PolicyGen, it is necessary to export the requests as XACML using a similar procedure to that outlined in § 4.4.4. Interestingly, the requests can be exported in JSON-encoded XACML 2.0 as well as the more traditional XML representation, if that is desired. Also, `DomainManager` can export requests in *property specification* format equivalent to that employed by the policy author when specifying the policies. See Listing 4.4 for an example of the syntax of such a policy specification. The advantage of the property specification format is that it enables easier comparison between template policies and the requests that were generated from them.

> RequestGen generates exhaustive combinations of the SUBJECT, RESOURCE, ACTION and ENVIRONMENT domain entities, subject to any cross-entity semantic constraints, and these are exported as individual requests.

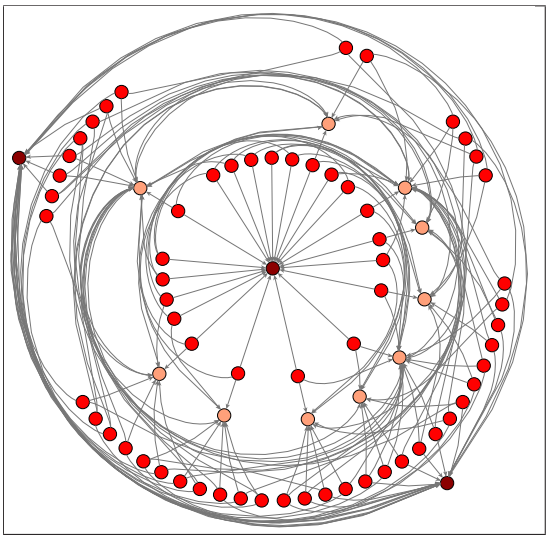
4.5.5 Step 5—create FINE (instance-based) requests

The procedure for generating instance-based requests is analogous to that for generating instance-based policies from the `granularity` = `COARSE` policies, which

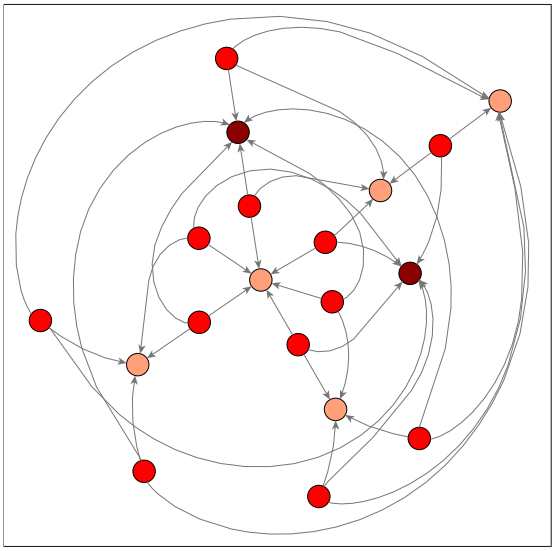
(a) Full *Without* TSC and *Without* TC reduction.



(b) Full *With* TSC and *With* TC reduction.



(c) Minimal *Without* TSC and *Without* TC reduction.



(d) Minimal *With* TSC and *With* TC reduction.

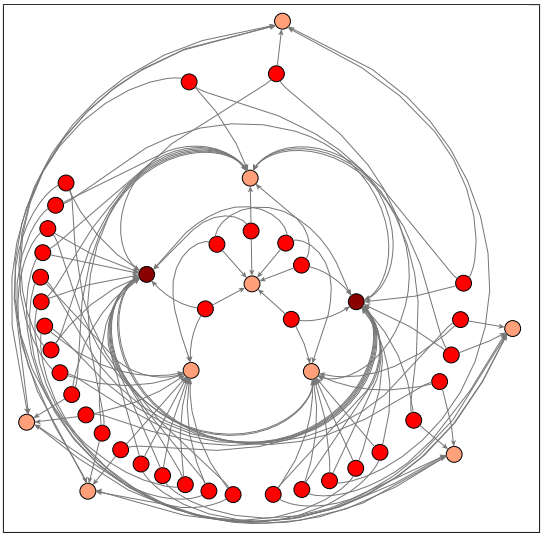


Fig. 4.17 Comparison of COARSE Requests, for different settings of `extraTcType`, `useTscReduction` and `useTcReduction`.

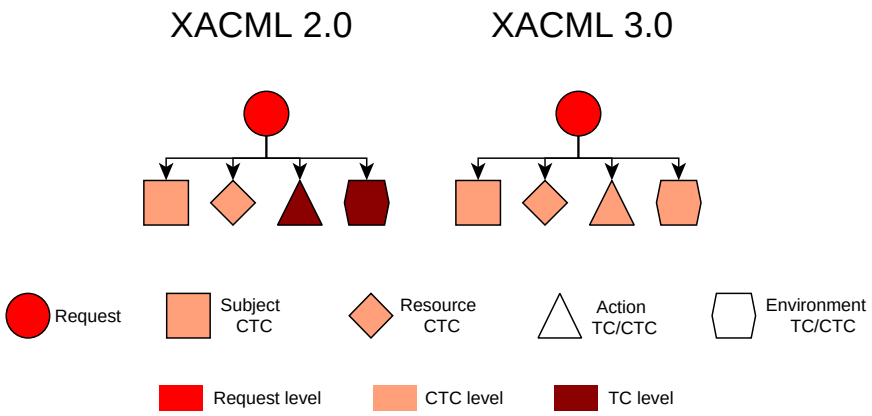


Fig. 4.18 Comparison of the structure of XACML 2.0 (note the mix of SUBJECT and RESOURCE `CTC`s and ACTION and ENVIRONMENT `TC`s) versus XACML 3.0 (`CTC`s only) requests.

themselves are slightly modified (owing to the addition of semantic constraints in the form of extra `TC`s) versions of the template policies specified by the `DomainManager` user.

The main difference between policy and request generation is that requests are built "layer-by-layer" for the reasons outlined in § 4.5. Stages 1 to 4 are described in § 4.5.1,4.5.2, 4.5.3 and 4.5.4.

Instance-based requests are refined from the attribute-based requests in a top-down manner. The top level structure of XACML 2.0 and XACML 3.0 requests is compared in Figure 4.18. Note that, at this level of detail, the structure of attribute-based and instance-based requests is identical for a given version of XACML.

For each COARSE entity whose parent is Request, `DomainManager` looks up the instance nodes that satisfy the entity's instance query. For example, the instance nodes satisfying a SUBJECT `CTC` instance query might be `Alice` and `Bob`. The corresponding instance `TSC`s would be `Member.name = Alice` and `Member.name = Bob`. The instance `TC`s would look the same: `Member.name = Alice` and `Member.name = Bob`. In this regard, we note that there is only one instance `TSC` per instance `TC` given the equality constraints that are currently supported by `DomainManager`. This is because an entity (a Person in this case) cannot, simultaneously, correspond to more than one unique instance. The instance `CTC` would be `Member.name = Alice OR Member.name = Bob`.

When the COARSE entity is a TC (say a XACML2.0 ACTION), care is needed, because only one ACTION instance can be used in that place. This is in contrast to an instance CTC, where the set of instances is unbounded. Therefore DomainManager needs to ensure that the COARSE TC is semantically consistent before it can be refined to an instance-based (FINE) TC. Generally it is easier to achieve this with CTC nodes because TC nodes can be added (or the TC reduction can be limited, so they are not removed)) to ensure that the instance nodes belong to a single AssetGroup or ActionType such as Document.

It should be noted that each COARSE CTC induces an instance CTC which has its own hierarchy of TCs and TSCs. For XACML 2.0 requests, each COARSE Action and Environment request model TC node does the same.

For each derived node, an IS_REFINED_FROM edge is added, relating that instance node to the attribute node which was its source. As with the policy model, for each instance-based request model node, the Granularity property takes the value FINE; for the source attribute-based request model node, the value is COARSE.

> RequestGen generates an instance-based request corresponding to each attribute-based request, where the instances depend on the STATIC domain model and are combined according to the structure of the source attribute-based request.

4.5.6 Varying the request complexity

As described in § 4.5.5, the instance-based requests are derived from the attribute-based requests by querying the database for each of the attribute-based REQUEST model nodes. This is the same procedure that is followed in the POLICY model, but the interpretation is different. For policies, the full set of instance-based nodes is required each time. Otherwise it is impossible to guarantee that the semantics of the template policy will be honoured for all requests. However, the request does not need to be as comprehensive. Often requests need to reference a small number of instances. Therefore we can control the *request complexity* by controlling how many of the instances that match the relevant instance query (such as "Look for Persons where Member.function = Sales AND Member.role = Manager") are

used. `DomainManager` distinguishes between `requestComplexity` = `Sgl`, `Dbl` and `ALL`, depending on whether the instance query result set is limited to one (e.g., Alice) or two (e.g., Alice and Bob) matching instances, or is unlimited (e.g., the entire Department of which Alice and Bob are members). Note that the `requestComplexity` applies only to instance-based requests but otherwise can be varied independently of other settings such as the (static) domain size, or `useTcReduction`, etc.

Therefore, `DomainManager` generates three FINE requests (labeled `Sgl`, `Dbl` and `ALL`) for each COARSE request.

> Different request complexities of instance-based requests can be generated together and labeled as such, making it easier to compare the effects of different request complexities.

A further consequence of the instance-based query is that sometimes, particularly when the instances are limited (e.g., `Sgl`) and/or the the domain size is small, the results of the query can be the same for several different instance queries. For example, "Alice" might satisfy the query `Member.level` = `Senior AND Member.function` = `Finance`. She might also satisfy a seemingly unrelated instance condition such as `Join.year` < 2010. When the attribute-based request is assembled from such general conditions, these queries are treated as distinct. However, if the `requestComplexity` is `SGL`, the resulting instance `CTC`s will each be `Member.name` = `Alice`. Therefore the set (more correctly: *bag*) of refined instance-based requests will contain duplicate entries. This is unfortunate for two reasons:

1. the dimension of the space of generated requests is less than expected, so there is the possibility that the requests fail to sample the PDP execution paths effectively.
2. if we assume all the instance-based requests are unique, this could lead to clustered service times (the measured outputs) just because the inputs (in this case the requests) are clustered. This is misleading and could result in spurious observations being made about the data.

`DomainManager` takes account of this feature (the near certainty that the derived instance queries will contain some duplicates) by using the following algorithm:

```
1. Find groups of duplicate requests, say those labeled A,B,C.
2. In the file system:
   Keep exported request A and delete the exported requests B and C.
3. In the database:
   Add IS_EQUIVALENT_TO relationship between B and A and between C and A.
```

More details of this algorithm are presented in § B.2.

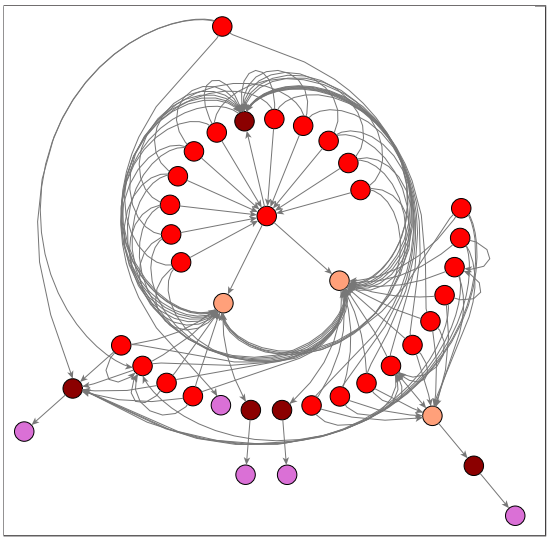
> The requests that are submitted to the PDP for performance testing can be assumed to be unique at the instance-level. Consequently, if two requests are found, consistently, to have very similar service times, this feature is worthy of investigation.

The `requestComplexity` factor is a characteristic of access control policies in practice as well as being a technical setting for RequestGen. Generally, simple access requests are associated with questions like "Can X do Y with Z?". More complex scenarios require much more context to be provided to the `PDP`. For example, to avoid conflicts of interest, the access request might look like "Can X1, X2, X3 and X4 do Y with Z?". This is an example of a request with higher complexity, in much the same sense as the `requestComplexity` factor used by RequestGen in `DomainManager`. Another way such complex requests might arise is if the requesting entity is a proxy for a set of users and/or resources—in this way, access requests are "batched together" and a single decision is given for all entities participating in the request. Therefore, by carefully tuning the `requestComplexity` parameter, it is possible to model some interesting scenarios.

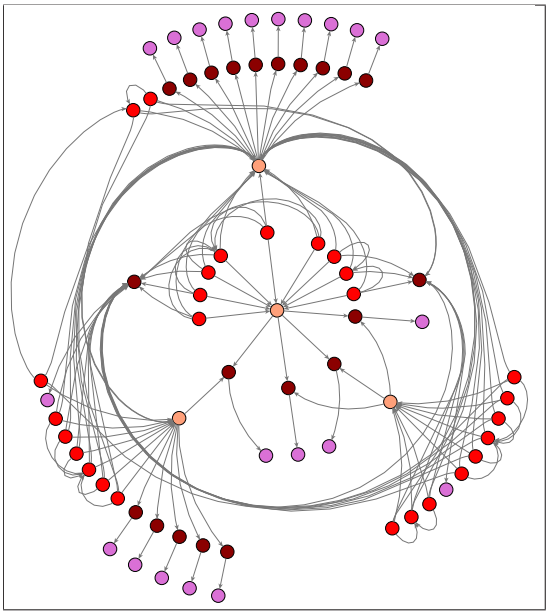
Figure 4.19 compares the generated FINE requests, showing the effects of the combination of the Boolean-valued `useTscReduction` and `useTcReduction` factors, as well as comparing the least complex (`SGL`) and most complex (`All`) settings. Conservative `extraTcType` = `Full` COARSE policies were used by RequestGen in each case. When refining these COARSE policies, the *small* static domain was used.

Figures 4.19a and 4.19d can also be compared with the equivalent COARSE requests (Figures 4.17a and 4.17b, respectively). The layout algorithm used when plotting rearranges the nodes and edges, but the number of Request, `CTC` and Action `TC` nodes does not change. The most interesting change is the addition of Request-to-Request `IS_EQUIVALENT_TO` relationships, which are particularly noticeable in Figure 4.19a.

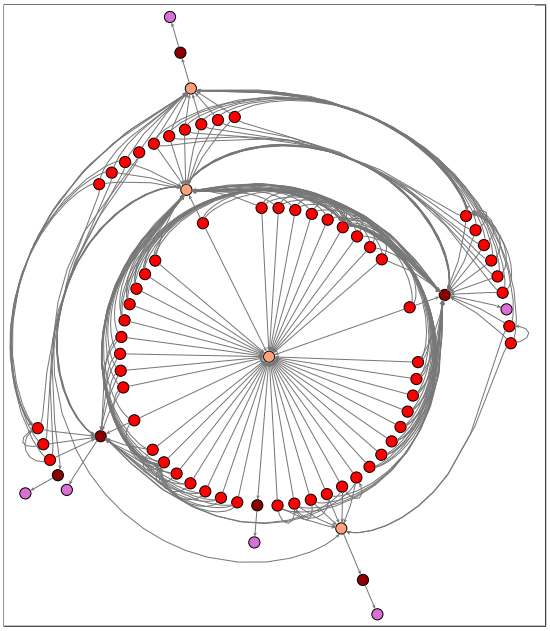
(a) Full *Without* `TSC` and *Without* `TC` reduction; `requestComplexity` = `SGL`.



(b) Full *Without* `TSC` and *Without* `TC` reduction; `requestComplexity` = `ALL`.



(c) Full *With* `TSC` and *With* `TC` reduction; `requestComplexity` = `SGL`.



(d) Full *With* `TSC` and *With* `TC` reduction; `requestComplexity` = `ALL`.

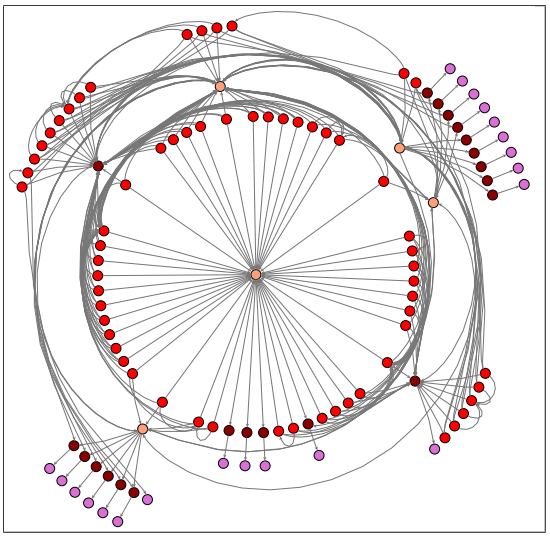


Fig. 4.19 Comparison of FINE (instance-based) Requests, ranging from the smallest and simplest (`useTscReduction` = `useTcReduction` = False; `requestComplexity` = `SGL`) to the largest and most complex (`useTscReduction` = `useTcReduction` = True; `requestComplexity` = `ALL`).

Note that comparatively few instance-based TSCs are used: 5, 6, 21 and 21 in Figures 4.19a, 4.19c, 4.19b and 4.19d, respectively. Yet RequestGen in DomainManager is able to generate 27 (Figures 4.19a and 4.19b) and 63 (Figures 4.19c and 4.19d) requests. For Figure 4.19a there are only 4 unique requests; for Figure 4.19c there are 8, for Figure 4.19b there are 6 while for Figure 4.19d there are 12. The rate of unique FINE requests per generated COARSE request is interesting: 0.15, 0.13, 0.22, 0.19. Thus for a relatively small static domain and few policy rules, typically less than one in 5 of the generated requests are unique. One interpretation of this finding is that the scope to generate broadly based requests, starting from the policies, is limited, particularly when the static domain has small size. Another, perhaps more favourable interpretation, is that there were relatively few rules in the policy to begin with, and the number of unique FINE requests that can be generated is limited by the semantic complexity of the underlying policy. Thus, in an indirect way, RequestGen is also able to provide insight (in the form of visualisations such as those in Figure 4.19) into the policy set being used in ATLAS performance experiments.

4.6 DomainManager Evaluation

4.6.1 Graph measures

Since the generated policies and requests are represented as graphs, graph-theoretic measures can be used to estimate policy set size and complexity. Many of the better-known graph measures are more suited to small-world networks. However, policies and requests have a different structure, being *forests* (collections of trees). The proposed graph measures are *leaf count* and *total path length*. For the latter, all paths start from the root(s) of the corresponding graph structure (either a policy set or a request) and each step in the path has unit length.

Such graph measures summarise the underlying structure but ignore semantic considerations. For example, some rules (hence branches in the policy tree) are evaluated more frequently and this feature is not included in either measure. Thus (static) graph measures do not take account of such evaluation-time (policy-with-request) behaviour. However the measures can be calculated easily by DomainManager and may even be able to give a rough indication of relative performance.

Since the domain model we use is a graph representation, visualisation can provide insight into what policy properties "look like" and why the performance has a particular profile. For example, Figure 4.7 indicates why, when `Granularity = FINE`, both policy leaf count and path length grow so rapidly with domain size, and can also be used to motivate the choice of other graph measures that might be relevant. Inspection of sets of these diagrams, and those of the policy and context models, can help when interpreting the measured service times.

4.6.2 Service time analysis

The policy and request generation algorithms are applied in the following scenario. A security administrator ("Alice") wishes to estimate the effect of (static) domain size on policy evaluation performance. She designs an experiment in which the domain size factor (denoted `DS`) has levels {`S`, `M`, `L`} (small, medium and large). She also has secondary questions concerning the effects of different policy and request generation choices, so she adds four extra factors:

- `SC` (static constraints), which indicates whether the policy generator adds the (`M`) `minimal` set of static constraints or goes further and over-specifies them (`F`: `full`), thereby increasing policy redundancy;
- `RC`, which indicates whether there are single ('`Sgl`') or multiple ('`Mlt`') conditions per request `TC`;
- `TscR`, which indicates whether reduced request `TSC`s are used when generating requests (`t`) or not (`f`);
- `TcR`, which indicates whether reduced request `TC`s are used (`t`) when generating requests or not (`f`).

Her experiment runs on a server (Ubuntu 14.04.1 LTS, Java 7, 4 cores, 2Gb memory) running SunXacml PDP (subversion revision 137). She collects her measurements using STACS (Butler et al., 2010, 2011).

Listing 4.4 is the template policy used in the experiment. There are 4 `Action.type`s labeled 'Document' and 2 labeled 'Communication'. For domain size `S`, `M`, `L` there are 10, 60 and 110 assets respectively, and 6, 40 and 74 persons respectively. The generated policies for domain size `S`, `M`, `L` have 157, 834 and 1534 conditions if `SC=F`

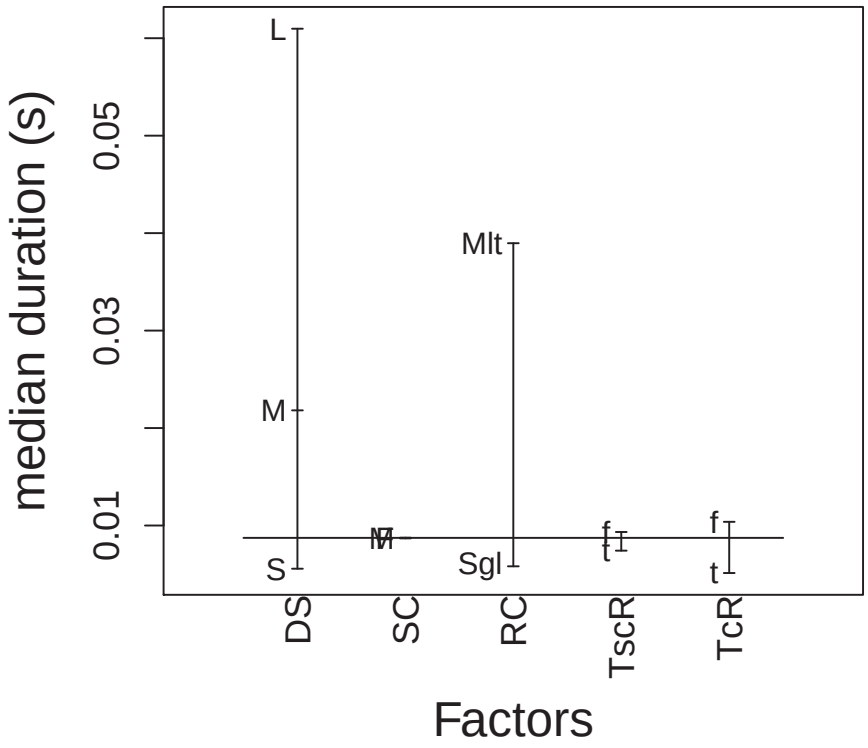


Fig. 4.20 Median service times for each level of each factor, as well as the overall median. Both the domain size `DS` and request complexity `RC` are seen to have significant impacts on the median service time, where as the settings for `SC`, `TscR` and `TcR` have a much weaker influence—the median service for each level of these factors departs very little from the grand median.

and 78, 444 and 823 conditions if `SC`=`M`. The number of *unique* generated requests depends on the domain size, the request cardinality and the $2^2 = 4$ combined settings of `TscR` and `TcR`. It ranges from 14 requests (`DS`=`S`, `RC`='Sgl', and `TscR`=`TcR`='N'; 3 `TSC`s per request) to 128 requests (`DS`=`L`='f', `RC`='Mlt', and `TscR`=`TcR`='t'; average 150 `TSC`s per request). The requests were issued in random order to the PDP. Nine replicate service time measurements were collected per unique combination of parameters.

Figure 4.20 indicates that service times tend to increase markedly with domain size (`DS`) and request complexity (`RC`). Figure 4.21a confirms this heuristic and suggest that as the domain size increases there are clusters of requests whose services times increase

(a) Service time density: `DS` in {'S','M','L'} (b) Service time density: `RC` in {'Sgl','Mlt'}

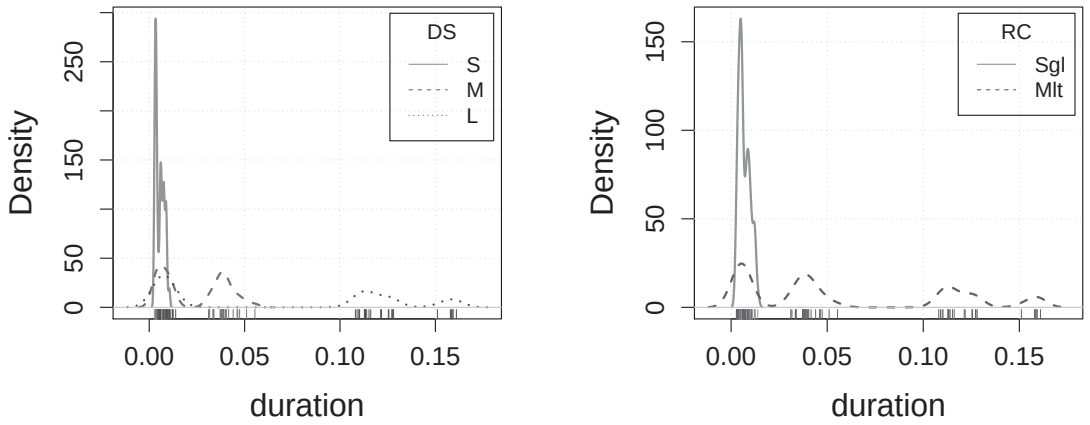


Fig. 4.21 Selected service time density plots. Three clusters of service times are apparent in each plot and appear to be associated with larger domains (hence policies) and higher cardinality requests.

significantly. This information, together with request frequency data from production systems, can subsequently be used to assess the likelihood that the access control system is meeting its service level agreements. The trends are not surprising as the SunXacml PDP always decides requests by performing a full scan of the policies. Meanwhile, Figure 4.21b suggests that single request cardinality leads to shorter and more predictable service times.

This preliminary analysis indicates the effects of some factors that were not available and hence not considered in out earlier papers Butler et al. (2010, 2011). However it is not complete, because it is likely that there is an interaction between domain size (hence policy set size) and request complexity (hence request size), but this analysis does not take account of it. The actual effect of a factor is computed as the sum of the main effects (of that factor on its own) and the contributions from the interactions in which it is a member. If those interactions effects are significant, the resulting effect could differ significantly from the main effects (ignoring interactions). Therefore, a new analysis component (denoted `PARPACS`) was developed to provide a more comprehensive and statistically robust analysis of how these new factors influence access control service times. This new component is presented in Chapter 5.

4.7 Summary

This chapter addresses one of the main drawbacks with (Butler et al., 2011) and similar papers—namely, that it is difficult to extrapolate from performance measurements gathered using a small, domain-specific set of policies and requests to provide insights into likely performance in an arbitrary deployment scenario.

The evaluation in §4.6 shows how the policy and request generation algorithms can be applied in practice, and how the types of analysis described in Chapter 3 can be applied to a more relevant domain configuration. This is a significant contribution, as previous studies of access control performance have been hampered by the lack of a) policies and requests tailored to the domain under study and b) a means of generating suites of such policies and requests, fully parametrised and graded by difficulty.

Note that `DomainManager` is tuned to the problem of specifying policies and requests for use in access control policy evaluation experiments. Hence it would probably need to be replaced if `ATLAS` were to be used for other types of experiments, such as predicting database query performance.

Chapter 5

Analysing enterprise access control performance with PARPACS

Table 5.1 Research questions addressed in Chapter 5

ID	Question
RQ1	How can access control evaluation performance be measured for use in performance experiments? – What form does the service time distribution take? – What simulations can be performed to explore the effect of different request arrival patterns? – What analysis can be performed when the systems under test use different languages, frameworks and encodings?
RQ2	How can domain models be specified and used to express enterprise access control scenarios? – How can different variants of domain models be specified in a flexible and easy to use way? – How can access control evaluation performance be compared at different domain sizes?
RQ3	How can the data from performance experiments be used to understand and predict access control evaluation performance? – What types of exploratory data analysis are suitable for the performance experiments? – What are the steps needed to build statistical models predicting access control performance?
RQ4	What are the main factors affecting access control evaluation performance? – What are the effects of PDP choice, domain size and resources? – What are the effects of domain size, policy and request characteristics?

5.1 Adding more factors

One of the most interesting features of `DomainManager` and its associated property graph model is the fact that it enables rich enterprise access control scenarios to be modelled. Some of the factors that have been considered to date include:

basic The following factors are available when the researcher has minimal control over the policies and requests, so the policies and requests are used as provided:

- **PDP** As has been seen in §3.3.2, PDPs differ in respect of both mean service time and induced request clusters. This is even more apparent when the PDPs use different technologies (§ 5.3 and § 5.4);
- **request groupings** §3.4.6.3 shows that there are performance differences between the `single` and `multi22` request groups in respect of their performance with the `continue-a` policies;
- **host** §3.4.6.3 also shows that performance is affected by differences in CPU, memory and other resources. Generally, performance increases as more resources become available, but there are exceptions, see § 5.4;
- **encoding format** §3.5.1 indicates that the encoding used (XML versus JSON) might also have an effect, e.g., because one is easier to parse than the other, although this factor might be confounded with the PDP factor;

advanced When the policies and requests are generated using `DomainManager`, many additional factors can be considered in performance experiments. We have already seen the following

- **Policy specification redundancy** Policy semantics can be over- or under-specified. For example, rule clauses can be added that are redundant in the sense that their removal does not change the policy decision from Permit to Deny or vice-versa. Optionally, `DomainManager` is able to generate over-specified policies.
- **Request Complexity** Instance-level requests are generated by querying the property graph. Optionally, the result set of each query can be limited to 1, 2 or all instances of `TSC`s satisfying the query. Thus the size and complexity of the requests can thus be represented by the Request Complexity factor.

TSCr, TCr When generating requests, it is possible to tweak the generated requests to make them more or less similar to the policies from which they were generated. These factors can be used to represent different mixes of requests that might be more or less similar to their policy source.

However, as will be seen in this Chapter, many more factors are available. The analysis in previous chapters has used visualisation (predominantly service time distribution (density) plots, boxplots by factor-level combination and design plots) and ANalysis Of VAriance (ANOVA) tables (to indicate what factors appear to be statistically significant) and ANalysis Of Means (ANOM) tables (to estimate the service time for a given factor when other factors are held constant).

This type of analysis is limited and does not scale particularly well to larger and more complex sets of factors, for the following reasons:

- as factor numbers grow, analysing their interactions becomes more important because, even though more of the behaviour in the dependent variable (service time in this case) is explained by the extra factors, it becomes more difficult to ascribe that behaviour to the factors themselves. Interactions also need to be considered—they were not considered in their own right in the analysis described in previous chapters;
- other types of plot (such as residual plots, effects plots, Cook's Distance plots) are more appropriate when trying to answer more complex statistical questions, such as: Do all the service times come from a single distribution, or are there contaminants from another distribution that could be affecting the results?
- ANOVA and ANOM can be used to compare given factor-level combinations, but without a common *statistical model* it is difficult to make more general statements, particularly relating to factor-level combinations that were not measured in the experiment. Also, if the model includes significant interactions of non-significant factors, the ANOM results need very careful interpretation.

Therefore the `PARPACS` (Butler, 2015b) component of `ATLAS` was developed to fill these gaps. As will be seen in §5.3, it greatly enhances the utility of the access control performance experiments, and integrates well with the existing `DomainManager` and `STACS` components in the form of the `ATLAS` framework, see Figure 5.1.

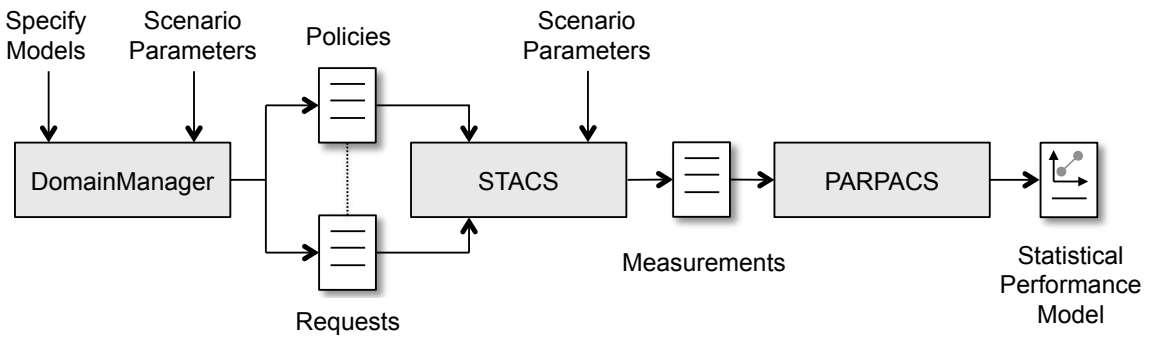


Fig. 5.1 Overview of the ATLAS framework. Users specify static, policy and request models along with scenario-specific parameters. DomainManager generates representative policies and associated requests, which provide the input for the STACS measurement process. PARPACS analyzes the raw measurement data and generates a statistical performance model.

5.2 PARPACS Overview

PARPACS provides statistical analysis of STACS measurement data; it has two main objectives:

1. to enable comparison of the decisions made by combinations of policies and requests—PARPACS can check that the semantic effects (access control decisions) are consistent across the parameters that vary in the experimental scenario;
2. to investigate the relationship between the factors in the experiment and the measured service times, and enable derivation of statistical models predicting access control performance.

For the latter objective, PARPACS extracts the required service times from the measurements obtained using STACS. To do this, it employs a SQL SELECT statement with conditions that select the required service times. In cases where the source measurements have more factors than are needed in a given analysis scenario, a (weighted) average can be taken over the unwanted factors and the resulting service times will have only the required factor settings. As an example, let the measured service times be associated with factors $F_{\text{measured}} = \{f_1, f_2, \ldots, f_p, f_{p+1}, \ldots, f_q\}$, and the desired service times be associated with factors $F_{\text{analysis}} = \{f_1, f_2, \ldots, f_p\}$. Then a simple query specifying just F_{analysis} will return multiple rows per factor level combination so, by convention, the mean of these service times is used as the

representative value. In practice, this is achieved by applying a relational projection with aggregation on `duration` by the `AVERAGE` function.

The resulting set of data is then loaded into an R (R Core Team, 2014) dataframe for analysis. The first step is to explore the data using plots and statistical tests. The objective of this exploration is to determine whether the data, in its present form, can be fitted successfully with a linear model. If the data is not suitable yet, the exploration indicates what steps to take to configure the data and/or the linear model so that the resulting fitted model faithfully captures the information and hence the predictive capacity of the data. The fitted model can then be used both to understand the underlying relationship between the controlled factors (such as domain size, memory allocation, etc.) and the time required to decide whether the request should be permitted or not. These insights can then be used to: 1) diagnose performance issues; 2) suggest how to resolve these issues; and 3) build systems that are more likely to achieve their performance objectives than those based on more informal/less objective performance models.

It should be noted that `PARPACS` attempts to build a statistical model, not a mathematical model (Mason et al., 2003, Chapter 1). This is because of the apparent difficulty of deriving a comprehensive mathematical model relating the controllable factors to the measured service times. The statistical model also includes a way to estimate the uncertainty in any of its predictions. However, the downside is that great care needs to be taken to validate the statistical model and to interpret its features. `PARPACS` is designed for maximum flexibility. For example, by editing a single statement in one module, it is possible to change the *model formula* used when fitting the linear model. It is then possible to rerun the analysis and compare with previous versions. This feature is essential to facilitate the use of `PARPACS` for optimization of PDP performance.

Building a reliable statistical model, particularly one with many terms, is a challenging exercise. The basic form of a linear statistical model is

$$\boldsymbol{y_i = f(a, x_i) + \varepsilon_i,} \tag{5.1}$$

where y_i represents the i^{th} observation of the dependent variable (service time in this case); x_i represents the terms in the model (a selection of factors, numeric predictors and their interactions) for the i^{th} observation; a represents a set of model coefficients

to be estimated when fitting the statistical model to the data, ε_i is the residual error between the statistical model and the i^{th} observation and f is a function (represented in R by its model formula) that is linear in a.

In ideal circumstances, f contains the minimum set of terms needed to ensure that ε is distributed as a Normal distribution, centred on zero and with a variance that is "small" relative to the variability in the data. These principles will be used throughout this chapter to ensure that a given model is reliable, before attempting a statistical interpretation. In addition to employing extensive model validation techniques (Butler et al., 1999), `PARPACS` uses effects plots (Fox, 2003) to aid understanding of factor and interaction terms in the model, and how they influence the service time.

`PARPACS` was designed to support very general performance models. As such, it is not limited to (models of) access control performance data. It is designed to support the fitting of any model that can be expressed as a model formula, such as those used by R. The `PARPACS` user captures the salient features of the statistical model in a set of 4 configuration files which each configure an aspect of how `PARPACS` operates. For example, one file defines the SQL statement used to extract the relevant performance data, another defines the model formula and related statistical model configuration, etc. Together, these 4 files define the *scenario* that is investigated using `PARPACS`. For the purpose of this dissertation, the scenarios relate to access control performance, but other scenarios (such as web service performance) can be investigated in the same way.

5.3 Investigating PDP and resource choices

5.3.1 Scenario Motivation and Overview

The following scenario has been chosen to indicate how `ATLAS` can help to diagnose performance problems in practice.

> Assume that an organization has deployed an access control system that is experiencing policy evaluation performance problems that manifest themselves as increased latency as seen by users. The organization is growing rapidly in size and complexity and senior management has commissioned a study into the scalability of the existing system and in particular, its ability to grow with increasing demand. A team of security

consultants/experts is tasked with identifying the reasons for the performance problems and of proposing some solutions.

The approach reflecting current best practice would be to treat the access control system as a black box and to run tests on equivalent deployments, adding system resources until adequate performance is achieved. If the system is viewed as a black box with a predictable policy evaluation time, it would then be possible to reason about the scalability of a given privilege management deployment. However, this approach has significant weaknesses. The main problem is that the performance of the underlying policy evaluation is difficult to characterize in terms of known quantities such as system resources. Indeed, *apparently* small changes to the policies can have dramatic effects on the service time per request. Therefore, apart from resulting in inefficiencies, it is often difficult to say whether this "point solution" has validity in the target deployment, particularly in the long term. It is precisely the lack of a robust per-request policy evaluation service time estimation procedure that motivates the use of the combined modelling/measurement procedure facilitated by `ATLAS`.

The security consultants decide to focus on one of the key interactions where access controls are needed. The customer organization (assume it is a Bank) frequently needs to buy in expertise from technical consultants, such as system integrators, experts in web design, etc. Often such third parties need privileged access to sensitive systems. While trust, confidentiality and similar agreements are in place between the Bank and suppliers of specialist services, it is also necessary for the Bank's own security staff to alter its access controls to ensure that external staff can provide the required service while not compromising the security of the Bank's data. Given this scenario the consultants use `ATLAS` to investigate the performance problems.

5.3.1.1 Influence of domain size on policies and requests

The consultants capture information about the problem domain and use this to configure the static model (typically describing attributes and relationships between persons (such as might be found in an LDAP directory), or describing resources (such as might be found in a document database with rich metadata)).

The consultants also capture information on the access rules specified by the business stakeholders. `DomainManager` is then used to create three different sets of policies (and hence requests), depending on the *size of the static domain*: small `S`, medium `M` and large `L`. A different form of size measurement is concerned with the complexity, measured as the number of conditions per request target component, which we term the request cardinality, which can be single `Sgl` or multiple `Mlt`.

5.3.1.2 Choice of PDP

The consultants note that the organisation uses the reference classic SunXACML PDP. They note that other PDPs exists, notably Enterprise XACML (which claims to perform better because it indexes the policies for faster lookups) and SunXACML 2.0 has been released with a revised codebase that is claimed to be easier to maintain (since it uses the Spring Framework for many tasks, replacing custom-built and hard to maintain code). Consequently, the consultants consider performing experiments in which three PDPs are compared: "classic" SunXACML `SX`, SunXACML 2.0 `SX2` and Enterprise XACML `EX`.

5.3.1.3 Availability of computing resources: memory and number of cores

The consultants also decide to test whether adding memory (RAM) and/or processor cores can offset any performance shortfall arising from the larger domain sizes, etc. Often when levels of performance decrease, the response of the system administrators is to provide more computing resources (more/faster processors, more memory, larger/faster disks, more bandwidth, etc.). The consultants wish to investigate the degree to which adding extra resources such as these would help to offset the expected reduction in performance when the domain size increases.

5.3.2 Review of Policy and Request Generation

Policy and request generation are the responsibility of the PolicyGen and RequestGen (§ 4.4 on page 143 and § 4.5 on page 159) components of `DomainManager`. In this section we briefly recall the main features of these procedures and highlight the main settings that are the subject of the consultants' performance investigation.

Listing 4.1 on page 130 is a listing specifying that, say for **DomainSize** = `medium`, 60 `Document` (`Marketing Plan`, `Corporate Strategy`) asset entities should be created, with generated names beginning `Part`, `Chapter`, `Section` and `Webpage`, etc. Listing 4.4 on page 147 is a listing of the template policy used in this scenario. As can be seen, it shares the hierarchical nature of XACML (groups contain rules and/or other groups) and much of its terminology (`Subject`, `Resource`, etc.), but the encoding is very different. Logical statements such as `SUBJECT.0` can be combined with other statements of the same *class* (such as `SUBJECT.2`) either by:

- conjunction (AND), e.g., `R3.Subjct.0` takes two values `SUBJECT.2` and `SUBJECT.0` which are ANDed together; or by
- disjunction (OR), e.g., `R3.Action.0` is ORed with `R3.Action.1`.

In this example, there is a single template policy set `G3` containing three template policies `G0`, `G1`, `G2`. Several variants of the policy are policy, differing in relation to choice of *combining algorithm* and placement of the fall-through Deny clause(s); the setting that controls these choices is **PolicyRef**. The template policy is modified to add semantic constraints according to the setting of **ExtraTcType**, such as `ExtraTcType` = `full`.

The requests are generated by collecting the logical statements used in the template policy rules (such as `SUBJECT.2` and `RESOURCE.3`) and assembling them into attribute-based requests, where the interpretation of the logical statement changes from a *filter* (when used in a policy) to a *question* (when used in a request) like "CAN *subject_terms action_terms resource_terms*?" In turn these attribute-based requests can be used to generate instance-based requests. At this point, there is an option: to list all the matching instance-based logical statements, or just one that represents that set of statements. For example, an attribute-based logical statement such as "`Member.level` = *Senior*" could map to any subset of "`Member.name` = *Alice01* OR `Member.name` = *Alice03* OR `Member.name` = *Alice06*". If we consistently choose just

one, say "`Member.name` = *Alice03*" whenever such an option arises, we say the resulting request has single cardinality (factor: **RequestCardinality** = `Sgl` (single)). Otherwise, if all such optional statements are used, it has multiple request cardinality (factor: `RqCrd` = *Mlt* (multiple)). Such "multi"-requests might arise when requests are batched together either for reasons of performance (perhaps it is faster to decide on one composite request rather than many smaller related requests) or because the security property requires such a composite check (such as would be the case with ethical wall policies (Brewer and Nash, 1989)).

5.3.3 Obtaining measured service times

Algorithm 5.1 Outline of the nested loop used in STACS for measurement runs. Requests are batched for evaluation at the PDP—we do not currently measure queuing delays for arriving requests.

```
for all {memory,nProc} ∈ {{2,4,6,8}, {4,8}} do
  for all Dsize ∈ {S,M,L} do
    for all Pdp ∈ {SX, SX2, EX} do
      readPolicies()
      for all RqCrd ∈ {S, M} do
        for all Rep ∈ 0...8 do
          shuffledRequests ← shuffle(1...n_req)
          for all Req ∈ shuffledRequests do
            request ← readRequest(Req)
            t ← measureServiceTime(Dsize, Pdp, RqCrd, request)
            saveInDbTable(memory, nProc, Dsize, policyRef, Pdp, RqCrd, Rep, Req, t)
```

The procedure followed by `STACS` is shown in Algorithm 5.1. Note that the changes in `memory` and `nProc` take place when the VM on which STACS is deployed is restarted with the revised `memory` and `nProc` settings. `STACS` provides an adapter for each of the three PDP implementations under test, so that each can perform the "readPolicies()", "readRequest()" and "measureServiceTime()" operations mentioned in Algorithm 5.1.

Lastly, it should be noted that the deeply nested loops (see Algorithm 5.1) and the scale of the domain settings (see Table 5.2, which compares the policies and requests generated by `DomainManager` using the template policy in Listing 4.4 with the simpler `continue` set presented in (Krishnamurthi, 2003) and used for performance experiments in (Butler et al., 2011; Griffin et al., 2012)) mean that *between 1 and 2 million policy evaluations are performed per VM setting combination.* Interestingly, it was found that the greater stability achieved by running the experiment on a VM

Table 5.2 Size and scale of the experimental runs.

Dsize	Metric	Generated	Continue
	#(VM setting combinations (memory, nProc))	8	8
	#(Number of Rules per policy set)	9	298
S M L	Avg. #(Conditions per policy set)	112 639 1163	678
	#(Request variants)	1280	400
	#(reps)	9	100
	Avg. #(Conditions per request (RC = Sgl))	4	3
S M L	Avg. #(Conditions per request (RC = Mlt))	15 80 145	6

server in a data center rather than a user machine meant that #reps could be reduced from 100 (for earlier continue-a runs) to 9 when using policies and requests generated by DomainManager. Note that these operations are augmented with other tasks (notably, managing the service time data) during a STACS run. Thus STACS run times, on typical hardware, are exhaustive and can take several hours, during which time the CPU usage of the server is high. This observation supports the assertion that such runs need to be performed offline in a dedicated testbed, rather than against the production access control deployment.

5.3.4 Deriving the performance model

Before estimating the performance model, it is necessary to examine the data, to identify any unexpected features and to prepare the ground so that the statistical model used for predicting performance is a faithful representation of the measured data and, by extension, of the service times that might be encountered in practice.

While it is possible to make qualitative judgments regarding the factors and their effects based purely on the data, a model is needed to predict performance for other settings and, more importantly, to estimate the uncertainty in all predictions (even for those where the settings were part of the data used to estimate the model). Fitting a

(a) All 9 replicates. (b) Middle 3 replicates.

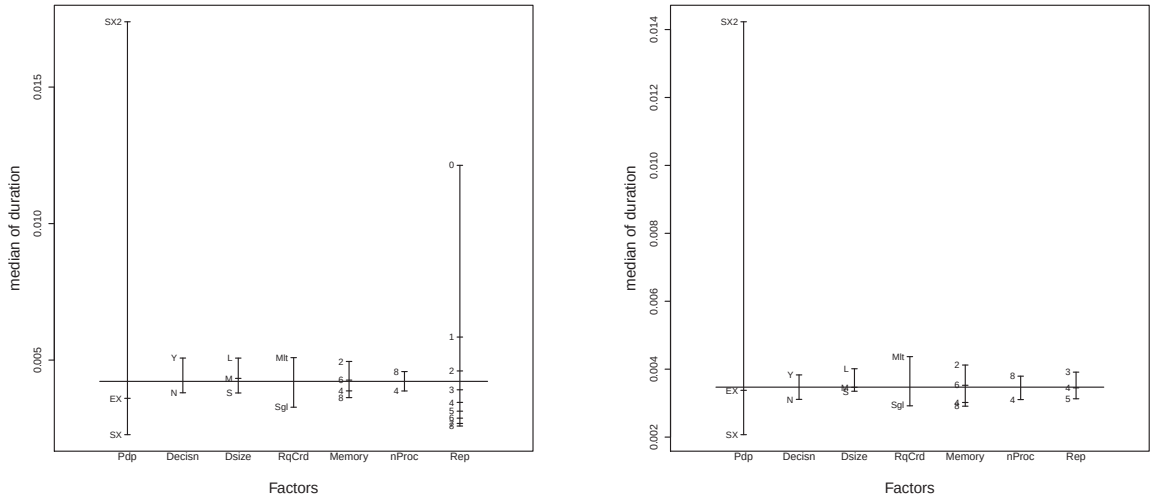


Fig. 5.2 Median service times for each level of each factor, as well as the overall median, when either all replicates are included (Figure 5.2a) or only the middle three replicates are included (Figure 5.2b).

model to such a large and complex data set requires care. A good fit is expected to have the following characteristics (Croarkin and Tobias, 2015, §1.2;4.4.4) and (Butler et al., 1999):

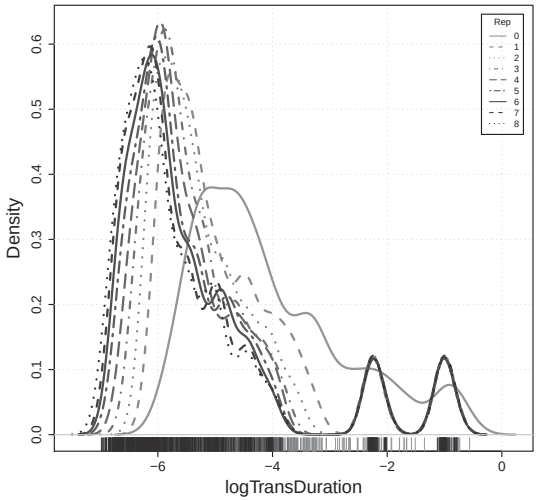
- the residuals $e_i = y_i - f(\mathbf{x}_i)$ have equal variance σ^2 across the domain of the data;
- the distribution of the residuals is unimodal and symmetric around zero, preferably Normal;
- the residuals have no structure—all structure in the data is captured in the model.

These desirable properties are used to check whether the model is adequate, in which case internal validity is assured.

5.3.4.1 Step 1: Restriction of `reps`

The simplest exploratory plot is a *design plot* (R Core Team, 2014, `plot.design()`) of the main factors and their levels, with the median of the service times per factor-level setting, together with the overall median, as shown in Figure 5.2. Note that, despite the fact that the requests arrive at the PDP in random order (due to the "shuffle()"

(a) All 9 replicates.



(b) Middle 3 replicates.

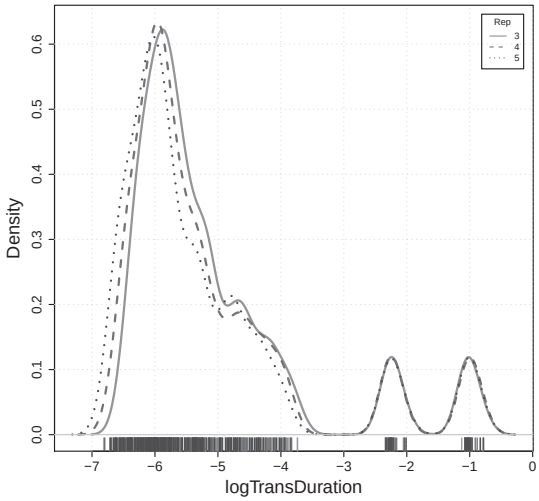
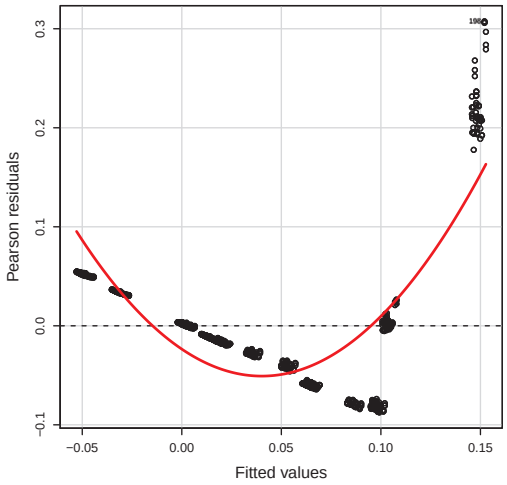


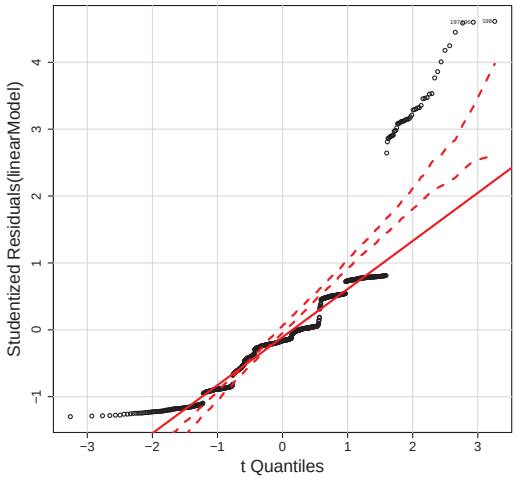
Fig. 5.3 Logarithm of the service time distributions, comparing different groups of replicates: all 9 replicates versus the middle three replicates only. When all replicates are included, the distributions show more variability in shape.

operation in Algorithm 5.1), there is a clear downward trend in service times as the replication index increases. It is known that the Java Virtual Machine (JVM) uses caching of commonly used objects, so we believe that, over time, the JVM optimizes its object cache, resulting in shorter average execution times—it "remembers" requests it has seen before. Note that this feature is provided by the JVM without any need for intervention by the developer. In an enterprise scenario we would expect a majority of frequent requests and a smaller number of less frequently occurring requests, so the (average) object cache hit rate would be quite high as seems to be the case with the middle 3 replicates here. In supporting evidence, we note that the service times also converge in a distributional (shape) sense as the `Rep` index increases, suggesting this is a systematic effect across all other factors, see Figure 5.3. The logarithm of service time durations is used in these density plots, otherwise the range of service time durations is so great that it is difficult to see the differences. Clearly, the distributions are quite different when the full range of replicates is used, but they have settled into a more consistent pattern when only the middle replicates are concerned. This, per request analysis, is consistent with the median analysis in Figure 5.2. Therefore, we analyze further only the middle 3 (of 9 replicates) as a compromise between one extreme, where no caching occurs because all requests are new arrivals, to the other

(a) Residuals: main factors only.

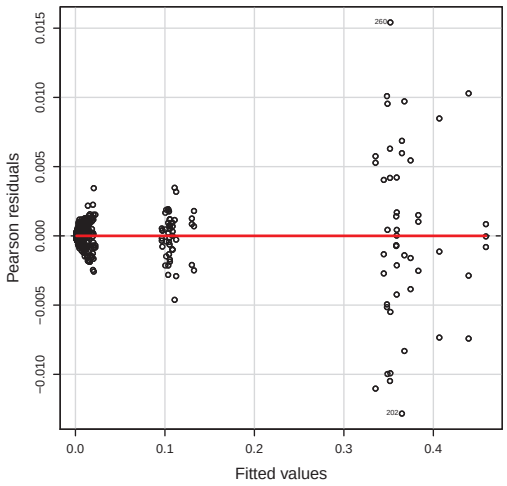
(b) Quantile-Quantile plot: main factors only.





(c) Residuals: all factors and their interactions.

(d) Quantile-Quantile plot: all factors and their interactions.



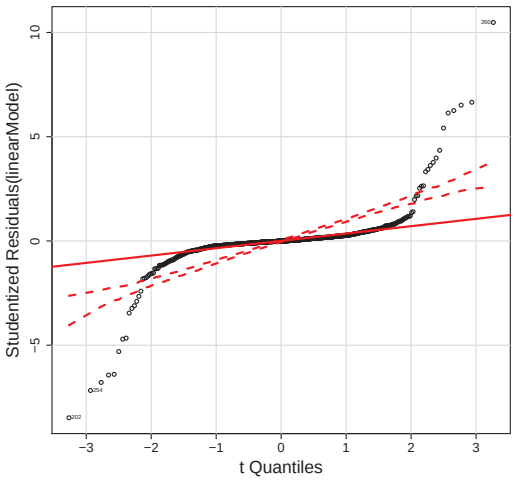


Fig. 5.4 Comparison of the Main-Only and Full Factorial models for the *untransformed* middle replicates (indexed 3,4,5): residuals and Quantile-Quantile plots. If the residuals had a Normal distribution, the quantile-quantile plot would be a straight line.

extreme, where (nearly) all evaluations become cache lookups because the requests are repeat arrivals.

5.3.4.2 Step 2: Adding interaction terms

The first consideration is whether the model contains terms (factors in this case) that are sufficient to explain the variability of the data. In the first (main factors only) model, only the factors in Figure 5.2b are included as terms in the model. In the second, extra terms are added: all 2-, 3-, 4-, 5- and 6-way interactions, so the model is now full factorial. As can be seen in Figure 5.4a, the (main-only) set of residuals still has lots of structure, most of which no longer exists in the second (full factorial) model, implying that the extended model accounts for more of the structure in the data with less of that structure left in the residuals. Another way of analysing the residuals is to consider whether they follow a Normal distribution. In that regard, a convenient method of visualising differences between a standardised empirical distribution (of the residuals of the fit to the measured service times) and a reference distribution (the Normal distribution centred on zero with unit variance) is to use a quantile-quantile plot (Lane, 2015, §8.1). The quantile-quantile plots confirm that there is systematic underfitting (see Figure 5.4b) unless the additional interaction terms are added. Referring to Figure 5.4d, there is very good agreement between the residuals and the desired Normal distribution, except in the tails, where it appears the residual distribution has fatter tails than the Normal distribution. This "lack of fit" is addressed later, but for now we note that the full factorial model is used in preference to the simpler "main factors-only" model.

5.3.4.3 Step 3: Transforming the data

Analysis of the residuals indicates that the model does not capture all the "behavior" in the data (see Figure 5.4c), and the residuals are not Normally distributed (see Figure 5.4d). The scope to add more terms, to make the model more flexible and reduce its bias, is limited, because the model is already full factorial. However, one of the standard techniques in regression analysis, particularly when presented with a lack of fit with no obvious terms to be added to the model, is to consider a transformation of either the predictor variables or the dependent variable. The latter is often easier to apply and a suitable Box-Cox transformation (Fox and Weisberg, 2011, §6.4.1) parameter λ can be estimated by examining the log-likelihood plot in Figure 5.5.

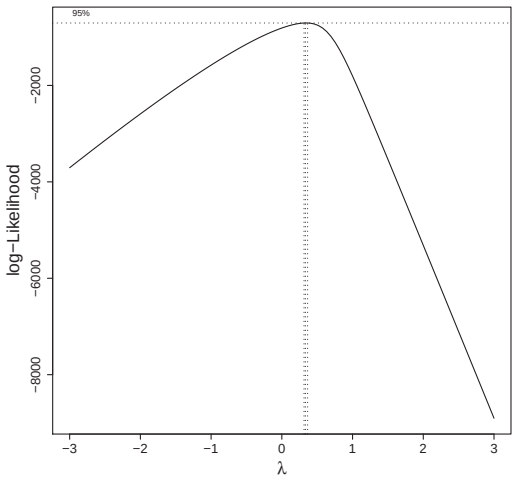


Fig. 5.5 Maximum-likelihood estimation of the Box-Cox parameter λ for the full factorial model; this plot motivates the choice of $\lambda = \frac{1}{3}$.

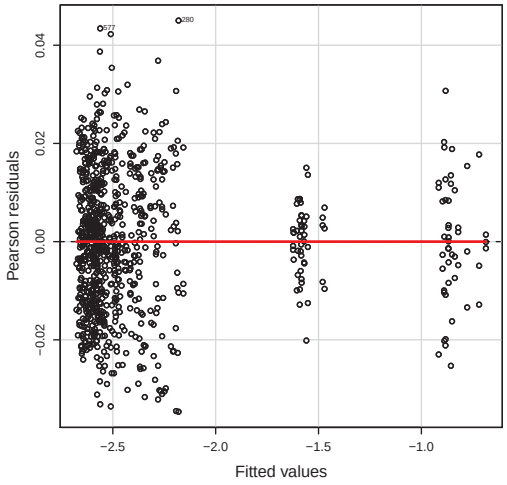
The Box-Cox transformation has the following definition

$$\begin{aligned} y^{\text{trans}} &= \frac{y^{\lambda} - 1}{\lambda}, \text{ when } \lambda \neq 0; \\ &= \log y \text{ otherwise.} \end{aligned} \tag{5.2}$$

The dependent variable, to which the Box-Cox transformation is to be be applied, is the service time duration in this case. As can be seen from Figure 5.5, the log likelihood is maximised when λ is approximately $\frac{1}{3}$. Therefore, we apply the transformation and consider what improvements, if any, are obtained when the transformed data is submitted for anaylsis. In this case, the improvement we seek is that the variance of the residuals is more constant over the range of the service time data. § 6.4.1 on page 216 defines other types of improvement, and how different procedures can be used to define power transformations to achieve those improvement objectives.

Comparing Figure 5.4c with Figure 5.6a we can see that the distribution of the residuals has less variation over the range of the measured service times, so the structure remaining in the residuals is now negligible, as desired. Comparing Figure 5.4d with Figure 5.6b we see that agreement between the residuals and reference Normal quantiles extends far into the tails, suggesting that the residuals are much closer to being Normally distributed, as desired. Good improvement can be seen when comparing Figure 5.7b against Figure 5.7a, where it can be seen that there is no

(a) Residuals.



(b) Quantile-Quantile plot.

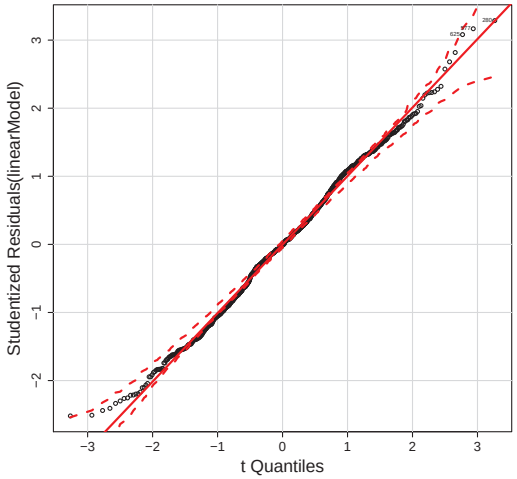


Fig. 5.6 Analysis of the residuals of all factors and their interactions of the *transformed* model when $\lambda = \frac{1}{3}$, for comparison with equivalent plots for untransformed data (Figure 5.4c and , respectively).

Table 5.3 Levene test for homogeneity of variance: Reps 3–5, full factorial model, $\lambda = \frac{1}{3}$.

	Df	F value	Pr(>F)
Group	287	0.49	1
	576		

significant trend and the variance is approximately constant over the range of the fitted data, as desired. The Levene test for homogeneity of variance (Croarkin and Tobias, 2015, §1.3.5.10) suggests, as enumerated in Table 5.3, that there is no reason to doubt that subsets of the residuals have equal variance, even if data were to be collected again and a new model (of the same type) fitted to the data.

Also note that, even with the Box-Cox transformation, the residuals suggest that the interaction terms are still needed, and the combination of the two refinements yields a better model than either refinement alone.

The combined Global Validation of Linear Model Assumptions (GVLMA) test (Peña and Slate, 2006) suggests that most assumptions of the linear model are met. The exception is the heteroscedasticity of the residuals: the variance of the error (as estimated by the residuals) is not constant over the range of the fit. However, we discount this concern because the plots suggest that much of the remaining

(a) Untransformed data (b) Transformed data

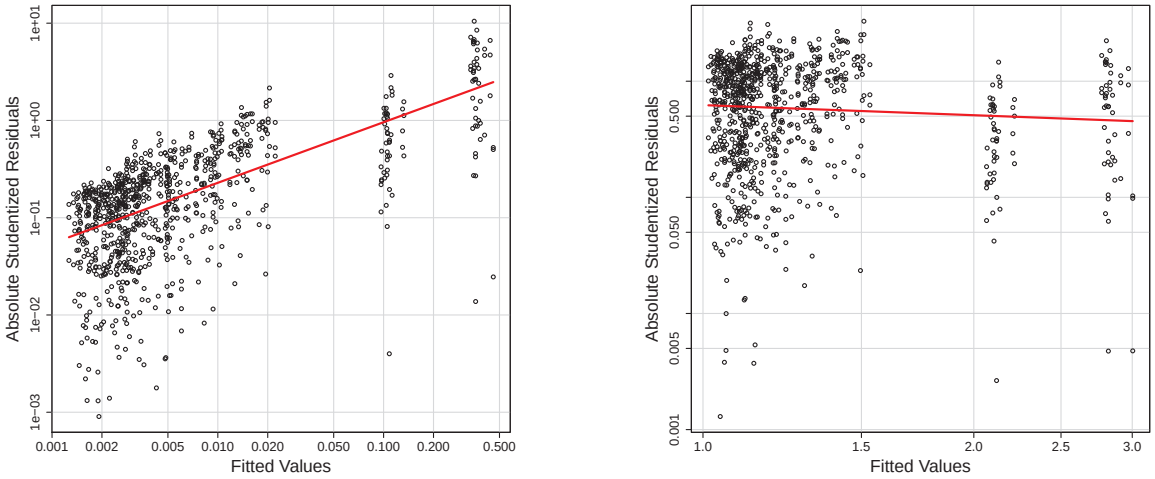


Fig. 5.7 Spread-level plot of full factorial model residuals, comparing those derived from the original untransformed data with those from the transformed ($\lambda = \frac{1}{3}$) data.

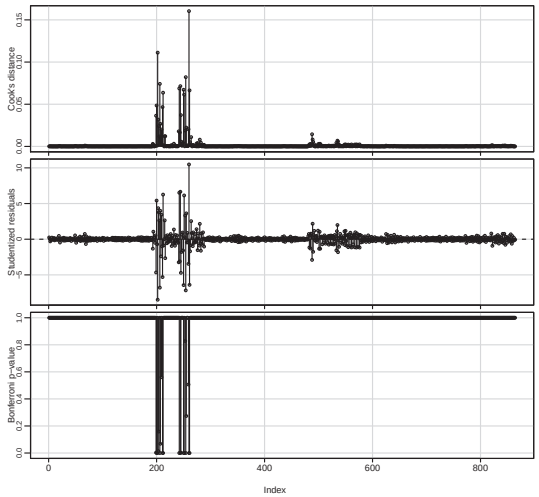
Table 5.4 Global Validation of Linear Model Assumptions: Reps 3–5, full factorial model, $\lambda = \frac{1}{3}$.

	Value	p-value	Decision
Global Stat	10.2	0.037	Assumptions NOT satisfied!
Skewness	0.559	0.455	Assumptions acceptable.
Kurtosis	2.40	0.122	Assumptions acceptable.
Link Function	0.00	1.00	Assumptions acceptable.
Heteroscedasticity	7.26E+00	0.007	Assumptions NOT satisfied!

non-constant error variance is caused by the fact that there are unavoidable "gaps" in the data (see Figure 5.6a), where the variance is unknown and hence the test cannot assume that it is constant across those gaps.

The dramatic reduction in the number of outliers indicated in Figure 5.8b is very encouraging, as is the fact that the Cook's distance plot shows remarkable consistency across all observations. This plot can be compared with the untransformed data case (see Figure 5.8a) where clear evidence of the presence of outliers, lack of fit and high-leverage points is presented.

(a) Untransformed data ($\lambda = 1$).



(b) Transformed data ($\lambda = \frac{1}{3}$).

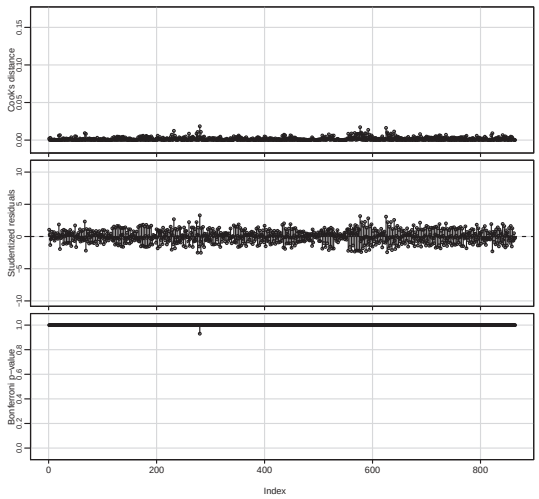


Fig. 5.8 Outlier, leverage and influence analysis for the full factorial model applied to untransformed data ($\lambda = 1$) and transformed data ($\lambda = \frac{1}{3}$). With the transformed model, each observation has about the same *leverage* (measured by Cook's distance (Fox and Weisberg, 2011, §6.3.3)) as its peers, hence the fit is very stable, as desired. The same feature applies to the studentised residuals (where they have been scaled by the estimated standard error, which itself is much smaller for the transformed model). Lastly, the Bonferroni-corrected probability of each observation being generated by chance (and not being a contaminant from a different distribution, and hence an outlier) is very close to 1 in all cases except one, where the probability is approximately 0.93, which exceeds the critical value 0.05. Hence, we accept the null hypothesis that there is no strong evidence of an outlier in the transformed data, relative to the full factorial model.

.

There is overwhelming evidence that the model has internal validity, and Figure 5.8 presents *partial* evidence of its external validity.

5.4 Scenario predictions

The nature of the model is that the terms are treated either as *factors* (`PDP` and `Decision`) or as *ordered factors* (`Dsize`, `RequestComplexity`, `memory` and `nProc`). Each of these (ordered) factors has a finite number (2, 3 or 4) of levels. In the case of unordered factors, the model estimates should be viewed as *point* estimates: they are valid only for that level of the factor. As an example, the model can predict the performance of any one of the existing PDPs for which measurements were taken, but the performance of a new PDP cannot be predicted based on the available data from the existing PDPs. In the case of ordered factors, it is possible to *interpolate* existing model estimates in order to estimate the performance at a new intermediate setting. As an example, if the domain size increases to lie somewhere in the range [`S`, `M`, `L`], it would be possible to estimate the performance at this intermediate size setting, *without the need to collect additional data.*

For unordered factors such as `PDP`, new settings require new measurements and estimation of a new statistical model. For ordered factors such as `Dsize`, `RequestComplexity`, `memory` and `nProc`, the *existing* statistical model can be used to estimate the performance at intermediate settings of those factors.

After estimating a model, and validating that model, the model may be used to predict the access control performance under specified conditions. In this regard, the performance is measured as the mean service time to process an access request. The security consultants may wish to find the answers to questions of the following form:

- Does the system meet its performance objective, such as that the expected service time $E(T) = \bar{t} \leq t_{\text{tol}}$?
- What is/are the limiting settings at which the expected service time just achieves its performance objectives: $E(T) = \bar{t} = t_{\text{tol}}$?

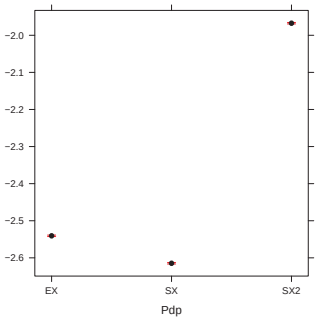
The critical feature is that the model estimates the *effects* of each factor level combination (see Fox (2003)). In the case of a single ordered factor, it is possible to use an interpolation scheme such as barycentric interpolating polynomials (Berrut and Trefethen, 2004) to estimate its effect at intermediate values. More generally, if several parameters (such as `Dsize` and `ReqCrd` (equivalent to *request complexity*)) are varied, the corresponding effects table should be consulted. If the service time duration is to be predicted at one of the grid points, its value (and uncertainty) can be selected from the table. If not, a multivariate grid interpolation procedure, such as that described in (Gasca and Sauer, 2000) can be used to estimate the service time duration at intermediate points. Because of the high degree of precision (equivalently: small uncertainty) in the effect estimates at the grid points, the uncertainty is dominated by any systematic errors arising from the choice of polynomial fit between the grid points.

The data measured in the experiment is sufficiently rich that a full factorial model can be estimated, and hence the effects can be estimated for each combination of each factor at the factor levels used in the experiment. Figure 5.9 highlights some of the more interesting features. Firstly, the `PDP`, `Dsize` and `RqCrd` factor main effects are significant. Note that the error bars at the estimated points are tiny compared to the variation in the response (i.e., the service times) with each of these factors. Since `DSize` is an ordered factor with 3 levels, we can see that, if other factors are held constant, the service time increases roughly linearly with the domain size (see Figure 5.9b). If we look at the two-factor interactions, such as `Dsize` with `PDP` in Figure 5.9d, we see that the effects are additive: the `SX2 PDP` performs particularly poorly as the domain size increases, consistent with the observation earlier that its performance is not scalable in respect of domain size. If we consider both `Dsize` and `RqCrd`, which each measure an aspect of the size and complexity of the domain, Figure 5.9e indicates that this is also additive: larger domains (for policy purposes) and more complex requests result in longer policy evaluation times, and we can even quantify the difference.

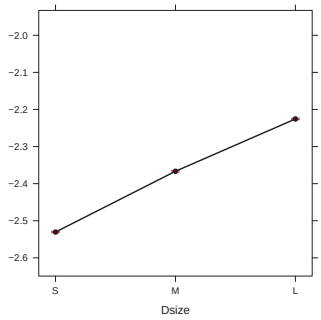
Not all interactions show such properties—sometimes changing one factor (such as `Decision` or `memory`) has little difference on another (such as `RqCrd`): see Figure 5.9g where the two traces are very similar, apart from a vertical translation due to differing `RqCrd`. We also notice that the variability in performance with respect to `memory` is much less than that for `RqCrd`. Also, contrary to expectation, *service times do not decrease monotonically as memory is added*, but the slight increase in service times when `memory` is 6Gb is negligible in this case.

Lastly, if we select the three-factor interaction of `Dsize`, `PDP` and `RqCrd` we can see that their additive nature persists even for such higher-order interactions (see Figure 5.9h). That is, the worst-possible performance occurs with the "slowest" PDP with the largest domain and the most complicated requests, and the contributions of each of these factors is mostly additive.

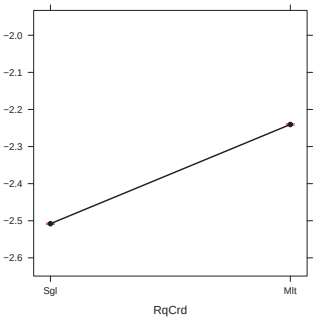
(a) Effects plot: `PDP`.



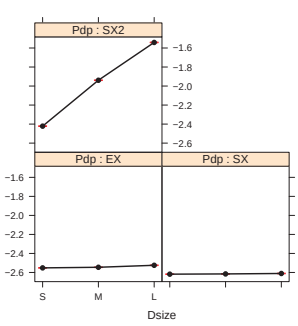
(b) Effects plot: `Dsize`.



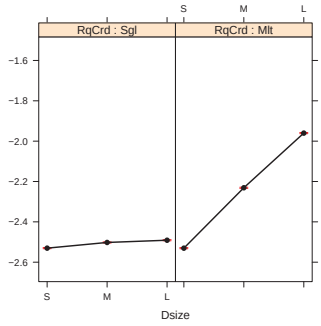
(c) Effects plot: `RqCrd`.



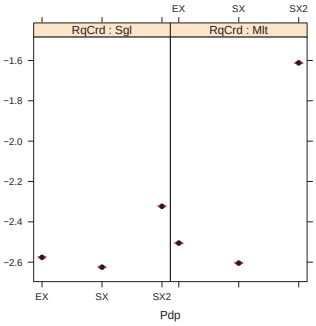
(d) Effects plot: `Dsize:PDP`.



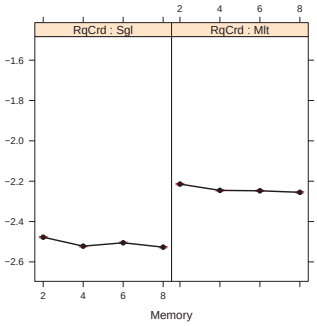
(e) Effects plot: `Dsize:RqCrd`.



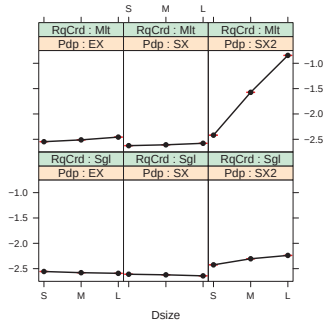
(f) Effects plot: `PDP:RqCrd`



(g) Effects plot: `RqCrd:memory`.



(h) Effects plot: `Dsize:PDP:RqCrd`.



(i) Effects plot: `Dsize:PDP:Decision:RqCrd:memory`.

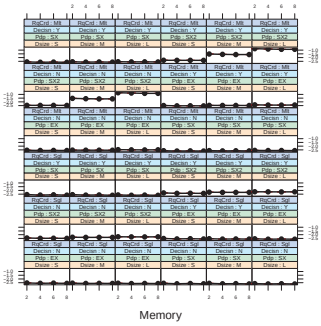


Fig. 5.9 Selected effects estimated from transformed data with $\lambda = \frac{1}{3}$. The y-axis is in units of the Box-Cox transformed service times. Each of the panel plots shows the effect of the highlighted factor (or factor interaction) term in the statistical model, taking account of the contribution both of that model term *and* the interaction terms that include that term (Fox, 2003). Indeed, the *effect* of a model term is the marginal contribution to the service time for that level of that term, including its interactions, averaged over all non-related terms in the model. Therefore, when predicting performance from a factorial model with many significant interaction terms, it is easier to interpret the *effects computed from the model coefficients* than the coefficients themselves. Each panel plot enables even high-dimensional effect surfaces to be visualised by means of a series of linked 2-D effect plots. For example, Figure 5.9f indicates how the (transformed) service times vary with `Dsize`, for each of the six data subsets (two `RqCrd` levels and three `Pdp`). Clearly, the SunXacml 2.0 PDP performs worse than the other two PDPs, and this effect is accentuated when the incoming requests have multiple (rather than single) cardinality.

5.5 PARPACS Summary

The purpose of ATLAS is to help its users to understand what factors affect access control system performance; compare several scenarios in a testbed; and provide a robust basis for predicting the performance in a production deployment. The last of these is perhaps the most difficult because it requires both internal and external validity. Internal validity is relatively easy to achieve in a designed experiment (Mason et al., 2003) such as this because, *inter alia*, it is possible to control many factors directly, and confounding can be minimized in a full factorial experiment. External validity is more difficult to prove, because it relates (in this case) to whether the experiment findings will generalize to actual deployments. Thus, in the description of the results that follows in this section, we look for any hints that external validity is compromised, in which case the (notional) security consultants would not have a sound basis for the advice they provide their clients.

> The use of designed experiments and statistical safeguards in ATLAS promote the internal validity of its predictive models. Gross violations of external validity can be identified (e.g., suspicious patterns of outliers) but it is not possible to make any guarantees. This is because external validity depends on the mapping from actual deployment scenarios to the scenarios supported by ATLAS.

PARPACS plays an essential role in ATLAS, as it helps to make sense of complex (20+ factors), large scale ($O(10^6)$ measurements) experiments. It is extensible because specifications like the model formulas can be defined outside PARPACS, making comparison of models as easy as comparison of factors. Furthermore, a suite of different statistical procedures and visualisation plots is available; the researcher can configure PARPACS to use those that are most appropriate to the scenario under study.

As mentioned earlier, PARPACS can, in principle, be used for many different types of performance experiment. Aspects relating to a specific scenario have been abstracted from the rest of PARPACS, so users need to edit only 4 configuration files[1] to customize PARPACS for a given scenario. That said, there is a big difference between editing configuration files and knowing what configuration settings to choose. The former task

[1] A fifth file is needed if a Box-Cox transformation is required.

is relatively easy compared to the latter. PARPACS offers many different statistical analyses. The majority of these analysis procedures are designed to validate the model, by checking that it is a faithful representation of the data. These procedures are complex and sometimes seem to contradict each other, though they can usually be found to be consistent based on emerging insights from other analyses and visualisations. Sometimes a procedure or plot will indicate how a particular model deficiency can be addressed, but often other procedures are needed to interpret the problem and indicate a possible solution. Most of these procedures are highly sensitive to the context in which they are applied. As with Big Data initiatives more generally, considerable statistical/data science expertise is needed to make the best use of the analytical procedures, visualisations and predictions available in PARPACS. However, given suitable knowledge and training, it remains a very powerful tool in the hands of an expert user.

Given the availability of STACS, DomainManager and PARPACS, the stage is set for an investigation of the influence of policy authoring choices on access control evaluation performance. This is the main topic of Chapter 6.

Chapter 6

Influence of policy settings on access control performance

Table 6.1 Research questions addressed in Chapter 6

ID	Question
RQ1	How can access control evaluation performance be measured for use in performance experiments? – What form does the service time distribution take? – What simulations can be performed to explore the effect of different request arrival patterns? – What analysis can be performed when the systems under test use different languages, frameworks and encodings?
RQ2	How can domain models be specified and used to express enterprise access control scenarios? – How can different variants of domain models be specified in a flexible and easy to use way? – How can access control evaluation performance be compared at different domain sizes?
RQ3	How can the data from performance experiments be used to understand and predict access control evaluation performance? – What types of exploratory data analysis are suitable for the performance experiments? – What are the steps needed to build statistical models predicting access control performance?
RQ4	What are the main factors affecting access control evaluation performance? – What are the effects of PDP choice, domain size and resources? – What are the effects of domain size, policy and request characteristics?

6.1 Introduction to the extended evaluation

ATLAS was used in Chapter 5 to investigate the effects of major changes to the access control deployment. Such changes relate to:

- the use of different PDPs;
- altering the memory available to the server instance;
- gross changes to the policies (notably, the size of the static domain being protected);
- different request complexity settings (cardinality changes from 1 to unbounded);

Interesting findings presented in § 5.4 on page 201 include the following:

1. Adding more computing resources (such as memory) do not always benefit performance;
2. Efforts to improve the manageability of a codebase can harm performance (SunXACML v2.0 is not as performant as "classic" SunXACML v1.4);
3. Larger static domains result in larger instance policies and less performance.

However, one of the features that appears self-evident, and which is assumed by many authors, is that policy authoring choices affect performance. The evaluation in Chapter 5 does not address this issue; it is addressed in this chapter instead.

It should be noted that the existing service time measurements collected by STACS and used in Chapter 5 are adequate for the analysis of the present chapter, because they already include the additional factors relating to policy authoring and related concerns. Indeed, the policies and requests used in Chapter 5 had several different "flavours" that were not relevant to the analysis of that chapter. Consequently, for the analysis in Chapter 5, PARPACS computed the partial averages of the measured service times with respect to those extraneous policy authoring factors, see § 5.2. Note that the findings of that chapter relate to those partial averages of service time measurements.

In the present Chapter, those policy authoring factors take centre stage and so partial averaging with respect to these policy authoring factors is not needed. Instead, the "raw" service time measurements are used by PARPACS and the findings that are identified and discussed later in the chapter relate to these "raw" service time measurements.

6.2 Access control decision analysis

One of the advantages of the domain model is that it can support iterative policy authoring. It is often desirable for a policy set to be *robust* in the sense that perturbing its input (the requests) has minimal impact on the output (the responses). This is particularly valuable in enterprise access control, where the requests will evolve as the business environment changes. Thus as new requests arise, the policies should make decisions that are "reasonable" in the sense that they enforce the overall security objectives of the enterprise. If a new request is sufficiently different from the conditions expressed in the policies, it may be unclear what the decision should be in this case. The PDP should then indicate that an exceptional request has been received, in which case the policy author should take action.

One way of measuring this desirable quality of a policy set is to define *stability criteria* and to check how well they hold for a given policy set. In particular, the following criteria are considered: that each request with a given request `id` (and hence derived from the same attribute-level request) has the same response regardless of

- `DS` (domain size is `small`, `medium`, or `large`)
- `PR` (how the policy rules are laid out): all 4 combinations of `First Applicable` and `Permit Overrides` combining rules with `Distributed Deny` and `One Deny` guard rules. The levels used are `FD`, `FO`, `PD`, `PO`, where the first letter relates to the combining rule (`First Applicable` and `Permit Overrides`) and the second character relates to the guard clause placement (`Distributed Deny` and `One Deny`).
- `PS` (Policy Specification level: `extraTcType` is one of `minimal` or `full`)
- `pdp`: the response should depend on standard features only of the `PDP`

While the settings above should not affect the semantics, other settings relating to request generation do. They include:

- request generation adjustments: `AS` = (`TscR`, `TcR`) (each being either `f` or `t`) etc.;
- request cardinality constraints: the request complexity `RC` can be `Sgl`, `Dbl` or `Mlt` (the latter being unconstrained).

The request generation adjustments affect the number and nature of the attribute-level requests, so there is no reason why response $r_i(a_1) = r_i(a_2), \forall i$ where $\texttt{AS} = a_1$ and $\texttt{AS} = a_2$ are different request generation adjustment settings. Indeed, in the scenario considered in §6.3, there are 28 distinct attribute-level requests when $\boldsymbol{a_1} = (\texttt{TscR}, \texttt{TcR}) = (\texttt{f}, \texttt{f})$ (in which case $i = 1, \ldots, 28$) and 440 distinct attribute-level requests when $\boldsymbol{a_2} = (\texttt{TscR}, \texttt{TcR}) = (\texttt{t}, \texttt{t})$ (so $i = 1, \ldots, 440$). Clearly there is no way that responses $r(\ldots, a_p)_i = r(\ldots, a_q)_i$ unless $p = q$ in which case the request indices are drawn from the same range $1, \ldots, r$, say.

If instance-level requests $r_i(a_p)$ and $r_j(a_p)$ belong to the same duplicate group of instance-level requests (see § 4.5.6 on page 173), they will have the same response (by definition of requests that are duplicates of each other).

These properties can be checked easily because `STACS` returns both the access decision and the service time in the `ServiceTimeResponse` database view. `PARPACS` derives the `DecisionAnalysis` table from the `ServiceTimeResponse` database view and exports the results of queries on `byStaticRef.csv` to check the following assertions:

- The same Response is given, for each request (indexed by `reqId`) issued against the same PDP and policies, where the requests differ only in relation to `DS`. Thus $r(\ldots, \texttt{AS}_k, \texttt{RC}_j, \texttt{DS} = \texttt{S})_i = r(\ldots, \texttt{AS}_k, \texttt{RC}_j, \texttt{DS} = \texttt{M})_i = r(\ldots, \texttt{AS}_k, \texttt{RC}_j, \texttt{DS} = \texttt{M})_i, \forall i, j, k$. This is checked by querying the responses and grouping them by `staticRef` and $\texttt{RC}_j$i and $\texttt{AS}_k$;
- The same Response is given, for each request (indexed by `reqId`), issued against the same PDP and policies, where the requests differ only in relation to `PR`. Thus $r(\ldots, \texttt{AS}_k, \texttt{RC}_j, \texttt{PR} = \texttt{FD})_i = r(\ldots, \texttt{AS}_k, \texttt{RC}_j, \texttt{PR} = \texttt{FO})_i = r(\ldots, \texttt{AS}_k, \texttt{RC}_j, \texttt{PR} = \texttt{PO})_i = r(\ldots, \texttt{AS}_k, \texttt{RC}_j, \texttt{PR} = \texttt{PO})_i, \forall i, j, k$. This is checked by querying the responses and grouping them by `policyRef` and $\texttt{RC}_j$ and $\texttt{AS}_k$;
- The same Response is given, for each request (indexed by `reqId`), issued against the same PDP and policies, where the requests differ only in relation to `PS`. Thus $r(\ldots, \texttt{AS}_k, \texttt{RC}_j, \texttt{PS} = \texttt{minimal})_i = r(\ldots, \texttt{AS}_k, \texttt{RC}_j, \texttt{PS} = \texttt{full})_i$. This is checked by querying the responses and grouping them by `policySpecification` level (labeled as `extraTcType`) and $\texttt{RC}_j$ ans $\texttt{AS}_k$.

If these access decision checks, summarised in Table 6.2, apply, it shows that the semantics of different "flavours" of generated policies and requests are predictable and match what was expected at specification time. Thus the policy-request combinations

Table 6.2 Consistent Decisions

Parameter	Desired outcome
DS	Same decision for corresponding ({small, medium, large}) *instance-based* requests
PR	Same decision for corresponding ({FA-DD, FA-OD, PO-DD, PO-OD}) requests
PS	Same decision for corresponding ({minimal, full}) requests

have predictable responses and hence the semantics are *stable*. It is then possible to focus on service time analysis, safe in the knowledge that the semantics are well behaved. These checks are performed for each set of measurements that are used in a performance analysis.

6.3 Outline of the extended scenarios

Table 6.3 outlines the main properties and their values as used in the evaluation tests. The DS label describes the "size" of the static model, where size relates to the number of instances per attribute combination. The PR label identifies variants of a policy definition. In this case, it takes the form XY, where X identifies the policy and rule combining algorithm used (FirstApplicable or PermitOverrides) and Y (DistributedDeny or OneDeny) identifies the strategy for placement of the guard Deny clause: distributed through the policy set or just at the end. The PS label indicates the degree to which extra TCs are added to the template policies to ensure that the static semantic constraints are enforced. For PS = minimal, the only TCs added are those that are *necessary*. If PS = full, some additional TCs are added that over-specify the semantic constraints while not changing the decisions made by the policies.

The RC label describes settings for request generation. The RC labels indicate *how many* matching instances are included in each request target component: one, two or all. This relates to the fact that the number of instances in the graph that match an attribute query can vary dramatically—from none to a large number. Depending on the scenario, a series of individual requests could be issued, or a single composite request issued instead. Two Boolean indicators TscR and TcR capture the possible component reduction adjustments made when generating requests, see § 4.5 and

Table 6.3 Scenario parameters

Property	id	Values	Comments
Domain Size	`DS`	`S`, `M`, `L`	size of static model
Policy Ref	`PR`	`FD`, `FO`, `PD`, `PO`	[*Rule-combining algorithm*]-[*how the default clause is handled*]
Policy Spec level	`PS`	`minimal`, `full`	different levels of extra `TC`s used to enforce static semantic constraints
contextSubRef	`RC`	`Sgl`, `Dbl`, `Mlt`	Single-, double- and (unlimited) multi-cardinality request complexity
Tsc Reduction	`TscR`	`f`,`t`	has `TSC` reduction been used when generating requests?
Tc Reduction	`TcR`	`f`,`t`	has `TC` reduction been used when generating requests?
pdp	`Pdp`	`SX`	`SunXACML PDP`
memory	`Mm`	2GB, 4GB	total memory allocated to the server
nProc	`nP`	4, 8	number of processor cores
policy LC	`pLC`	$\mathbb{N}$	policy leaf count, ranges over the natural numbers
policy TPL	`pPL`	$\mathbb{N}$	policy total path length, ranges over the natural numbers
request LC	`pLC`	$\mathbb{N}$	request leaf count, ranges over the natural numbers
request TPL	`pPL`	$\mathbb{N}$	request total path length, ranges over the natural numbers

Appendix B. As PDP, the "classic" SunXACML v1.4 with traditional DOM-based parsing (SX) is used; service time data for the other PDPs is excluded (filtered) from the analysis. The memory and nProc parameters label the amount of computing resource available to PDP during policy evaluation.

Summarising, this set of parameters is rich enough to explore the following Scenario Questions:

SQ1 If particular changes are made to the organisation of a policy set (c.f., the PR setting) which *could* lead to evaluation shortcuts, what are the effects on access control evaluation performance?

SQ2 Regarding PS, generally PS=minimal instance-level policies have fewer clauses than PS=full instance-level policies, other factors being equal. What are the effects on access control evaluation performance?

SQ3 What are the effects of instance-based request complexity (c.f., RC) on decision outcomes and evaluation performance?

SQ4 How does static model size influence instance-level policy performance?

In our survey of the literature (§ 2.1.7 on page 35, 2.2.8 on page 50, 2.2.9 on page 51 and 2.2.10 on page 52), we have not discovered any other publications where such questions *specific to policy formulation* could be addressed in a comprehensive manner. The remainder of this Chapter presents the answers we found, using ATLAS, to these questions.

6.4 Access control performance analysis

Four STACS runs were performed, for each of the four combinations of memory and number of processing cores in Table 6.3. In each STACS run, there was a nested loop with the innermost loop being over nine replicate runs of each request to be submitted to the PDP, with the policies chosen for that set of requests. However, the order in which the requests are submitted was randomised a) to reduce effects due to sequencing (repeating the same request nine times introduces unwelcome serial

Table 6.4 Specification of the two model versions and the two model scopes, showing the right hand side of each model formula. Predictors such as `poly(pLC,2)` use R syntax for a quadratic polynomial in `pLC` (policy leaf count in this case). `I2`, `I4` and `A4` represent the `modelVer`=2 insignificant and the `modelVer`=4 insignificant and aliased terms, respectively.

ver	scope	model formula (right hand side only)
2	`mainOnly`	`DS+Dn+PR+PS+TscR+TcR+RC+Mm+nP`
	`sig`	`(DS+Dn+PR+PS+TscR+TcR+RC+Mm+nP)^3 - I2)`
4	`mainOnly`	`Dn+PR+PS+TscR+TcR+Mm+nP+poly(pLC,2)+poly(pPL,2)+poly(rLC,2)+poly(rPL,2)`
	`sig`	`(Dn+PR+PS+TscR+TcR+Mm+nP+poly(pLC,2)+poly(pPL,2)+poly(rLC,2)+poly(rPL,2))^2 - A4 - I4`

correlation in the data) and b) to reflect typical usage, where requests do not arrive at the PDP in any specific order.

Two families of model formulae were used, labeled as `modelVer`=2 and `modelVer`=4. In the first, all terms are categorical (discrete-valued) factors; in the second, some numerical predictors replace the equivalent categorical factors. In the case of `modelVer`=4, the numerical predictors `poly(pLC,2)+poly(pPL,2)+poly(rLC,2)+poly(rPL,2)` relate to the size of the instance-based policy sets and requests and replace the discrete-level `DS+RC` factors which serve a similar purpose in the case of `modelVer`=2.

Both models are linear functions of their predictors but the decomposition of the variance differs. In the first model, *ANOVA* techniques are suitable because all predictors have discrete levels. In the second, the mix of predictor types requires the use of *ANalysis of COVAriance (ANCOVA)* techniques, where the variance is first partitioned using the covariates (numerical predictors) and then by the categorical factors. Consequently, interactions between factors and numerical predictors are more difficult to analyse, and interactions between numerical predictors need to be treated by mapping the predictors onto pseudo-factors clustered over ranges of their independent variables.

For each `modelVer` there are two variants: one with main factors only (labeled `modelScope`=`mainOnly`), the other including all main factors with added significant interactions (labeled `modelScope`=`sig`); see Table 6.4.

The `sig` variants were obtained by 1) considering all interactions up to and including 3-way for `modelVer`=2 and up to 2-way interactions for `modelVer`=4, then 2)

(a) All 9 replicates.

(b) Middle 3 replicates.

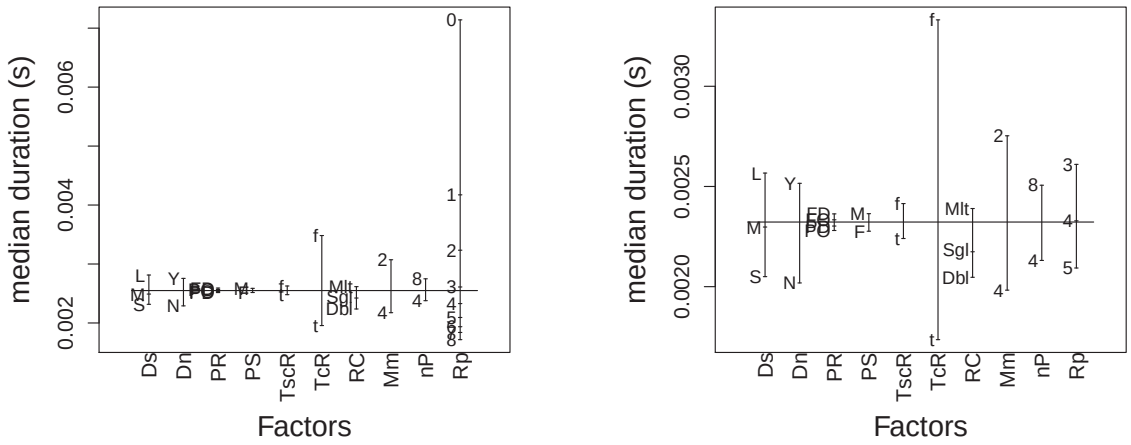


Fig. 6.1 Median service times for each level of each factor when `modelVer`=2, as well as the overall median, when either all replicates are included (Figure 6.1a) or only the middle three replicates are included (Figure 6.1b). Note the change of scale between the two plots.

iteratively removing those that contributed the least to the fit, as measured by the Akaike Information Criterion (AIC) (Akaike, 1974; Butler et al., 1999) of a particular statistical model fitted to the service time measurements. This procedure (dropping a term, recalculating the AIC of the new fit, then either stopping or selecting the next term to drop) is repeated until further improvement in the AIC is impossible. At that point the model formula is assumed to have the best balance (among models in that class) between closely following the given measurements in that data set and generalising to other instances of the data, were the experiments to be repeated.

In the case of `modelVer`=4, there is an added complication in that some of the interactions are aliased and hence cannot be estimated from the data. The aliased interaction terms are `PS:poly(pLC,2) + PS:poly(pPL,2) + poly(pLC,2):poly(pPL,2)`; these terms (labelled `A4` in Table 6.4) need to be removed *before* the insignificant terms are removed. By contrast, the observation matrix (Mason et al., 2003) for `modelVer`=2 has full rank so all terms can be estimated.

6.4.1 Refining each model

Two further model refinements are implemented. Firstly, it became clear from initial exploration of the results of the experiment that, even with the randomization of the order in which requests arrived, there is a pronounced downward trend in service times as the replication id increases, even as all other settings are kept fixed, see Figure 6.1a. It is known that the Java Virtual Machine (JVM) uses intelligent garbage collection internally to improve performance. The purpose of garbage collection is to find a good compromise between keeping objects in memory (so that they can be used again in an efficient manner) and releasing the memory for new objects (so that the application does not run out of memory). Since this is a dedicated run of a JVM-based PDP in STACS, there is no other application software competing with the PDP in the JVM. Thus it is likely that the highly controlled testbed environment makes it easier for the JVM garbage collection algorithms to predict what objects should be held in memory and what should be dropped. However, in a typical enterprise deployment, it is likely that many requests would be similar to each other in any case and so some caching of access decisions would occur in practice. As a compromise, we choose only the middle three (of nine) replicates for analysis, and discard the other 6 replicates, see Figure 6.1b. By filtering data in this way, the three replicates in use are more representative of those in the desired scenario. With this interpretation, it is also acceptable not to treat the replication id as a factor in the analysis and to treat the three measurements per setting combination as true statistical replicates.

The second refinement is a response to the apparent underfitting common to the four models outlined in Table 6.4. Since the scope to add new terms is limited (recall that the model formulae have been reduced from larger, less efficient expressions), heteroscedasticity is visible in Figure 6.3 and the most serious problems appear to be in the upper tails (see Figure 6.2), transformation of the dependent variable (service time duration in this case) is indicated. A Box-Cox transformation (Fox and Weisberg, 2011, §6.4.1) applied to the dependent variable y with parameter λ is a power transformation typically applied to the dependent variable; see Equation 5.2 on page 197 in § 5.3.4.3.

The purpose of the power transformation can be 1) to stabilise the variance (reduce the heteroscedasticity), 2) to make the response mean a linear function of the predictors, or 3) to make the distribution of the residuals more symmetric. In

(a) Quantile-Quantile residual plot, sig modelVer=2.

(b) Quantile-Quantile residual plot, sig modelVer=4.

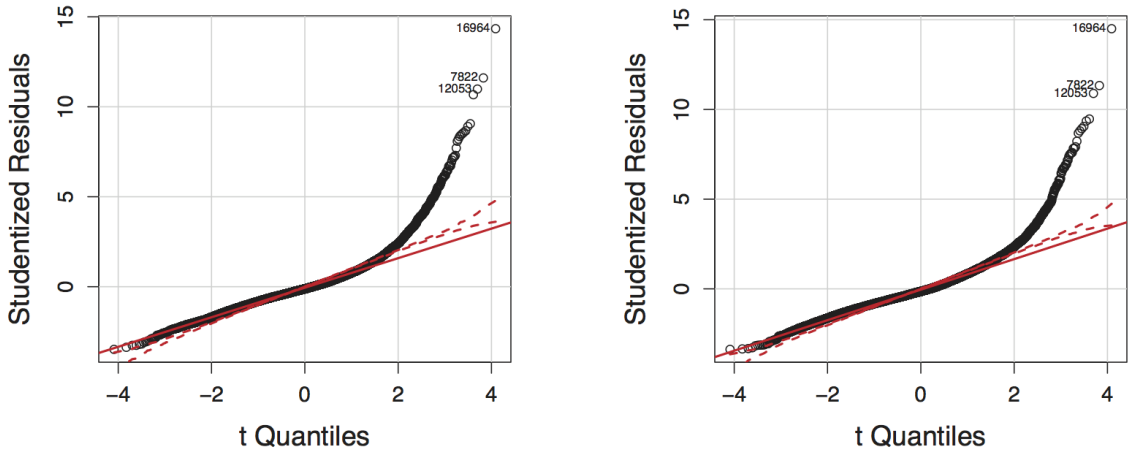


Fig. 6.2 Quantile-quantile diagnostic plots, comparing the standardised residuals against a standard Normal distribution, for `modelVer`=2 and 4, respectively. Only minor differences are visible. In each case the relevant `modelScope`='sig' formula is used and it seems that the residual distribution is right-skewed compared to a Normal distribution.

§ 5.3.4.3, the purpose was stabilising the variance (i.e., purpose 1) in this list). For the present chapter, the other purposes are also valid and so were considered separately. Procedures from the R `car` package (Fox and Weisberg, 2011) were used to estimate λ, these procedures being `powerTransform()`, `inverseResponsePlot` and `symboxPlot` respectively. As might be expected, the procedures did not agree on the optimal λ, being -0.2, 0.1 and -0.3 respectively for `modelVer`=2 and -0.3, -0.1 and -0.2 respectively for `modelVer`=4. Of the three alternative λ values for each `modelVer`, the variance-stabilising transformation λ from the `powerTransform()` procedure was chosen, for the following reasons:

- The best fit to the residuals in Figure 6.3 is very close to being a constant through the origin. Although there is a small amount of curvature in the best fit to the residuals, there is little evidence against the requirement that the fit should be a linear function of the predictors.
- There is evidence from Figure 6.3 and Figure 6.2 that the residual distribution is skewed in favour of more large residuals, but that this is correlated with larger service times. Therefore, we choose to stabilise the variance over the range of

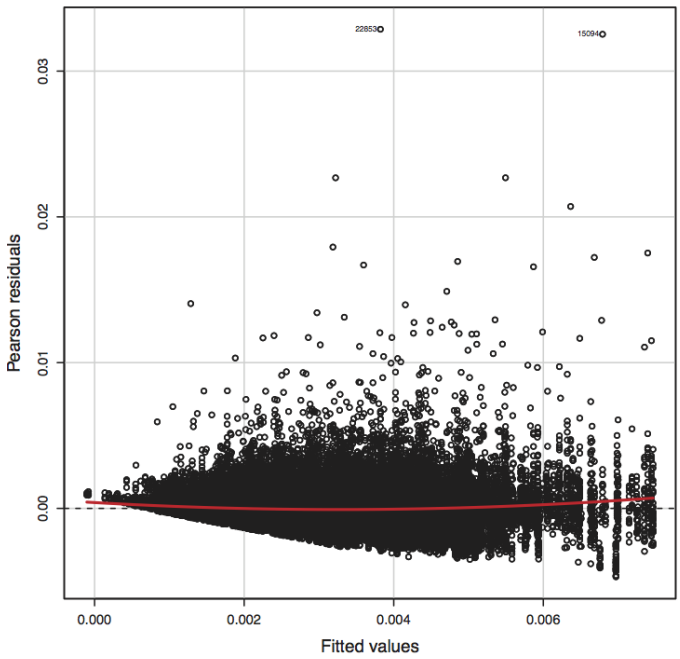


Fig. 6.3 Residual plot for `modelVer`=2 (`modelVer`=4 looks similar) and `modelScope`='sig' showing how residuals tend to increase as the dependent variable (service time duration) increases and there also appear to be upper outliers.

measured service times in the hopes that this step will also make the distribution of the residuals more symmetric.

The residuals of the fitted models obtained from the transformed data have more desirable features than the residuals obtained from the untransformed data. Comparing Figure 6.5 with Figure 6.3,

- the fit to the residuals is a constant through the origin, suggesting the linear model is adequate;
- the residuals have little or no structure (so most of the structure is captured in the model itself);
- there are fewer discordant observations, so some of the formerly unexplained observations are now explained by the model and so might be considered "mainstream".

(a) Quantile-Quantile residual plot, sig modelVer=2.

(b) Quantile-Quantile residual plot, sig modelVer=4.

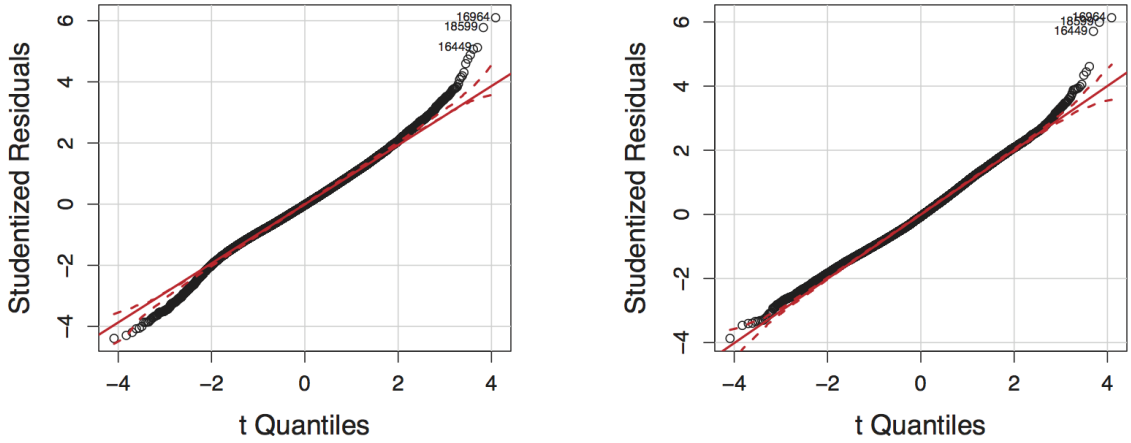


Fig. 6.4 Quantile-quantile diagnostic plots, comparing the standardised residuals against a standard Normal distribution, for `modelVer`=2 and 4, with $\lambda = -0.3$ and $\lambda = -0.25$ respectively. Compared to Figure 6.2, there is much better agreement between the residuals and a standard Normal distribution.

Comparing Figure 6.4 to Figure 6.2, agreement between the fit and the data extends much further into the tails, so analysis of the residuals suggests the revised model is significantly better and, in particular, is a better basis for prediction.

> The improved statistical models can be used to estimate the effect of (combinations of) the scenario parameters and hence to predict the consequences for access control performance.

As with § 5.4, effects plots (Fox, 2003) are used to visualize the estimated service time arising from different settings of factors, numeric predictors and their interactions.

6.5 The Extended Scenario Questions

The enhanced models, summarised as the two model versions in Table 6.4, contain many factors and numerical predictors. Indeed, when interactions are considered, hundreds of terms can be estimated. However, the majority of these terms do not have

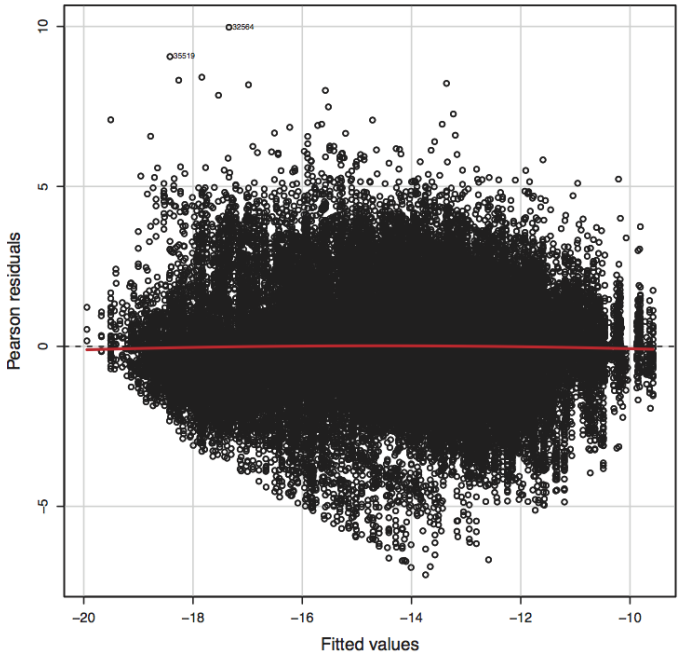


Fig. 6.5 Residual plot for `modelVer`=2 (`modelVer`=4 looks similar) and `modelScope`='sig'. Compared to Figure 6.3 the residuals form a symmetric cloud and there are fewer outliers.

a simple interpretation. Therefore, we take a different (demand-led) approach: what are the *detailed* research questions that, if answered, increase understanding of how service times depend on the factors being measured in the experiment. The detailed questions are termed *Scenario Questions*,

- to avoid confusion with the overall Research Questions of this dissertation, and
- to emphasise that they might correspond to scenarios studied by the security consultants introduced in § 5.3.

The first two scenario questions relate to the effect of policy authoring choices, the third to request handling and the last (of four) relates to the effect of domain size, represented as a graph measure, on measured service times.

(a) `modelVer=2`. (b) `modelVer=4`.

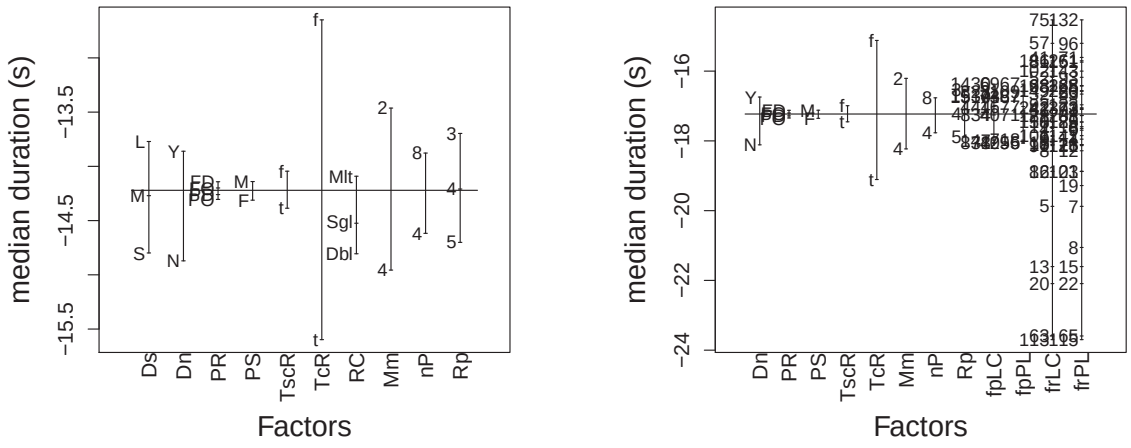


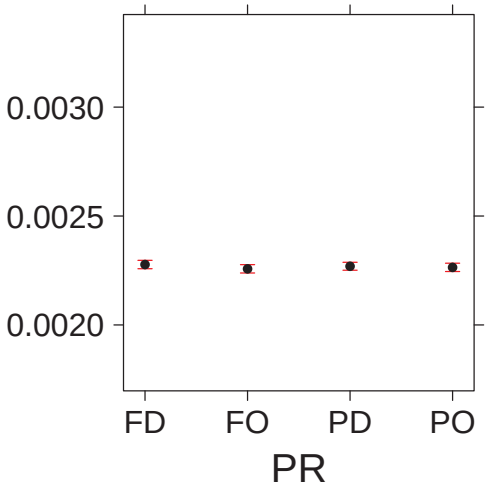
Fig. 6.6 Median transformed service times for each level of each factor when `modelVer`=2 (Figure 6.6a) and when `modelVer`=4 (Figure 6.6b). In the latter, the numeric parameters `pLC`, `pPL`, `rLC` and `rPL` have been represented as factors, for easy comparison with the factors in the model.

6.5.1 SQ1: Influence of Rule Combination and Placement (`PR`)

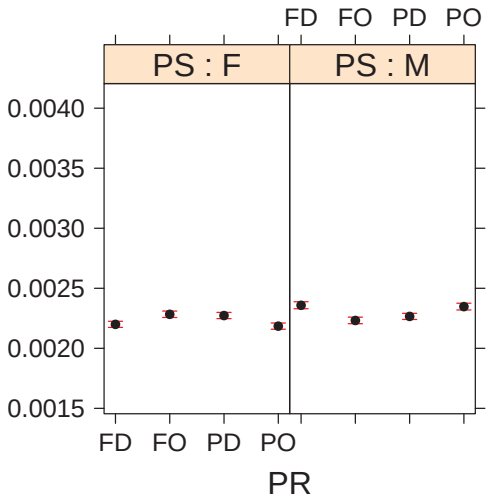
When policy authors specify a template (attribute-level) policy such as that in Figure 4.4, they can choose to arrange the rules any way they wish. One common pattern in access control policy authoring is to specify an "A" group of rules which all have a rule effect of either permitting or denying access, together with a smaller "B" group of guard conditions (each having the opposite rule effect to that found within the "A" group). It is likely that there is an optimal arrangement of "A" and "B" groups such that the amortized number of rule evaluations is minimized, thereby improving performance. The problem is that it is unclear what that optimal "A"-"B" rule structure should be *a priori*. In Figure 4.4 a `Permit-Overrides` combining rule is used, with just a single `Deny` clause, hence `PR` = `PO-OD` in this case. Three other policy arrangements are possible that are labeled with suitable `PR` values.

As can be seen from Figure 6.6 the `PR` factor has the least direct effect of all the factors considered in both models. Figure 6.7 indicates that, while the main effects are insignificant (see Figure 6.7a), some of the two-way interactions are significant. For example, the relative ordering of the effects as a function of `PR` level changes both in

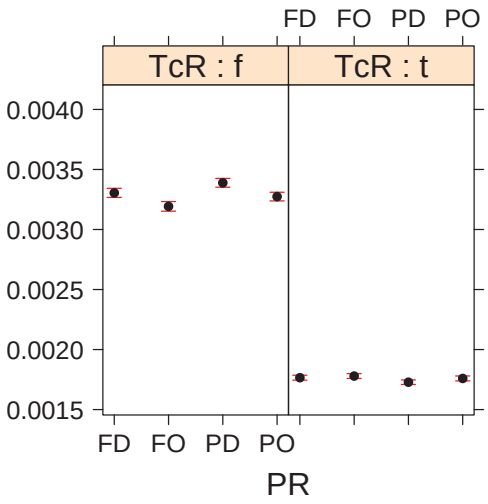
(a) `Effects plot, modelVer=2, PR.`



(b) `Effects plot, modelVer=2, PR:PS.`



(c) `Effects plot, modelVer=2, PR:TcR.`



(d) `Effects plot, modelVer=4, PR:poly(pLC,2).`

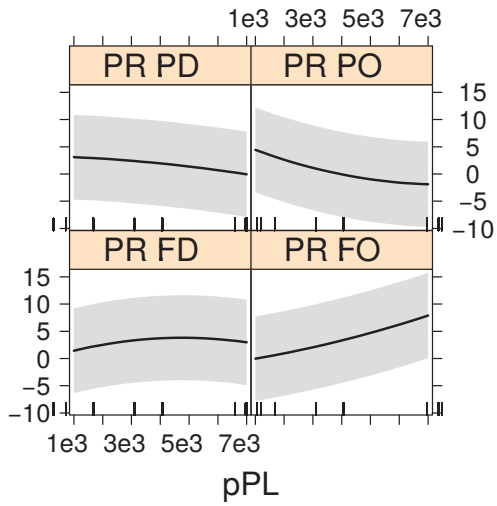


Fig. 6.7 Selected `PolicyRef` (`PR`) main and 2-way interaction effects from `modelVer`=2 and `modelVer`=4. The main effects are not significant but the chosen interaction effects plots are significant. For factors, the 95% confidence limits of an estimated quantity are indicated by error bars. For numerical predictors, grey shading serves the same purpose as error bars do for factors.

Figure 6.7b and Figure 6.7c. Performance when `PR` is `FA-OD` or `PO-DD` dis-improves when `PS` changes from `full` (more conditions in general) to `minimal` (fewer conditions in general). This is counter-intuitive, in that usually the assumption is that more conditions result in longer evaluation time. What might be happening here is that the

location (in the policy set) of the extra conditions is also significant because, if removed, short-cut evaluation might no longer be possible. Figure 6.7c indicates that characteristics of the requests should also be considered when choosing the best combining algorithms in the policies. For example, note the relative ordering of the PR factor levels when TcR = f: ("down", "up", "down"), compared to the the ordering when TcR = t: ("up", "down", "up"). Figure 6.7d shows that the effects of the numerical predictors can even change from convex to concave functions based on the the PR setting. Thus an aggregate value such as policy leaf count pLC, which depends mostly on domain size rather than the arrangement of the policies, can show quite different effects across its domain.

Summarising, the way the combining algorithms are used influences performance, but only in conjunction with other settings, and it is very difficult to predict what the best PR would be for a specific policy set unless service time measurements are available.

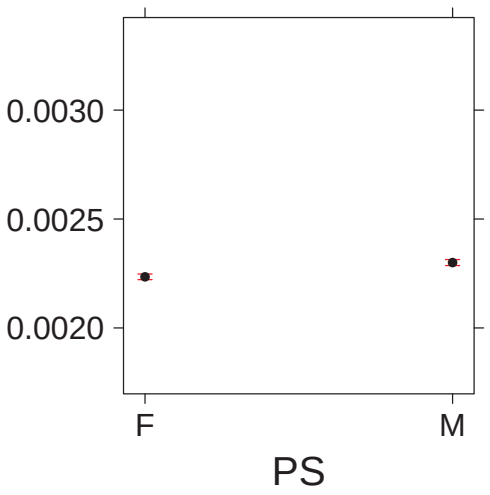
6.5.2 SQ2: Influence of Policy Specification Level (PS)

The template policy in Listing 4.4 does not ensure that static semantic constraints hold. When DomainManager derives the COARSE policies from the template policy, it adds TCs as necessary to enforce the static semantic constraints. When PS ≡ extraTcType = minimal, the extra TCs are just enough to enforce the semantic constraints. Otherwise, DomainManager adds as many TCs as it can (PS ≡ extraTcType = full), such that the semantic constraints hold and the decisions are not affected. One way of interpreting PS ≡ extraTcType = full is that the policies are *over-specified*, such as might occur when policy authors attempt to maintain large, complex policy hierarchies. We understand there is a strong temptation for policy authors to add extra rules "just in case".

Another interpretation is that policy authors accept that some policy redundancy exists and wish to predict the possible performance benefits if this redundancy were to be removed. The purpose of this investigation is to determine whether such effort (by policy authors or by DomainManager) is worthwhile.

Figure 6.6 indicates that the PS factor has relatively little effect but Figure 6.8a suggests it is still statistically significant (note that there is no overlap between the estimated service times when PS=full and PS=minimal. Figure 6.7b and Figure 6.8b

(a) `Effects plot, modelVer=2, PS.`



(b) `Effects plot, modelVer=2, DS:PS.`

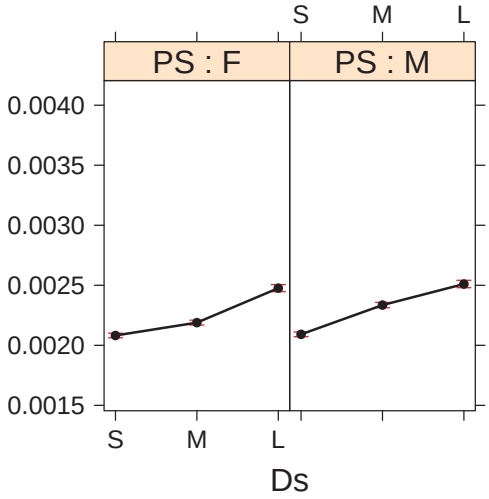


Fig. 6.8 Selected `PolicySelection` level (`PS`) main and 2-way interaction effects from `modelVer`=2. The main effects and interaction with domain size are significant, the latter because the expected service times do not increase unless `DS` is 'M' (medium).

suggest that `PS` has a small interaction with `PR` and with `DS`, but otherwise seems independent of the other factors. It is aliased with the policy graph measures, such as leaf count. Indeed, the leaf count when `PS=minimal` is generally a little over half that when `PS=full`, showing that there is a lot of scope within DomainManager to reduce the number of clauses. Unfortunately, the predicted service times when `PS=F` *is less than* that when `PS=M`. Such a finding is surprising, because most policy authors assume that removing redundancy must result in shorter service times. Clearly this is not always the case. Of course, reducing redundancy generally does make the policies easier to manage, but that may be at the cost of reduced performance.

In some cases, removing redundant policy rules can harm performance.

Summarising, to optimise policy evaluation performance, it is generally not enough to optimise the policy set alone, e.g., by statically analysing the policies and removing redundancy and similar conflicts. This is because the configuration of the requests play a role—some redundancy might actually benefit performance. The only way to *know* whether a semantically-redundant rule improves performance is to conduct comparative experiments in a testbed such as `STACS`.

6.5.3 SQ3: Influence of Request Cardinality (RC)

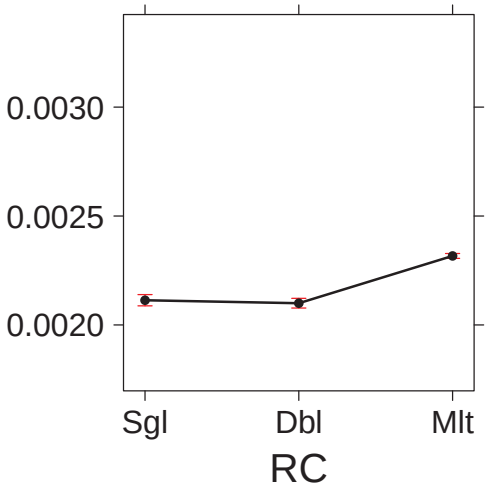
Sometimes the context needed to specify a request can be extensive, especially in cases where multiple requests are batched together. Combining requests in this way might be intended to help performance. Alternatively, the intention might be to satisfy a higher-order security property such as *binding of duties*, where subject S_1 and S_2 might need to cooperate to perform actions A_1 and A_2 on resources R_1 and R_2. This higher-order access request combines several simple requests of the form "Can X do Y with Z?".

When DomainManager generates requests, the attribute-level definitional clauses are derived from the policy set and their complexity varies from an empty clause (a tautology) up to the full complexity of the attribute-level policy element from which it was derived. One attribute-level request is created for each unique combination of such attribute-level definitional clauses. When deriving instance-level requests, more than one instance-level definitional clause can often be derived from a given attribute-level definitional clause. Consequently, the request generator can produce different requests depending on how many instance-level clauses are included for each attribute-level request clause. The generated requests are labeled with requestComplexity (labelled RC in plots). In PARPACS, measured service times use the equivalent request complexity (RC) label, where RC can take the values Sgl, Dbl and Mlt depending on whether the instance-based request clause cardinality is only 1, no more than 2 or unlimited.

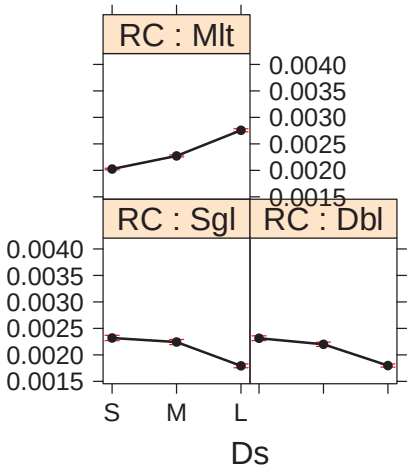
The RC factor is part of modelVer=2 but not of modelVer=4, because its definition overlaps with that of numerical terms like pLC. If both RC and rLC were included, any model-fitting procedure would struggle to estimate the model coefficient of each term independently of the other.

Generally, the difference between RC=Sgl and RC=Dbl is negligible (see Figure 6.9a and Figure 6.9c). Indeed, Figure 6.9c indicates that the times for RC=Sgl are also very similar to RC=Mlt when Mm=2Gb, but if more memory is added, there is a dramatic reduction in the RC=Sgl times, but not as much reduction in the RC=Mlt times. Therefore, when memory is limited, the JVM seems to throw out all request objects from memory (regardless of their size) but when more memory is available, it preferentially keeps the smaller requests in memory. Perhaps when memory is essentially unlimited, all requests would be kept in memory and the performance of the larger (RC=Mlt) requests would approach that of the smaller requests. This is

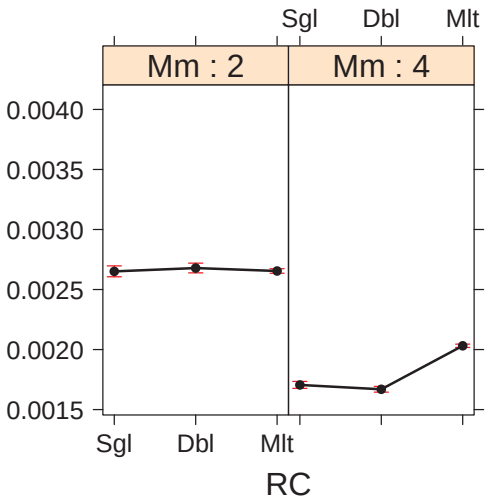
(a) `Effects plot, modelVer=2, RC.`



(b) `Effects plot, modelVer=2, DS:RC.`



(c) `Effects plot, modelVer=2, RC:Mm.`



(d) `Effects plot, modelVer=4, Mm:poly(rLc,2).`

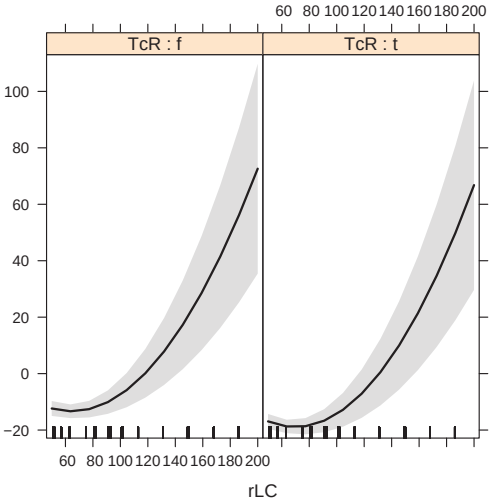


Fig. 6.9 Selected `contextRef` (`RC`) main and selected 2-way interaction effects from `modelVer`=2, with the `poly(rLC,2)` main effects from `modelVer`=2 for comparison. Generally, the difference between `RC`=`Sgl` and `RC`=`Dbl` is minimal compared to the difference between `RC`=`Sgl` and `RC`=`Mlt`.

consistent with the comments in §6.4.1 that JVM memory management plays a significant part in PDP performance.

> If there are many high complexity requests, adding memory can result in significantly better performance.

Figure 6.9b shows a surprising interaction between domain size (DS) and request complexity (RC). For smaller requests ($\texttt{RC} \in \{\texttt{Sgl}, \texttt{Dbl}\}$), measured service times actually *decrease* as the domain size increases. For $\texttt{RC} = \texttt{Mlt}$, the measured service time increases, as might be expected. This might be due to the fact that requests with few definitional conditions tend to match many policies unless the policies themselves have a large number of security conditions, in which case policy-request matches are fewer, so can take longer to find. It suggests that the best performance is obtained if the polices have many conditions and the requests have few, or if the policies have few conditions and the requests have many.

Figure 6.9d indicates that the request leaf count (rLC) has a very limited effect on service time; this conclusion also applies to any interaction terms involving rLC. The lack of explanatory power is consistent with Figure 6.6b, where the median for each value (level if viewed as a factor) of rLC shows no obvious pattern: $\{113, 63, 13, 20, 5, 16, \ldots, 41, 81, 75\}$. Thus a simple measure of the size of the request is not sufficient to predict, either on its own or in conjunction with other factors, the time it takes to process that request. Furthermore, similar comments apply to the related request path length (rPL). By contrast, the request cardinality (RC) is a much better predictor because it encodes a structural feature of each request, rather than measuring what appears to be an irrelevant detail. Unfortunately, this means that it is difficult to see how to predict PDP performance based on measured structural characteristics of the policies and requests alone (i.e., without taking account of the matching semantics, which would generally require evaluation of the policies and requests, preferably by a PDP in a testbed such as that provided by STACS).

> Simple, statically-derived structural measures of policies and requests, such as pPL and rLC respectively, have poor predictive power on their own. That fact, together with the presence of significant interactions among the other factors, indicates measurement-based predictions, as derived by the STACS and PARPACS components of ATLAS, are needed in practice.

6.5.4 SQ4: Influence of domain size (DS, pLC)

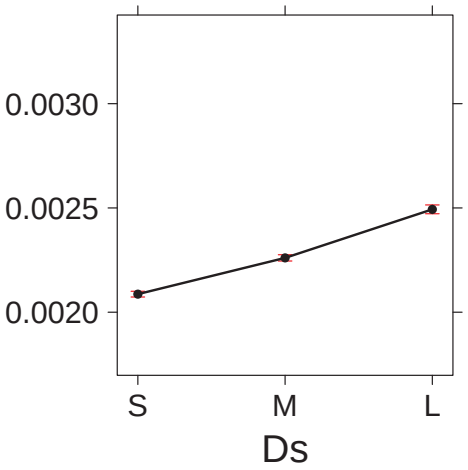
The size of the the static domain clearly influences the size of the generated instance-level policy set, and also the size of individual requests, particularly when RC = Mlt. For modelVer=2, domain size appears as an explicit DS factor. For modelVer=4, measures of graph entity size such as pLC and pPL (for policies) and rLC and rPL (for requests) are used instead. The purpose of this research question is to determine whether there is a way to predict whether domain size (however it is represented) can predict policy evaluation performance (measured as service times).

Figure 6.10a it appears that service times increase roughly linearly with domain size, if all other factors are kept constant. However, this is only part of the story. There are occasions when performance *increases* as the domain size DS) increases from 'medium' (M) to 'large' (L), see Figure 6.10b when TcR=f, particularly so when PS=M. One explanation might be that the policy is generally smaller when PS=M and the request becomes more general and so matches fewer policy clauses when TcR=f, so there are fewer rules to combine.

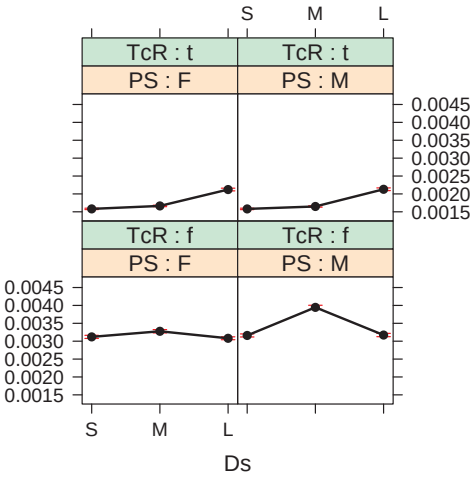
This is even more dramatic in the case of Figure 6.10c, where the service times *decrease* as the domain size increases from small to medium to large when the request cardinality (RC) is limited to Sgl and Dbl. This is particularly true when there is less memory available (Mm=2 rather than Mm=4). Less working memory means that keeping objects in memory becomes less attractive, and keeping track of many matches (of non-specific requests against specific policies) becomes more expensive (i.e., it increases the service time).

The policy leaf count (pLC) is positively correlated with domain size (DS), but it is not as good a predictor of PDP performance since the associated effects are small and the uncertainties are large. Nevertheless it is interesting that the "best estimate" of effects (small as they are) switches from negative to positive curvature as memory is increased.

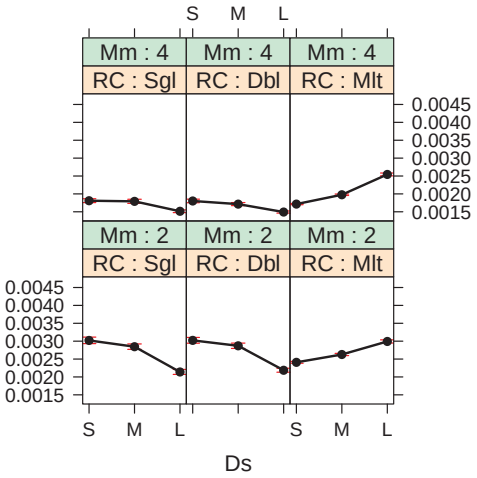
(a) `Effects plot, modelVer=2, DS.`



(b) `Effects plot, modelVer=2, DS:PS:TcR.`



(c) `Effects plot, modelVer=2, DS:RC:Mm.`



(d) `Effects plot, modelVer=4, Mm:poly(pLC,2).`

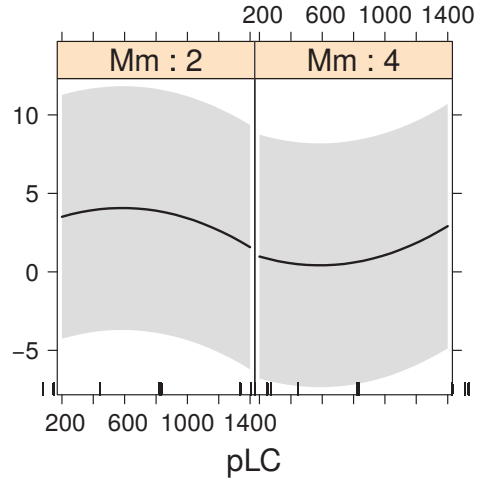


Fig. 6.10 Selected `RC` main and 2-way interaction effects from `modelVer`=2, with the `poly(pLC,2)` main effects from `modelVer`=4 for comparison. The difference between `RC`=`Sgl` and `RC`=`Dbl` is minimal compared to that between `RC`=`Sgl` and `RC`=`Mlt`.

6.6 Summary of the extended scenarios

The analysis and scenarios in §5.4 focused mainly on inter-PDP, domain size and resource effects. By contrast, the four scenario questions (§6.5.1, §6.5.2, §6.5.3 and §6.5.4) drill down in finer detail to policy and request characteristics. One consequence is that, with more terms involved, hence larger data sets being submitted for analysis, there is less averaging of the data and more detailed features emerge. At

the very least, we learn that it is difficult to make general statements about access control performance, because the main factors tell only part of the story. However, with suitable analysis, it is possible to get an answer to a *specific* question.

In some cases there are *apparent* differences between the findings of the basic scenarios in Chapter 5 and those of the more detailed scenarios in this chapter. For example, according to § 5.4 on page 201, adding memory has relatively little effect, but as we see above, when memory is limited and certain policy and request properties hold, adding memory *can* improve performance. Note that the same service time measurement data was used in both analyses. The different conclusions result from the presence of significant interaction effects so, if those effects are ignored, much of the variability in the data is unexplained (and hence averaged over the remaining main factors, becoming statistically invisible). As a consequence, care must be taken to "ask the right question" of the form "If other factors (and interactions) *can be ignored*, how does interaction X:Y (say) affect access control policy evaluation time?".

> `ATLAS`, and `PARPACS` in particular, is designed to provide statistical answers to precisely phrased questions, at whatever level of detail/filtering is required by its user.

It should also be noted that, even though both the scale and scope of the data used in the scenarios in this chapter are much greater than those of the summarised data in § 5.3, some features persist. Indeed, filtering by `rep` number (choosing just the middle three) and Box-Cox transformation of the service times help when fitting reliable statistical models in *both* situations. This is particularly interesting as these features appear to be `multiscale` in nature: they can be found at many levels of detail, unlike the scenario questions where specific statistical treatment is required.

The results in this chapter provide insights into policy evaluation and are statistically robust. However, in a real deployment, performance will be complicated by 1) queueing effects and 2) the fact that actual requests belong to a scenario-specific distribution: some requests arrive more frequently than others. These two factors ensure that actual performance will differ from the expected performance predictions from `ATLAS`. Therefore, these two considerations should be taken into account when setting up measurement-based simulations based on the ATLAS performance estimates. Such simulation results are more likely to transfer to actual deployments.

Chapter 7

Conclusions and Recommendations

This Chapter reviews the Research Questions and summarises the answers that are derived elsewhere in this dissertation. We then highlight the main Research Contributions and link them to the answers to the Research Questions. Having summarised the main achievements of the work to date, we then present some possible extensions, ending with some conclusions about access control evaluation performance.

7.1 Review of the Research Questions

Table 1.1 on page 4 outlines the Research Questions addressed in this dissertation, and is replicated as Table 7.1 in this chapter for convenience.

We address each of the Research Questions in turn below.

7.1.1 RQ1: How can access control evaluation performance be measured for use in performance experiments?

For the purpose of this dissertation, access control performance is interpreted in terms of the time taken to decide whether to permit or deny an access request, see § 3.2. The time taken for this operation adds to latency of any attempt to use a protected resource. This dissertation focuses on the time taken by the PDP to make the access decision.

Having defined *what* to measure, the next question is to define *how* to measure it. This is where the `STACS` measurement testbed (§ 3.3 on page 65) plays a role. As a testbed, it enables a researcher to measure policy evaluation performance by computing the elapsed time per request evaluation at the PDP. `STACS` also records the full context associated with a given request evaluation, including details of the PDP, policies, the request itself and the server instance on which the PDP is hosted. Each request evaluation generates a row in a `ServiceTime` table in a relational database.

Table 7.1 High-level Research Questions addressed in this dissertation

ID	Question
RQ1	How can access control evaluation performance be measured for use in performance experiments?
RQ2	How can domain models be specified and used to express enterprise access control scenarios?
RQ3	How can the data from performance experiments be used to understand and predict access control evaluation performance?
RQ4	What are the main factors affecting access control evaluation performance?

These records are kept for offline analysis. Figure 3.1 on page 67 shows the `STACS` architecture.

Having annotated measurements is not enough, in the sense that it is also necessary to support measurement *scenarios*, as mentioned below.

7.1.1.1 RQ1.1: What form does the service time distribution take?

§ 3.3.2 indicates that the service time distribution is generally not a simple distribution, such as a decaying exponential, say. Instead, it appears that service time requests are clustered at particular times. We believe this is because there are some requests for which the PDP follows the same search/matching path in the policies. Thus, while a group of requests might look different to each other, they are similar in terms of how they match the policies available to the PDP. Thus a typical service time distribution is best seen as a *mixture* of simpler distributions. The cluster positions and heights are derived from the service time measurements, which themselves depend on all the factors in the experiment.

7.1.1.2 RQ1.2: What simulations can be performed to explore the effect of different request arrival patterns?

§ 3.4 on page 72 describes several scenarios that feature measurement based simulation. One simulation (§ 3.4.5 on page 79) is used to investigate the effect of different admission control algorithms for a specific mix of requests (hence service time distribution), but with the mean arrival rate being varied from $\rho = 0.5$ to $\rho = 1.25$

and back to $\rho = 0.5$. This arrival pattern is meant to replicate intermittent overload conditions and the belief is that, by basing the simulation on the temporal distribution of the measured service times, the findings of the simulation are applicable to actual deployments having that request profile. The second simulation (§ 3.4.6 on page 84) assumes that the mean arrival rate stays the same, but the relative frequency of the requests change. Hence the service time distribution, which is weighted by the number of requests per cluster, changes and the question we seek to answer is how sensitive is the average latency to changes in request (and hence service time) distribution. Both of these scenarios extend the per-request service time measurements from `STACS` to become composite estimates of performance that change when the underlying conditions affecting request arrival rates change.

7.1.1.3 RQ1.3: What analysis can be performed when the systems under test use different languages, frameworks and encodings?

Even when relatively little is known about the policies and requests used in a performance experiment, it is still possible to compare various other factors, notably the `PDP` itself. § 3.4.6.3 on page 87 indicates that `STACS` is able to rank PDPs by their performance. Another possibility is to compare two very different PDPs: the reference `SunXACML PDP` (Java-based, working with XML-encoded XACML policies and requests) and a prototype `njsrpdp` (Javascript-based, working with JSON-encoded XACML policies and requests). `STACS` is able to quantify the relative performance advantage of the prototype PDP, and is partially able to explain that advantage by considering the differences in resource usage (§ 3.5 on page 91).

7.1.2 RQ2: How can domain models be specified and used to express enterprise access control scenarios?

RQ1 focused on collecting service time measurements, and on relatively simple analyses, typically relating to PDP comparisons and estimating performance as a function of both service time distribution and request arrival rate changes. Policies and requests taken from a different domain are used, given the absence of publicly available enterprise access control policy and request sets. Therefore we set out to build a domain model for enterprise access control that is flexible enough to express

typical policies and requests *and* to specify variants of these artifacts in order to compare the effects of different access control system specifications *in that domain*. We designed a relational model that had static, policy and request components, but which could be linked together as necessary (§ 4.2 on page 117). However, we found that the linked model was better realised as a property graph, because linking data using such models is trivial (using additional edges between the data nodes) and extending a property graph model to use additional properties per entity is much easier to accomplish than is the case with relational models (§ 4.3 on page 126).

To specify the models, it is necessary to:

1. define the static domain (creating instances of agents, assets and actions, and assigning attributes to each instance—§ 4.3.1 on page 130);
2. specify the rules in a hierarchical structure, expressing them in terms of the attributes in the static domain—these are the template policies;
3. derive the instance-based policies from 1) and 2) using the PolicyGen procedure in `DomainManager` (§ 4.4 on page 143);
4. derive the attribute- and instance-based requests from the attribute- and instance-based policies, respectively, using the RequestGen procedure in `DomainManager` (§ 4.5 on page 159).

In addition to the basic specification of the policies and requests, many different settings (factors) can be configured when generating the policies and requests. These range from quite technical settings such as `extraTcType`, through structural settings such as `requestCardinality` to major changes in settings such as `exportLanguage`. Thus it is possible to compare variants of deployment scenarios in the testbed.

7.1.2.1 RQ2.1: How can different variants of domain models be specified in a flexible and easy to use way?

The key to achieving this objective is a rigorous separation of concerns, so parameter a can be specified to have values a_1 or a_2 independently of parameter b. As an example, `extraTcType` can be set to any of three values, and `tcReduction` can be set to either of two values, so $3 \times 2 = 6$ combinations of settings are possible. All of the policy generation settings can be composed in this way (§ 4.4 on page 143).

Since requests are generated from policies, they depend on the policy settings as well as the settings that apply only to requests, such as `requestComplexity` (§ 4.5 on page 159).

7.1.2.2 RQ2.2 How can access control evaluation performance be compared at different domain sizes?

Because domain size is decoupled from policy semantics, it is easy to generate instance policies for different domain sizes. These instance policy sets are then suitable for performance comparisons (§ 4.4.3).

7.1.3 RQ3: How can the data from performance experiments be used to understand and predict access control evaluation performance?

As mentioned above, `STACS` and `DomainManager` provide the flexibility to conduct experiments that collect large volumes of service time measurements, and ensure that each measurement is fully annotated with all the context (attribute values) associated with that measurement. The scenarios are specified in terms of composable parameters, hence attributes. Because of this degree of experimental control, experiments are nominally full factorial in nature. Hence statistical procedures in the `PARPACS` component (§ 5.2 on page 185) can be used to estimate the effects of the main factors and as many of the multi-way interactions such that the observation matrix remains full rank).

7.1.3.1 RQ3.1: What types of exploratory data analysis are suitable for the performance experiments?

The first step in the analysis is to explore the data using statistical tests and plots, with the goal of deriving a (statistical) performance model (§ 5.3.4 on page 192). Among the exploratory plots used are design plots for the main factors, service time density plots, residual plots and variants of these. A guided statistical procedure is used to find an acceptable fit to the data. Some of the steps include removing unnecessary data, transforming the service time measurements and including the main

factors and all their interactions. All these model terms can be estimated because the observation matrix has full rank.

Many of the findings relating to access control performance were discovered while exploring the data. Examples include the benefits of the JVM holding objects in memory and the fact that there are more factors governing performance than just the main (controlled) factors, but if interaction terms are added to the model, it can account for some of this variability.

7.1.3.2 RQ3.2: What are the steps needed to build statistical models predicting access control performance?

Some of the steps to build statistical models occur while exploring the data, as described above. However, building an acceptable statistical model includes checking that various statistical properties hold (§ 5.3.4 on page 192 and § 6.4.1 on page 216). As soon as a reliable statistical model has been found, it can be used to estimate (and hence predict) access control performance. Because of the large number of terms in the model, we present our predictions as *effect plots* for selected factors and their interactions. Examples of such plots can be found in § 5.4 on page 201 and § 6.5 on page 219.

7.1.4 RQ4: What are the main factors affecting access control evaluation performance?

The factors affecting performance include:

- the mix of requests;
- the choice of PDP;
- the size of the domain;
- the complexity of the requests being handled;
- memory;
- the presence or not of redundant rule clauses.

However some of these (notably memory and rule redundancy) are more significant in interaction terms.

Other factors such as policy tweaks, the access decision, etc., tend to have relatively small effects.

7.1.4.1 RQ4.1: What are the effects of PDP choice, domain size and resources?

PDP choice has a very large effect, see Figure 3.11 on page 99 and Figure 5.2 on page 193.

The domain size also has a large effect, see Figure 5.9b on page 204. Its interaction with the `Pdp` and `RqCrd` factors is also significant (see Figures 5.9d 5.9e on page 204, respectively).

Memory does affect service time, but its interactions are more significant. Indeed, if there is already enough memory available, adding more can result in lower performance (Figure 5.2 on page 193).

7.1.4.2 RQ4.2: What are the effects of domain size, policy and request characteristics?

After the choice of PDP, policy and request characteristics are probably the next best predictors of performance, see Figure 6.1.

7.1.5 Extension to general client-server performance

These research questions are quite specific in the sense of applying to access control performance. However, it is possible to rephrase them so that they apply to other domains. As an example, equivalent research questions relating to database performance might read:

RQ1-db How can *database query* performance be measured for use in performance experiments?

RQ2-db How can domain models (in the form of data to be stored in the database, and representative queries against this data) be specified and used to express *database query* scenarios?

RQ3-db How can the data from performance experiments be used to understand and predict *database query* performance?

RQ4=db What are the main factors affecting *database query* performance?

As can be seen, variants of the Research Questions used in this dissertation can be used in many other client-server settings where server performance is a concern.

7.2 Summary of main contributions

The main contributions have been presented in tabular form in Table 1.2 on page 8, for findings that are expected to apply to other enterprise access control scenarios, and Table 1.3 on page 10 for innovative models and infrastructure that is intended to enable future research advances.

Some highlights are listed below:

- Service times are clustered and the clustering is particularly sensitive to the choice of PDP, as well as to latent properties of the policies and requests (§ 3.3.2 on page 67);
- Given the service time distribution, the researcher can use simulations to investigate the effects of varying request arrival rates and/or mixes of requests (§ 3.4.6 on page 84;
- The prototype `njsrpdp` has shorter service times and uses less memory and CPU resources than either `SunXACML PDP` or `EnterpriseXACML PDP` (§ 3.5.3 on page 98);
- Although it was not possible to decouple the efficiency advantages of using the Node.js prototype PDP from those accruing from the use of JSON policies and requests, it is clear that reducing the size of each JSON request has a significant effect, but making the policy encoding more efficient has negligible effect (§ 3.5.4 on page 103);

- Turkmen and Crispo (2008) suggests that `EnterpriseXACML PDP` has better evaluation performance than `SunXACML PDP` on their test data. § 3.4.6.2 on page 86 uses `continue` policies and agrees with this. Also, the service time distribution suggests that `SunXACML PDP` has fewer request clusters and so is more predictable. However, when the `multi22` requests are excluded and outliers are removed, § 3.5.2 on page 96 suggests that the position is reversed. This suggests that `SunXACML PDP` has more extreme values but, for relatively simple policies and requests, might even outperform `EnterpriseXACML PDP`, see § 3.4.6.2 on page 86 and § 3.5.2 on page 96;
- `SunXACML PDP` v1.2 has much better scalability than `SunXACML PDP` v2.0. One of the main changes with `SunXACML PDP` v2.0 is to delegate XML processing to a JAXB provider and to elements of the Spring framework, as this provides a better abstraction for element finding and hence should make maintenance and extension easier. Unfortunately, this comes with a high price in performance terms, where the effects plots for `PDP` (Figure 5.9a on page 204), `PDP:DS` PDP × DomainSize interaction: (Figure 5.9d on page 204) and `PDP:DS:RC`A (PDP × DomainSize × RequestCardinality interaction: (Figure 5.9h on page 204) strongly indicate a problem with the scalability of `SunXACML PDP` v2.0 that is much more serious than that with `SunXACML PDP` v1.2. (§ 5.4 on page 201)
- The JVM caches evaluation results automatically. The garbage collection rate of the JVM is sufficiently low (on a dedicated PDP server) that subsequent requests (of the same kind) can get the result from the cache much faster than calculating it anew each time, see Figure 5.3 on page 194 and Figure 6.1 on page 215 (§ 5.3.4.1 on page 193).
- Simple linear predictive models are not sufficient. A linear model can be used for prediction, but it needs more than just the main factors: second, third and higher order interactions are also significant. The researcher can control the main factors but, when building the model that fits the data, there is clear evidence of underfitting unless many of the interaction terms are also included, see Figure 5.4 on page 195 (§ 5.3.4.2 on page 196)
- A linear model with factors and interactions underfits the data unless the dependent variable (service time in this instance) is transformed to make the distribution of the residuals more constant over its range. Ideally, the residuals

of a linear least squares fit have constant variance over the range of the dependent variable and have their distribution is symmetric and centred on 0, etc. When these assumptions do not apply, a transformation can help, as it does here. Compare Figure 5.4d on page 195 (untransformed) against Figure 5.6b on page 198 (transformed). See also § 5.3.4.3 on page 196;

- Measured service times are needed to predict performance. It is possible to measure many characteristics of policies and requests, but it is difficult to build a reliable predictive model from just those terms. It would be much more convenient to use the (static) characteristics of policies and requests to predict performance, precluding the need for extensive measurement experiments. Unfortunately the predictive power of many of these characteristics is weak. See Figure 6.6 on page 221 where it is obvious that the leaf count and path length parameters have no obvious correlation with service time, and this feature is made even more clear in effect plots such as Figure 6.7 on page 222 (§ 6.5.1 on page 221);
- Some of the "rules of thumb" relating policies to performance are misleading. Small changes to policy structure have minimal effects on performance: when writing policies, there are many ways of tweaking the rules to achieve the same results. However such changes seem to have little effect on performance. See Figure 6.6 on page 221 where it is clear that the `PR` factor, which encodes several types of policy structure, has almost no effect on performance compared to other factors. (§ 6.5.1 on page 221);
- Removing duplicate rule clauses can *increase* service times. Most researchers assume that removing duplicate rule clauses should reduce service times. However this is not always the case, see Figure 6.7 on page 222 and Figure 6.8 on page 224, possibly because certain evaluation shortcuts disappear when the duplicate clauses are removed. This suggests that sometimes it is better to leave some redundancy in the policy set, in lieu of refactoring the policy set to optimise the placement of those logical conditions (§ 6.5.2 on page 223);
- Increasing the Request Cardinality (hence request size) and simultaneously increasing the policy size is not guaranteed to increase service times compared to smaller policies and requests. See Figure 6.9b on page 226 where we believe this is down to the fact that a search problem can stop as soon as either a match has

been found, or cannot be found. If a request is "large" it needs to match a "large" part of the policy but a smaller request could match more of a large policy, in principle. Indeed, the complexity of each request receives little attention in the literature but it seems that the best performance is obtained when *either* the policies *or* the requests have many logical clauses (`Ds` = '`S`' and `RC` = '`Mlt`' *or* `Ds` = '`L`' and `RC` = '`Sgl`'), but not both large (§ 6.5.3 on page 225);

- Adding more system resources does not always improve performance. Most researchers assume that PDP resource demands are elastic in the sense that adding more memory/CPU helps the PDP to scale upwards as load increases. This is not always true. When more than adequate resources are available, e.g. in the case of Figure 5.2b on page 193 and Figure 5.9g on page 204, it appears that changing the resources has no effect. Real enterprise scenarios might be expected to have greater semantic complexity (not just size, as here) and so perhaps additional memory would be advisable then. When memory and CPU are more limited, see Figure 6.1b on page 215, it is clear from the `Dsize:RqCrd:Mem` interaction effects plot (Figure 6.10c on page 229 that the dependence on memory is not monotonic. This suggests that beyond a certain point (predicted by the experiment) it might be better to scale outwards rather than upwards. Alternatively, consider policy decomposition techniques (Decat et al., 2012; Deng et al., 2014; El Kateb et al., 2012), particularly for larger domains (§ 6.5.4 on page 228).

7.3 Recommendations for Future Work

Given the research contributions presented in §7.4, it can be asked whether more research contributions are possible from the same source. The answer is yes; seven areas for future work are outlined in this Section.

7.3.1 Work in Progress; Short Term

Work in this category has either started and is largely complete, or is well understood and has relatively low risk.

7.3.1.1 Comparison of XACML 2.0 vs XACML 3.0

The XACML 3.0 specification (Rissanen, 2013) has been published and work has started on extending `DomainManager` to be able to export XACML 3.0 policies and requests, in addition to the XACML 2.0 described in this dissertation. Note that the features of the exported policies and requests are those that are shared by XACML 2.0 and 3.0.

As well as extending `DomainManager`, additional PDP adapters are needed for `STACS`. Candidate open source XACML 3.0 PDPs include `Balana` (Asela, 2015), `ATTxacml` (Dragosh and Knust, 2015), `xacml4j` (Sevelis, 2014) and `sne-xacml` (Ngo, 2014).

An adapter for `Balana` has already been added to `STACS`. Interestingly, `Balana` can evaluate both XACML 2.0 and 3.0 policies and requests, so it is possible to compare it with other XACML 2.0 PDPs using a common set of XACML 2.0 test artifacts, and to use the XACML 3.0 versions of those artifacts to compare its XACML 3.0 policy evaluation performance with its XACML 2.0 policy evaluation performance. We have some preliminary results that suggest that `Balana`'s XACML 2.0 and 3.0 performance are very similar, and as a XACML 2.0 PDP, it performs as well as `SunXACML PDP` v1.2 from which it was derived.

Since `sne-xacml` was developed with high performance as an explicit objective, it would be interesting to compare it against its XACML 3.0 peers, such as `Balana`. A suitable research question might be: "To what extent does the optimised BDD formulation of `sne-xacml` improve performance compared to more traditional formulations?"

7.3.1.2 Comparison of Javascript/JSON versus Java/XML PDPs

§3.5 describes a set of experiments in which the performance of a limited prototype PDP (`njsrpdp`) was compared with that the reference `SunXACML PDP` implementation. `njsrpdp` was implemented in Javascript and works with JSON policies and requests only. As a prototype, it did not implement all the XACML 2.0 specification, just enough for the `continue` policies and the `single` requests; even the `multi22` requests were out of scope.

In parallel with the work described in this dissertation, a colleague (Mr Fan Zhang) has taken the `SunXACML PDP` implementation and re-implemented it in Javascript. His `njspdp` accepts XML-encoded policies and either XML- or JSON-encoded requests. The new PDP is a full implementation of the XACML 2.0 specification: it passes the XACML 2.0 conformance test suite (Kuketayev, 2005). Performance comparisons with `SunXACML PDP` suggests `njspdp` is significantly faster.

Since the new PDP is essentially a fork of `SunXACML PDP` v1.2 but using Javascript patterns, idioms, syntax and libraries instead of their Java counterparts, we propose the following research question: "What is the performance benefit of migrating `SunXACML PDP` from Java on the JVM to the Javascript/Node.js ecosystem, and what is the added benefit (if any) of encoding requests in JSON rather than XML?"

7.3.1.3 Making DomainManager easier to use

The plethora of files and settings used to configure a `DomainManager` scenario run (to generate policies and requests) can be daunting for new users. A colleague (Mr Shahzada Ali Saleem) has built a prototype web application to capture the required configuration settings in a more friendly way, using context-sensitive web forms and information flows. His GUI saves `DomainManager` scenario configurations in text files in the standard format format required by `DomainManager`. However, the scope for user error and misconfiguration in these files is greatly reduced, compared to editing them manually.

7.3.2 Medium Term

Work in this category has not started and would require significant software engineering to complete. It would also require a new suite of experiments to capture the measurements and interpret the results.

7.3.2.1 Attribute-level versus instance-level evaluation

§4.4 describes a technique to address the impedance mismatch between attribute-level policies and instance-level requests. `PolicyGen` (a component of `DomainManager`) starts from the attribute-level template policies and generates instance-level policies

that are consistent with the access requests submitted to the PDP. This has the benefit that we can resolve the mismatch at the policy authoring stage, rather than at the policy evaluation stage. However it also means that policy generation should be run frequently, to ensure that the generated policies are up to date.

Another way to resolve this mismatch is to forego the generation of instance-level policies from attribute-level policies. Instead, the procedure would be to convert instance-level requests to attribute-level requests, by looking up their attributes in a PIP before sending them to the PDP for evaluation. There are at least three potential benefits to this approach:

1. Policy refreshes would be less frequent. There is no need to refresh the policies unless an attribute-level rule is added, removed or updated. Such changes would not happen as often as changes elsewhere in the domain model.
2. The policy set is smaller, so *other factors being equal*, the service time for each evaluation might be less.
3. The mapping from instance-level to attribute-level requests would be many-to-one, so many of the (formerly instance-level) requests would appear as duplicates to the PDP. Consequently caching of responses becomes much more attractive.

The main drawback is that more computation is needed at policy evaluation time, *before* the request arrives at the PDP.

It would be very interesting to compare the two approaches. A research question might be "What is the performance benefit of evaluating policies and requests at attribute-level rather than instance-level, and is it better overall when the extra evaluation-stage lookup times are included?"

7.3.2.2 Access control in the Internet of Things

Another colleague, Mr Michael Wall, is looking for a MSc dissertation topic relating to access control in the *Internet of Things*. We have discussed various options, but the topic is most likely to relate to modifying policies so that they can be evaluated in a distributed fashion on resource-constrained devices, where increased energy efficiency is as important as reduced service times per request.

7.3.3 Longer Term

Work in this category has not started and could potentially form the basis of new lines of research.

7.3.3.1 Analyse distributed PDP performance

At present `STACS` can be used to measure access control performance on a single PDP at a time. Furthermore, the PDP under test operates in the same JVM or Node.js instance as `STACS`. However, in enterprise deployments, access control systems are distributed across different servers. Communication between servers is via TCP sockets or web service calls. Thus service time also needs to include request and response transport overheads. Depending on the request arrival distribution and transport protocols, queueing is also possible. In that case, it is also possible to consider the effects of different queueing service disciplines, and the benefits of multiple service points (PDPs in this scenario) and hence whether the PDP function should be scaled outwards.

The OPNET™ simulations in §3.4 helped to predict some of the queueing behaviour of *single* PDP deployments. If this were combined with measurements of multiple and/or distributed PDP deployments, it would greatly expand the types of analysis currently available in `ATLAS`.

7.3.3.2 Performance-aware policy authoring

At the moment, we understand that policy authoring focuses primarily on functional requirements such as completeness and correctness. However, given knowledge of typical requests, it might be possible to arrange the rules in such a way that it is easier to find a match. One way might be to transform the internal graph representation of the (template) policy to a Multi-Terminal Binary Decision Diagram, which is also a graph but in a condensed format. The enhanced `DomainManager` could then export that condensed graph as an "optimised" XACML policy set. Other types of policy optimisation could also be tried, such as policy reconfiguration, etc.

7.3.3.3 Extension to database performance analysis

Since one of the common use cases for enterprise access control is to protect data that is retrieved from a database, it would be interesting to consider access control performance in this wider context. Row-oriented, disk-based relational database management systems such as those offered by Oracle® Microsoft® and similar vendors had become dominant. DBA certification courses included a module on performance tuning. However, databases themselves, and the infrastructure supporting them, are now much more varied. In relation to infrastructure, the traditional model of the database being hosted on a large server with attached storage has given way to distributed database deployments, hosted on virtualised infrastructure, possibly off-premises, with storage being decoupled from management functions and iwith multiple layers of abstraction (query rewriters, load balancers, etc.) being added between the client and server. Even the database management system itself has changed, with a growing list of alternatives including *NewSQL* (where the query language and logical model remains the same, but the physical implementation is dramatically different, e.g., being column-oriented (e.g., MonetDB), or in-memory (e.g., VoltDB), or massively parallel (e.g., NuoDB)). Other database storage systems eschew the relational model and offer query languages that might look like SQL, but have different semantics. Such databases are often termed Not-only SQL (NoSQL) and include key-value stores (e.g., redis), document databases (e.g., mongodb®), column-family (e.g., HBase, built on hadoop), semantic databases/linked data stores (e.g., GraphDB™) and graph databases (e.g., neo4j™) (Robinson et al., 2015).

With all these differences in models, infrastructure and technology, there is a wealth of factors to consider before deploying a database. Many of these factors will affect performance. Simplistic comparisons, using standard benchmarks and metrics, tell only part of the story. What is needed is a means of generating meaningful scenario-specific data for populating the database, coupled with an event model for generating representative queries against that data. By conducting performance experiments, and fitting a statistical model to the performance data collected from these experiments, it should be possible to make predictions about performance and scalability, and to estimate the uncertainty in those predictions.

As stated in previous chapters, the ATLAS framework provides a good basis upon which to start planning and implementing a performance analysis framework for other domains:

Performance testbed The generic components of STACS are relevant, but a lot of software engineering challenges remain, e.g., to build a set of adapters that hide many of the specifics of how to formulate queries and interpret responses.

Data and query generation DomainManager indicates the scope of what would be required, but a completely new component would be needed to generate data and query artifacts for database performance experiments.

Reporting and Analysis PARPACS has the software engineering in place to be extended into new settings such as database performance analysis. However, subject matter experts would be needed to configure the new scenarios, and a data scientist would be needed to work with such experts to interpret the results.

Therefore, this extension would be considerably more challenging than even the extension to integrate performance prediction into policy authoring.

7.3.4 Future work summary

While the research to date has led to significant research contributions and interesting findings, there is still scope for new research to be conducted based on what has been achieved to date.

In that regard we intend to look for research collaborations and funding opportunities to continue our research on client-server performance analysis and on access control performance analysis in particular.

7.4 Access Control Evaluation Performance: General Principles

Having studied access control performance, and attempted to advance the state of the art, we strongly believe in the following principles:

An Experimental Testbed is Needed. The lack of an agreed benchmark, either in terms of a testbed, or indeed a reproducible means of running experiments on that testbed should it exist, meant that the evaluation in each case was less convincing than it could have been. For the research to be reproducible, it is necessary to provide more details on the experimental conditions. This is especially true in relation to performance experiments. The use of a common testbed like `STACS` would greatly ease the reporting of experimental evaluations, e.g., it means that service times are measured and aggregated in a consistent fashion, so that part of the results presentation does not need further elaboration. The researcher is then free to discuss more significant details of the performance experiment, such as what computing resources were available, or what policies and requests were used;

Relevant policies are needed. Turkmen and Crispo (2008) note that it is very difficult for researchers to obtain actual policies and requests, owing to the sensitivity of the policy and request data itself. `DomainManager` provides a partial solution: it uses policy refinement principles to generate instance-level policies from attribute-level policies. However, it still needs template policies as input. Such policies are best written by subject matter experts. Perhaps such experts might be persuaded to share their policies with the research community, since they would not need to reveal instance-level details about their enterprises.

Requests consistent with those policies are needed. Enterprise access request data is likely to remain sensitive. However, `DomainManager`'s `RequestGen` procedure can generate requests from policies. These requests are consistent with the policies used to generate them (by construction) and yet they are also somewhat different (owing to `TC` and/or `TSC` reduction). Perhaps in future `RequestGen` could have further options for bulk request generation;

Formal performance analysis is needed. We have already seen that `STACS` enables researchers to collect service time measurements from experiments. Furthermore `DomainManager` enables those researchers to control what policies and requests are used in those experiments. All of this effort is to no avail unless researchers can analyse the measurements and interpret the results. The first requirement, with any such analysis, is to select and condition the data in preparation for that analysis. `PARPACS` can be configured to use a variety of data adapters. The next requirement is to explore the data, plotting it and learning

about its features: what main effects are larger than others? what does the service time distribution look like? etc. Given this understanding of the main features, `PARPACS` enables the researcher to fit extended linear models to the prepared data. Because of a rigorous separation between control and function within `PARPACS`, control settings such as model terms, desired diagnostic plots etc. can be set externally and `PARPACS` will respect those settings and provide customised analyses. Other authors appear to use a more limited analysis procedure that what is available in `PARPACS`, so it appears that `PARPACS` represents a considerable improvement over what is available to other researchers who investigate access control evaluation performance;

An integrated toolchain is needed. Given the scope and complexity of `STACS`, `DomainManager` and `PARPACS`, it is essential for them to work together. This is achieved by ensuring that control flow, data formats, etc. are consistent. Use of control tables in a sqlite database ensures that all analyses are versioned and traceable to the relevant `STACS` and `DomainManager` runs. The components work together in such a seamless fashion that it is possible to consider that they belong to an overarching framework, which we call `ATLAS`. As such, they can be used to solve engineering problems, such as dimensioning an access control system prior to deployment.

We believe these principles provide a sound foundation for future research on access control evaluation performance.

Bibliography

Abadi, D., Agrawal, R., Ailamaki, A., Balazinska, M., Bernstein, P. A., Carey, M. J., Chaudhuri, S., Chaudhuri, S., Dean, J., Doan, A., Franklin, M. J., Gehrke, J., Haas, L. M., Halevy, A. Y., Hellerstein, J. M., Ioannidis, Y. E., Jagadish, H. V., Kossmann, D., Madden, S., Mehrotra, S., Milo, T., Naughton, J. F., Ramakrishnan, R., Markl, V., Olston, C., Ooi, B. C., Ré, C., Suciu, D., Stonebraker, M., Walter, T., and Widom, J. (2016). The Beckman Report on Database Research. *Commun. ACM*, 59(2):92–99.

Abou-Tair, D. e. D. I., Berlik, S., and Kelter, U. (2007). Enforcing Privacy by Means of an Ontology Driven XACML Framework. In *IAS '07: Proceedings of the Third International Symposium on Information Assurance and Security*, pages 279–284, Washington, DC, USA. IEEE Computer Society.

Agha, G. A. and Kim, W. (1999). Actors: A unifying model for parallel and distributed computing. *Journal of Systems Architecture*, 45(15):1263 – 1277.

Akaike, H. (1974). A new look at the statistical model identification. *Automatic Control, IEEE Transactions on*, 19(6):716–723.

Akers, Jr., S. B. (1959). On a Theory of Boolean Functions. *Journal of the Society for Industrial and Applied Mathematics*, 7(4):487–498.

Ammann, P., Offutt, J., and Huang, H. (2003). Coverage Criteria for Logical Expressions. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 99, Washington, DC, USA. IEEE Computer Society.

Anderson, A. H. (2005). Core and hierarchical role based access control (RBAC) profile of XACML v2.0. OASIS Standard.

Asela (2015). "Balana" The Open source XACML 3.0 implementation.

Berrut, J.-P. and Trefethen, L. N. (2004). Barycentric Lagrange Interpolation. *SIAM Review*, 46(3):501 – 517.

Bertolino, A., Daoudagh, S., Lonetti, F., and Marchetti, E. (2012). Automatic XACML Requests Generation for Policy Testing. In *Proc. IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST 12)*, pages 842–849.

Bertolino, A., Gao, J., Marchetti, E., and Polini, A. (2007). TAXI–A Tool for XML-Based Testing. In *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pages 53–54.

Brewer, D. F. C. and Nash, M. J. (1989). The Chinese Wall Security Policy. In *IEEE Symposium on Security and Privacy*, pages 206–214.

Brossard, D. (2014a). Json profile of XACML 3.0 Version 1.0.

Brossard, D. (2014b). Understanding XACML combining algorithms. Developer Blog.

Butler, B. (2015a). DomainManager application. Git Code Repository. Accessed 2015-11-12.

Butler, B. (2015b). PARPACS application. Git Code Repository. Accessed 2015-11-12.

Butler, B. (2015c). STACS application. Git Code Repository. Accessed 2015-11-12.

Butler, B., Cox, M., Forbes, A., Harris, P., and Lord, G. (1999). Model validation in the context of metrology: a survey. NPL Report 19/99, National Physical Laboratory, Teddington, UK. UK NMS Software Support for Metrology Programme.

Butler, B. and Jennings, B. (2015). Measurement and Prediction of Access Control Policy Evaluation Performance. *Network and Service Management, IEEE Transactions on*, 12(4):526–539.

Butler, B., Jennings, B., and Botvich, D. (2010). XACML Policy Performance Evaluation Using a Flexible Load Testing Framework. In *Proc. 17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 648–650. ACM. Short paper.

Butler, B., Jennings, B., and Botvich, D. (2011). An experimental testbed to predict the performance of XACML Policy Decision Points. In *Proc. IM 2011 - TechSessions*.

Chadwick, D., Zhao, G., Otenko, S., Laborde, R., Su, L., and Nguyen, T. A. (2008). Permis: a modular authorization infrastructure. *Concurr. Comput. : Pract. Exper.*, 20:1341–1357.

Cisco (2012). Voice and Unified Communications.

Craven, R., Lobo, J., Lupu, E., Russo, A., and Sloman, M. (2011). Policy refinement: Decomposition and operationalization for dynamic domains. In *Proc. 7th International Conference on Network and Service Management (CNSM 2011)*, pages 1–9.

Croarkin, C. and Tobias, P. (2015). NIST/SEMATECH e-Handbook of Statistical Methods. Accessed 2015-03-10.

Crockford, D. (2006). JSON: The Fat Free Alternative to XML. In *15th International WWW Conference*, Edinburgh.

Crockford, D. (2011). ECMAScript Language Specification.

Dahl, R. (2011). Node.js: Evented IO for V8 javascript.

Davy, S., Barron, J., Shi, L., Butler, B., Jennings, B., Griffin, K., and Collins, K. (2013). A Language Driven Approach to Multi-System Access Control. In *Proc. IM 2013 - AppSessions*, Ghent, Belgium.

Davy, S., Jennings, B., and Strassner, J. (2007). The policy continuum–a formal model. In *Proceedings of the Second IEEE International Workshop on Modelling Autonomic Communications Environments (MACE 2007)*, pages 65–79.

Davy, S., Jennings, B., and Strassner, J. (2008). The policy continuum—Policy authoring and conflict analysis. *Computer Communications*, 31(13):2981–2995.

Special Issue on "Self-organization and self-management in communications as applied to autonomic networks".

Decat, M., Lagaisse, B., and Joosen, W. (2012). Toward Efficient and Confidentiality-aware Federation of Access Control Policies. In *Proc. 7th Workshop on Middleware for Next Generation Internet Computing (MW4NG '12)*, pages 41–46. ACM.

Deng, F., Chen, P., Zhang, L.-Y., Wang, X.-Q., Li, S.-D., and Xu, H. (2014). Policy Decomposition for Evaluation Performance Improvement of PDP. *Mathematical Problems in Engineering*, 2014:14.

Dougherty, D. J., Fisler, K., and Krishnamurthi, S. (2006). Specifying and Reasoning About Dynamic Access-Control Policies. In *IJCAR*, pages 632–646.

Dragosh, P. and Knust, H. (2015). AT&T XACML 3.0 Implementation. Now part of the Apache OpenAz incubator project at http://incubator.apache.org/projects/openaz.html.

Egelman, S., Molnar, D., Christin, N., Acquisti, A., Herley, C., and Krishnamurthi, S. (2010). Please continue to hold: An empirical study on user tolerance of security delays. In *WEIS*.

Eifrem, E., Hunger, M., and Robinson, J. (2015). Neo4j database.

El Kateb, D., Mouelhi, T., Le Traon, Y., Hwang, J., and Xie, T. (2012). Refactoring Access Control Policies for Performance Improvement. In *Proc. 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*, pages 323–334. ACM.

Esposito, R. and Cole, M. (2013). How Snowden did it. http://www.nbcnews.com/news/other/how-snowden-did-it-f8C11003160. NBC News article.

Ferraiolo, D. and Kuhn, R. (1992). Role-Based Access Control. In *Proc. 15th NIST-NCSC National Computer Security Conference*, pages 554–563.

Ferraiolo, D. F., Barkley, J. F., and Kuhn, D. R. (1999). A role-based access control model and reference implementation within a corporate intranet. *ACM Trans. Inf. Syst. Secur.*, 2:34–64.

Ferrini, R. and Bertino, E. (2009). Supporting RBAC with XACML+OWL. In *Proc. 14th ACM Symposium on Access Control Models And Technologies (SACMAT 09)*, pages 145–154. ACM.

Fisler, K., Krishnamurthi, S., Meyerovich, L. A., and Tschantz, M. C. (2005). Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, New York, NY, USA. ACM.

Florescu, D. and Kossmann, D. (2009). Rethinking cost and performance of database systems. *SIGMOD Rec.*, 38(1):43–48.

Fox, J. (2003). Effect Displays in R for Generalised Linear Models. *Journal of Statistical Software*, 8(15):1–27.

Fox, J. and Weisberg, S. (2011). *An R Companion to Applied Regression*. Sage, Thousand Oaks CA, 2nd edition.

Gasca, M. and Sauer, T. (2000). On the history of multivariate polynomial interpolation. *Journal of Computational and Applied Mathematics*, 122(1–2):23–35. Special Issue: Numerical Analysis in the 20th Century Vol. II: Interpolation and Extrapolation.

Geist, R., Offutt, A. J., and Harris, Jr., F. C. (1992). Estimation and Enhancement of Real-Time Software Reliability Through Mutation Analysis. *IEEE Trans. Comput.*, 41(5):550–558.

Griffin, L., Butler, B., de Leastar, E., Jennings, B., and Botvich, D. (2012). On the performance of access control policy evaluation. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2012)*, pages 25–32. IEEE.

Griffin, L., Ryan, K., de Leastar, E., and Botvich, D. (2011). Scaling Instant Messaging communication services: A comparison of blocking and non-blocking techniques. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 550 –557.

Gryb, O. (2008). XACML Light Reference. http://oleg.internetkeep.net/xacml/doc/XACMLightReference.html. Accessed 2010-04-19.

Haller, P. and Odersky, M. (2009). Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202 – 220. Distributed Computing Techniques.

Hornik, K. (2009). The R FAQ. ISBN 3-900051-08-9.

Hothorn, T. and Everitt, B. S. (2009). *A Handbook of Statistical Analyses Using R, Second Edition*. Chapman and Hall/CRC, 2 edition.

Hu, V. C. (2008). *ACPT: Access Control Policy Tool*. NIST, Gaithersburg, MD.

Hu, V. C., Ferraiolo, D., Kuhn, R., Friedman, A. R., Lang, A. J., Cogdell, M. M., Schnitzer, A., Sandlin, K., Miller, R., and Scarfone, K. (2013). NIST 800-162 (Draft): Guide to Attribute Based Access Control (ABAC) Definition and Considerations. Technical report, NIST, Gaithersburg, MD. Draft: Public comment period: April 22, 2013 through May 31, 2013.

Hu, V. C., Kuhn, D. R., Xie, T., and Hwang, J. (2011). Model Checking for Verification of Mandatory Access Control Models and Properties. *International Journal of Software Engineering and Knowledge Engineering*, 21(01):103–127.

IBM (2012). Unified Communications.

Jajodia, S., Samarati, P., Subrahmanian, V. S., and Bertino, E. (1997). A unified framework for enforcing multiple access control policies. In *Proceedings of the 1997*

ACM SIGMOD international conference on Management of data, SIGMOD '97, pages 474–485, New York, NY, USA. ACM.

Jamin, S., Danzig, P. B., Shenker, S. J., and Zhang, L. (1997). A measurement-based admission control algorithm for integrated service packet networks. *IEEE/ACM Trans. Netw.*, 5(1):56–70.

Jennings, B. (2001). *Network-oriented Load Control for SS.7/IN*. PhD thesis, Dublin City University.

Jennings, B., Arvidsson, Å., and Curran, T. (2001). A token-based strategy for co-ordinated, profit-optimal control of multiple IN resources. In *Teletraffic Engineering in the Internet Era (Proc. 17th Int'l Teletraffic Congress - ITC17)*, volume 1, pages 245–258. Elsevier.

Jin, X., Krishnan, R., and Sandhu, R. (2012). A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In Cuppens-Boulahia, N., Cuppens, F., and Garcia-Alfaro, J., editors, *Proceedings of the 26th Annual IFIP WG 11.3 conference on Data and Applications Security and Privacy*, volume 7371 of *Lecture Notes in Computer Science*, pages 41–55. Springer Berlin Heidelberg.

Johnson, M. (2012). *Toward Usable Access Control for End-users: A Case Study of Facebook Privacy Settings*. PhD thesis, Columbia University.

Kelbert, F. and Pretschner, A. (2012). Towards a policy enforcement infrastructure for distributed usage control. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 119–122, New York, NY, USA. ACM.

Kleinrock, L. (1975). *Queueing Systems, Volume 1: Theory*. Wiley-Interscience.

Kohler, M. and Brucker, A. D. (2010). Caching strategies: An empirical evaluation. In *International Workshop on Security Measurements and Metrics (MetriSec)*, pages 1–8. ACM Press, New York, NY, USA.

Kolovski, V., Hendler, J., and Parsia, B. (2007). Analyzing web access control policies. In *Proc. 16th international conference on World Wide Web (WWW '07)*, pages 677–686. ACM.

Krishnamurthi, S. (2003). The CONTINUE Server (or, How I Administered PADL 2002 and 2003). In Verónica Dahl and Philip Wadler, editor, *Proc. Symposium on the Practical Aspects of Declarative Languages (PADL 03)*, Lecture Notes in Computer Science 2562., pages 2–16. Springer.

Kuhn, D. R., Kacker, R. N., and Lei, Y. (2010). NIST 800-142: Practical Combinatorial Testing. Technical report, NIST, Gaithersburg, MD.

Kuketayev, A. (2005). Oasis XACML 2.0 Conformance tests. Accessed 2010-01-14.

Lampson, B. W. (1974). Protection. *SIGOPS Oper. Syst. Rev.*, 8:18–24.

Lane, D. M. (2015). Online Statistics Education: A Multimedia Course of Study. Accessed Feb 2015.

Lang, B., Zhao, N., Ge, K., and Chen, K. (2008). An xacml policy generating method based on policy view. In *Third International Conference on Pervasive Computing and Applications (ICPCA 2008)*, volume 1, pages 295–301.

Lerner, R. M. (2011). At the forge: Node.JS. *Linux J.*, 205.

Li, N., Hwang, J., and Xie, T. (2008). Multiple-implementation testing for XACML implementations. In *Proc. 2008 workshop on Testing, Analysis, and Verification of Web services and applications (TAV-WEB '08)*, pages 27–33. ACM.

Lin, D., Rao, P., Bertino, E., and Lobo, J. (2007). An approach to evaluate policy similarity. In *Proc. 12th ACM Symposium on Access Control Models And Technologies (SACMAT '07)*, pages 1–10. ACM.

Liu, A. X., Chen, F., Hwang, J., and Xie, T. (2008). Xengine: a fast and scalable XACML policy evaluation engine. In *Proc. ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2008)*, pages 265–276. ACM.

Lodderstedt, T., Basin, D. A., and Doser, J. (2002). SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 426–441, London, UK. Springer-Verlag.

Marouf, S., Shehab, M., Squicciarini, A., and Sundareswaran, S. (2011). Adaptive Reordering and Clustering-Based Framework for Efficient XACML Policy Evaluation. *IEEE Transactions on Services Computing*, 4(4):300–313.

Martin, E. (2006). Automated test generation for access control policies. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 752–753, New York, NY, USA. ACM.

Martin, E., Hwang, J., Xie, T., and Hu, V. (2008). Assessing quality of policy properties in verification of access control policies. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 163–172.

Martin, E., Xie, T., and Yu, T. (2006). Defining and Measuring Policy Coverage in Testing Access Control Policies. In Ning, P., Qing, S., and Li, N., editors, *Information and Communications Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 139–158. Springer Berlin Heidelberg.

Masi, M., Pugliese, R., and Tiezzi, F. (2012). Formalisation and implementation of the xacml access control mechanism. In *Proceedings of the 4th international conference on Engineering Secure Software and Systems*, ESSoS'12, pages 60–74, Berlin, Heidelberg. Springer-Verlag.

Mason, R. L., Gunst, R. F., and Hess, J. L. (2003). *Statistical Design and Analysis of Experiments, with Applications to Engineering and Science*. Wiley-Interscience, 2 edition.

Mazzoleni, P., Bertino, E., Crispo, B., and Sivasubramanian, S. (2006). XACML policy integration algorithms: not to be confused with XACML policy combination

algorithms! In *Proc. Eleventh ACM Symposium on Access Control Models And Technologies (SACMAT '06)*, pages 219–227. ACM.

Miseldine, P. L. (2008). Automated XACML policy reconfiguration for evaluation optimisation. In *Proc. Fourth international workshop on Software Engineering for Secure Systems (SESS '08)*, pages 1–8. ACM.

Moffett, J. and Sloman, M. (1993). Policy hierarchies for distributed systems management. *Selected Areas in Communications, IEEE Journal on*, 11(9):1404–1414.

Moses, T. (2005). eXtensible Access Control Markup Language TC v2.0 (XACML). Accessed 2016-02-10.

Motik, B., Patel-Schneider, P. F., and Parsia, B. (2012). Owl 2 web ontology language: Structural specification and functional-style syntax.

Mouelhi, T. (2015). Policysplitter tool. Accessed 2015-03-10.

Ngo, C. (2014). High performance XACML PDP Engine.

Ngo, C., Makkes, M. X., Demchenko, Y., and de Laat, C. (2013). Multi-data-types Interval Decision Diagrams for XACML Evaluation Engine. In *Proc. 11th International Conference on Privacy, Security and Trust (PST 2013)*.

NIST/NSA (2010). NIST Interagency/Internal Report (NISTIR) - 7657: A Report on the Privilege (Access) Management Workshop. Technical report, NIST, Gaithersburg, MD.

OASIS XACML-TC (2005a). ALL XACML 2.0 documents. Accessed: 2015-03-10.

OASIS XACML-TC (2005b). XACML 2.0 Policy Schema. Accessed 2009-11-25.

OASIS XACML-TC (2005c). XACML 2.0 Request Schema. Accessed 2009-11-25.

OASIS XACML-TC (2014). OASIS XACML TC webpage. Accessed 2014-07-30.

Peña, E. A. and Slate, E. H. (2006). Global Validation of Linear Model Assumptions. *Journal of the American Statistical Association*, 101(473):pp. 341–354.

Peterson, G. and Nair, S. K. (2015). Getting the OWASP Top Ten Right with dynamic authorization. White Paper 84. Accessed 2015-03-10.

Pina Ros, S., Lischka, M., and Gómez Mármol, F. (2012). Graph-based XACML evaluation. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 83–92, New York, NY, USA. ACM.

Proctor, S. (2004). Sun's XACML Implementation - Programmer's Guide for Version 1.2. Accessed 2015-10-03.

R Core Team (2014). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rahmouni, H., Solomonides, T., Mont, M. C., and Shiu, S. (2009). Privacy Compliance in European Healthgrid domains: An Ontology-Based Approach. In

22nd IEEE International Symposium on Computer-Based Medical Systems, 2009. CBMS 2009., pages 1–8.

Ramli, C. D. P. K., Nielson, H. R., and Nielson, F. (2013). XACML 3.0 in Answer Set Programming. In Albert, E., editor, *Logic-Based Program Synthesis and Transformation*, volume 7844 of *Lecture Notes in Computer Science*, pages 89–105. Springer Berlin Heidelberg.

Ramli, C. D. P. K., Nielson, H. R., and Nielson, F. (2014). The Logic of XACML. *Science of Computer Programming*, 83(0):80 – 105. Formal Aspects of Component Software (FACS 2011 selected & extended papers).

Rao, P., Lin, D., Bertino, E., Li, N., and Lobo, J. (2009). An algebra for fine-grained integration of XACML policies. In *SACMAT '09: Proceedings of the 14th ACM Symposium on Access Control Models And Technologies*, pages 63–72, New York, NY, USA. ACM.

Rissanen, E. (2013). eXtensible Access Control Markup Language (XACML) Version 3.0. Accessed: 2015-06-10.

Rissanen, E., Levinson, R., and Lockhart, H. (2010). XACML v3.0 Hierarchical Resource Profile Version 1.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-hierarchical-v1-spec-cd-03-en.pdf. OASIS Standard.

Rissanen, E. and Lockhart, H. (2014). XACML v3.0 Administration and Delegation Profile Version 1.0. OASIS Standard.

Rizvi, S., Mendelzon, A., Sudarshan, S., and Roy, P. (2004). Extending query rewriting techniques for fine-grained access control. In *Proc. 2004 ACM SIGMOD international conference on Management of Data (SIGMOD '04)*, pages 551–562. ACM.

Robinson, I., Webber, J., and Eifrem, E. (2015). *Graph Databases, Second Edition*. O'Reilly Media.

Rochaeli, T. (2009). *An Automated Policy Refinement Process Supported by Expert Knowledge*. PhD thesis, TU Darmstadt.

Rodriguez, M. A. and Neubauer, P. (2010). Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41.

Russello, G., Dong, C., and Dulay, N. (2008). A workflow-based access control framework for e-health applications. In *Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on*, pages 111–120.

Sevelis, V. (2014). Implementation of OASIS XACML 2.0 & 3.0 specification in Java programming language.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y. (2007). Pellet: A practical owl-dl reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51 – 53. Software Engineering and the Semantic Web.

Stepien, B., Matwin, S., and Felty, A. P. (2011). Advantages of a non-technical XACML notation in role-based models. In *Proc. Ninth Annual International Conference on Privacy, Security and Trust (PST 11)*, pages 193–200. IEEE.

Thakkar, D., Hassan, A. E., Hamann, G., and Flora, P. (2008). A framework for measurement based performance modeling. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 55–66, New York, NY, USA. ACM.

Tilkov, S. and Vinoski, S. (2010). Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80 –83.

Tschantz, M. C. and Krishnamurthi, S. (2006). Towards reasonability properties for access-control policy languages. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, SACMAT '06, pages 160–169, New York, NY, USA. ACM.

Turkmen, F. and Crispo, B. (2008). Performance evaluation of XACML PDP implementations. In *Proc. 2008 ACM workshop on Secure Web Services (SWS '08)*, pages 37–44. ACM.

Wand, M. P. and Jones, M. C. (1994). *Kernel Smoothing (Monographs on Statistics and Applied Probability)*. Chapman & Hall/CRC.

Wang, Z. (2010). Enterprise Java XACML. http://code.google.com/p/enterprise-java-xacml/wiki/DevelopmentPlan. Accessed 2010-04-19.

Wolter, C., Schaad, A., and Meinel, C. (2007). Deriving XACML Policies from Business Process Models. In Weske, M., Hacid, M.-S., and Godart, C., editors, *Web Information Systems Engineering – WISE 2007 Workshops*, volume 4832 of *Lecture Notes in Computer Science*, pages 142–153. Springer Berlin / Heidelberg.

Zhang, N., Ryan, M., and Guelev, D. P. (2004). Synthesising verified access control systems in XACML. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 56–65, New York, NY, USA. ACM.

Appendix A

Policy Refinement for Bulk Policy Generation

§ 4.4 describes, at a high level, the steps needed to generate FINE (instance-based) policies, particularly the main steps involved, see § 4.4.1, § 4.4.2, § 4.4.3 and § 4.4.4, in that order.

The present chapter provides more details on how COARSE (attribute-based) policies can be refined to FINE (instance-based) policies and is supplementary to § 4.4.3. In particular, it provides a more formal justification of parts of the algorithm used by PolicyGen when refining a COARSE policy. This chapter considers the general problem of how FINE entities at level i can be derived from COARSE entities at the same level together with FINE entities at the lower level $i-1$. As an example, FINE `CTC`s can be derived from COARSE `CTC`s, when combined with the relevant FINE `TC`s.

A.1 Refining a coarse policy

The policy DSL fragments can be parsed and transformed into the property graph representation quite easily, because the target hierarchies themselves are graphs. Each `TSC` equality clause can be linked to the static domain as follows. Let $c_{i,j}$:= `Type.attribute = value`. Using an index on `Type`, it is possible to find the property graph node which has `attribute = value`, and then to add a relationship between the $c_{i,j}$ node and that node. Staying with the example policy in Figure 4.4, an example clause such as $c_{1,0} := i$ (`Asset.confidentiality = High`) has a `:satisfied_by` relationship with the corresponding static model `lookup` graph node. There could be many of these static model nodes, hence `:satisfied_by` relationships incident on that attribute-level clause.

The next step is to lookup the instances sharing the attribute-value pair specified in the `TC` clause. Staying with the same example of $c_{1,0}$:= (`Asset.confidentiality` =

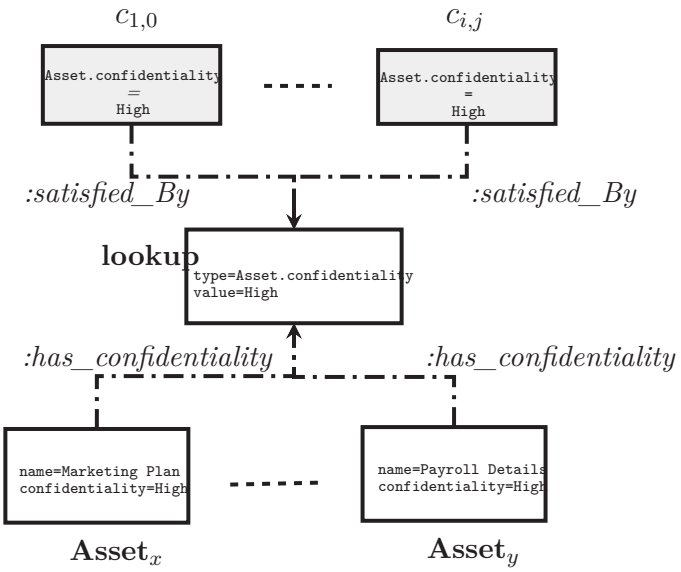


Fig. A.1 Example of how instance-level target subcomponent clauses can be derived from attribute-level target subcomponent clauses

`High`), it is possible to navigate the property graph to lookup the instances of `Asset` where the attribute `Asset.confidentiality` has the value `High`. The lookup process is shown in Figure A.1 As can be seen in the Figure, for a given attribute-level clause $c_{i,j}$[1], there may be many instance-level clauses and this results in instance-level policy sets that are typically much larger than the originating attribute-level policy sets. The derived instance-level clauses represent alternative matches and hence are combined by disjunction (*OR*).

Since *each* attribute-based `TSC` clause $c_{i,j}$ expands to a disjunction of instance-based `TSC` clauses, and the `TSC`s themselves are combined by disjunction to form `TC`s, we have

$$\begin{aligned} c_i &\doteq \bigvee_j c_{i,j} \\ c_{i,j} &\equiv \bigvee_k \tilde{c}_{i,k} \\ \text{then } c_i &\equiv \bigvee_j \left(\bigvee_k \tilde{c}_{i,k} \right) \end{aligned} \tag{A.1}$$

[1] A node with a pale background signifies a policy model entity; a white background indicates a static model entity

Some simplifications may be possible by applying logical identities, e.g., $\forall i, p$, we have $\tilde{c}_{i,p} \doteq \tilde{c}_{i,p} \vee \tilde{c}_{i,p}$ and $\tilde{c}_{i,p} \vee \neg\tilde{c}_{i,p} \equiv \top$.

Each `TSC` $c_{i,j}$ has the form `StaticEntity.Property` = `valu`, where a *Subject* `StaticEntity` is one of `Member` or `organisation`, a *Resource* `StaticEntity` is `Agent` and an *Action* `StaticEntity` is `Action`. Static properties include `name`, `Member.role`, `Asset.confidentiality` and `Action.type`. Then there may exist zero, one or many relationships of the form

$$\{c_{i,j}\,[\texttt{StaticEntity.Property} = \texttt{valu}] \xrightarrow{\text{hasInstance}} \texttt{StaticEntity}\,[(\texttt{Property} = \texttt{valu}]\} \quad \text{(A.2)}$$

where $c_{i,j}$ is a `TSC` and $Entity[a = b]$ is the set of entity instances where property a takes the value b. We define two procedures

enumeration This is where a *coarse* attribute-based `TSC` $c_{i,j}$ becomes *a set of fine* instance-based `TSC`s $\{\tilde{c}^{(k)}_{(i,j)}\}$.

aggregation This is where either an attribute-based or an instance-based target sub entity is combined with its peers to yield its parent entity. Thus `TSC`s can be aggregated to `TC`s, which can be aggregated to `CTC`s, which can be aggregated to a `Target`. Further aggregation (across `Target`s) is complicated and is best left to `PDP` implementations.

Aggregation of attribute-based target sub entities is unlikely to simplify a policy set to any great extent because the terms are distinct and the (human) policy author might be expected to exclude unnecessary terms like A and B in $(A \cap B) \cup C$, since that expression reduces to $C, \forall\{A, B\}$ whenever $A \neq B$.

By contrast, aggregation of instance-based target sub entities can be much more productive because of the identities relating logical operations to set operations, viz.

$$\{\tilde{c}^{(\alpha)}_{i,j}\} \wedge \{\tilde{c}^{(\beta)}_{i,k}\} = \{\tilde{c}^{(\alpha)}_{i,j}\} \cap \{\tilde{c}^{(\beta)}_{i,k}\} \quad \text{(A.3)}$$

$$\{\tilde{c}^{(\alpha)}_{i,j}\} \vee \{\tilde{c}^{(\beta)}_{i,k}\} = \{\tilde{c}^{(\alpha)}_{i,j}\} \cup \{\tilde{c}^{(\beta)}_{i,k}\} \quad \text{(A.4)}$$

This reformulation of the policies has two consequences

1. the instance-level target hierarchy is less deeply nested because `CTC`s comprise instance-level `TC`s only; there is no need for a conjunction (*AND*) of `TSC`s.
2. the matching conditions become more explicit and can be re-interpreted as set membership conditions, e.g., when matching Subjects (or equivalently, Resources, Actions or Environments), does the request's entity instance belong in the set specified in the instance-based target component clause?

Otherwise the structure of the policy set is unchanged, e.g., the same rule nesting, rule effect and combining algorithms apply.

Appendix B

Bulk Request Generation Algorithm

§ 4.5 presented a step-by-step procedure for generating COARSE and FINE requests from COARSE policies and the static domain model. The steps are described informally and visualisation of the evolving graph representation of the request model is used to aid understanding. Many of the details of the algorithms are deferred to the present chapter, which presents a more formal, algorithmic description of RequestGen. This chapter is split into two parts: 1) the algorithms for deriving COARSE request model entities, hence the COARSE requests themselves, from the same COARSE (attribute-based policies) that were used by PolicyGen, and 2) how to recognise and remove duplicate FINE (instance-based) requests that were refined from the COARSE requests.

B.1 Algorithms for generating requests from a policy set

As described in §4.5, it is necessary to generate requests that are semantically consistent with the policies used in the performance experiment. Furthermore, using the policy model as a basis, and not the static model, makes this alignment between policies and requests easier to achieve.

Algorithm B.1 indicates how `TC`s c_i^{context} can be derived from their policy-model counterparts. Step 0 is to derive the request model `TSC`s from the policy model `TSC`s, changing their properties as necessary. The first real step, as shown in Algorithm B.1, is to search the graph database for all policy `TC`s. One of the semantic restrictions to be applied is that, for each `Action TC`, all the `Action TSC`s $c_{i,j}^{\text{context}}$ referenced by it must share the same `Action.type`. That is, actions relating to *different* AssetGroups cannot

Algorithm B.1 Selected procedures used in the algorithm to derive *context* TCs $\{c_i^{\text{context}}\}$ from existing policy TCs $\{c_i^{\text{policy}}\}$

Require: Policy TSCs $\{c_{i,j}^{\text{policy}}\}$; Policy TCs $\{c_i^{\text{policy}}\}$

Ensure: Persist *context* TCs $\{c_i^{\text{context}}\}$ and Relationships $\{c_i^{\text{context}} \xrightarrow{\text{derfivedFrom}} \{c_k^{\text{policy}}\}\}$

procedure DERIVEREQUESTTC(`tct`)
  $\{c_i^{\text{policy}}\} \leftarrow$ FINDSOURCETCSET(`tct`)
  if `tct` = `Action` **then**
    $\{c_i^{\text{context}}\} \leftarrow$ RESTRICTACTIONTC($\{c_i^{\text{policy}}\}$)
  else
    for all $\{c_i^{\text{policy}}\}$ **do**
      $\mathcal{P}(\{c_{i,j}^{\text{policy}}\}) \leftarrow$ DERIVEPOWERSET($\{c_{i,j}^{\text{policy}}\}$)
      for all $\mathcal{P}(\{c_{i,j}^{\text{policy}}\})$ **do**
        $c_i^{\text{context}} \leftarrow$ DERIVEDESTTC(`tct`)
        Accumulate c_i^{context} in $\{c_i^{\text{context}}\}$
        Create $\{c_i^{\text{context}} \xrightarrow{\text{derivedFrom}} \{c_k^{\text{policy}}\}\}$

procedure FINDSOURCETCSET(`tct`)
  Search db for $\{c_i^{\text{policy}}\}$ given `tct`
  for all $\{c_i^{\text{policy}}\}$ **do**
    if $\left|\{c_{i,j}^{\text{policy}} \xleftarrow{\text{hasTSC}} c_i^{\text{policy}}\}\right| > 0$ **then**
      Accumulate c_i^{policy} in $\{c_i^{\text{policy}}\}$
  Persist and **return** $\{c_i^{\text{policy}}\}$

procedure RESTRICTACTIONTC($\{c_i^{\text{policy}}\}$)
  for all $\{c_i^{\text{policy}}\}$ **do**
    for all $\{c_{i,j}^{\text{policy}}\}$ **do**
      Lookup `Action.type` for $\{c_{i,j}^{\text{policy}}\}$
      Accumulate `Action.type` in {`Action.type`}
    if |{`Action.type`}| = 1 **then**
      Accumulate c_i^{policy} in $\{c_i^{\text{policy}}\}$
  Persist and **return** $\{c_i^{\text{policy}}\}$

procedure DERIVEDESTTC(`tct`,$\{c_{i,j}^{\text{policy}}\}$)
  Derive *context* attributes from $\{c_{i,j}^{\text{policy}}\}$
  Persist and **return** $\{c_i^{\text{context}}\}$

participate in the *same* Action TC; if any multi-ActionType TC is found, it should be omitted.

If the context TCs are based solely on the policy TCs, the resulting request set might not have sufficient *diversity* to represent rich, dynamic domains. Thus we consider ways of adding modified TCs. In that regard, the power set of the set of TSCs referenced by a given TC has an interesting interpretation. Each proper subset represents a (sub) *component reduction* and hence a less specific set of conditions. Because the reduced context TC is less specific, it no longer matches the policy TC which was its source, yet it still uses (some of) the same basic terms. Consequently, the derived *reduced target component* TcR shares much of the "meaning" of its source, but is sufficiently different that it no longer matches. By definition, the power set generates *all* possible reduced TCs and hence maximises the available coverage. This feature (i.e., component reduction) is necessary to ensure that some requests match (particularly those that were derived with minimal changes from the policy TCs) and others do not, making for a more realistic mixture of requests and hence a variety of decisions.

To support future analytical requirements, the Many:One IS_DERIVED_FROM relationship links each context TC to its source policy TC. The context TCs are saved in the database and hence are available for aggregation.

Algorithm B.2 describes how context CTCs (C^{context}) are derived from the combination of policy CTCs and the context TCs (c_i^{context}) derived according to Algorithm B.1. Many of the procedures used to generate context CTCs are similar to those used to generate context TCs. In particular, the context CTCs are based on the policy CTCs having a mimimum of one policy TC. An analogous *component reduction* procedure is used to generate a greater diversity of context CTCs. Both the CTCs and the derivedFrom relationships are stored in the graph database as before.

However, there are differences in relation to what semantic restrictions apply. The first semantic difference results from the fact that, in XACML 2.0 requests, the Subject and Resource parts of the requests can each be a set of CTCs, but the Action and Environment parts can each be a set of TCs. Thus there is no need to generate Action and Environment request CTCs.

Consequent upon that, the semantic restriction regarding Resource-Action combinations should be applied to Resource CTCs only. Again, there is a restriction to ensure that only one AssetGroup is associated with each resource CTC, and also that

Algorithm B.2 Selected procedures used in the algorithm to derive *context* collected target components $\{C^{\text{context}}\}$ from existing policy collected target components $\{C^{\text{policy}}\}$

Require: Policy target components $\{c_i^{\text{policy}}\}$; Policy collected target components $\{C^{\text{policy}}\}$

Ensure: Persist *context* collected target components $\{C^{\text{context}}\}$ and $\{\tilde{C}^{\text{context}} \xrightarrow{\text{derfivedFrom}} \{C^{\text{policy}}\}\}$

procedure DERIVEREQUESTCTC(`tct`)
  $\{c_i^{\text{policy}}\} \leftarrow$ FINDSOURCECTCSET(`tct`)
  for all $\tilde{C}^{\text{policy}} \in \{C^{\text{policy}}\}$ **do**
    $\mathcal{P}(\{c_i^{\text{policy}}\}) \leftarrow$ DERIVEPOWERSET($\{c_i^{\text{policy}}\}$)
    for all $\tilde{c}^{\text{policy}} \in \mathcal{P}(\{c_i^{\text{policy}}\})$ **do**
      if $\left|\mathcal{P}(\{c_i^{\text{policy}}\})\right| > 0$ **then**
        $C^{\text{context}} \leftarrow$ DERIVEDESTCTC(`tct`, $\tilde{c}^{\text{policy}}$)
      Accumulate C^{context} in $\{C^{\text{context}}\}$
  if `tct` = `Resource` **then**
    $\{C_{\text{valid}}^{\text{context}}\} \leftarrow$ RESTRICTRESOURCECTC($\{C^{\text{context}}\}$)

procedure FINDSOURCECTCSET(`tct`)
  Search db for $\{C^{\text{policy}}\}$ given `tct`
  for all $\tilde{C}^{\text{policy}} \in \{C^{\text{policy}}\}$ **do**
    if $\left|\{\tilde{C}^{\text{policy}} \xleftarrow{\text{hasTC}} C_i^{\text{policy}}\}\right| > 0$ **then**
      Accumulate $\tilde{C}^{\text{policy}}$ in $\{C^{\text{policy}}\}$
  Persist and **return** $\{C^{\text{policy}}\}$

procedure DERIVEDESTCTC(`tct`,$\{c_i^{\text{policy}}\}$)
  Derive *context* attributes from $\{c_i^{\text{policy}}\}$
  Persist and **return** $\{C^{\text{context}}\}$

procedure RESTRICTRESOURCECTC($\{C^{\text{context}}\}$)
  Initialise $\{C_{\text{valid}}^{\text{context}}\} \leftarrow \{\}$
  for $\tilde{C}^{\text{context}} \in \{C^{\text{context}}\}$ **do**
    if `attr`($\tilde{C}^{\text{context}}$) = `Asset.type` **then**
      for $c_{i,j}^{\text{context}} \in \tilde{C}^{\text{context}}$ **do**
        Find an asset a where `Asset.type`(a) = `valu`($c_{i,j}^{\text{context}}$)
        Find `group`($a_{i,j}$) using $a_{i,j} \xrightarrow{\text{hasGroup}}$ `group`($a_{i,j}$)
        Accumulate $\{a_{i,j}^{\text{group}}\}$; $a_{i,j}^{\text{group}} \doteq$ `group`($a_{i,j}$)
      if $\left|\{a_{i,j}^{\text{group}}\}\right| = 1$ **then**
        Accumulate $\tilde{C}^{\text{context}}$ in $\{C_{\text{valid}}^{\text{context}}\}$
  Persist and **return** $\{C_{\text{valid}}^{\text{context}}\}$.

each Resource CTC should be labeled with its associated Asset (which is semantically equivalent to the ActionType of any Action TC that can be combined with this Resource).

Algorithm B.3 describes how the context Action and Environment TCs and the context Subject and Resource CTCs can be assembled to form the generated requests. The first step is to collect the relevant CTCs and TCs from the graph database. Again, any "empty" Subject and Resource CTCs, or Action and Environment TCs are excluded since they are not representative of most domains, where requests tend to be quite specific.

The requests are assembled using a full enumeration of all possible combinations of Subject, Resource, Action and Environment as described above, *less* any semantic restrictions, particularly those affecting Resource-Action combinations.

The Resource CTCs and Action TCs need to be classified by AssetGroup (equivalently: ActionType). The database lookup algorithm is similar to that followed in Algorithms B.2 and B.1, respectively. However, there is a complication assigning a single ActionType to a Resource CTC if, as is often the case, that CTC has at least one TC that does not resolve to a clause involving a single Asset. As an example, a clause requiring that Asset.confidentiality = High could apply to Asset instances having either Communication or Document AssetGroup. If such a clause were encountered as one of the constituent TSCs in a TC, but at least one of the other constituent TSCs could be classified with an AssetGroup, the overall TC can be assigned that AssetGroup successfully because the TSCs are ANDed together. However, the presence of even one unclassified TC constituent is a problem for a CTC, because TCs are combined by disjunction (OR) and hence a single AssetGroup cannot be assigned to the overall CTC. One solution to this problem is to expand the set of Resource CTCs. This is done by combining each unclassified resource CTC C_0^{R} with a classified resource CTC C_p^{R}, by ANDing them at the TSC level and then aggregating up to the CTC level. The resulting $\tilde{C}_{\mathrm{R}}^{+}$ CTC inherits its classification from the C_p^{R} classified CTC.

The number of generated requests can be computed as follows. Let $n_{\mathrm{S}} \doteq |\{C\}_{\mathrm{S}}|$ and $n_{\mathrm{E}} \doteq |\{c\}_{\mathrm{E}}|$. Also let $n_{\mathrm{R}}^{(i)} \doteq \left|\{C\}_{\mathrm{R}}^{(t_i)}\right|$ and $n_{\mathrm{A}}^{(i)} \doteq \left|\{c\}_{\mathrm{A}}^{(t_i)}\right|$ where t_i is the i^{th} Asset (equivalently, ActionType).

Then the number of generated requests, n is

$$n = n_{\mathrm{S}} \left(\sum_i n_{\mathrm{R}}^{(i)} n_{\mathrm{A}}^{(i)} \right) n_{\mathrm{E}} \tag{B.1}$$

Algorithm B.3 Selected procedures used to assemble requests from (collected) target components $\{c_i^{\text{context}}\}$ and $\{C^{\text{context}}\}$

procedure ASSEMBLEREQUESTS()
  $\{\{C^{\text{S}}\},\{C^{\text{R}}\}\} \leftarrow$ GETSRCTC()
  $\{\{c_i^{\text{A}}\},\{c_i^{\text{E}}\}\} \leftarrow$ GETAETC()
  $\{\hat{C}\}_{\text{R}} \leftarrow$ CLASSIFYRESOURCECTC($\{C\}_{\text{R}}$)
  $\{\hat{c_i}\}_{\text{A}} \leftarrow$ CLASSIFYACTIONTC($\{c_i\}_{\text{A}}$)
  $\{\tilde{C}\}_{\text{R}}^{+} \leftarrow$ EXPANDRESOURCECTC($\{\hat{C}\}_{\text{R}}$)
  for $c_i^{\text{A}} \in \{C_i\}_{\text{A}}$ **do**
    $a_{\texttt{type}} \leftarrow \texttt{type}(C_i^{\text{A}})$
    for $\hat{C}^{\text{R}} \in \{\tilde{C}\}_{\text{R}}^{+}(a_{\texttt{type}})$ **do**
      for $C^{\text{S}} \in \{C\}_{\text{S}}$ **do**
        for $c_i^{\text{E}} \in \{C\}_{\text{E}}$ **do**
          Persist Request$(C^{\text{S}}, \hat{C}^{\text{R}}, c_i^{\text{A}}, c_i^{\text{E}})$

procedure GETSRCTC()
  Search db for $\{C^{\text{S}}\}$ and $\{C^{\text{R}}\}$
  for $\texttt{tct} \in \{S, R\}$ **do**
    for $C^{\texttt{tct}} \in \{C^{\texttt{tct}}\}$ **do**
      if $\left|\{C^{\texttt{tct}} \xrightarrow{\text{hasTc}} c_i^{\texttt{tct}}\}\right| < 1$ **then**
        Omit $C^{\texttt{tct}}$
  return $\{\{C^{\text{S}}\},\{C^{\text{R}}\}\}$

procedure GETAETC()
  Search db for $\{c_i^{\text{A}}\}$ and $\{c_i^{\text{E}}\}$
  for $\texttt{tct} \in \{A, E\}$ **do**
    for $c_i^{\texttt{tct}} \in \{c_i^{\texttt{tct}}\}$ **do**
      if $\left|\{c_i^{\texttt{tct}} \xrightarrow{\text{hasTsc}} c_{i,j}^{\texttt{tct}}\}\right| < 1$ **then**
        Omit $c_i^{\texttt{tct}}$
  return $\{\{c_i^{\text{A}}\},\{c_i^{\text{E}}\}\}$

procedure CTCANDCTC($C_{(1)}^{\text{R}}, C_{(2)}^{\text{R}}$)
  for $c_{i,(1)}^{\text{R}} \in \{c_{k,(1)}^{\text{R}} \mid C_{(1)}^{\text{R}} \xrightarrow{\text{hasTC}} c_{k,(1)}^{\text{R}}\}$ **do**
    for $c_{i,(2)}^{\text{R}} \in \{c_{k,(2)}^{\text{R}} \mid C_{(2)}^{\text{R}} \xrightarrow{\text{hasTC}} c_{k,(2)}^{\text{R}}\}$ **do**
      $\{\tilde{c}_{i,j}\} \leftarrow \{c_{i,k,(1)}^{\text{R}} \mid c_{i,(1)}^{\text{R}} \xrightarrow{\text{hasTSC}} c_{i,k,(1)}^{\text{R}}\} \cup \{c_{i,k,(2)}^{\text{R}} \mid c_{i,(2)}^{\text{R}} \xrightarrow{\text{hasTSC}} c_{i,k,(2)}^{\text{R}}\}$
      Create new $\tilde{c}_i$ where $\tilde{c}_i \xrightarrow{\text{hasTSC}} \{\tilde{c}_{i,j}\}$
  Create new $\tilde{C}$ where $\tilde{C} \xrightarrow{\text{hasTC}} \{\tilde{c}_i\}$
  Persist and **return** $\tilde{C}$

procedure CLASSIFYRESOURCECTC($\{C^{\text{R}}\}$)
  for $C^{\text{R}} \in \{C\}^{\text{R}}$ **do**
    for $c_i^{\text{R}} \in \{c_k^{\text{R}} \mid C^{\text{R}} \xrightarrow{\text{hasTC}} c_k^{\text{R}}\}$ **do**
      for $c_{i,j}^{\text{R}} \in \{c_{i,k}^{\text{R}} \mid c_i^{\text{R}} \xrightarrow{\text{hasTsC}} c_{i,k}^{\text{R}}\}$ **do**
        Accumulate $r_{\texttt{type}} = \{\texttt{type}(c_{i,j}^{\text{R}})\}$
    if $\left|r_{\texttt{type}}\right| = 0$ **then**
      $\texttt{ActionType}(C^{\text{R}}) \leftarrow$ NotAssignedYet
    else
      Search db for Asset A with $\texttt{type}(A) = r_{\texttt{type}}$
      $\texttt{ActionType}(C^{\text{R}}) \leftarrow \texttt{Asset}(A)$
  return $\{\{C^{\text{R}}\},\{\texttt{ActionType}(C^{\text{R}})\}\}$

procedure CLASSIFYACTIONTC($\{c_i^{\text{A}}\}$)
  for $c_i^{\text{A}} \in \{c_i\}^{\text{A}}$ **do**
    for $c_{i,j}^{\text{A}} \in \{c_{i,k}^{\text{A}} \mid c_i^{\text{A}} \xrightarrow{\text{hasTSC}} c_{i,k}^{\text{A}}\}$ **do**
      Accumulate $a_{\texttt{type}} = \{\texttt{type}(c_{i,j}^{\text{A}})\}$
    Search db for Action A with $\texttt{type}(A) = A_{\texttt{type}}$
    $\texttt{ActionType}(C^{\text{A}}) \leftarrow \texttt{ActionType}(A)$
  return $\{\{c_i^{\text{A}}\},\{\texttt{ActionType}(c_i^{\text{A}})\}\}$

procedure EXPANDRESOURCECTC($\{\hat{C}^{\text{resource}}\}$)
  Pass 1:
  for $C^{\text{R}} \in \{C^{\text{R}}\}$ **do**
    Derive the relation that partitions $\{C^{\text{R}}\}$ as $\{\texttt{ActionType}_p \longrightarrow \{C^{\text{R}}\}_p\}$

  Pass 2:
  for $C_0^{\text{R}} \in \{\text{NotAssignedYet} \longrightarrow \{C^{\text{R}}\}\}$ **do**
    for $\texttt{ActionType} \in \{\texttt{ActionType}\}$ **do**
      for $C_p^{\text{R}} \in \{\texttt{ActionType} \longrightarrow \{C^{\text{R}}\}\}$ **do**
        if $\texttt{ActionType}_p \neq$ NotAssignedYet **then**
          Accumulate CTCANDCTC($C_p^{\text{R}}, C_0^{\text{R}}$) in $\tilde{C}_{\text{R}}^{+}$

  Pass 3:
  for $\texttt{ActionType} \in \{\texttt{ActionType}\}$ **do**
    for $C_p^{\text{R}} \in \{\texttt{ActionType} \longrightarrow \{C^{\text{R}}\}\}$ **do**
      if $\texttt{ActionType}_p \neq$ NotAssignedYet **then**
        Accumulate C_p^{R} in $\tilde{C}_{\text{R}}^{+}$

The following settings: `extraTcType` = `full` and `useTscReduction` = `useTcReduction` = True tend to result in the most requests being generated from a given policy set. In the case of the policy set in Listing 4.4 this evaluates to

$$n = 11 \times (2 \times 15 + 2 \times 5) \times 1 = 440$$

requests.

B.2 Removing duplicate instance-based requests

The `CTC`s (`Subject` and `Resource`) and `TC`s (`Action` and `Environment`) of each generated instance-based request are obtained by querying the property graph. Sometimes different attribute-based queries return the same instances, particularly when a limit is imposed on the number of `TC`s assigned to each `CTC` in each request.

Let

$$C_i \to \tilde{c}_\alpha \cup \tilde{c}_\beta \cup \tilde{c}_\gamma \ldots \tag{B.2}$$

$$C_j \to \tilde{c}_\alpha \cup \tilde{c}_\beta \cup \tilde{c}_\delta \ldots \tag{B.3}$$

$$\tag{B.4}$$

Then if C_i and C_j are both limited to the first two terms, the resulting restricted `CTC`s are identical ($C_i^{(2)} = C_j^{(2)}$) and if this occurs for all `Subject`, `Resource`, `Action` and `Environment` parts of each request, the resulting requests are equal: $R_i^{(2)} = R_j^{(2)}$.

In such cases, treating $R_i^{(2)}$ and $R_j^{(2)}$ as distinct for all i and j is misleading and can lead to spurious peaks in the service time distribution. In turn, this could lead to invalid statistical interpretation of those service times.

Consequently, `DomainManager` looks for groups of such duplicate instance-based requests. For each duplicate group, it

1. records the `id` of the source attribute-based request used to generate each request in a duplicate instance-based request group,
2. deletes all but one request (i.e., the request with the lowest `id`) in that duplicate instance-based request group.

By following this procedure, all instance-based requests submitted for performance measurement are distinct, but it is still possible to link those measured service times back to the attribute-based requests which were their source. Moreover, if any peaks arise in the service time distribution, it is caused by more subtle semantic interaction between a group of instance-based requests (whose membership is not known until *after* they have been submitted to the `PDP`) and the policy set used by the access control system.

List of Acronyms

Symbols

3NF

Third Normal Form. 118

A

ABAC

Attribute-Based Access Control. *See Glossary:* Attribute-Based Access Control, 27, 38, 39, 46, 47, 51, 271

ACPT

Access Control Policy Tool. *See Glossary:* Access Control Policy Tool, 11, 12, 53, 55, 271

ACTS

Automated Combinatorial Testing for Software. *See Glossary:* Automated Combinatorial Testing for Software, 12, 271

AIC

Akaike Information Criterion. *See Glossary:* Akaike Information Criterion, 214, 271

ALFA

Axiomatics Language For Authorization. *See Glossary:* Axiomatics Language For Authorization, 51, 52, 53, 271

ANCOVA

ANalysis of COVAriance. 214

ANOM

ANalysis Of Means. 184

ANOVA

ANalysis Of VAriance. 184, 214

ATLAS

A TooL for dimensioning Access control Systems. *See Glossary:* A TooL for dimensioning Access control Systems, 11, 175, 181, 184, 187, 188, 205, 207, 213, 227, 230, 245, 246, 249, 272

B

BDD

Binary Decision Diagram. *See Glossary:* Binary Decision Diagram, 41, 44, 242, 272, *see*

BoD

Binding of Duties. *See Glossary:* Binding of Duties, 26, 38, 45, 272

BPEL

Business Process Execution Language. *See Glossary:* Business Process Execution Language, 45, 272

BPMN

Business Process Model and Notation. *See Glossary:* Business Process Model and Notation, 45, 272

BYOD

Bring Your Own Device. *See Glossary:* Bring Your Own Device, 24, 272

C

CRUD

Create, Read, Update, Delete. 11, 12

CTC

Collected Target Component. *See Glossary:* Collected Target Component, 138, 141, 142, 145, 149, 152, 155, 156, 160, 164, 165, 167, 168, 169, 170, 172, 173, 174, 175, 259, 261, 265, 267, 269, 272

D

DAG

Directed Acyclic Graph. 148

DL

Description Logic. *See Glossary:* Description Logic, 40, 47, 273

DNF

Disjunctive Normal Form. *See Glossary:* Disjunctive Normal Form, 273

DomainManager

Domain model Manager. *See Glossary:* Domain model Manager, 10, 11, 12, 16, 17, 112, 113, 118, 120, 121, 127, 128, 136, 144, 145, 149, 151, 155, 158, 159, 160, 161, 164, 167, 168, 169, 170, 172, 173, 174, 175, 177, 181, 182, 183, 184, 189, 191, 206, 223, 225, 234, 235, 241, 242, 243, 245, 247, 248, 249, 269, 273

DSL

Domain-Specific Language. 51, 131, 152

E

EBNF

Extended Backus-Naur form. 158

ETL

Extract, Transform, Load. *See Glossary:* Extract, Transform, Load, 273

F

FIFO

First In, First Out. 10, 63, 73, 82

G

GMP

Good Manufacturing Practices. *See Glossary:* Good Manufacturing Practices, 273

H

HTTP

HyperText Transfer Protocol. 11

I

IAM

Identity and Access Management. *See Glossary:* Identity and Access Management, 22, 274

J

JAXB

Java And Xml Binding. 158, 239

JSON

JavaScript Object Notation. 37, 45

JVM

Java Virtual Machine. *See Glossary:* Java Virtual Machine, 7, 9, 62, 236, 274

M

MTBDD

Multi-Terminal Binary Decision Diagram. *See Glossary:* Multi-Terminal Binary Decision Diagram, 41, 45, 47, 245, 274

N

NoSQL

Not-only SQL. 245

P

PAP

Policy Administration Point. 35

PARPACS

Performance Analysis, Reporting and Prediction of Access Control Systems. *See Glossary:* Performance Analysis, Reporting and Prediction of Access Control Systems, 10, 11, 12, 17, 113, 180, 184, 185, 186, 187, 205, 206, 208, 210, 225, 227, 230, 235, 247, 248, 249, 275

PDP

Policy Decision Point. xii, 7, 8, 9, 12, 13, 15, 24, 27, 28, 29, 31, 32, 34, 37, 39, 41, 42, 44, 45, 47, 48, 49, 52, 54, 57, 59, 61, 63, 64, 65, 66, 67, 69, 71, 72, 73, 75, 76, 79, 80, 81, 82, 83, 84, 86, 87, 91, 93, 94, 95, 100, 108, 109, 136, 137, 175, 201, 209, 211, 231, 232, 233, 236, 237, 238, 243, 244, 245, 261, 269

PEP

Policy Execution Point. 29, 31, 32, 48, 49, 54, 65, 67, 79, 80, 81, 82, 93, 107, 116, 137

PIP

Policy Information Point. 27, 46, 116, 244

PRP

Policy Retrieval Point. 29, 93

R

RBAC

Role-Based Access Control. *See Glossary:* Role-Based Access Control, 38, 39, 46, 47, 275

S

SoD

Separation of Duties. *See Glossary:* Separation of Duties, 20, 25, 27, 39, 45, 46, 275

STACS

Scalability Testbed for Access Control Systems. *See Glossary:* Scalability Testbed for Access Control Systems, vii, 10, 11, 12, 15, 16, 17, 49, 62, 66, 67, 66, 67, 71, 73, 76, 79, 82, 84, 90, 93, 96, 97, 107, 108, 109, 110, 111, 112, 113, 184, 185, 191, 206, 208, 210, 213, 224, 227, 231, 232, 233, 235, 242, 245, 247, 248, 249, 276

SVM

support vector machine. *See Glossary:* support vector machine, 276

T

TC

Target Component. *See Glossary:* Target Component, xvii, 138, 139, 141, 142, 145, 148, 149, 151, 152, 155, 156, 159, 160, 161, 162, 164, 165, 167, 168, 170, 172, 170, 172, 173, 175, 178, 211, 223, 248, 259, 260, 261, 263, 265, 267, 269, 276

TSC

Target SubComponent. *See Glossary:* Target SubComponent, 138, 139, 141, 142, 145, 148, 149, 150, 155, 158, 160, 161, 162, 164, 165, 168, 172, 173, 175, 178, 183, 211, 248, 259, 260, 261, 263, 265, 267, 276

X

XACML

eXtensible Access Control Markup Language. 14, 27, 28, 29, 31, 33, 35, 37, 38, 39, 41, 42, 44, 45, 46, 47, 50, 51, 52, 53, 54, 57, 65, 67, 73, 80, 82, 86, 94, 95, 115, 116, 136, 233

xjc

Xml to Java Compiler. 158

XSD

Xml Schema Document. 158

XTC

XACML Testing Client. 93

XTS

XACML Testing Server. 67, 79, 93, 107

List of Symbols

λ

mean arrival rate, used in queueing theory. 73

$\mu^{(s)}$

mean exit (service) rate, used in queueing theory. 73

ρ

ratio of mean arrival rate to mean service rate, used in queueing theory. 73

Glossary

A

Access Control Policy Tool

Suite of tools provided by NIST and colleagues which can be used to generate a policy and related requests.. 11, 271

AccessNode

Prototypical property graph node, with basic properties and methods that are available to all the typed domain model nodes that extend AccessNode. 128

AccessRelationship

Prototypical property graph relationship joining two AccessNodes, with basic properties and methods that are available to all the typed domain model relationships that extend AccessRelationship. 128

Action

Model entity (static, policy and request) that represents the operation that an Agent wishes to apply to an Asset. An example would be: Read (a document). 26, 120, 121, 127, 134, 135, 138, 141, 261, 263, 265, 267, 269

Action.type

Model property (static, policy and request) that represents the type of operation that an Agent wishes to apply to an Asset. It is a property of Action and each instance is associated with an ActionType node. 261, 263

ActionType

Model entity (static, policy and request) that represents the type of operation that an Agent wishes to apply to an Asset. Examples include: Document and Task: Read a Document and Delegate a Task. 135, 265, 267

Actor

Formal concurrency model, where Actors are independent computational units. On receipt of a message, an Actor can send a finite number of messages to other

Actors and/or create a finite number of Actors and/or decide its won behaviour when it receives its next message. 36

Agent

Static model entity that represents the actor performing an action. Member and Organisation entities are examples of Agent. 120, 121, 135, 261

Akaike Information Criterion

Akaike's Information Criterion, a measure of model fit that seeks to balance two objectives: closeness to the data (associated with small residuals) with the ability to generalise to other data sets of the same kind (associated with parsimony: small number of model terms).. 214, 271

antlr

a tool for generating parsers for a given context-free grammar, such as might be used to convert from one language to another.. 158

Asset

Static model entity that represents the object that an Agent to which an Agent wishes to apply an Action. An example would be: MarketingPlan (to read). 120, 121, 127, 130, 134, 135, 259, 267

Asset

Static model entity that represents a group of Assets. An example would be: Documents, Group Chats. 135, 167, 265, 267

Asset.confidentiality

Static model property that represents the confidentiality attribute of an Asset– -ypically affects read-type operations. Examples include: Low, Medium. 130, 139, 140, 141, 142, 259, 261, 267

Asset.integrity

Static model property that represents the integrity attribute of an Asset - typically affects write-type operations. Examples include: Low, Medium. 130

Asset.type

Static model property that represents the type attribute of an Asset. Examples include: PriceList, CommunicationChannel. 130, 140, 165, 265

A TooL for dimensioning Access control Systems

A set of applications: DomainManager, STACS and PARPACS, together with essential testbed components like databases, that work together to enable researchers to investigate the performance of access control systems. 11, 272

adminssion control

In telecoms, admission control is a mechanism used to check whether there are sufficient resources for a communication event to occur. More generally, admission control can be used to prevent a queue growing beyond a specified size.. 10

Attribute-Based Access Control

Access control specification where policies are specified in terms of the attributes of the entities to which the rules are applied. Any entry property can be an attribute.. 27, 271

Automated Combinatorial Testing for Software

Suite of tools provided by NIST and colleagues which searches for faults in software by considering single factor and low-degree interaction factor faults, as this is usually sufficient.. 12, 271

Axiomatics

Axiomatics (http://www.axiomatics.com) is the leading independent supplier of ABAC solutions. Its headquarters are in Stockholm and it is very active in the OASIS XACML TC.. *see* ABAC, OASIS & XACML, 27, 28, 51

Axiomatics Language For Authorization

A language developed by Axiomatics which can be compiled to XACML but has much less verbose syntax.. 51, 271

B

Binary Decision Diagram

Directed Acyclic Graph used to represent, in compressed form, an arbitrarily complex Boolean function. 41, 272

Binding of Duties

Constraint that enforces the requirement that if person S_1 works on task t_i, he/she must also work on task t_j where $i \neq j$ in a given workflow.. 26, 272

Bring Your Own Device

Growing prevalence of employees using their personal electronic devices (phones and tablets) to access (privileged) resources in their employer's network. 24, 272

Business Process Model and Notation

Graphical notation with standard shapes and connectors to represent business process models. BPMN 2 introduced execution semantics to supplement the existing diagram elements.. 45, 272

Business Process Execution Language

OASIS standard language to express how services should be orchestrated, to enable both "programming in the large" and "programming in the small" of large systems of loosely-coupled components. 45, 272

C

Collected Target Component

Policy and Request domain entity, comprising a disjunction of Target Components. 138, 141, 272

Condition

Policy model entity corresponding to the Condition element of a XACML Rule, Policy or PolicySet. It is a logical condition that is applied only of the associated Target conditions are true. 136, 137

D

Description Logic

A Description Logic (DL) models concepts, roles and individuals, and their relationships, using taxonomies and axioms.. 40, 273

Disjunctive Normal Form

Logical expression written as a disjunction of the conjunction of simple logical conditions.. 273

distribution, hyperexponential

statistical distribution $f(x); f(x) > 0 \forall x; \int_{-\infty}^{\infty} f(x)dx = 1$ consisting of the weighted sum of exponential distributions with non-coincident "centres". 73

Domain model Manager

Application that uses static, policy and context specifications to populate a property graph, that infers instance policies from template policies, and which can export policies and requests in standard policy languages. 10, 273

E

Environment

Policy and Request model entity corresponding to the Environment element of a XACML 2.0 Target or Request element, used for general/contextual conditions. It is an example of a CTC in a policy Target and a TC in a request. 138, 141, 265, 267, 269

Ethical Wall

Previously known as "Chinese Wall": communications boundary between staff in an enterprise, designed to prevent the flow of information (often in one directon) that might lead to the receiver having a conflict of interest. 21

Extract, Transform, Load

Data integration pattern used in data warehousing and related domains to 1) take data from a source system (capturing its context as metadata), 2) transform it according to metadata mapping rules and 3) to store the transformed data in a target system. 273

extraTcType

A template policy might not always specify certain semantic restrictions that apply to the static domain, e.g., that certain actionTypes apply only to certain assetGroups. These conditions can be added to the template policy rules in the form of additional TargetComponents. This term relates to the scope of the extra TCs: either minimal (just enough) or full (overspecified).. 209

G

Good Manufacturing Practices

Quality standards/guidelines for to ensure that food and pharmaceuticals are safe for human use - they include extensive requirements for data integrity and traceability. 273

Granularity

Attribute-based policies, such as those specified in template policies are defined to have COARSE granularity. Instance-based policies are defined to have FINE granularity, because their conditions are more specific. 155, 156

I

Identity and Access Management

Identity and Access Management is the security discipline that enables the right individuals to access the right resources at the right times for the right reasons. 22, 274

J

Java Virtual Machine

Abstract machine that enables the host to run Java (and other languages that compile to a compatible bytecode) programs.. 7, 9, 62, 274

M

Member

Static model entity that represents an individual Agent that wishes to perform some action to an Asset. An example would be: Manager (to read a document). 120, 127, 134, 135, 261

Member.function

Static model property that represents the function attribute of Member - typically his/her affiliation. Examples include: Sales, Finance. 134, 135

Member.role

Static model property that represents the role attribute of a Member - typically his/her job title in a given context. Examples include: Developer, Clerk. 134, 135, 261

MemberGroup

Static model entity that represents a group of Members, less ormal than an Organisation, and easier to be setup, dissolved and changed. Example: Project team. 120, 134

model formula

Compact symbolic formula introduced in Chapter 2 of Chambers and Hastie (1991). Example: $y \sim x_1 * x_2$, where a response y is to modeled as a linear statistical model in terms of two predictors x_1 and x_2 and their interaction $x_1 : x_2$. 186, 187, 205

multiscale

Numerical data obtained from experiments has a granularity that depends on its context. Generally, less context implies more summarisation implies coarser granularity. A phenomenon has multiscale characteristics if it can be found at different granularity levels. 230

Multi-Terminal Binary Decision Diagram

Directed Acyclic Graph used to represent, in compressed form, an arbitrarily complex function over a discrete set. It simplifies to a hen that set has two elements.. 41, 245, 274

N

NP-hard

from wikipedia: class of problems that are "at least as hard as the hardest problems in NP", where NP is Non-deterministic Polynomial time. 44

O

OASIS

OASIS is a not-for-profit consortium that brings people together to agree on intelligent ways to exchange information over the Internet and within their organizations. 27, 29, 37, 47

Obligation

A XACML policy can *optionally* include obligations that constitute actions to be taken at the instigation of the PEP, depending on the decision taken by the PDP. Examples include logging the fact that access was granted.. 137

observation matrix

Matrix $A = A(\mathbf{x})$ defined by $\mathbf{y} = A\mathbf{p} + \varepsilon$, where $\mathbf{y}$ is the dependent variable, $\mathbf{p}$ is the vector of model parameters (to be estimated) and ε is the vector of model residuals, centred on 0. If $\mathbf{y}$ has a nonlinear dependence on $\mathbf{p}$, $A = A(\mathbf{x}; \mathbf{p})$ is the (linearised) Jacobian of that functional relationship.. 215

OPNET

Discrete Event simulation package. *see* DL, 15, 72, 84, 86, 91, 245

organisation

Static model entity that represents an Agent comprising a persistent group of individuals that wishes to perform some action to an Asset. An example would be: Bank (to join a pension scheme). 120, 127, 134, 135, 261

OWL

Web Ontology Language, standardised by W3C mainly for use in the semantic web/linked data community, but can also be used to specify ontologies more generally.. 39

OWL-DL

Web Ontology Language corresponding to DL, designed to provide the maximum expressiveness possible while retaining computational completeness, decidability, and the availability of practical reasoning algorithms. *see* DL, 40

P

njspdp

Node.js PDP: a *full* XACML 2.0-compliant PDP developed by Fan Zhang in Javascript, using XML-encoded policies and either XML- or JSON-encoded requests, deployed in a Node.js instance. 242

sne-xacml

Open source XACML 3.0 PDP with a focus on high-performance that implements the Multi-data-types Interval Decision Diagrams presented in Ngo, Demchenko and de Laat (2013). 242

xacml4j

Open source XACML PDP that supports both XACML 2.0 and XACML 3.0 and the JSON profile for the latter. 242

ATTxacml

Open source XACML 3.0 implementation developed by AT&T. It includes a PDP, PAP and other components. The PDP component has been selected to form the reference PDP implementation used by the Apache OpenAz incubation project begun by Hal Lockhart, co-chair of the OASIS XACML TC. 242

Balana

Open source XACML 3.0 PDP implementation, derived from the SunXACML reference XACML 2.0 PDP, and part of the WSO2 Identity and Access Management suite. 242

njsrpdp

Node.js with redis PDP: an *early prototype* PDP developed by Leigh Griffin in Javascript, using JSON-encoded policies and requests, deployed in a Node.js instance. xii, xv, 15, 93, 94, 95, 97, 98, 100, 103, 105, 107, 233, 238, 242

EnterpriseXACML PDP

XACML PDP that passes the XACML 2.0 conformance test suite and is claimed to perform better than SunXACML PDP because it indexes the policies and so does not have to scan the full policy set, for each request, to find a match. xii, 49, 69, 71, 86, 87, 88, 91, 95, 96, 97, 98, 100, 105, 107, 238

SunXACML PDP

XACML PDP developed in Java originally by Seth Proctor and colleagues at Sun Microsystems, offered to the community in open source form as the reference PDP for XACML 1.0 and 2.0. Work on the original version 1.4 stopped with subversion revision 137. Version 2.0 was re-engineered to use the Spring framework for greater flexibility. xii, 12, 15, 34, 42, 43, 44, 45, 48, 49, 66, 67, 69, 68, 69, 71, 72, 86, 87, 88, 91, 95, 96, 97, 98, 100, 105, 107, 114, 211, 233, 238, 239, 242, 243

Xengine PDP

PDP developed in Java that implements most of the XACML 2.0 standard but, for conforming policies, its performance is very fast because of the use of numericalisation, rewriting rules as first-applicable and evaluating rules using MTBDD. *see* MTBDD, 42, 45, 48

Performance Analysis, Reporting and Prediction of Access Control Systems

Application that enables its users to query the results (both service times and decisions) of a STACS run and which offers extensive statistical procedures and visualisation to understand the performance of a given access control system. 10, 275

Play Framework

Play (https://www.playframework.com/) is built on Akka and a lightweight, stateless architecture. It offers predictable and minimal resource consumption for highly-scalable applications. 36

power set

The power set $\mathcal{P}(A)$ of a set A is the set of all subsets of A, including the empty set $\varnothing$ and A itself. If $|X|$ is the cardinality of any set X, then $|\mathcal{P}(A)| = 2^{|S|}$.. 161

Principal

the access-requesting entity: it could be a group, a person or an agent acting for either. 20

proportional thinning

admission control procedure where a fixed percentage of overload requests are not admitted to the server, the objective being to control the queue size. 79, 109

R

Request

When interacting with protected resources, agents (people and/or software acting on the behalf) might require access to some of those resources. The privilege management system provides a reference monitor that intercepts this activity and forwards each access *request* to the PDP for a decision.. 142

Resource

Policy and Request model entity corresponding to the Resource (related to the static model Asset) element of a XACML 2.0 Target or Request element. It is an example of a CTC. 26, 138, 141, 265, 267, 269

Role-Based Access Control

Access control specification where policies are specified in terms of combinations of attributes (termed roles) of the subjects.. 38, 275

Rule

Policy model entity having a Target and a Rule Effect (e.g., Permit) corresponding to the XACML 2.0 policy element of the same name.. *see* Target, 26

S

Scalability Testbed for Access Control Systems

Application that enables its users to configure and run controlled, repeatable performance experiments in a testbed. vii, 276

Separation of Duties

Constraint that distributes a workflow across two or more entities, to reduce the likelihood of mistakes or abuse of power. As an example, the person who makes an investment decision is not the person who makes the actual investment.. 20, 275

service time

The timings obtained via the Adapter capture the total time spent by the PDP per request a) converting the XACML-encoded request into the PDP's internal representation in memory, b) searching the policy set for matching policies and c) returning the decision as a XACML-encoded response. xii, 60, 61, 64, 65, 68, 69, 68, 71, 73, 75, 77, 86

Subject

Policy and Request model entity corresponding to the Subject (related to the static model Member and Organisation) element of a XACML 2.0 Target or Request element. It is an example of a CTC. 26, 138, 141, 145, 265, 267, 269

support vector machine

Statistical pattern recognition technique. 276

T

Target

Policy model entity corresponding to the Target element of a XACML Rule, Policy or PolicySet. It is a Conjunction of CTC elements. 54, 136, 137, 138, 141, 142, 145, 149, 150, 155, 156, 261

TargetComponentType

Policy and Request model entity associated with XACML 2.0 CTC hierarchies each of which must be associated with a single Target Component Type: Subject, Resource, Action or Environment. 145

Target SubComponent

Policy and Request domain entity, a simple logical "Subject-Action-Resource" condition; the lowest level in the Target hierarchy. 138, 276

Target Component

Policy and Request domain entity, comprising a conjunction of Target SubComponents. xvii, 138, 276

U

usage control

Access control is concerned with once-off checks of what can be done. Sometimes the access policies are sticky (i.e., are bound to the digital asset they protect) and so are invoked each subsequuent reuse of that asset, to ensure that such usage is consistent with the original access conditions.. 6

V

Vert.x

Vert.x is a lightweight, high performance application platform for the JVM that uses message-passing with nonblocking I/O and the Actor concurrency model.. *see* Actor, 36

Index