UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Policy Driven Development: SOA Evolvability through Late Binding

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy

in

Computer Science

by

Barry Demchak

Committee in charge:

Professor Ingolf Krüger, Chair
Professor William Griswold
Professor Kevin Patrick
Professor Ramesh Rao
Professor Stefan Savage

2013

The Dissertation of Barry Demchak is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2013

## DEDICATION

First, to God

… without whose grace this contribution would not be possible.

Second, to Franca

… my loving and patient wife.

Third, to all of my family and friends

… who encouraged me every step of the way.

And finally, to those who seek truth

… with integrity, creativity, sensitivity, and love.

# EPIGRAPH

*Anyone who lives within their means*

*suffers from a lack of imagination.*

Oscar Wilde
(On the true meaning of graduate school)

*All the world's a stage.*

*And all the men and women merely players;*

*They have their exits and their entrances,*

*And one man in his time plays many parts,*

William Shakespeare
As You Like It (Act 2, scene 7, lines 139-143)
(An early definition of SOA)

TABLE OF CONTENTS

## LIST OF ABBREVIATIONS

| | |
|---|---|
| ABAC | Attribute Based Access Control |
| ACL | Access Control List |
| ADL | Architecture Definition Language |
| AEV | Application and Environment-related Value |
| AO-ADL | Aspect Oriented Architecture Definition Language |
| AOD | Aspect Oriented Design |
| AOP | Aspect Oriented Programming |
| AORE | Aspect Oriented Requirement Engineering |
| AOSA | Aspect Oriented System Architecture |
| AOSD | Aspect Oriented Software Design |
| API | Application Programming Interface |
| BPEL | Business Process Execution Language |
| BPMN | Business Process Modeling Notation |
| CABIG | Cancer Biomedical Informatics Grid |
| CCS | Calculus of Computing Systems |
| CI | Cyberinfrastructure |

| | |
|---|---|
| CIS | Context Infrastructure Service |
| CPN | Colored Petri Net |
| CSP | Communicating Sequential Processes |
| CSS | Cascaded Style Sheets |
| CYCORE | Cyberinfrastructure for Comparative Effectiveness |
| DBMS | Database Management System |
| DLL | Dynamic Linked Library |
| DSL | Domain Specific Language |
| DOM | Document Object Model |
| EA | Early Aspects |
| EAA | Enterprise Application Architecture |
| EB | Exposure Biology |
| EI | Enterprise Integration |
| EPAL | Enterprise Privacy Authorization Language |
| EPC | Event-driven Process Chain |
| ESB | Enterprise Service Bus |
| EU | European Union |

| | |
|---|---|
| FOSD | Feature Oriented Software Development |
| FR | Functional Requirement |
| GAARDS | Grid Authentication and Authorization with Reliability Distributed Services |
| GEI | Genes, Environment and Health Initiative |
| GPS | Global Positioning System |
| GUI | Graphical User Interface |
| GUID | Globally Unique Identifier |
| GWT | Google Web Toolkit |
| HIPAA | Health Insurance Portability and Accountability Act |
| HTTP | Hypertext Transfer Protocol |
| IA | Information Assurance |
| IRB | Institutional Review Board |
| IM | Interservice Message |
| ISO | International Standards Organization |
| IV | Independent Value |
| LTL | Linear Temporal Logic |

| | |
|---|---|
| JAAS | Java Authentication and Authorization Service |
| JMS | Java Message Service |
| JSON | JavaScript Object Notation |
| MSC | Message Sequence Chart |
| MDA | Model Driven Architecture |
| MDE | Model Driven Engineering |
| MSOD | Multisession Separation of Duties |
| NFR | Nonfunctional Requirement |
| NIH | National Institutes of Health |
| NSF | National Science Foundation |
| OCL | Object Constraint Language |
| OMG | Object Management Group |
| OO | Object Orientation |
| OOD | Object Oriented Design |
| OOI-CI | Ocean Observatory Initiative Cyberinfrastructure |
| OOP | Object Oriented Programming |
| OWL | Web Ontology Language |

| | |
|---|---|
| PALMS | Physical Activity Location Measurement System |
| PALMS-CI | PALMS Cyberinfrastructure |
| PBD | Policy Based Design |
| PDD | Policy Driven Development |
| PDP | Policy Decision Point |
| PEP | Policy Enactment Point |
| PERMIS | Privilege and Role Management Infrastructure Standards |
| PI | Principal Investigator |
| POJO | Plain Old Java Object |
| RA | Research Assistant |
| RAS | Rich Application Service |
| RBAC | Role Based Access Control |
| RDBMS | Relational Database Management System |
| RDL | Requirements Definition Language |
| REST | Representational State Transfer |
| RIS | Rich Infrastructure Service |
| RPC | Remote Procedure Call |

| | |
|---|---|
| RS | Rich Service |
| RSDP | Rich Service Development Process |
| SCA | Service Component Architectures |
| SDC | Service/Data Connector |
| SIV | Service Interaction-related Value |
| SOA | Service Oriented Architecture |
| SOAP | Simple Object Access Protocol |
| SOARS | Student Online Administration and Reporting System |
| SoS | System of Systems |
| SQL | Structured Query Language |
| SYN/ACK | Synchronize Acknowledgement |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| UCSD | University of California, San Diego |
| UDDI | Universal Description Discovery and Integration |
| UI | User Interface |
| UML | Unified Modeling Language |
| URN | User Requirements Notation |

| | |
|---|---|
| US | United States of America |
| VM | Virtual Machine |
| WFMS | Workflow Management System |
| WS-CDL | Web Services Choreography Description Language |
| WS-POLICY | Web Services Policy Language |
| WSCL | Web Services Conversion Language |
| XACML | Extensible Access Control Markup Language |
| XHTML | Extended Hypertext Markup Language |
| XML | Extensible Markup Language |
| YAWL | Yet Another Workflow Language |

LIST OF FIGURES

LIST OF TABLES

# ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Ingolf Krüger, for his constant and generous encouragement, moral support, advice, and overall leadership. This work would not have been possible without his help and good ideas. I also thank my advisor and Professors William G. Griswold, Ramesh R. Rao, and Kevin Patrick for their financial support and for their generosity in widening my view of the computing world. I thank Massimiliano Menarini for his support in getting this thesis written and for the years of collaboration and friendship as my office mate, coauthor, and mentor. I also thank my friend and former office mate, Lucas Lee, for years of encouragement and good humor. Finally, I thank all the members of my dissertation committee for reading this manuscript and providing feedback.

Chapter 4, in part, is a reprint of material as appeared in 3 papers:

1) A paper currently being prepared for submission for publication of the material. B. Demchak, C. Farcas, E. Farcas, I. Krüger. The dissertation author was a co-investigator and co-author of this material.

2) B. Demchak and I. Krüger, "Policy Driven Development: Flexible Policy Insertion for Large Scale Systems," in 2012 IEEE International Symposium on Policies for Distributed Systems and Networks, Chapel Hill. IEEE Computer Society, Jul. 2012, pp. 17-24. The dissertation author was the primary investigator and author of the text used in this chapter.

3) B. Demchak, J. Kerr, F. Raab, K. Patrick, and I. H. Krüger, "PALMS: A Modern Coevolution of Community and Computing Using Policy Driven Development," in 45th Hawaii International Conference on System Sciences (HICSS), Maui, Hawaii. Jan. 2012. The dissertation author was the primary investigator and author of the text used in this chapter.

Chapter 5, in part, is a reprint of material as appeared in 3 papers:

1) B. Demchak and I. Krüger, "Policy Driven Development: Flexible Policy Insertion for Large Scale Systems," in 2012 IEEE International Symposium on Policies for Distributed Systems and Networks, Chapel Hill. IEEE Computer Society, Jul. 2012, pp. 17-24. The dissertation author was the primary investigator and author of the text used in this chapter.

2) B. Demchak, J. Kerr, F. Raab, K. Patrick, and I. H. Krüger, "PALMS: A Modern Coevolution of Community and Computing Using Policy Driven Development," in 45th Hawaii International Conference on System Sciences (HICSS), Maui, Hawaii. Jan. 2012. The dissertation author was the primary investigator and author of the text used in this chapter.

3) B. Demchak, C. Farcas, E. Farcas, and I. Krüger, "The Treasure Map for Rich Services," in Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration (IRI), Las Vegas, USA. IEEE, Aug. 2007, pp. 400-405. The dissertation author was a co-investigator and co-author of this material.

## VITA

| | |
|---|---|
| 1975-1977 | Principal, Chama Corporation |
| 1979 | Bachelor of Arts, University of California, San Diego |
| 1980-1982 | Director of Software, Advanced Computer Design |
| 1982-1986 | Principal, Software Construction, Inc |
| 1986- | Principal, Torrey Pines Software, Inc |
| 2008 | Master of Science, University of California, San Diego |
| 2013 | Doctor of Philosophy, University of California, San Diego |

## PUBLICATIONS

B. Demchak and I. Krüger, "A Model-Driven Engineering Approach to Requirement Elicitation for Policy-Reactive Cyberinfrastructures," Tech. Rep. CS2012-0988, University of California, San Diego, Computer Science and Engineering Department, Sep. 2012.

B. Demchak and I. Krüger, "Policy Driven Development: Flexible Policy Insertion for Large Scale Systems," in 2012 IEEE International Symposium on Policies for Distributed Systems and Networks, Chapel Hill. IEEE Computer Society, Jul. 2012, pp. 17-24.

I. Krüger, B. Demchak, and M. Menarini, "Dynamic Service Composition and Deployment with OpenRichServices," Software Service and Application Engineering, vol. 7365, M. Heisel, Ed. pp. 120–146, Springer Berlin / Heidelberg, 2012.

B. Demchak, J. Kerr, F. Raab, K. Patrick, and I. H. Krüger, "PALMS: A Modern Coevolution of Community and Computing Using Policy Driven Development," in 45th Hawaii International Conference on System Sciences (HICSS), Maui, Hawaii. Jan. 2012.

N. Nikzad, C. Ziftci, P. Zappi, N. Quick, P. Aghera, N. Verma, B. Demchak, K. Patrick, H. Shacham, T. S. Rosing, I. Krueger, W. Griswold, and S. Dasgupta, "CitiSense - Adaptive Services for Community-Driven Behavioral and Environmental Monitoring to Induce Change," Tech. Rep. CS2011-0961, University of California, San Diego, Jan. 2011.

B. Demchak, V. Ermagan, E. Farcas, T.-J. Huang, I. Krüger, and M. Menarini, "A Rich Services Approach to CoCoME," The Common Component Modeling Example, Comparing Software Component Models, A. Rausch, R. Reussner, R. Mirandola, and F. Plásil (Eds.), Lecture Notes in Computer Science, vol. 5153, pp. 85-115, Springer Berlin / Heidelberg, Aug. 2008.

B. Demchak, V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini, "Rich Services: Addressing Challenges of Ultra-Large-Scale Software-Intensive Systems," in Proceedings of the ICSE 2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), Leipzig, Germany. New York, NY, USA: ACM, May 2008, pp. 29-32.

B. Demchak and I. H Krüger, "Rich Feeds for RESCUE," in Proceedings of the 5th International Conference on Information Systems for Crisis Response and Management (ISCRAM 2008), F. Fiedrich and B.K Van de Walle (Eds.), Washington, D.C. May 2008.

B. Demchak and I. H Krüger, "Composable Chat: Towards a SOA-based Enterprise Chat System," Tech. Rep. CS2008-0918, UCSD, Apr. 2008.

B. Demchak, W. G. Griswold, and L. A. Lenert, "Data Quality for Situational Awareness during Mass-Casualty Events," in Proceedings of the American Medical Informatics Association Annual Fall Symposium 2007, Chicago. Nov. 2007, pp. 176-180.

B. Demchak, C. Farcas, E. Farcas, and I. Krüger, "The Treasure Map for Rich Services," in Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration (IRI), Las Vegas, USA. IEEE, Aug. 2007, pp. 400-405.

M. Arrott, B. Demchak, V. Ermagan, C. Farcas, E. Farcas, I. H. Krüger, and M. Menarini, "Rich Services: The Integration Piece of the SOA Puzzle," in Proceedings of the IEEE International Conference on Web Services (ICWS), Salt Lake City, Utah, USA. IEEE, Jul. 2007, pp. 176-183.

S. W. Brown, W. G. Griswold, B. Demchak, and L. A. Lenert, "Middleware for Reliable Mobile Medical Workflow Support in Disaster Settings," in Proceedings of the American Medical Informatics Association Annual Fall Symposium 2006, Washington. American Medical Informatics Association, Nov. 2006, pp. 309-313.

## FIELD OF STUDY

| | |
|---|---|
| Major Field: | Software Engineering |
| Sub-fields: | Software Architecture |
| | Service-Oriented Systems |

## ABSTRACT OF THE DISSERTATION

Policy Driven Development:
SOA Evolvability through Late Binding

by

Barry Demchak

Doctor of Philosophy in Computer Science

University of California, San Diego, 2013

Professor Ingolf Krüger, Chair

Software maintenance is a significant cost driver in the value proposition of large scale software systems such as cyberinfrastructures (CIs) – it is often the longest and most expensive phase of software production. Because software maintenance and delivery cycles are often long and risky, stakeholder requirements are often not realized in timeframes meaningful to stakeholders. Such delays impair software system value due to lost opportunities, costs exceeding benefits, and stakeholder disenfranchisement.

As a solution, my dissertation proposes a new methodology called Policy Driven Design (PDD), which enables the composition of stakeholder requirements onto an unprepared application at runtime. PDD models an application as a collection of base *workflows* that implement stakeholder requirements. It defines a *policy* as a decision that chooses amongst alternative workflows – stakeholder requirements can be expressed and realized as policies injected into a base workflow.

An important source of delays under existing methodologies is *early binding*, which occurs when requirements (as policies) are integrated into applications during design and coding phases, often causing entanglement and scattering at both abstract and coding levels, and resulting in delays and mis-implementations.

PDD introduces *late binding* as the injection of requirements (as policies) into running systems without incurring traditional development risks and delays. PDD policies are expressed using Domain Specific Languages tailored to requirement domains, thereby enabling stakeholders to participate directly in defining, vetting, and evolving policies.

To demonstrate and evaluate PDD, I designed and implemented a successful real world cyberinfrastructure (PALMS-CI) using PDD principles. PALMS' late binding demonstrated policy injection with acceptable overhead in common cases. Its workflow support proved effective in significantly reducing entanglement and scattering, and its DSL support demonstrated stakeholder enfranchisement resulting in quick and accurate requirement realization.

PDD leverages Aspect Oriented Software Design (AOSD) and Service Oriented Architecture (SOA) principles, and contributes:

- a SOA foundation for policy definition and injection
- a family of policy languages that enable programmer/stakeholder collaboration
- a working cyberinfrastructure that realizes PDD and serves as a platform for future development

Given the intensifying contradiction between greater system complexity, increasing stakeholder demands, and shorter delivery timeframes, PDD uncovers a low cost route to high value.

# CHAPTER 1

# POLICY DRIVEN DEVELOPMENT IMPROVES EVOLVABILITY

The topic of this dissertation is a new development methodology called Policy Driven Development (PDD), which is my approach to designing complex systems so as to improve their evolvability over time. In this dissertation, I argue that under existing software development methodologies, large scale computing systems do not realize stakeholder requirements quickly enough to meet the needs of large and diverse stakeholder groups, and that the viability of such systems is threatened as a result.

I observe that for systems expressed as Service Oriented Architectures (SOAs), stakeholder requirements can be represented as workflows, and systems can be formed by composing workflows together. I argue that an important cause of slow system evolution is performing this composition *early* in the software development cycle, thereby resulting in inflexible systems that exhibit long delays between requirement discovery and the delivery of systems incorporating them. To address this problem, I propose the PDD methodology, which results in the composition of requirements *late* in the software development cycle, thereby dramatically reducing delays between requirement discovery and delivery.

To achieve this, PDD envisions a unique collaboration between programmers and stakeholders, where both cooperate to cast requirements as *policies* represented as workflows and the conditions under which they are activated. PDD then leverages and extends existing features of SOAs to enable the injection of policies into *running* systems. As a result, stakeholder requirements can be realized with speed, accuracy, and low cost unachieved with existing methodologies.

My contributions to this vision include a SOA foundation for policies and policy injection, a family of policy languages that enable programmer/stakeholder collaboration, and a real-world case study that realizes them and serves as a platform for future development.

In this chapter, I briefly describe unsolved issues affecting the evolution of large scale systems having large and diverse stakeholder groups and how these issues threaten the viability of such systems. Specifically, I focus on the relationship between the evolution of both stakeholders' requirements and the systems that reflect them. I discuss various approaches to requirement realization in this context, and describe the PDD vision, which includes a realignment of the roles of programmers and stakeholders in evolving such systems. Finally, I explain my contributions and give an outline for the remainder of this dissertation.

## 1.1 The Evolvability Problem at Scale

Since shortly after programs were first written, software maintenance [1] has been a significant cost driver in the software value proposition – it is often the longest and most expensive phase of software production [2] [3] [4]. Software maintenance encompasses many post-delivery activities, including bug fixes, adaptation to environmental change, and evolution responsive to changes in requirements. While the purpose of software is to realize articulated requirements (including those that change or are presented anew), application developers often address the unspoken requirement of "evolvability" as an afterthought, if at all [5]. In the process, stakeholders become disenfranchised because their new and evolving requirements are not met on a timely basis, thereby limiting productivity and flexibility that could contribute to their own success.

I define evolvability as the ability of a system to address new needs, easily fulfill new requirements, and do so at relatively low cost [6] [7] [8]. Improvements in evolvability can result in the liberation of resources to advance stakeholder interests, which may include faster time to market, increased efficiency, reliability, robustness, more (and more sophisticated) feature sets, and fewer bugs, to name but a few.

Many improvements in software engineering have directly impacted evolvability, including advances in programming languages [9]; programming models [10] [11] and system architectures [12] [13] [14]; and development models [15] [16], design patterns [17], and modeling languages [17] [18]. Even so, as these advances extend the volume and complexity of requirements that can be addressed by software systems, stakeholders offer even more (and more complex) requirements (requiring faster delivery, to boot).

Evolvability is especially critical for cyberinfrastructures (CIs) and, by extension, other large scale systems, where the timely and accurate identification and realization of emergent requirements drives system success [19] [20] [21]. A cyberinfrastructure [22] [23] is a computing ecosystem that simultaneously fulfills the diverse requirements of classes of producers, consumers, and operators; its purpose is to serve a diverse community of stakeholders having interests that bind them and which form the basis for the community. It is a scalable Internet-based computing environment that supports data acquisition, storage, management, integration, mining, and visualization, in addition to related computing and information processing services. In order for a stakeholder to contribute data, consume data, or contribute resources, the CI must meet the stakeholder's requirements even as those requirements evolve – to the extent that both current and emergent requirements are satisfied, the ecosystem thrives and the community benefits. As a community enabler, the CI also serves as a means to recruit other stakeholders, which strengthens the community and contributes additional requirements. Through this synergy, the community and the CI co-evolve, and the coevolution often depends on the rapid realization of these new requirements [24]. In a virtuous cycle, continued participation of some stakeholder groups attracts and enables participation by other stakeholder groups, and the CI better serves the community. In a vicious cycle, unmet requirements that threaten the participation of one stakeholder group eventually threaten the participation of others, resulting in the disaffection and eventual opt-out of some or many stakeholders.

Cyberinfrastructures have long lifetimes (measured in decades), and not only accrue maintenance issues and new requirements from existing stakeholder groups, but also accrete new stakeholder groups having new and unanticipated classes of requirements. These requirements represent both changes to existing system features and the creation and composition of completely new feature sets.

## 1.2 Policies and Evolvability – Then and Now

To motivate a discussion of evolvability, I briefly describe an ancient real world system (SOARS) that was not easily evolvable, and use it to introduce existing policy-based solutions and explain their limitations. I then introduce other relevant approaches and give a brief overview of their limitations. (A more detailed survey of existing approaches is presented in Chapter 2.) In Section 1.3, I present PDD as my vision for a solution.

### 1.2.1 SOARS: A History of the Wrong People Making Right Decisions

When I was 18 years old, I was a principal in a company that created and sold the SOARS[1] software, a state-of-the-art system that automated the work of a small-to-medium size university, replacing largely paper-based processes. I was responsible for programming a collection of major subsystems, while my four partners created other subsystems. The year was

[1] Student Online Administration and Reporting System, a product of Chama Corporation written in timeshared Basic

1975, and even then, university operations encompassed complex workflows and rules. The development process was an iteration of a basic sequence: collect and negotiate requirements, instantiate the requirements in code, and present the results to the customer for evaluation and rework.

A particularly complex domain was student accounts – managing billing, payments, and credits for students depending on their attendance, financial aid, payment arrangements, and so on. There were simply too many cases to express conveniently as requirements, and I, as the programmer, made many guesses and deductions regarding how to conceptualize, organize, and handle each case. While the deliverables were judged correct and greatly beneficial by the student accounts office, it occurred to me that I was making a great number of decisions that I was unqualified to make, and a portion of which were likely wrong or subject to revision. The consequences of my wrong decisions could have been minor (requiring re-coding) to catastrophic (losing data or having critical functionality incorrect at critical times). Independent of my decisions, university administrators frequently asked for new features and asked that working features be changed to work differently. Furthermore, bug fixes and new feature implementations were usually deployed in a formal release process that occurred infrequently, thereby deferring (and possibly compromising) their benefit to users.

This vignette is true, and is representative of the state of enterprise application development processes both before 1975 and after, continuing to present day. At that time, the purpose of an enterprise software program was to realize stakeholder requirements so as to deliver some (usually economic) value to the stakeholders – generally, a greater value than the alternatives. Major drivers of application value were the quality of requirement elicitation, the fidelity of delivered code relative to those requirements, and the delay between requirement elicitation and code delivery. The nascent discipline of software engineering was often more concerned with technique than with delivering or optimizing value – the hot topics were modularity and information hiding [25], use of `goto` statements [26], and the art and styles of programming [27]. Additionally, software projects were executed most often using a plan-driven approach, which involved long latencies between requirement specification (as either formal or informal exercises) and actual system delivery. Often, upon system delivery, initial requirements were found to be insufficient, wrong, outdated, or misinterpreted, thereby necessitating subsequent (time consuming) remedial development and release cycles. Such software projects often resulted in expensive cost and calendar overruns, and many projects failed.

While the software value proposition today is the same as in 1975, the value drivers have changed significantly, and they continue to evolve. Today's stakeholders are more numerous, diverse, and have more complex requirements than those for legacy applications. These requirements encompass increasing degrees of scalability, manageability, distributability, evolvability, auditability, reusability, and reliability while incorporating information assurance [28] values such as security, availability, integrity, authenticity, confidentiality, non-repudiation. While some requirements can be realized independent of others, many stakeholders' requirements crosscut basic system functionality and other stakeholders' requirements. Significantly, stakeholder expectations of application quality (as reflected in code fidelity and coverage of the requirements space) and delivery times have remained relatively constant.

Since the mid 1970s, disciplines, techniques, methodologies, and tools have evolved to accommodate stakeholders' increasingly complex requirements while reducing software development risk. Disciplines such as requirements engineering [29] and system architecture have contributed to the capture, refinement, elaboration, disambiguation, evolution, and realization of requirements. Techniques such as modular design, object orientation, aspect orientation, standards-based design, service orientation, and functional languages have enabled programmers to manage complexities associated with distributed and heterogeneous systems, cross-platform development, and embedded systems more efficiently and effectively. The agile family of methodologies has enabled programmers to better leverage stakeholders to reduce delivery times and focus these deliveries on high value propositions. Finally, tools such as integrated development environments (IDEs), modeling systems, theorem provers, version and configuration control, and improved languages have reduced the risk in creating large and complex code bases that realize large and complex requirement sets.

Even so, this evolution has occurred as incremental improvements to the 1970s programming discipline – stages of the development process have been deconstructed and elaborated, optimized, or rearranged in order to create significant economic benefits. For example, agile methodologies were developed as alternatives to plan-driven methodologies specifically to enable rapid response to changing stakeholder requirements. As a process improvement, it combines a requirement prioritization process with constant stakeholder engagement and frequent system releases to deliver applications that meet stakeholder requirements prioritized on a timely basis according to stakeholder values. It substantially reuses existing requirements engineering and architecture disciplines, though it emphasizes techniques whose incremental cost closely matches the incremental value provided.

In modern large scale systems such as CIs, successful development depends on modern disciplines, techniques, methodologies, and tools to deliver timely and relevant computing capabilities to stakeholders whose mission depends on them. However, taken singly or in combination, they do not address the need stakeholders have for rapid and reliable realization of requirements. Consequently, stakeholders' requirements go unmet, and the health of the CI degrades accordingly. For example, though agile development methodologies intend to quickly enact stakeholder requirements, they are a poor fit for CI development and maintenance, as CI communities tend to be large, diverse, geographically dispersed, and unable to engage in the day-to-day interaction and evaluation needed for requirement definition, refinement, and prioritization. Additionally, because CI communities are often distributed worldwide, CIs are often required to be highly available on a 24 hour a day, 7 days a week basis, thus making even the high frequency agile release schedule very expensive. Indeed, the risks associated with deployment of even small changes make the prospect of releases themselves unattractive. Furthermore, to the extent that a CI has cyber-physical [30] components, the modification and redeployment of such components can be particularly risky and expensive in terms of energy and time. Consequently, re-deployment occurs only infrequently, which defeats the agile proposition of shareholder enfranchisement through rapid response.

### 1.2.2 How Policy-based Solutions Have Fallen Short (So Far)

A common technique for implementing stakeholder requirements and avoiding release latencies is replacing hard-coded branch expressions (incorporated into application workflows at development time) with policy decisions based on policy expressions supplied at runtime. Policy decisions are commonly used to implement access control or application feature selection based on a user's credentials or on the state of the application or operating environment. More generally, they use policy expressions to choose between two or more workflows at a decision point in a base workflow. While this technique can address requirements known at development time, it has limited value at runtime – it can only select amongst workflows known at development time. Addressing changing or emerging requirements that result in new or changed workflows – or in new decision points – generally requires application modifications and re-deployment, which are often lengthy, costly, and error-prone processes. Furthermore, existing policy techniques do not account for the integration of independent policies provided by multiple, independent stakeholder groups.

An implicit assumption in implementing a new or changed requirement is that not only will the application correctly and completely implement the requirement, but the implementation of unrelated requirements will be unaffected. Under most development methodologies (including waterfall and agile) continued correct operation is demonstrated by regression tests. However, because such tests do not typically cover all branch paths and corner cases, they *suggest* correctness but do not *prove* it. As a CI evolves and becomes more complex, the system-wide guarantees offered by regression tests become weaker, and such applications require improved strategies for achieving low risk of maintenance. Similarly, modern program development tools include theorem provers that can be used to evaluate properties implying code correctness. However, such tools are limited by the size and complexity of the code under test, and their use becomes less feasible as the CI evolves and becomes more complex. Furthermore, regression tests and theorem provers work well when code contains hard-coded branching expressions, and are less robust against decision points evaluating the universe of runtime-supplied policy expressions.

An issue that diminishes the value of embedded policy decisions, regression testing, and theorem proving is the entanglement of multiple independent concerns in a single workflow. While many programming languages encourage entanglement for the sake of compactness, performance, and intelligibility, entangling such concerns risks intelligibility over time as a workflow representing a concern becomes more complex. When complex workflows are combined, the clarity of policy decisions diminishes, the number and complexity of regression tests multiplies, and the tractability of theorem proofs is reduced. Consequently, the costs of maintenance and the risks of undetected errors increase.

Finally, insofar as the requirement elicitation process involves a handoff of information from stakeholders to programmers, requirements are subject to omission, misstatement, incomplete statement, and mis-implementation, leading to application errors whose remediation results in costly and risky application or policy re-deployment [31]. To date, facilities enabling intimate collaboration between programmers and stakeholders (e.g., domain-specific policy languages, policy editors, visualizers, and validity checkers) are under-developed.

### 1.2.3 Better Evolvability Leads to More Productive Stakeholders

Through enabling the rapid and accurate realization of stakeholder requirements in cyberinfrastructures, PDD aims to maintain and improve the cyberinfrastructure value proposition along a number of dimensions:

- Stakeholders can leverage critical capabilities in timeframes that suit their needs
- Stakeholders can participate directly in the realization of requirements, thereby improving the requirements elicitation process, and leading to implementations that address actual requirements more quickly
- Cyberinfrastructures can quickly accommodate requirements that increase in number and complexity from stakeholder groups that increase in number and diversity.
- Cyberinfrastructures can become simpler and more reliable, while their code becomes more reusable and scalable even as they continue to leverage new technologies and capabilities that exist outside of the cyberinfrastructure.

### 1.2.4 Why has the Evolvability Problem Not Been Solved?

Existing methodologies, techniques, and strategies are challenged because the architectures and code they produce are often entangled, brittle, and scattered, particularly as they encode multiple stakeholders' requirements realized on workflows spread throughout an application. Such architectures and code cannot evolve to address long-lived requirement streams quickly, consistently, and with high fidelity to requirements. Errors in requirement implementation are particularly problematic, as they represent lost time and opportunity for the community (at best) or economic damage and lost credibility (at worst). Such errors arise from numerous sources, including poor coding; challenges in understanding and modifying entangled, brittle, and scattered designs and coding [32]; poor testing; and improper understanding or statement of requirements either by technical staff, stakeholders, or both. Perversely, in the time taken to address requirements (either properly or improperly), requirements may change, become obsolete, or may be superseded.

Paradoxically, efforts to broaden the appeal of a CI by recruiting more stakeholders (as users) often results in an influx of more and varied requirements, which leads to longer implementation times and poorer quality implementations. Consequently, communities relying on CIs maintained using existing disciplines are often underserved due to the time it takes to realize their requirements or due to mismatches between real requirements and actual implementations. Ultimately, this discourages growth of the CI and the community it serves.

Essentially, as the seeds of a CI's success, stakeholders and requirements are also the seeds of its failure.

### 1.2.5 How Other Existing Approaches Fall Short

The concepts of code entanglement, brittleness, and scattering are well documented by the Aspect community (represented by disciplines such as Aspect Oriented Design, Aspect Oriented Programming, Aspect Oriented Software Design, and others described in subsequent chapters), and the Aspect community has proposed a family of solutions (i.e., aspect weaving) that gives valuable and significant insights into improving code maintenance and, to an extent, the management and implementation of requirements themselves.

Commonly, software engineering problems are often approached by means of divide-and-conquer or indirection. The Aspect solutions seek to divide an application into workflows that can then be recombined, but do not sufficiently address issues such as conditional injection of workflows at runtime, deep composition of workflows, and state management that enables System of Systems architectures.

Additional evolvability challenges have related, but deeper roots, originating in programming philosophies invented with the first programs – particularly committing decisions and workflows so early in the development process that changing them to suit emergent requirements impacts working and unrelated code inordinately. While this practice has virtues in code and project management, and application testing and verification, it results in highly complex programs requiring the attention of highly skilled designers and programmers (so-called traditional programmers) -- they are often culturally and physically separate from stakeholders, especially when the stakeholder community is large and diverse.

The traditional practice of software engineering has long recognized that quickly implementing new requirements reduces time to market and drives software value. Consequently, in strategic situations, statically coded `if` statements (and their equivalents, representing early bound requirements tightly coupled with specific implementations and with other requirements) have given way to a combination of late binding and indirection-oriented approaches that leverage declared interfaces to enable a coarse grained realization of partial application behaviors. Late bound (or not-so-early bound) interfaces are leveraged to create loose coupling in methodologies and techniques such as modular programming, object oriented programming, dynamic link libraries, plugins and plugin architectures, hooking strategies, Web Services, REST architectures, and service oriented architectures (SOAs).

While potent, these solutions have come up short by requiring the constant participation of traditional programmers as guardians and stewards of highly complex programs that get only more complex as the number, complexity, and diversity of requirements increase. Consequently, they also keep stakeholder communities at a distance instead of enlisting and leveraging them as solution bearers.

Many solutions envision system evolution in terms of replacing old systems with new systems in full or in part, thereby incurring operational risks and inconveniences to stakeholders that often have nothing to gain in return (because they don't benefit directly from the new requirements implemented in the new systems).

In Chapter 2, I examine different classes of solutions that bear partially or substantially on these issues, and I demonstrate that they are insufficient (individually or in obvious combination) to solve the CI evolvability problems posed above. I start by examining potential contributions found in fundamental abstractions (i.e., models of computation) and software engineering methodologies, and then survey particular mechanisms and best practices (i.e., patterns).

## 1.3 PDD – A Vision for Rapid Requirement Injection

To answer the challenges of rapid requirement realization and stakeholder engagement in a CI context, I have created a new methodology called Policy Driven Development (PDD), which seeks to fulfill stakeholder requirements via application changes effected while an application is running. It leverages the observation that stakeholder requirements are often realized by a combination of decisions and follow-on workflows.

Under traditional methodologies, requirement realization follows a common process: decisions and workflows are programmed, then tested, and then deployed as code. From a runtime perspective, this programming amounts to an early binding of requirement implementations, which places a drag on application evolution. PDD seeks to enable rapid evolution by shifting portions of the application development process from the static domain to the dynamic domain, where the workflows and decisions implementing stakeholder requirements are specified at runtime. This late binding of requirement implementations offers the potential of lower latency between requirement specification and realization, invites more direct participation by the stakeholder in the implementation process, and results in a tighter coupling between stakeholder requirements and their implementation.

As such, PDD represents the bifurcation of the existing application development process into complimentary static and dynamic domains. To facilitate the rapid realization of requirements in the dynamic domain, developers in the static domain are encouraged to create short and simple workflows that focus on implementing discrete concerns. The development process in the dynamic domain focuses on combining and constraining such workflows to realize stakeholder requirements.

### 1.3.1 How PDD Differs from Existing Methodologies

Under traditional methodologies (e.g., modular programming and object oriented design), an application consists of a collection of workflows that are related as to purpose and/or state. The sequence of activities in a workflow is determined at design time and is fixed in deployable code – the actual activities executed in the workflow depend on decisions (i.e., if() statements and their equivalents) embedded in the workflow, following a Strategy pattern [33] (as briefly described in Appendix C). For example, for workflows expressed in Java, activities can be expressed as assignments and function calls, and decisions can be expressed as if() statements that choose between workflows. Fundamentally, realizing a new stakeholder requirement in an existing workflow involves augmenting, changing, or removing existing workflow code or changing the decisions that choose between workflows. Using traditional programming techniques, these decisions are often entangled, leading to the entanglement of the concerns they control. As requirements change, the maintenance of static Strategy patterns often preserves or increases entanglement, thereby increasing testing time, risking fidelity to stakeholder requirements, and leading to long intervals between CI re-deployments.

The key insight of PDD is that by deferring the creation of the Strategy pattern until runtime (i.e., as a late-bound Strategy pattern), new stakeholder requirements can be realized on an *executing* application without incurring the delays and risks endemic to early-bound Strategy patterns. Specifically, it envisions the runtime injection of a decision (using a dynamic Inversion of Control pattern [34]) into an otherwise unprepared workflow, where the decision chooses between continuing the workflow, executing a different workflow, or both. Workflows can execute using parameters supplied by the decision or externally, and can be initially deployed with the application, deployed after the application, supplied as part of the decision, or created dynamically. Consequently, by augmenting, changing, or removing decisions and associated workflows at runtime, PDD enables the realization of stakeholder requirements without suffering the development and deployment latencies incurred by existing approaches. Furthermore, PDD encourages the separate definition and maintenance of injectable decisions and workflows, which discourages the entanglement that leads to increased testing, development risk, and long deployment latencies in traditional methodologies.

### 1.3.2 PDD's Perspective on Workflows

PDD envisions the creation of applications as collections of simple workflows (possibly related as to purpose and/or state) that represent basic application functionality. The workflows (called *base workflows*) represent an application's most rudimentary requirements, forming a skeleton that is fleshed out at runtime by composing additional requirements (represented by other decisions and workflows) onto it based on criteria also defined at runtime. An example of a trivial application workflow is a calculator that accepts a numerical input, adds 10 to it, and outputs the result. An auditing requirement can be realized by interjecting an auditing workflow between the addition and output activities. The auditing workflow would record some pertinent information either synchronously or asynchronously relative to the base calculator workflow. Furthermore, the execution of the auditing workflow could be predicated on some decision criteria such as a user's membership in a group.

The participation of multiple independent stakeholder groups admits the likelihood of multiple stakeholder groups having different policies tendered to the same decision location – or, similarly, a single stakeholder group tendering multiple policies to the same location. This necessitates a strategy for composing coincident policies. Following the calculator workflow example, a Scientists group could insist that any calculator use by a member of the Engineers group be logged for audit, and Engineers could insist that calculator use by Scientists be logged. Possible compositions of these policies include evaluating the Scientist policy only if there is no Engineer policy, or vice versa, or always evaluating both policies. In general, PDD forms a policy composition by executing a composition policy tendered to a particular workflow location. Composition policies are defined, authored, and maintained using the same process as other policies, and follow the same principles of policy injection. In this case, the stakeholder controlling the composition policy would have an oversight role that crosscuts all other stakeholder groups, and which would likely default to the CI development team.

Under PDD, the decision of whether to interject a workflow, where in the base workflow to make the decision, and which workflow to inject are all made at runtime based on externally supplied predicates called *policy tuples*. Loosely speaking, a policy tuple consists of a decision location and a *policy,* and a policy consists of decision criteria and a collection of candidate workflows. A policy is said to be *injected* into the workflow at the decision location. A policy tuple is authored in the context of the application's existing workflows – the decision criteria may have access to the base workflow's transient or permanent state, and can communicate this state to the candidate workflows. Whereas in traditional application development, a requirement would be implemented by explicitly recoding the application (e.g., using an if() statement and a function call), under PDD it would be implemented as a policy tuple applied onto the executing application. This amounts to a late-bound, loosely coupled Strategy pattern resulting in low latency between requirement elicitation and enactment.

To take advantage of late decision binding, application workflows must be exposed for inspection at runtime, it must be possible to interrupt transitions between workflow activities, and it must be possible to inject a workflow between activities or replace an entire workflow. Furthermore, while workflow activities themselves can be atomic (as in adding 10 to a numerical input), they can also be composed of workflows, thus extending the opportunities to realize late binding benefits deeply into an application. With traditional imperative and functional programming languages, these prerequisites can be met through introspection, though few (if any) such languages have compilers that generate appropriate metadata. However, at a conceptual level, Service Oriented Architectures (SOAs) provide a clean mapping between workflows and logical or deployment architectures, and include interception features that enable this injection. Consequently, I discuss PDD in terms of services and SOAs, though services and SOAs are not required to realize PDD, per se.

### 1.3.3 Policy Programmers Enfranchise Stakeholders

The ability to create and inject decisions and workflows at runtime encourages the participation of new classes of stakeholders in the application definition, authorship, and maintenance process. Under traditional application development, policy and workflow authorship are generally in the purview of professional developers, as they are both expressed using general purpose programming languages and systems whose operation requires specialized training. Under PDD, the decision location, decision criteria, and workflow choices are decoupled and can be expressed in languages most appropriate for both their authors and their purpose. This admits the possibility of using domain specific languages (DSLs [35]) that enable stakeholders themselves to define or closely scrutinize decision criteria, if not actual workflows, thus freeing professional programming resources to focus on system design and complex workflow implementation in the static domain. Such DSLs can be highly congruent with domain concepts, thereby fostering a high fidelity between stakeholder requirements and their actual implementation.

Additionally, runtime policy authorship invites the creation of a new role in the stakeholder ecosystem: the *policy programmer,* who collects and refines stakeholder requirements; defines, implements, and refines appropriate policy DSLs; and writes and maintains PDD policies on behalf of other stakeholders. A policy programmer must be familiar with stakeholder domains, application workflows, and the technologies used for authoring policies and workflows, then injecting them into base workflows. The policy programmer role represents a specialization and combination of the programmer and stakeholder roles, and adds value by relieving other stakeholders of policy-level implementation.

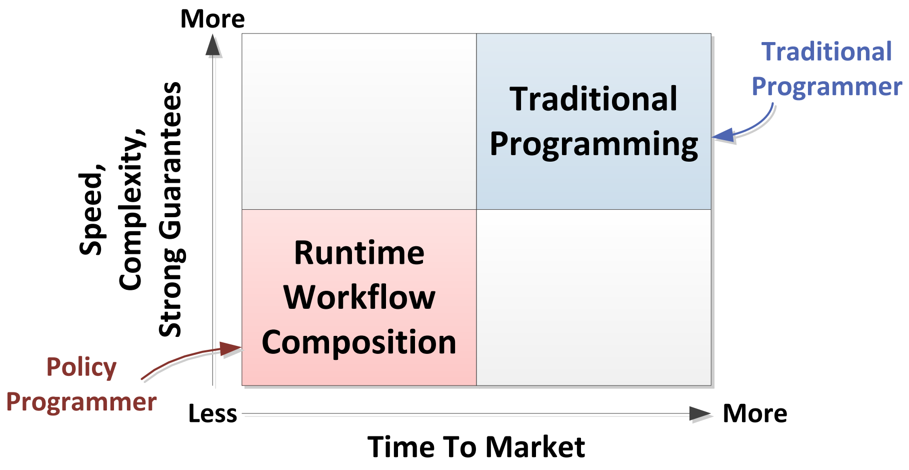


Figure 1. Relationship of PDD to Traditional Programming

Under my vision, as shown in Figure 1, the benefit of the policy programmer role is time to market while maintaining fidelity to stakeholder requirements. As traditional programming relies on well-developed tools that generate fast and efficient code and can coordinate with model checkers to deliver basic guarantees, policy programming does not yet have such support. Consequently, it offers a tradeoff between execution time (as described in Section 6.3), strong guarantees (as discussed in Sections 1.1 and 7.5), and time to market. Additionally, as policy programming tools evolve to provide stronger guarantees, more complex policies become more routinely feasible.

### 1.3.4 Lifting the Policy Abstraction

At an abstract level, composition of workflows realizes a System of Systems architecture, where each workflow may comprise or be a part of a standalone application, and combining workflows results in a complex system that represents a fusion of two or more concerns. Each workflow maintains its own state according to its own rules, defining state transitions and lifecycles appropriate for its function, and possibly sharing state with other workflows. The fusion results from composing an *injected* workflow into a *target* workflow at a location in the target workflow via a policy.

The injected workflow qualifies as a standalone application, as it draws input (from the target workflow); makes some decision, performs some function, or both; and then outputs results (to the target workflow). Using the example of the calculator workflow, the injected audit workflow accepts credentials saved as part of the calculator state, and uses them to determine whether to record an audit event on a local database. Furthermore, the workflow that records audit events is part of a collection of workflows comprising an independent audit application (as described in Section 5.6.4), where other workflows implement the query, visualization, and maintenance of audit events. The System of Systems consists of the combination of the calculation application and audit application, as joined by the injected workflow.

Inasmuch as a policy contributes a workflow, the injected workflow itself can serve as a target for the injection of another policy. Using the example of the calculator's audit workflow, consider a new application requirement to manage data as a database in the Cloud. Given a workflow that stores data in the Cloud, the new requirement can be met by injecting the Cloud workflow into the audit event workflow, replacing the activity that stores audits event on the local database. Composing policies on policies in this way enables the realization of System of Systems architecture whose targets are themselves System of Systems architectures.

Finally, given that a workflow implements one or more stakeholder requirements, or that a single requirement can be implemented by more than one workflow, composite policies injected into multiple decision locations (in either a single workflow or in multiple workflows) enable the implementation of crosscutting requirements as defined in [32].

### 1.3.5 PDD's Focus on Simple Workflows

Consistent with Aspect Oriented approaches, PDD's focus on injection of policies promotes the creation of applications as simple workflows that can act as either injection targets or injected workflows, themselves. By combining applications that expose simple workflows, PDD encourages the creation of hierarchical Systems of Systems, where workflows are well characterized, are separately maintained and validated, are reusable, and form the basis for the rapid realization of stakeholder requirements.

### 1.3.6 The PDD Hypothesis

Realizing my PDD vision requires leveraging and advancing existing techniques and methodologies, and I frame the exercise as addressing hypotheses based on a general proposition that CIs can be modeled as workflows that reflect requirements. My hypotheses are:

- New requirements can be realized in existing applications by composing new workflows onto existing workflows.
- Such compositions can be implemented via the injection of policies, which themselves consist of a decision that selects between alternative workflows.
- The expression of a policy as a DSL enables stakeholder participation in the process of realizing requirements.
- Policy injection can be performed on a deployed application, thereby realizing stakeholder requirements quickly and accurately.

## 1.4 Contributions of this Dissertation

The contributions of this dissertation derive from proving or disproving the hypotheses, particularly in the context of the PALMS case study described in Chapter 5. They include:

1. An engineering approach to the realization of stakeholder requirements in SOA-based cyberinfrastructures (CIs) via runtime policy injection
2. A demonstration of a SOA-based CI that enables runtime policy injection
3. A demonstration of the creation and use of Domain Specific Languages (DSLs) to articulate injectable policy
4. An evaluation of runtime policy injection (in the context of PALMS)
5. An evaluation of the use of DSLs (in the context of PALMS)
6. Insights for improving the performance of injected policies and widening the stakeholder audiences they address

## 1.5 How to Read this Dissertation

The bulk of this dissertation is devoted to explaining the foundational underpinnings of PDD (Chapter 4), a case study that demonstrates PDD (Chapter 5), and an evaluation of PDD as implemented in the case study (Chapter 6), as shown in gold in Figure 2.

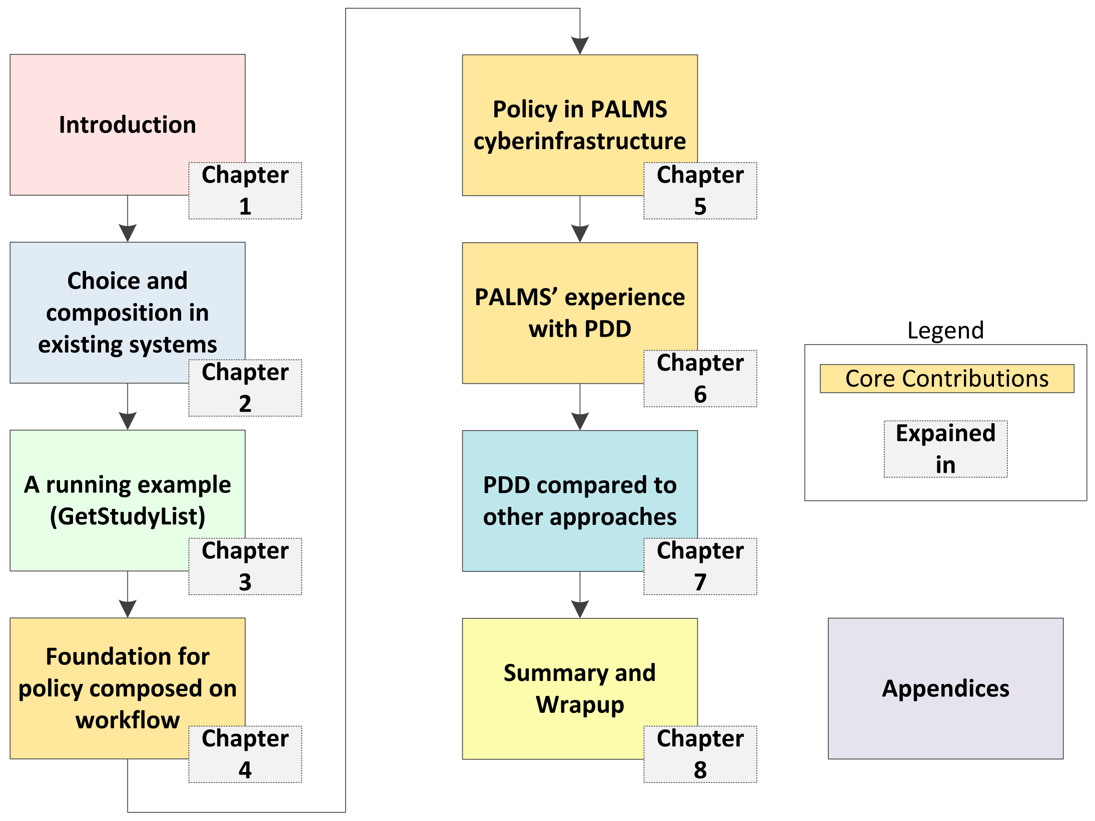

Introduction
Chapter 1
Choice and composition in existing systems
Chapter 2
A running example (GetStudyList)
Chapter 3
Foundation for policy composed on workflow
Chapter 4
Policy in PALMS cyberinfrastructure
Chapter 5
PALMS' experience with PDD
Chapter 6
PDD compared to other approaches
Chapter 7
Summary and Wrapup
Chapter 8
Legend
Core Contributions
Expained in
Appendices

Figure 2. Dissertation Flow

Because PDD addresses requirement composition at such a fundamental level, I devote significant time and material to describing the state of the art (Chapter 2) and comparing PDD's results to existing contributions (Chapter 7). Additionally, Chapter 7 delivers numerous insights into how fundamental PDD features can be supported through additional work, and how PDD itself can be improved in the future.

For grounding and efficiency of discussion, I frame much of this work relative to the GetStudyList workflow, which is a simplification of a common workflow found in the PALMS case study, and which is described in Chapter 3.

Each chapter builds on the material presented in previous chapters, so a straight-through reading of chapters will result in a clear understanding of PDD fundamentals, their implementation, and the evaluations and comparisons at the end.

For a simple understanding of how PDD works and its implementation in the PALMS case study, reading Chapter 4 and Chapter 5 is sufficient.

Note that sections are liberally cross-referenced, so reading for targeted information is feasible.

## 1.6 A Vision of SOARS in a PDD World

As described in Section 1.2.1, SOARS was built more than 35 years ago, and replaced manual processes that were simpler than many of today's processes – they collected less data, involved fewer steps and contingencies, involved fewer stakeholder groups, and had lower expectations regarding flexibility, scalability, availability, robustness, and evolvability.

Were SOARS to be implemented today, it might be implemented as a CI due to the large numbers and types of stakeholders now found in a university, the need for high availability and scalability, and the need for alignment between the capabilities of CIs and stakeholders requirements.

Whereas the SOARS of 1975 was framed as a productivity tool for university administration, a modern SOARS would likely be framed as an infrastructure supporting a vibrant, highly interconnected, and dynamic university community. To the extent that modern SOARS could evolve to accommodate stakeholder requirements on a timely basis, the community would thrive (as described in Section 1.2.1 and exemplified in Chapter 6).

By structuring modern SOARS as a collection of reusable, hierarchically defined workflows using PDD principles, SOARS itself would represent a *class* of applications, where a particular instance would be derived by coupling SOARS workflows (created in the static domain) with a set of policies that coordinate the workflows (created in the dynamic domain) – stakeholder requirements would be realized quickly and with high fidelity through policy injection in the dynamic domain.

Under PDD, the SOARS technical staff's job would be to factor an initial set of stakeholder requirements into workflows, then seed the policy set to bind workflows according to those stakeholder requirements. To the extent that follow-on requirements and requirement changes represent constraints on existing workflows or compositions of new or existing workflows and systems, policy changes represent application evolution at low development cost and deployment latency.

Given this, policy programmers (described in Section 1.3.3) represent a critical role that enables an alignment between stakeholder requirements and CI capabilities by positioning stakeholders themselves as integral to both the definition of requirements and the timely and correct implementation of systems based on principles of PDD.

As a result, modern SOARS could be much more responsive to new and changed stakeholder requirements than the SOARS of 1975, even as the CI scales to meet the demands of new and existing stakeholder groups. For stakeholders, the result is greater, more effective, and more timely collaboration.

## 1.7 Summary

In this introduction, I framed the critical problem of rapid realization of stakeholder requirements in terms of system evolvability, and posed my methodology, Policy Driven Development (PDD), as a means to address evolvability in large scale systems. I briefly described existing policy and non-policy approaches that fail the challenge in one way or another, and I propose my PDD vision where policy programmers collaborate with stakeholders to quickly and accurately realize new and changed requirements by defining and injecting policies into running systems. As a straw man, I described the ancient SOARS system, which was not easily evolvable – in subsequent chapters, I revisit this example to illustrate evolvability in resulting from existing approaches and PDD.

In the next chapter, I survey existing approaches and explain how they come up short against the evolvability challenge; an explanation of the PDD solution begins with Chapter 3.

# CHAPTER 2

# EXISTING APPROACHES TO CHOICE AND COMPOSITION

The problem of realizing requirements quickly and efficiently in computer systems arrived with the creation of the first computer system, and a great deal of attention has been paid to defining and improving theories, methodologies, techniques, and processes that bear upon creating these systems. They address defining what a system is, how requirements are gathered and transformed into systems, how systems are organized and modeled, how models are transformed into working code, how code is deployed, and proving the correspondence between all of these results and the original requirements. In one way or another, each of these issues bears on achieving or maintaining crucial alignment between system requirements and their implementation, even as a system evolves to realize new or changed requirements.

A dominant paradigm for software development is to bind requirements to system design and software coding *early* in the authorship process, and to deliver and deploy verified and validated software as the final product. In both waterfall and agile processes, requirements are factored into designs, designs are evolved into workflows, workflows are coded as programs, and programs are tested and deployed monolithically or in modules. As explained in Chapter 1, the insight of PDD is to bind new and changed requirements (as workflows) to base workflows *late* in the authorship process – such workflows can be constructed and injected into delivered software even as the software executes.

Under both early and late binding paradigms, the nature of the junction between a base workflow and an injected workflow is key. Figure 3 shows this relationship as a UML class diagram (explained in Appendix A), where policy injection addresses the association of new workflows (representing new requirements) with existing workflows (representing existing requirements). It includes the criteria (i.e., a decision) and composition rules for doing so – a *decision* determines whether an injected workflow (or which of possibly several workflows) is executed. Additionally, while workflows express activity sequences, decisions and computations encoded within the workflow may depend on state maintained by or for the workflow. Consequently, injection of an independent workflow is strongly tied to the injection of independent state.

In this dissertation, I describe how PDD addresses the foundations and implementation of late bound workflows via policy injection, which in turn enables rapid, stakeholder-centric system evolution. In this chapter, I examine existing methodologies and technologies as contributions tuned to support *early* requirement binding but which may apply to a *late* binding paradigm. Chapter 4 describes a foundational basis for PDD, and Chapter 5 presents an implementation case study. In Chapter 7, I revisit the existing contributions (described in this chapter) in direct comparison to the foundations and case study presented in Chapter 4 and Chapter 5.

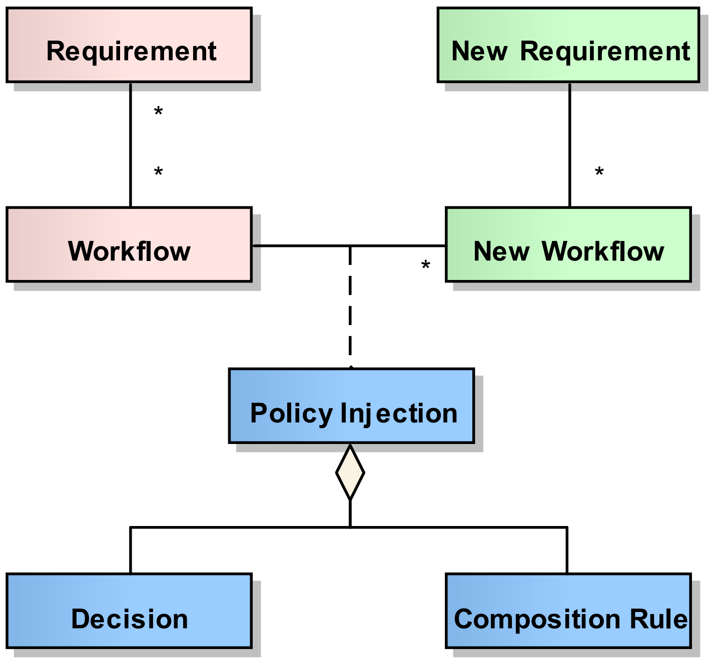


Figure 3. Requirements, Workflows, and Policy Injection

To frame an examination of existing contributions, consider a straw man situation that demonstrates the value of late binding, as drawn from the SOARS example in Section 1.2.1. Under SOARS, students could register for classes on a single day called "registration day", where each student would appear in an auditorium and visit a registrar (who would enter the student's class selections into an online database) followed by a bursar (who would retrieve the class selections, print a bill, and collect fees). The bursar's program would execute the workflow shown in Figure 4a, which included the hidden step of posting the student's billing to the campus general ledger. From the students' perspective, the success of "registration day" was critical because without class enrollment and (eventual) bill paying, they could not attend class. Similarly, from the separate perspectives of the registrar, bursar, and business office, unsuccessful, inconsistent, or incorrect data capture could result in lost revenue and thousands of extra hours of manual data entry, recoding, and auditing. As re-running a registration day would be prohibitively expensive, last minute SOARS changes to accommodate additional stakeholder requirements were forbidden.

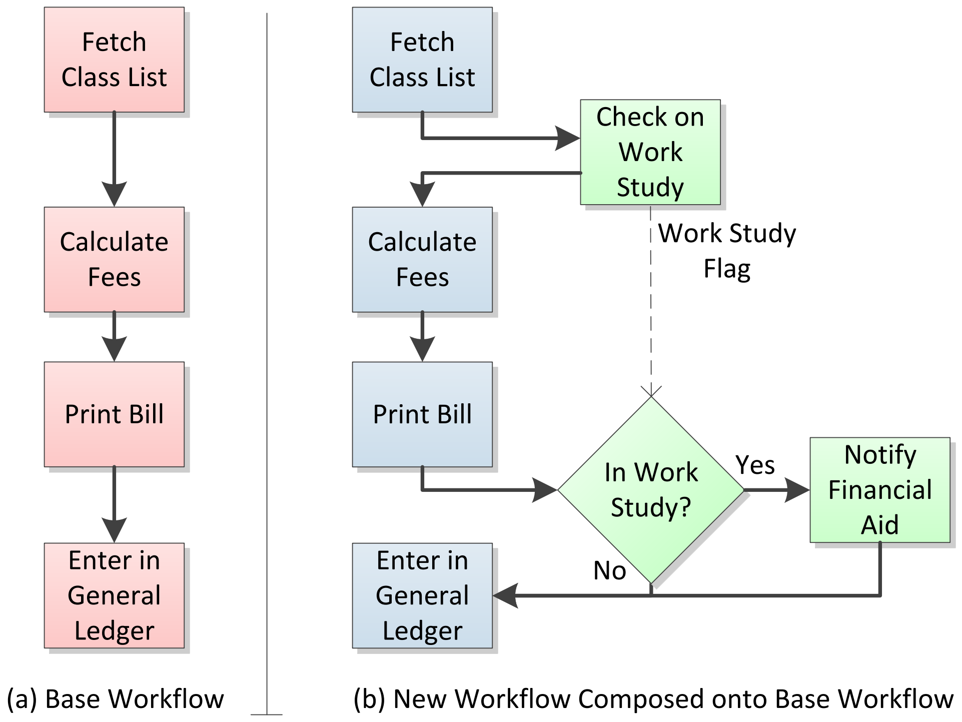


Figure 4. Hypothetical SOARS Bursar Workflow

However, as with many systems supporting a growing and diverse stakeholder community, SOARS requirements changed frequently, and often included the integration of requirements from previously excluded stakeholders. In a hypothetical (but realistic) situation, the financial aid department realizes on the day prior to registration that it could save hundreds of hours by receiving automatic notification of students that enroll in work study classes and have a printed bill. As shown in Figure 4b, this requirement could be recognized as a separate concern realized as the composition of a stateful workflow (consisting of checking the class list, setting a flag, and later deciding whether to send the notification) onto a base workflow. Correct composition requires, among other things, interspersing activities in the new workflow with appropriate activities in the base workflow.

Using the technology underlying SOARS, the injection of this workflow would require reprogramming and re-deploying SOARS, which likely would not be allowed because of the enormous risks to other stakeholders should the registration day workflow be interrupted or broken (as could happen under even the simplest of program modification and re-deployment scenarios). Under a PDD scenario, the new requirement could be realized via workflow injection at runtime, with minimal risk to other stakeholders.

This straw man example represents one of many high value scenarios addressable via late binding, and illustrates a valuable requirement that could not be anticipated by the SOARS programming staff and could not be addressed on a timely basis using traditional programming methods under acceptable risk/reward scenarios. Variations of this scenario include allowing injection of other workflows at the same time (e.g., automatically e-mailing a copy of the printed bill to parents), or replacing a portion of the base workflow (e.g., posting to a different general ledger, depending on attributes associated with the student's identity).

To examine existing methodologies and technologies as contributions that may support this *late* binding paradigm I focus on how they support decisions and workflow composition, and organize them into the broad categories in Table 1.

Table 1. Contributions of Existing Methodologies and Technologies

| Category | Section | Importance of Contribution |
|---|---|---|
| **Models of Computation** | 2.2 | give rise to fundamental approaches to computing |
| **Software Development Methodologies** | 2.3 | justify both strategic and tactical mechanisms that realize workflow injection |
| **Mechanisms** | 2.4 | encode and realize workflow injection |
| **Patterns** | 2.5 | encode and realize workflow injection |

As shown in Figure 5, I begin by giving working definitions of the key PDD concepts of workflows and requirements (in Sections 2.1.1 and 2.1.2). Sections 2.2 through 2.5 address the contributions listed in Table 1. I end with Section 2.7, where I present a summary of how well these contributions address late binding, where gaps exist, and give a rationale for the capabilities developed in PDD as described in Chapter 4 and implemented in Chapter 5.

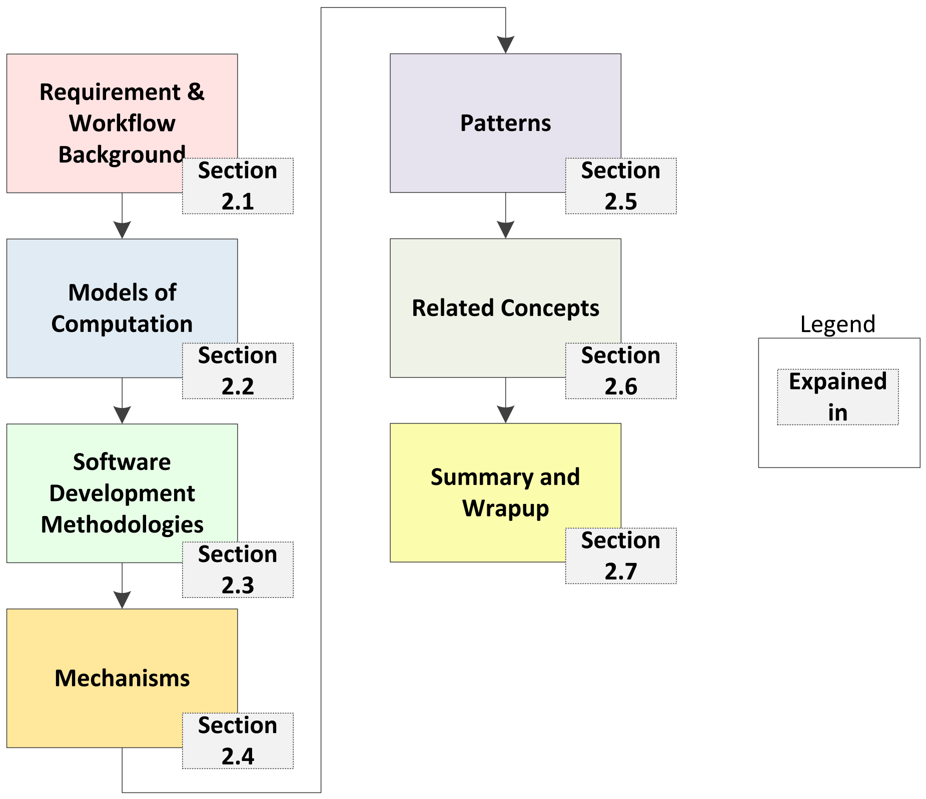


Figure 5. Chapter 2 Flow

I evaluate each contribution according to how it could support the runtime injection of workflows (as representatives of requirements) and associated state, particularly regarding the implementation of key capabilities:

- selection and execution of a workflow in a given context
- composition of one workflow onto a base workflow
- decomposition of a workflow action into a finer grained workflow
- composition of workflows onto composed workflows
- composition of multiple workflows in a given context
- maintenance of persistent state by or amongst composed workflows
- abstractions that enable the specification of workflow selection and composition, focusing on those that avoid entanglement of otherwise separate concerns

- reflection features that enable workflow selection and composition to be specified at runtime
- simulation of composed workflows and proof of relevant properties
- deployment of workflow selection and composition

I identify 17 different facets of this support, organized for convenience into five groupings:

**Workflow specification:** the ability to specify a sequence of calculations as a unit separate from other workflows (*separate workflows*). A *workflow interface* is an abstraction that specifies semantics, and may be expressed as dependencies (e.g., typed parameter lists) and results (e.g., typed return values) independent of the means by which results are achieved. *Contract enforcement* is a mechanism that verifies that interface pre-conditions and post-conditions are met.

**Workflow Injection:** the ability to compose a choice (including a decision and alternate workflows – *workflow selection*) into an existing workflow at a *specified site*. *Choice on choice* is the ability to compose a choice onto an injected workflow. *Composite choice* is the ability to evaluate multiple choices composed on the same site. *Centralized concerns* is the ability to compose one or more choices pertaining to a concern without incurring scattering or entanglement, and *visualization* is the ability to inspect and understand the composition relationship of two or more workflows.

**State Management:** the ability to reference and modify state whose context is unique to a workflow instance (*workflow unique*), shared amongst workflow instances (*instance shared*), global to all workflows (*global*), and to define state lifecycle that depends on its use (*custom lifecycle*) (e.g., dynamic allocation and deallocation).

**Verification and modeling**: the ability to verify that a policy implements a requirement for all workflows (*verification*), and the ability to demonstrate system properties in the presence of injected workflows (*model checking*).

**Deployment:** the ability to deploy choice reliably and authoritatively (*secure deployment*), and the providing a guarantee that a consistent set of policies is in use (*consistent deployment*).

Note that while Figure 3 establishes the entities and relationships at stake, it leaves open the many possible strategies and details of how and when to instantiate and evolve them.

Note that while this section discusses existing contributions that might apply to late binding, there are a number of related contributions that are either subsumed by contributions described in this section or are tangential to them. For completeness, I address a number of these in Section 2.6.

For background specifically on computational models, software development methodologies, and other existing contributions, see Appendix B.

## 2.1 Background

In this section, I describe how workflows are appropriate abstractions for implementing single requirements or collections of requirements in the context of computing systems, and that additional or changed requirements (including crosscutting concerns) can be implemented as workflows composed onto these base workflows. This discussion frames both the investigation of existing contributions in this section, the presentation of PDD foundations in Chapter 4, the case study described in Chapter 5, and the comparison of PDD relative to existing contributions in Chapter 7.

At a high level, a *policy* can be considered to be a rule that defines how or when such a composition occurs, and may result in the addition of a new workflow to a base workflow, or the replacement of the base workflow with the new one. More concretely, a policy consists of a choice (or decision) that results in the insertion of a workflow into another workflow, where the choice is based on the state of the computing system, and it may select the execution one of a number of candidate workflows in addition to or instead of the base workflow. The term I use to encompass this choice and the resulting workflow is *policy injection*.

The speed and efficiency of computer system evolution are strongly influenced by the process by which requirements are mapped to workflows and are then composed onto base workflows – these factors are key in maintaining high stakeholder productivity and satisfaction. In the following sections, I survey several approaches to realizing new and changed requirements as policy-mediated workflows, focusing on minimizing the time required to deliver implementation of new or changed requirements to stakeholders, and maximizing the likelihood that the implementation will address the stakeholders' true requirements.

### 2.1.1 What is a Requirement?

As defined in [36], the primary measure of the success of a software system is the degree to which it meets the purpose for which it was intended, and the purpose of the study of software requirements engineering is to:

- identify the stakeholders and their needs (either stated or implied, including constraints)
- model, document, and communicate these needs to interested parties (including implementers)
- detect and resolve conflicting needs
- participate in the verification and validation of designs and software that implement requirements

Additionally, requirements engineers must reprise these activities as the stakeholder population and its requirements evolve.

Commonly, requirements are categorized as functional requirements and non-functional requirements, though other taxonomies exist [37]. Functional requirements (FRs) relate inputs to outputs using some transformation, and can be expressed in a number of ways, including user stories [38], use cases [39], UML actor diagrams [40], formal documents [41], business process modeling [42], and many others. Non-functional requirements [43] (NFRs, or quality requirements; for example, information assurance [28] requirements such as security, safety, accuracy, confidentiality, privacy, and availability) often express constraints on the inputs, outputs, and realization of functional requirements, and are often stated late in the requirement elicitation process. NFRs often crosscut either FRs or other NFRs.

Considering that software systems are designed to realize requirements, discovery of FRs and NFRs are primary drivers of software cost and delivery time – in incomplete or late requirement discovery can lead to retracing of development steps, leading to cost and schedule overruns where the costs depend on the development process in use. For example, late requirement discovery in a waterfall process [44] may cause the rework of significant design and implementation work. Other processes anticipate the discovery of new requirements and the refinement or evolution of existing requirements, thereby reducing the cost of late requirement discovery: Agile processes [45] decrease the time required to address such requirements by producing frequent, incremental releases; and spiral processes [46] [47] partition the development process into stages organized as a restartable pipeline, which leverages and reuses work previously done. (These processes are not mutually exclusive, and a particular development may take advantage of the strengths of different processes at different times.)

Numerous strategies [48] [41] have been created for the early elicitation, capture, and tracking of both FRs and NFRs. Particularly, elicitation of NFRs are notoriously difficult because they are often incidental to FRs, are difficult to quantify and test, must often be inferred during FR elicitation, or must be gathered using processes tailored to the application development process. Among such tailored processes, [49] integrates a NFR-oriented view into UML class diagrams for an application's logical model, thereby discovering relationships between FRs and trust, entitlement, decision points, and sources of authority, leading to improvements in use case and logical model coverage. [50] applies to early elicitation phases, and seeks to identify NFRs by discovering *unwanted* behaviors using *misuse* cases to identify triggers, assumptions, preconditions, threats, mitigations, and risks so as to account for them early in the application design. Other approaches elicit trust models [51], testing strategies [52] [53], model-aligned security [54] [55], and access control [56].

Elicitation processes that address domain-specific concerns (e.g., [49], [51], [54], [55], and [56]) produce requirements that can be considered as aspects composed on a requirement base [57] conceptually similar to the composition of aspects in Aspect Oriented Programming [32] [58], architectural connectors onto architectures [59], and features in Feature-Oriented Software Development (FOSD) [60] [61] [62] [63] [64]. In each case, the composition of requirements, architectures, features, or code contemplates the integration of two concerns, where I identify one of the concerns as the *base* or *target concern*, and the other as the *injected concern*.

Whereas the base concern is defined by a subset of FRs fundamental to the application or a class of applications, an injected FR or NFR embellishes the base concern by constraining the base concern's behavior, adding behaviors to it, or both. When injected requirement domains are agnostic as to the specific requirements or functions of a base set (for example, encryption between two points), they can be conceptualized independently of the base set except for associating them with particular requirements. In contrast, requirement domains may interact with base requirements (for example, fault detection or access control) by depending on contextual information consequent to base requirements or supplying context relevant to base requirements. Similarly, injected requirements may interact with each other.

Ultimately, the objective of requirement elicitation is the construction of high level workflows that represent entities, activities, structure, and relationships embedded in the requirement set. Choosing base concerns relative to the requirement set carries over to defining base workflows, and similarly injected concerns correspond to injected workflows. Using the workflow analogy, concerns representing aspects of requirements can be refined into collections of *sub-concerns* having well-defined relationships between them.

Given a set of concerns, the problem of choosing the base concern is closely related to the well-known problem of the dominant decomposition [65]. Choosing one concern as the base drives the definition of injected concerns relative to the base. To the extent that injected concerns compose cleanly onto the base, the underlying system can be understood, built, tested, and maintained efficiently in modular fashion. Strict modularity is violated when an injected concern depends on some property of the base concern or it must be injected onto multiple sub-concerns of the base. Such injected concerns are called dependent and *crosscutting*, respectively. A poor choice of base concerns leads to many dependent and crosscutting concerns, resulting brittle and entangled systems. Discovery of crosscutting concerns late in the requirement elicitation process often causes extensive design rework in redefining the base concern (and injected concerns) to maximize modularity and minimize entanglement.

Additionally, the detection and resolution of conflicts between requirement sets is an active area [66], as is the detection and resolution of feature interactions in FOSD [61]. Furthermore, complex requirements can be decomposed into simpler requirements using various strategies [39] [67] that ultimately produce sets of possibly interacting requirements. Given a set of finely grained requirements or features, complex requirements and features can be constructed using compositional calculi [68] and goal orientation [67]; finely grained requirements and features can be reused in multiple compositions.

Changes to requirement sets can take a variety of forms, ranging from adding new concerns, removing concerns, or altering existing concerns. Such changes can be reflected in existing workflows by altering the workflows to reflect the new requirement base, or by composing new workflows onto the existing workflows, where the resulting workflows provide additional functionality or replace existing functionality.

### 2.1.2 What is a Workflow?

There is wide disagreement regarding the definition of a workflow, as the term serves a number of interests and purposes. A simple definition compatible with the viewpoints of most interested parties is presented by the creators of the Orc [69] workflow orchestration language:

> *A workflow consists of a set of activities generating output in the form of data or events which may trigger further actions. These activities can be executed in sequential or parallel order.*

In Orc, data is an abstract container passed between activities, and an event is a container with no content.

I adopt YAWL's [70] complimentary definition of activity:

> *A description of a unit of work that may need to be performed as part of a workflow*

A workflow activity can be characterized as a computational link between pre-conditions and post-conditions linked to the fulfillment of one or more requirements [71].

I observe that a *procedural* view of a workflow is an orchestration of activities that satisfies the data flow and control dependencies for a task [72], and that an activity itself may be implemented as a workflow, which fulfills the workflow activity's pre- and post-conditions.

Equivalently, I observe that a *functional* view of a workflow is a relation that transforms one set of inputs into a set of outputs. Conversely, a set of input channels is related to a set of output channels via a transform represented by a workflow -- the main function of a workflow is to pair valid inputs with valid outputs.

Whether the workflow definition drives the input and output channel definitions, or vice versa, the result is the same -- the procedural and functional perspectives are duals of each other.

As in [71] [73], the relationship between activities can be modeled as a graph, where each activity is represented as a node, and an edge connects two nodes iff one node (called a *source*) emits data that the other node (called a *target*) consumes and acts upon. Therefore, the target depends on data from a source – an edge is directed, and indicates the direction of flow from source to target. A source can be defined to connect to one or more target, and can be defined to emit data to some or all of its targets simultaneously. Similarly, a target that accepts data from multiple sources can be defined to execute upon the receipt of data from some or all sources. Any target receiving data executes in parallel relative to other targets receiving data. A *choice* node is a source that is defined to emit data to some targets and not others based on some criteria evaluated at runtime.

Composition and decomposition of activities are described by [69], [73], and [74], where an activity can be decomposed into a self-contained workflow (called a *subflow*) or can be an atomic operation. Conversely, a workflow can be considered an activity in another workflow. As such, workflows follow a Composite pattern [33] (as briefly described in Appendix C) where an atomic activity is the leaf, a workflow is the composite, and an activity is the component. Workflows can also be considered independent objects (as in a workflow management system [75]) and can be reused as components of other workflows. Given that a particular workflow represents a process at a given level of abstraction, the decomposition of an activity represents the refinement of the workflow via increased activity granularity, with data flows defined accordingly.

This general description of workflows is common to many workflow models (e.g., UML Activity diagrams [40], Business Process Modeling Notation (BPMN) [76], the Action Port Model [77], and Petri Nets [78]), where each model provides embellishments (e.g., exception handling, typed data exchange, interface definitions, and grouping notations) and naming conventions (e.g., fork-join, and-split, and-join, or-split, and or-join) that facilitate and emphasize different abstractions important to different communities (e.g., visual modeling, reduction and correctness proofs [79], and liveness and safety proofs).

Generally, the connections between nodes and the behavior of each node are defined at application design time pursuant to application requirements. Cyclic paths represent loops and exceptions, and a particular workflow can often be transformed into equivalent workflows that are semantically identical but are simpler to analyze or maintain. In [74], rules are given for normalizing workflows by eliminating redundancies and for constructing views as workflow subsets, though such transformations may produce misleading semantics if dynamic workflow behavior is not accounted for.

Insofar as a workflow activity is defined by its inputs, outputs, and action, one version of an activity is interchangeable with another so long as both versions accept the same inputs and produce the same outputs – their actions need not be the same. Furthermore, there is no restriction on the external resources (e.g., state) a workflow activity can access or change in order to perform its function, though some workflow systems attempt to document or constrain such access. Considering that a workflow activity can, itself, be decomposed into a workflow, the same relationships hold for an entire workflow: a workflow can be characterized by its inputs, outputs, and action, and can be replaced by an equivalent workflow.

Note that while a workflow identifies data flow and control flow dependencies, its definition allows for the existence of multiple simultaneous workflow instances, each processing different or similar data. The state of a workflow instance is composed of a) the data flows and control flows spawned by an initial workflow activity, and b) a collection of workflow state variables. Workflow state variables come into existence upon the execution of an initial workflow activity, are available to constituent activities, and are extinguished when all constituent activities have ceased.

For acyclic workflows, the dependency relationships between activities are apparent by inspection. This may also be true for workflows containing cycles, though cycles introduce ambiguity as to valid sequences of data exchanges between activities. State machines may be used to specify constraints on the ordering and content of data exchanges, thereby disambiguating the workflow. The sequence of valid exchanges amongst activities is a *protocol*.

Based on this, I observe that a workflow activity and a workflow are interchangeable, and that the SOA definition of a service fits both definitions, too. Consequently, I use the terms *workflow*, *workflow activity*, *relation*, and *service* interchangeably as duals where:

- ***Workflow*** emphasizes a description of data and control flow, as is useful in the business domain
- ***Workflow activity*** emphasizes a component of a workflow that may be decomposed
- ***Relation*** emphasizes the formal properties of workflows as transformations of a domain (inputs) into a range (outputs)
- ***Service*** applies to an implementation domain (e.g., SOA), and emphasizes interface specifications, protocols, and functionality

## 2.2 Models of Computation

Models of computation [80] present the principles important in the realization of a calculation without attending to details of concrete implementation – they include *machines* and *process algebras* [81].

Machines (described in Sections 2.2.1 and 2.2.2) represent a set of rules that combine to realize a computational result.

Process algebras [81] (described in Sections 2.2.3 and 2.2.4) are logic systems that enable a rigorous characterization of a workflow for the purpose of reasoning about their behavior, equivalence, and properties [82]). They describe both serial and parallel execution in small and large systems, focusing on interactions between processes, including processes distributed across a computing network. Considering that a workflow can be viewed in terms of both control flow and the transitions of ephemeral or persistent state, process algebras support reasoning about both control flow and state maintenance. A workflow can affect state private to a single workflow instance, shared amongst multiple workflow instances, or shared amongst different workflows. Process algebras may support any or all of these modes.

A number of computation models attempt to describe workflows, each from a different perspective, and many computation models provide foundation for other computation models.

In this section, I describe prominent computational models, and discuss how they address (and fail to address) decision making and workflow composition (including associated state management), which are central requirements of PDD's policy injection. The models I choose are foundational for classes of machines and process algebras that focus on describing control and data flow, specifically pertaining to addressing computations, process, and structure. Other models address abstractions that don't apply directly to PDD-based policy injection (e.g., StackAnalyzer and SCR) – [83] presents a list of over 100 process algebras.

### 2.2.1 Turing Machines

A Turing machine [84] is a very basic process description briefly explained in Section B.1.1. The concept of workflow state is related to the concept of a Turing machine state, but they are not the same. Abstractly, a workflow state exists as parameter values referenced by workflow actions and decisions. As Turing machines do not directly encode many parameterized actions and decisions, a state machine definition contains (often very large and complex) networks of states to represent the workflow state concept. In essence, networks of Turing machine states are used to represent combinations of workflow control flow and state values. As such, the realization of an application requirement may be encoded as a Turing machine, but would not be easily written or maintained.

A decision in a Turing machine is represented by a transition function, where a new state is chosen (deterministically or not) based on the current state and the current input. Intuitively, composition of two Turing machines (representing two workflows) can be achieved using algorithms as in [85] and [86]. Hypothetically, while a Turing machine state itself can be decomposed into a Turing machine, there is little, if any, support for this in standard Turing machine definitions.

Requirements are bound into Turing machine programs at the time the machine is created. Because standard Turing machine definitions do not provide facilities for runtime composition of workflows, runtime workflow injection is not supported.

### 2.2.2 Petri Nets

A Petri Net [78] [87] is a graphical notation briefly explained in Section B.1.2. Under Petri Nets, a transition is an abstraction of a decision based on abstractions of conditions (as input places) and producing abstractions of results (as output places), with all input and output places representing roles relative to the decision. A place abstraction can represent state shared between workflow instances or amongst all workflows, but does not conveniently represent state for a single workflow instance. A place can, itself, be decomposed into a lower level abstraction implemented by a Petri Net, where each of the place's incoming and outgoing arcs are mapped to places in the lower level Petri Net [88]. This amounts to an encapsulation that enables modeling of state as workflow and vice versa.

Using the service algebra proposed in [89], Petri Nets can be composed by using the following composition operators: ordered sequential, unordered sequential, alternative choice, iteration, parallel (with communication), join, service selection, and refinement. With the exception of service selection and refinement, each operator gives rules and semantics for combining two independent workflows.

The service selection operator is defined to choose amongst alternative workflows according to an independently specified selection function. In [89], the selection function is defined to operate in a Web Services context; accordingly, the selection operator queries each workflow for information about itself, and then calls the selection function to evaluate the workflow properties and choose a workflow. In a more generalized operator, the selection function would base its decision on whatever data the Petri Net runtime system might expose to the selection function.

The refinement operator is defined to substitute a workflow for an operation represented by a transition, thereby creating a specialization of the original workflow via hierarchical composition. This is accomplished by mapping the transition’s input and output places to input and output places within the substitute workflow (as distinct from [88], where a place is replaced.) This enables a base workflow to act as a template for a class of workflows. When used with the service selection operator, the refinement operator allows the selection from an array of workflow alternatives derived from a workflow template, which defines the workflow interface.

Both place- and transition-based refinement are accomplished via the mechanical manipulation of the Petri Net graph, so there is very little information on which to judge semantic compatibility of the refinement workflow as compared to its injection site. A refinement operation seeks to maintain arc connections without regard to the associated place or transition semantics.

Processes encoded as Petri Nets often integrate multiple concerns that themselves can be modeled by separate Petri Nets. The challenge of integrating separate concerns expressed as Petri Nets is partially addressed by the refinement operations proposed by [88] and [89], where [88] replaces a place (representing a data flow) with a workflow, and [89] replaces a transition (representing a control flow). In either case, they inject a single workflow at a single location in a base workflow and do not address the issue of injecting multiple workflows coordinating activities via shared state. View-oriented composition and decomposition techniques proposed in [74] address this, but for lack of a convenient integration language, they fail to create an integration path that itself is intelligible and maintainable in a programming setting.

Petri Nets (including the extensions described above) are statically defined process models, and as such, implementation of new or changed requirements involves a manual modification and release cycle. While the combination of the selection and refinement operators enable runtime decisions to determine actual workflows executed, these operators are placed statically in a workflow and provide no facility for after-delivery decisions that define which workflows to inject and where to inject them.

While Petri Nets and related service algebra can express and compose complex, stateful process flows, they are poor abstractions for application authorship, as the abstractions of place, arc, and transition are too fine grained for general programming. Additionally, they lack contract enforcement (e.g., type and range safety), organizational abstractions (e.g., classes and modules) crucial for efficient system development and maintenance, state maintenance for workflow instances, and convenient features for mathematical calculation. However, Petri Nets are useful for modeling concurrent systems and are well supported by modeling and simulation tools. For instance, by evaluating all possible markings for a Petri Net, simulation tools can determine liveness (where there is always at least one transition that can fire), safety (where no place ever contains more tokens than it can hold), and domain properties of interest as constrained by limits of time and space related to evaluation state space explosion.

Petri Nets have been extended in numerous ways, including facilities for modularization. Notably, Colored Petri Nets [90] (CPNs) enable types and data content attributes for tokens, enable types for places, allow conditions to specify whether an arc incoming to a transition can contribute to the transition being enabled, and allow expressions that compute tokens outbound from a transition. Consequently CPNs are much more expressive than Petri Nets, but for the purposes of refinement and runtime workflow selection and composition, they have the same characteristics as basic Petri Nets.

### 2.2.3 π -calculus

π-calculus [91] is a process algebra briefly explained in Section B.1.3. Under π-calculus, decisions are modeled incidentally as conditionals executed within a process. A decision that chooses between workflows chooses between multiple process or channel references present in a message (already received by the process) [92], or sets a process reference or channel reference contained in a message (which is passed to a waiting process). While its dynamic channel and process definition features enable clean implementation of a Strategy pattern [33], the universe of available strategies are those processes explicitly authored prior to program execution – they implement primitive forms of functors and polymorphism.

While the channel abstraction is a convenient and flexible implementation of a data structure, π-calculus relies on derived languages (e.g., Pict [93]) to define persistence, encapsulation, or life cycles necessary to create separate, stateful concerns. Notably, [94] proposes the Piccola composition language, which seeks to create a new component by encapsulating an existing component (using so-called *composition scripts*), and supports the composition using Active Objects (which represent stateful, concurrent, distributed, and mobile processes), components, inter-object glue, mappings between object models, message-level reflection, and a high level syntax. Piccola envisions employing these features to define agents and enable their interaction; it does not envision the definition and runtime composition of agents foreign to the application.

Because channel transmission and process invocation in π-calculus are bound at compile time, they represent instances of early binding not amenable to either composition of orthogonal concerns or the runtime definition and replacement of channels and processes. Additionally, channels modeled by π-calculus define the act of content transmission as atomic, thereby limiting opportunities for workflow injection at runtime via interception of messages in flight. Consequently, the implementation of new or changed requirements in π-calculus expressions (and derived languages) involves a manual modification and release cycle. Furthermore, because implementation of independent requirements may require encapsulation transformations (per [94]), the implementation of more than a few such requirements may lead to significant application-wide entanglement.

Note that [94] observes that for an encapsulation-style composition, channel values contributed by the core component must be transmitted alongside of values contributed by the encapsulating component, yet must remain separate so as to maintain their separate concerns. This problem is solved in [94] by introducing separate name spaces for the two. Furthermore, state shared amongst multiple instances of the same workflow, and state global to all workflows can be implemented via channel references embedded in messages.

While π-calculus does not support PDD's concept of runtime-defined workflow injection, it offers insights into the features needed to support the composition of separate concerns, including the management of concern-related state both during workflow execution and across workflow executions, state exchange between workflows, and syntax suitable for expressing composable concerns.

### 2.2.4 *λ*-calculus

λ-calculus is a process algebra briefly explained in Section B.1.4. Under basic λ-calculus, a workflow is created as a consequence of the expression reduction process, where a function can be reduced only if its arguments have already been reduced to constants – if an argument is a function, its usage must be reduced to a constant before its value can be used. Choice is encoded as three clauses: a "decision" function, a "then" function, and an "else" function. The "decision" function evaluates to either a "true" function or a "false" function; the "true" function returns the "then" function, and the "false" function returns the "else" function. The function that is ultimately returned (by the "true" or "false" function) is the result of the choice, which is then further evaluated. The "decision", "then", and "else" functions can be supplied as part of the authored application, as a text string assembled at runtime, or as a text string fetched at runtime from external store. As such, while the choice calculation can be specified either at the time of program authorship or at execution time, the actual decision site (and the program flow that results) is bound at execution time.

Variables in λ-calculus exist to bind expressions and have scope local to an enclosing function (and whatever functions it defines). Consequently, basic λ-calculus simulates global state by using local scoping, and functions cannot have side effects that equate to changing this state. Practically speaking, this defeats the modularization that is necessary for decomposing an application into maintainable and reusable objects [25] [95], including process and agents. Additionally, λ-calculus has no concept of concurrent execution distinct from serial execution, as its main focus is on reducibility. Consequently, it provides no features addressing synchronization and variable consistency. Extensions to λ-calculus add support for concurrent execution

[96]. Functional languages such as Schema, ML, and Clojure add support for global-, module-, and thread-based variable scope, functions with side effects, and support for synchronization and variable consistency.

Given this support, functional composition (including workflow modeling) is inherent in functional languages – combining two functions can be as simple as passing both to a third function, which then reduces the functions and combines their results. However, because variable binding and function calls are atomic at runtime, composition ultimately results from early bound decisions, and ad-hoc runtime composition of orthogonal concerns or runtime replacement is difficult. Consequently, the implementation of new or changed requirements in λ-calculus expressions (and derived languages) involves a manual modification and release cycle. Even though functional languages such as ML and Clojure provide encapsulation and polymorphism in the style of object orientation, integration of independent requirements into existing functions may still lead to significant application-wide entanglement.

Under basic λ-calculus, higher order functional programming is enabled by allowing variables to contain functions, functions to be passed as parameters to other functions, and functions to return functions. This enables and leads to various strategies for expressing workflows more flexibly and readably than in other calculi. Notably, monadic programming [97] [98] focuses on λ-calculus-compliant techniques that implement workflows and support exception handling, sharing of state, mutable values, input/output, and non-deterministic choice. The main intuition in monadic programming is that instead of a function $f$ returning a result $\alpha$, a function $f$ returns a proxy function $g$ that returns the result $\alpha$, and $\alpha$ is bound to a variable internal to $g$. Therefore, $g$ can return $\alpha$ on demand, but can also return other values $\beta$ bound to other variables internal to $g$, including error information, state information, logging text, and alternate results. Ultimately, a monadic expression (called a *monad*) can be transformed into a non-monadic λ-calculus expression, though one that would be difficult to read or maintain.

An example of monadic programming in Clojure is [99], which demonstrates a simple workflow that accumulates a log as it executes. Workflow steps are orchestrated by a *decider* function (called *bindM* in [97]), which interacts with the proxy function ($g$) to set and fetch its variables, including both function results ($\alpha$) and internal variable ($\beta$). While the decider function must execute the workflow steps as intended by the workflow author, it also interacts with the proxy function (or other functions) to implement either related or orthogonal concerns – such functions can be either first order or monadic in nature. Depending on the decider and how it interacts with the proxy functions, a decider can compose workflows onto a base workflow, cause replacement of steps in a base workflow, or inspect or alter values passed from one workflow step to another.

Because of the scoping rules of λ-calculus, state maintained by a monad can be defined either within the lexical scope of the monad or within the scope of the proxy functions it engages. State can be shared between monad executions by either executing the monads within an outer scope or by storing and retrieving it via an external entity (though, this would violate the intention that λ-calculus be free of side effects). Consequently, state can be shared amongst activities in a workflow, multiple instances of a workflow, or all workflows.

While monads offer powerful ways of implementing stakeholder requirements by combining otherwise independent workflows or altering existing workflows, such monads (particularly the decider and proxy functions) are subject to the same early binding limitations as basic λ-calculus.

## 2.3 Software Development Methodologies

Anecdotally, over time, stakeholders' expectations for the delivery time and maintenance cost for software systems has trended downward while their expectation for attention to NFRs has increased. Programming concerns were responsible for the development of early methodologies, including modularity, structured programming, and object-oriented programming. Recognition of the increasingly difficult and important task of requirement elicitation and management (reflecting more sophisticated and complex demands by increasingly diverse stakeholder communities) has helped drive modern methodologies, including object-oriented programming, and aspect-oriented programming.

In each case, the practice of existing methodologies expects that system design and software coding are performed early in the authorship process, and lead to discrete testing and delivery phases. In this section, I describe major methodologies that represent the mainstream of the evolution of software engineering, and focus on the degree to which each supports binding workflows late in the authorship process.

### 2.3.1 Modular Programming

As described in Section B.2.1, modular programming [100] enables the creation and maintenance of systems larger and more complex than the monolithic programming style of its time, it envisions a problem decomposition that is strictly hierarchical. As such, the implementation of crosscutting concerns eliminates many modular programming advantages by entangling separate concerns throughout a code base. Furthermore, the focus of modular programming is to ease the programmer's job, without attending directly to the definition or implementation of stakeholder requirements. Consequently, requirement realization is subject to a process where the programming and re-delivery activity is the bottleneck.

Though not envisioned in [100], an application can be updated or upgraded at runtime using techniques such as dynamic binding (e.g., Dynamic Linked Libraries), which enables alternative implementations of module-level abstractions. With sufficiently flexible interfaces and deployment planning, this enables programmers to replace a portion of an application without affecting the remainder of the application.

However, this technique serves new and changed requirements only to the extent they can be implemented as variations of existing modules orchestrated in existing workflows, and is subject to programming team bottlenecks. It does not bear directly on the realization of new requirements composed upon new or existing workflows, and does not avoid the programming and re-delivery bottleneck.

### 2.3.2 Structured Programming

As described in B.2.2, a major benefit of structured programming is the management of complexity of large applications via a divide-and-conquer approach. However, as with modular programming, it provides little support for the incorporation of crosscutting concerns (without entanglement), and is focused on binding requirements early in the application authorship process. Structured programming techniques can conflict with or support modular programming analyses, and can be used effectively to inform a fruitful modularization of application code.

Note that while structured programming does not address or provide mechanisms for late binding of code or requirements, it provides an analysis framework that justifies the injection of a decision as a choice coupled with a workflow. In a top-down analysis, it enables reasoning about where such an injection can occur; in a bottom-up analysis, it enables reasoning about the effects of an injection.

### 2.3.3 Object Oriented Programming

As described in Section B.2.3, Object-oriented programming [101] (OOP) uses of interfaces to define an object and uses runtime call resolution to promote the loose coupling of objects both at design time and at runtime.

While the mechanisms and benefits of OOP as compared to modular programming are extensive, a key enabler of workflow expression is the object interface definition, which has a similar form and function to interfaces found in modular programming. Additionally, data encapsulation and multiple object instantiation support enable the creation of multiple, independent workflows, thereby supporting the decomposition abstraction of structured programming.

OOP's polymorphism and runtime call resolution support the design time and runtime selection of one workflow over others. At design time, given the existence of multiple objects implementing a common interface, one workflow is chosen over another by explicitly instantiating an object corresponding to the workflow – leveraging the common interface allows the exploitation of common semantics, and leveraging polymorphism allows the exploitation of common base functionality. At runtime, the choice of workflows occurs via subtyping, where fungible workflows are represented by a base class, which resolves to a real object when accessed at runtime. (Commonly, a Factory pattern [33] is used to realize such an object.) As such, this subtyping implements a choice, where the decision is made during instantiation in the factory, and the workflow is executed at runtime when its method is called – this amounts to a distributed Strategy pattern [33] (as briefly described in Appendix C).

As with modular programming, the focus of OOP is a hierarchical decomposition of abstractions, which can be discovered using Object

Oriented Analysis and Design [102] and Model Driven Design [103] techniques. Additionally, the decomposition ultimately encodes workflows, expressed as calling relations between objects. While OO processes and features enable design, deployment, and maintenance of larger, more complex applications that satisfy requirements from larger and more diverse stakeholder populations, the implementation of crosscutting concerns leads to scattered and entangled code bases just as with modular programming.

The combination of OOP's data encapsulation, object instantiation, and subtyping facilities creates the means to associate state with workflows both for the lifetime of the workflow, for access by groups of workflows, and for access by all workflows. Additionally, the dynamic creation and destruction of objects enables state lifecycles customized to OOP-based workflow abstractions.

While the OOP interface and messaging paradigm combine to create loose coupling between objects at both design time and run time, such relationships are established by programmers, are essentially bound in the programming process, and ultimately represent requirements that are heavily refactored during the OOP design and implementation process. Additionally, the hypothetical exchange of messages between object is most often implemented as function calls not subject to ad-hoc interception. Consequently, as with modular and structured programming, the realization of new and changed requirements occurs only through programming and re-delivery activities, which remain bottlenecks.

### 2.3.4 Aspect Oriented Programming

As described in Section B.2.4, Aspect-oriented programming (AOP) [32] has a number of implementations, each defining its own join point model, pointcut language, and advice language, and the particular capabilities of AOP vary with the implementation. For example, under AspectJ, it is possible to specify the lifecycle of variables declared private to an aspect: persistent across all executions, associated with an advised object, and persistent across a control flow.

The use of AOP improves maintainability of complex code bases by maintaining separate concerns as distinct aspects, thereby promoting reasoning about each concern in isolation, and avoiding expensive re-modularization that occurs when entangled concerns are (manually) injected. The decision regarding whether to compose advice into a workflow is made by an *aspect weaver* – it matches aspects' pointcuts to target workflows, and injects advice at matching locations.

Most AOP implementations perform weaving during the compile phase prior to code deployment. As with OOP, the implementation of requirements via AOP composition and the relationships between aspects and base workflows are defined and maintained by programmers, and the realization of new and changed requirements occurs only through programming and re-delivery activities, which remain bottlenecks.

Furthermore, the high specificity of pointcuts enables the composition of advice throughout base code, which creates de-facto interfaces on which advice relies, but which can be easily and accidentally perturbed by common code maintenance activities performed by oblivious base code programmers – the result is invalid aspect compositions that result in application breakage that can be difficult to discover, diagnose, and repair [104]. Additionally, as either base code or aspects evolve, the independence of aspect code (relative to the base code and other aspects) can make a comprehensive understanding of an application more difficult than with non-aspect implementations – casual inspection of an advised code base does not reveal the capabilities of the aspect-woven application. While some AOP communities have created visualization tools [105] that inform interactions between base code and aspects, these tools are not available for all AOP implementations.

Experimental approaches to weaving create base/aspect compositions either at program load time [106] or dynamically [107] [108] [109], while the code executes. The load time approach results in the fulfillment of aspect-implemented requirements only by restarting all or part of an application. Both approaches require the intervention of programmers intimately familiar with the base workflow code and fluid and loosely defined interfaces – all of which preserves bottlenecks caused by the programming process.

Finally, while AOP provides for the composition of workflows onto a base workflow, it does not provide for composition of aspects on aspects, as discussed in [110]. Consequently, aspect weaving does not address the composition of requirements upon requirements.

AOP is one facet of an overall philosophy that maintains separate and crosscutting concerns distinct from base concerns, thereby enabling reasoning about individual concerns distinct from other concerns. It represents programming design and implementation strategies supported by Aspect Oriented Requirement Engineering (AORE) [111], Aspect Oriented System Architecture (AOSA) [112], and a host of similarly motivated activities aimed at different facets of application development (e.g., Aspect Oriented Software Design (AOSD) and Aspect Oriented Design (AOD)). In particular, AORE addresses requirement decomposition and factoring, and leads to the formulation of corresponding base and separate workflows that enable PDD to leverage dynamically composed workflows effectively.

## 2.4 Mechanisms

While the basic pattern for a choice is a decision and a consequent workflow, the concept of choice is so basic to both imperative and functional programming that it occurs in nearly all designs and programming constructs. In this section, I survey major choice mechanisms, and examine how they relate to the realization of workflows injected at runtime. First, I inventory simple and fundamental mechanisms (in Section 2.4.1) that serve both the imperative and declarative styles (in Section 2.4.2). Then, I describe how choice is injected into workflows at runtime as managed by execution frameworks (in Section 2.4.3) and assisted by policy engines (in Section 2.4.4). Finally, I show how workflow context is maintained in common distributed systems (in Section 2.4.5).

### 2.4.1 Fundamental Mechanisms

The most basic forms of a choice are represented by the conditional branch or jump table usually present in machine-level instruction set abstractions. In modern imperative languages, they have the form of `if()`, `switch()`, `goto()`, and related statements; they represent a close coupling between such decision predicates and an associated workflow. In the tradition of CSP's [113] external choice, predicates are based on parameters, local variables, global variables, application state, environment state, property files, database fields, or calculations based on these values.

The idea of choosing a workflow based on decision predicates serves both algorithmic function (as might occur in a bubble sort) and in the composition of separate concerns. My dissertation pertains to the latter.

Examples of values that drive predicates to select amongst workflows at runtime include:

- Constants available at compile time
- Attributes in directories (e.g., permissions, owners, and groups for files in a file system)
- Permissions in property files (e.g., Tomcat's catalina.policy file per JAAS [114])
- Attributes and permissions in registries (e.g., Microsoft's group policies present in an Active Directory, Facebook's privacy settings, and Oracle database permissions)
- Attribute Based Access Control (ABAC ) and Role Based Access Control (RBAC) [115]

Predicates based on resources external to an application are a common means of selecting amongst workflows. However, predicates encoded into conditional branches and jump tables are defined at the time of design and coding, as are the workflows they select. Consequently, they cannot be changed at runtime to reflect new or changed requirements.

Note that many choices pertain to controlling access to a particular resource (e.g., a file or process). Access control predicates often reference identity proofs (e.g., credentials) authenticated by some authority. The process or details related to generating these proofs are not part of this dissertation.

Other choices pertain to the selection of a workflow appropriate to a configuration or circumstance (e.g., look and feel driven by Windows' group policies).

### 2.4.2 Declarative Representations

Because of the close connection between a decision and a workflow, choices are often naturally posed in imperative terms recognizable as flow diagrams.

#### *2.4.2.1 Functional Languages*

However, choice can be represented declaratively, where flow is abstracted as dependence relationships (and inferred by an execution engine), as is common in functional programming languages (e.g., Clojure [116] and Orc [69]). In this context, the realization of a requirement derives from the composition of one function with or onto another, where the composed function represents a decision and an alternative function (analogous to a decision and alternative workflow in an imperative construction).

For example, a choice encoded under Orc can declare a predicate and some number of alternatives. Unlike an imperative interpretation, a declarative interpretation may or may not execute *all* alternatives; based on the predicate, it propagates the result of only one alternative (and assumes that all alternatives are side-effect free). Functions defined in Orc (and Lisp descendants) are specified as data, which may be specified at the design and coding phase or at runtime. Consequently, realization of new or changed requirements in an arbitrary workflow can occur at runtime – existing code (either as source or as an existing function reference) would be composed into a new function incorporating a decision and alternate workflows.

However, to date, such declarative systems often lack a means to specify where in an existing workflow to inject a choice, or a mechanism to form this composition at runtime once a site is identified.

#### *2.4.2.2 Modeling Languages*

Modeling languages are often used as declarative representations of application structure or sequence. Message Sequence Charts [117], for example, define protocols involving role abstractions – choice takes the form of alt blocks, where the decision is a predicate and alternate workflows are sub-protocols. Similarly, diagrams created using Uniform Modeling Language (UML) [40] (e.g., class and object diagrams) allow guards written on relationships between entities – guards are written in Object Constraint Language (OCL), and enable or disable the guarded relationship. Because models usually depict only partial structure or behavior, composition of models to form an executable system is often impossible or undecidable. To the extent that modeling languages are directly executable (as base and injectable workflows that can be composed at runtime), the question of how to combine predicates arises should multiple concerns affect the same workflow. When modeling languages allow the composition of either structure or sequence, one option is to combine such decisions nondeterministically, as an internal choice under CSP, which leads to unmanageable state, entity, and relationship explosions as seen with Turing machines and Petri Nets.

Modeling languages that support imperative representations (e.g., Business Process Modeling Notation (BPMN) [76] and UML via sequence, state, and activity diagrams) represent choice explicitly as decision blocks that encode first order logic to distinguish between alternative workflows, and can be augmented with other logics (e.g., temporal logic [118]). Composition of such models has been demonstrated for UML state diagrams [119] using aspect-oriented techniques, but has not been demonstrated as a runtime technique.

#### *2.4.2.3 Structured Query Language (SQL)*

Structured Query Language (SQL) is a declarative language suited to the manipulation of relational databases (RDBMS). As such, SQL statements fulfill aspects of stakeholder requirements projected onto the database domain. Whereas SQL statements can be statically coded in applications (and are therefore resistant to runtime modification), most SQL engines enable the storage of SQL statements as views residing in the RDBMS. In this form, an SQL statement is available for requirement composition via modifications to its `select` and `where` clauses, which can be performed at runtime. Whereas SQL statements represent partial requirements, the composition of constraints via these clauses is not well supported because of the difficulty of parsing and regenerating SQL statements. Fully supporting requirement realization cannot be accomplished via SQL alone.

### 2.4.3 Execution Frameworks

In this section, I describe how choice and composition are implemented in common execution frameworks that integrate loosely coupled services (and therefore could present policy injection features), or explicity evaluate and enact policy, or both. While such execution frameworks comprise a diverse collection, the frameworks presented below are representative of the class.

#### *2.4.3.1 SQL Engines*

SQL engines evaluate an SQL statement according to a basic three activity workflow: a) client issues a query, b) engine evaluates the query, and c) client processes the results. These engines allow the declaration of constraints and triggers as a means to compose supplemental workflows onto the base workflow. A constraint is a choice composed onto an operation that stores a field value, where the decision verifies that the value is semantically consistent with a column definition, other values in the row, and other rows in the table – the alternate workflow is an exception that indicates a constraint violation. A trigger is a choice composed onto a workflow's interactions (i.e., a-b or b-c), where the decision is based on the SQL operation being performed, and the alternate workflow is a combination of the base workflow and a call to a stored procedure. Ultimately, constraints and triggers support stakeholder requirements, and SQL engines are designed to accept and execute them at runtime, thereby allowing the realization of new or changed stakeholder requirements at runtime. While stored procedures can store and retrieve state, and they can access application data, they are oblivious to the application workflow context from which they're called, and therefore cannot coordinate activities within such a workflow. Ultimately, this workflow injection capability addresses only the data storage aspects of stakeholder requirements, which is usually a small part of an overall application.

#### *2.4.3.2 Enterprise Service Bus*

As described in Section B.2.5, in an Enterprise Service Bus (ESB) [120], a workflow is defined as a routing, which can be associated with a predicate that can enable it (similar to the use of OCL for UML-based sequence modeling) or identify actual routing targets based on information contained in the routed message. Choice can be injected into a base workflow by intercepting a message in flight and using its contents to decide whether to continue the original routing, route to a different target, or both. While an ESB can execute a workflow, it doesn't maintain workflow state or enable decisions based on workflow, environment, application, or other persistent state. Consequently, while ESBs can support the runtime injection of stakeholder requirements that leverage message-based information, additional features are necessary before they can support crosscutting requirements that rely on state having specialized lifecycles.

### 2.4.4 Policy Engines

As described in Section B.2.6, policy engines enable application developers to design decision and execution points into applications while deferring the decision definition until runtime. This deferral represents a separation between workflows and policies (as business rules), and is usually leveraged to implement access control decisions (e.g., denying access to a resource based on identity or environment). Adding or changing a decision point (PDP), execution point (PEP), or alternate workflows requires explicitly recoding and redeploying the application.

### *2.4.4.1 PERMIS (Privilege and Role Management Infrastructure Standards)*

In the PERMIS policy infrastructure [121] (described in Section B.2.6), the PDP and PEP represent decisions and workflow selection logic that is statically placed in a workflow at program development time, and which selects or parameterizes statically defined alternate workflows. As such, decision criteria and workflows are decoupled, thereby providing flexibility in isolating and exposing decision criteria (e.g., RBAC-based access control [115]) for dynamic definition. However, this does not enable policy and workflow injection that implements data flow filtering or feature composition responsive to stakeholder requirements that emerge at runtime.

PERMIS evaluates declarative RBAC-based policy based on first order logic expressed in XML, and PERMIS provides a GUI that assists a policy writer in assembling the different elements in an XML-based policy statement. A PERMIS policy is framed from the standpoint of a security practitioner, and not a domain expert interested in constraining or augmenting workflows. In this regard, PERMIS is representative of other policy systems, both in how they integrate into an application and their approach to policy language – the policy authorship process and the policy language itself generally does not align with a stakeholder's understanding of domain problems, and therefore would not be used to engage a stakeholder in creating and maintaining solutions to domain-oriented requirements.

### *2.4.4.2 BPEL Process Integration with Business Rules*

As described in Section B.2.6, BPEL [122] doesn't address scalability or verifiability, and it doesn't provide a mechanism for composing workflows onto workflows, a dynamic external policy mechanism and support system, or a sub-workflow concept, as PDD does. As a scripting language, BPEL provides no independent methodology, and particularly provides no guidance for integration of crosscutting concerns and features. (BPEL-SPE supports sub-processes.)

Under, Oracle's BPEL Process Integration with Business Rules [123] choice is implemented as a statically coded call to the *decision service* – first, the application declares its state (as *input facts*); second, it calls the decision service to evaluate the input facts according to the business rules; and third, the decision service returns a result (as *output facts*) to the application. Coding input facts, invoking the decision service, and interpreting the output facts are performed via calls explicitly coded into the application. The business rules are defined as Boolean and arithmetic expressions via a GUI wizard, and are stored in a rules repository. As such, the combination of the decision service and a business rule amounts to a service having an interface defined by the input and output facts, which are tightly bound to the application, though the particular rule is loosely bound by virtue of its presence in the repository. A rule can call an external (web services) function, which amounts to a workflow bound at runtime. Therefore, a rule qualifies as a choice, where a decision can select amongst workflows. However, adding, changing, or removing decision service calls requires recoding and redeploying the application. External functions enable policies to maintain their own state and access application and environment state, though workflow state is not automatically tracked.

#### *2.4.4.3 xESB: Integration of Policy with ESBs*

Under xESB [124] (described in Section B.2.6), a rule is phrased as a decision and an action, where the decision is based on the interaction message content, message context, and rule state. An action can include allowing the interaction to proceed, returning an error message, modifying the message and proceeding, delaying the interaction, or executing an external recovery process. Consequently, rules can be introduced onto a workflow without explicitly programming PDPs and PEPs – they are essentially injected onto *every* workflow. Because rules can be stateful, their injection onto a workflow effectively creates a System of Systems composition, with the injected rule comprising a small system. However, there is no facility that allows rules to be composed onto rules, thereby limiting the depth of composition.

An interaction message is constrained to a uniform, normalized format, which places limits on the processing needed to determine whether a rule applies during a particular interaction, and also limits the processing needed to evaluate the decision and execute the action. Consequently, the content of interaction messages and the scope of decisions and actions an xESB policy can express are also limited. As a result, xESB can quickly determine the applicability of a single policy in an interaction context, and can execute policies quickly and efficiently.

The xESB policy language is tuned to address access and usage control requirements, though without provisions enabling policy composition by unrelated stakeholder communities. Additionally, like PERMIS, its policy language is a formulation of first order logic that does not align with a stakeholder's understanding of domain problems, and therefore would not be used to engage a stakeholder in creating and maintaining solutions to domain-oriented requirements.

#### *2.4.4.4 Ponder2*

Ponder2 [125] is an event based system that uses an interceptor-based approach to executable policy statements associated with particular events. Policies are expressed in PonderTalk, a mature and robust variant of SmallTalk. Ponder2 allows the association of a method in a policy class with a particular object's outbound request, inbound request, outbound reply, or inbound reply channels.

As such, Ponder2 models applications as a network of connected components instead of a collection of workflows. To the extent that a decision must be made in the context of a service interaction, the policy must encode and evaluate the interaction relationship before determining how to operate.

Also, PonderTalk itself offers no state lifecycle that corresponds to a workflow variable. Finally, Ponder2 offers little support for the runtime specification of policy bindings or the composition of external workflows as separate concerns. Consequently, while Ponder2 offers the benefits of separation of business rules (encoded as separate policies) from application code, it cannot easily compose workflows, compose workflows onto workflows, or inject workflows at runtime to implement emergent stakeholder requirements.

### 2.4.5 Workflow Context in Distributed Systems

As programming languages enable the definition of workflows, they provide state management features (e.g., local, global, class, and package variables) whose lifecycles suit the language's purpose (as described in Section B.3). Analogously, modern frameworks and architectures that support distributed systems frame the maintenance of state based on the workflow assumptions underlying their target applications. In this section, I describe context maintenance in Struts, REST, and AJAX, as examples of different tradeoffs, particularly regarding how they enable state maintenance during a workflow and on behalf of workflows injected into a base workflow. I also examine how they enable tracking of workflow state as an application scales into a distributed system.

#### *2.4.5.1 Struts*

The Struts [126] system is a server-based Java environment that executes workflows on behalf of clients. Under Struts, state is maintained as

Java beans, with request, session, and application lifecycles as described in Section B.3.1.

Under Struts, a *single* workflow executes in a single thread on a single computer. Consequently, as workflows do not extend to distributed systems, enabling access to a workflow's request beans beyond a single thread is not implemented.

However, a Struts application can execute *different* workflows under different threads or on different computers. Consequently, session and application beans represent state available to all workflows executing on all computers in a distributed system, and are maintained by an independent, thread-safe service.

The Struts programming model assumes workflows are statically defined and deployed as a monolithic application. Consequently, it has no facilities to compose workflows dynamically into an application, and also has no facilities for maintaining state for such workflows.

#### *2.4.5.2 REST*

Under the REST (Representational State Transfer) [127] architectural style (described in Section B.3.2), application state resides on clients, which themselves can reside on separate computers. Workflows are implemented as interactions between a client and a server, which can, in turn, act as a client in an independent interaction. REST applications are therefore inherently scalable to distributed systems. To implement state that is accessible to all activities in a workflow, the state must be passed on all interactions comprising the workflow, and all servers must automatically propagate such state, which violates principles of separation of concerns. Consequently, REST applications favor state exchanged via localized client-server interactions over workflow-wide state.

REST encourages state common to all (or a subset) of clients and servers (similar to Struts' application beans) via calls to an application-defined server that keeps such state – it performs the work of Struts' independent state repository.

Workflow decomposition under REST can be accomplished by defining a server to be a proxy, which then orchestrates communication with some number of other servers to implement the original request – each downstream server request incorporates a combination of original request values and orchestration-local values according to the downstream server's interface specification.

Note that REST servers are accessed via URL, which functions as the Internet version of a routing system that supports workflow orchestration. Dynamic behavior can be implemented as a client calling a server whose URL is provided by another server. However, receiving the URL requires explicit coding within the client in anticipation of the dynamic behavior, which does not support the concepts of composing oblivious workflows or composing multiple workflows.

Consequently, because REST's implementation of workflow context and dynamic behavior does not support workflow composition, implementation of diverse stakeholder requirements (e.g., access control, auditing, provenance tracking, quality of service, and failure management) represents application-wide entanglement of concerns in REST applications.

#### *2.4.5.3 AJAX*

AJAX [128] [129] is a collection of technologies aimed at providing a fluid experience for users executing client-server application in a web browser, as described in B.3.3. Under AJAX, JavaScript executes client-side workflows based on client-resident state, which may be held as global variables and in closures, particularly closures associated with server requests. For example (following the illustration in Section B.3.3), in a REST model, a closure would maintain a copy of parameters passed to a server, thereby allowing the client to correlate a server reply with the parameters that generated it.

JavaScript closures represent workflow state private to a single workflow interaction, characterized by a client source sending a request to the server and a client target receiving the server reply – the closure pairs source state with the server reply. New state is manually generated for each client/server interaction. This contrasts with more pervasive workflow state, which persists across all interactions in a workflow. JavaScript alone does not provide a mechanism that maintains state for an entire workflow.

For internal workflows, JavaScript code is not automatically organized to segregate orchestrations as workflow activities with well-specified interfaces, and therefore does not admit injection of workflows representing new or changed requirements. Additionally, for JavaScript applications coded to execute multiple workflows concurrently, tracking of workflow state is done ad-hoc, and is not automatically amenable to tracking state of injected workflows.

For workflows in which a web browser interacts with an external server, application developers have a choice of creating a RESTful or non-RESTful server, with either implementation having the character described for REST in Section 2.4.5.2).

### 2.4.6 Ponder Policy Verification

As described in Section B.4, Ponder is a highly successful environment that implements the Ponder policy language [130], and represents a class of predicate-based policy languages that focus on system management and authorization requirements. Given the mission-critical nature of such policies in real applications, there have been numerous contributions towards rigorous analysis of policy specification and refinement (e.g., [131] [132]) enabling checking for modality conflicts (i.e., both enabling and disabling the same action), separation of duty conflicts (described in Section 7.4.1), improper refinement of a base policy, and redundancy in refinements.

Similarly, [133] transforms a collection of Ponder policies into a graph that relates attributes and conditionals. It discovers policies that partially or completely negate other policies, or situations where an action is both permitted and prohibited. For Ponder meta-policies, it detects situations where a policy depends on another policy, but the dependent policy is disabled.

Insofar as policy-based verification avoids obvious policy-driven application errors, such verification is valuable and necessary, analogous to syntax and model checking built into modern Integrated Development Environments (IDEs). However, such approaches do not perform similar analysis at a requirements level, particularly regarding the suitability (including completeness, correctness, and conflicts) of requirements relative to each other (including base and add-on requirements) or their fidelity to the policy expressions that represent them.

## 2.5 Patterns

Choice mechanisms can be defined and referenced generally in terms of design patterns, which codify best practices pertaining to methodologies and key implementation domains. While various pattern sets address different application architecture, design, and implementation concerns, the pattern sets I discuss represent key enterprise application concerns. For a design and programming perspective, I use the well-known Object Oriented Design (OOD) patterns [33] to frame the discussion. Workflow patterns [70], Enterprise

Application Architecture (EAA) patterns [134], and Enterprise Integration (EI) patterns [135] inform the use and context of these choice mechanisms.

Various pertinent pattern sets overlap or compliment these sets, but don't contribute additional choice mechanisms, including Orchestration Environment patterns [136], Service Oriented Architecture patterns [137], Service Interaction patterns [138], and others.

Note that many of these patterns are represented in computational models (Section 2.2), software development methodologies (Section 2.3), and mechanisms (Section 2.4) presented in this chapter. I deal with them separately here because they crosscut these other topics.

Two important and related principles are sometimes referred to as patterns, though they are not specified as such, and are not contained within a formal pattern set: Inversion of Control [34] (IoC) and Dependency Inversion [139] (DI). IoC assembles a workflow out of compatible components specified as interfaces that provide the basis for service contracts. IoC can suffice as the process by which workflows can be customized based on some configuration-time decision. IoC can realize interfaces using the DI principle, which replaces a reference to an interface with an instance that satisfies the interface. The workflow then executes using the realized interfaces.

In simple IoC and DI implementations, workflow configuration and realization is based on choices made at load time via configuration. However, the distinction between load time and runtime blur when the IoC's component assembly and DI's dependency realization are carried out at runtime immediately before executing a workflow, or even as their results are modified at runtime.

The configuration mechanisms of IoC and DI are, themselves, choice mechanisms that result in selecting amongst sub-workflows at configuration time (before a workflow executes). To the extent that IoC and DI can occur while a workflow is executing, they would qualify as workflow injection mechanisms. However, this is not how IoC and DI are generally interpreted, and neither principle on its own or in combination touch on the nature of workflow selection decisions or the runtime criteria on which they may be based.

### 2.5.1 Object Oriented Design Patterns

OOD patterns [33] assume the existence of basic decision (i.e., comparison and branching) primitives, OOP's distinctive inheritance and subtyping, and compile-time, link-time, or load-time resolution. The most prominent choice pattern is the Strategy pattern, supplemented by the Bridge and State patterns. The Strategy pattern uses the result of a decision to choose among candidate algorithms, where the algorithms each observe a common interface (according to the Bridge pattern) and can be called interchangeably. The State pattern conditions a workflow's definition according to values supplied to the workflow at instantiation or thereafter. These patterns can be realized in several ways (which can be combined):

- In the code design:
  - via simple mechanisms that pair the decision directly with a resulting workflow (e.g., if() or switch() statements)
  - via class structure where a base class has two or more specializations, and a pointer records the decision of which specialization to call (e.g., where an instance is created via a Factory pattern based on some criteria)
  - via a functor mechanism where the functor class represents an abstract workflow that is reified when the class is instantiated, or by State settings specified during execution
- In the compilation support system:
  - via command line and environment variables that causes a compiler or build system to choose amongst workflows
  - via a link editor that causes some modules or methods to be linked instead of others
- In the runtime system:
  - via loader configuration (including environment variables) that selectively loads one module instead of another
  - via deployment configuration that installs one set of loadable libraries instead of others (e.g., as dynamic link libraries)

Other patterns (e.g., Factory, Decorator, Façade, Plugin, Adapter, and Proxy) incorporate decision making that can be characterized as Strategy, Bridge, and State patterns where choice is encoded at program authorship time and can be influenced at runtime by State, but is not *defined* at runtime.

Note that Strategy and State patterns can exist even when the decision (or state storage) and consequent workflow are separated in time or are realized in different workflows (e.g., the separation resulting from a Factory pattern creating an object compared to the eventual use of the object).

While OOD patterns are intended to serve OOP, the Strategy, Bridge, and State (and other) patterns are common to patterns useful for other methodologies and mechanisms, including modular and functional programming. For example, under higher-order programming (e.g., λ-calculus), a Strategy pattern occurs when a function is assigned to a variable based on some decision criteria, a Bridge pattern occurs as a consequence of the function's parameter list and semantics, and a State pattern drives monad behavior. Even in these situations, the decisions are encoded at program authorship time, and are not *defined* at runtime.

The Template and Visitor patterns address different approaches to implementation of requirements as workflow customizations. The Template Method pattern calls for the compile-time injection of functions that implement workflow activities, and the Visitor pattern accomplishes a similar result by enabling the choice of function injection at runtime. In both patterns, the workflows and the injected functions are defined only at compile-time, and as such, constitute compile-time IoC and DI.

In any case, the OOD patterns are presented in terms of imperative languages (particularly OOP languages), and therefore are responsive to requirements identified in the development or maintenance process. While this implies the realization of new and changed requirements through redesign or reprogramming, then re-deploying code, such a process is *required* in these patterns.

### 2.5.2 Workflow Patterns

Workflow patterns describe the types of control and data flows that can occur in modeling business processes. Control flow patterns represent variations on choices that can be made in executing workflow activities, including the choice of one workflow instead of another, executing multiple workflows, or merging workflows. Data flow patterns describe the lifecycle and scope of data a workflow can access, how data can be used in a workflow, how it is transmitted between tasks, and how it affects the choice of task to execute.

Workflow patterns assume that a choice is made, regardless of the mechanism, and that workflows and data flows are orchestrated as a result. While they address different outcomes of choices, they are agnostic as to the process of realizing such workflows based on new or added requirements.

Consequently, while workflow patterns are often realized in static design and coding contexts, they apply equally to early and late bound choice.

### 2.5.3 Enterprise Application Architecture Patterns

EAA patterns seek to describe major architectural components of enterprise applications and how they relate to each other. Decisions and workflows are not emphasized in these patterns, though they underpin various patterns (e.g., Model View Controller).

Notably, the Separated Interface pattern corresponds roughly to the OOP Bridge pattern, but the OOP Strategy pattern is considered an implementation detail. Consequently, EAA patterns are agnostic as to the process of realizing such workflows based on new or added requirements.

### 2.5.4 Enterprise Integration Patterns

EI patterns address the integration of applications in a broad sense, including at the levels of information and business functions, processes, and services. As such, it depends on choices implemented as policies within workflows to affect either control flow or message content. For example, the Content-Based Router pattern, which performs routing based on the contents of a message – similar examples include the Filter, Recipient List, Splitter, Aggregator, and Resequencer.

## 2.6 Related Concepts

PDD represents a methodology and technology in the space of enterprise computing, which contains a number of systems, languages, and standards addressing both disjoint and overlapping concerns. While some of these are tangential to PDD and its support technologies, others relate more directly to PDD, but not to a degree that justifies in-depth comparison of their relationship to PDD. For convenience, I briefly present a number of these topics in this section.

### 2.6.1 SCA Infrastructure

SCA (Service Component Architectures) [140] is a suite of specifications that defines a protocol- and platform-neutral SOA language describing the interconnection of services. It does not have an interceptor concept, and does not propose features for service decomposition. Consequently, it doesn't address workflow composition, policy, or the infrastructure that supports them.

### 2.6.2 Spring Framework

Spring [141] is an application framework comprising a number of technologies relevant to building server-based enterprise applications. Spring uses the AspectJ pointcut language to compose concerns, and includes before, after returning, after throwing, finally, and around advices. Spring is declarative, and does not admit the PDD features of dynamic composition, policy evaluation, and message-based interception. Consequently, feature addition to a Spring application must be done at the source level, which creates development dependencies and defeats obliviousness.

### 2.6.3 ORC Language

Orc [69] is an orchestration language that enables invocation of sequential, parallel, and pruning flows. It enables vertical and horizontal integration. The larger language (including Cor) allows interception by closure. PDD features of policy evaluation, a context infrastructure, and workflow substitution are not represented in Orc, though could exist through extensions to Orc.

### 2.6.4 WS-Policy

WS-Policy [142] specifies an XML-based policy statement as a collection of policy alternatives containing policy assertions. Applied to web services, a service is said to support a requestor if one or all policy assertions are satisfied. PDD uses policies to filter messages and to drive the composition of workflows. WS-Policy doesn't leverage the context system (including the current message) that PDD can provide. WS-Policy can statically compose policies into an XML policy statement, while PDD achieves composition through XQuery expressions evaluated at runtime.

### 2.6.5 Business Process Modeling Notation (BPMN)

BPMN [143] [144] is a notation that graphically represents behavioral aspects of workflows and is capable of expressing workflow patterns identified by [70]. While there are notational and stylistic differences between BPMN and UML Activity Diagrams [40], their major difference is in the audiences they seek to address – BPMN for business process designers, and Activity Diagrams for modeling software development. Consequently, while the high level entities addressed by these notations are tailored to their audiences, the semantics of the workflows they express are similar. Neither has direct support for injection of crosscutting concerns, though extensions exist that address them for BPMN [145] (to compliment [146]) and for UML [147].

Given this functional equivalence, I use only UML Activity Diagrams (and derivatives) in this dissertation because they are more familiar to software engineering audiences, though my discussions apply equally well using BPMN.

### 2.6.6 Policy as Commitments

In this dissertation, I conceive of a policy as a combination of a decision that selects amongst alternate workflows. Such workflows represent a process as an orchestration of activities that satisfy a data flow and control dependencies (as with BPMN), often with workflow activities representing computational constructs.

A higher level view [148] conceives of policies as engagement relationships between business services operating on behalf of one party or another. In this paradigm, a policy is a collection of *commitments*, which are relationships between *parties* within the scope of an organizational *context*. A party can be a concrete or abstract entity, and a context represents an environment in which relationships exist, exceptions are understood and handled, and non-conformant behavior is sanctioned. Basic commitment operations include *creation* (where a party activates a commitment), *discharge* (where a party satisfies a commitment), *assignment* and *delegation* (where a party transfers a commitment to another party), and *release* (where a party removes another party from further participating in a commitment).

Commitment-oriented systems are based on both standardized and novel patterns expressed as workflows that orchestrate communication between parties (or their computationally-oriented proxies) and facilitate sharing state. Commitment-oriented policies model business relationships vividly and flexibly, independent of a particular implementation. Decisions are made at two levels: within workflows that implement the behavior of a party, and within workflows that express and enforce a commitment. Given this, commitment-oriented systems may leverage workflow-based decisions (as PDD-style policy, addressed in this dissertation), but they don't affect decision mechanisms or the workflows they select.

## 2.7 Summary

Table 2 presents a summary of the capabilities of computational models, methodologies, mechanisms, and patterns as applied to the realization of requirements as workflows through late bound choice. Specifically, it identifies potential solutions in the key areas set out at the beginning of this chapter. These capabilities are discussed in previous sections -- in this section, I summarize those discussions and provide further insight into their value in addressing late binding issues. Section 2.7.6 ties the discussions together.

Note that the table includes an additional row, which corresponds to the PDD capabilities represented as contributions of this dissertation. Section 2.7.6 gives a brief glimpse into their implementation in Chapter 4 (as foundation) and Chapter 5 (as case study).

Table 2. Summary of Existing Work

| | Workflow Specification | | | Workflow Injection | | | | | | State Management | | | | Modeling | | Deployment | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
| **Computation Models** | | | | | | | | | | | | | | | | | |
| **Turing Machines** | ✘ | ✘ | ✘ | manual | manual | manual | manual | ✘ | ✘ | ✔ | ✘ | ✔ | ✔ | ✘ | ✘ | - | - |
| **Petri Nets** | ✔ | ✔ | ✘ | manual | manual | manual | manual | ✘ | ✘ | ✘ | ✔ | ✔ | ✔ | ✘ | ✔ | - | - |
| **π-Calculus** | ✔ | ✔ | ✘ | manual | manual | manual | manual | ✘ | ✘ | ✔ | ✔ | ✔ | ✔ | ✘ | ✔ | - | - |
| **λ-Calculus** | ✔ | ✔ | ✘ | manual | manual | manual | manual | ✘ | ✘ | ✔ | ✔ | ✘ | ✔ | ✘ | ✔ | - | - |
| **Methodologies** | | | | | | | | | | | | | | | | | |
| **Modular Programming** | ✔ | ✔ | ✘ | manual | manual | manual | manual | ✘ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✘ |
| **Structured Programming** | ✔ | ✔ | ✘ | manual | manual | manual | manual | ✘ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✘ |
| **Object Oriented Programming** | ✔ | ✔ | ✔ | manual | manual | manual | manual | ✘ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✘ |
| **Aspect Oriented Programming** | ✔ | ✔ | ✔ | static | static | ✘ | ✘ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✘ |
| **Mechanisms** | | | | | | | | | | | | | | | | | |
| **Fundamental** | - | - | - | manual | manual | manual | manual | ✘ | ✔ | - | - | - | - | ✔ | ✔ | - | - |
| **Declarative Representations** | ✔ | ✔ | ✘ | manual | manual | manual | manual | ✘ | ✔ | ✔✘ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✘ |
| **Execution Frameworks** | ✔ | ✔ | ✔ | runtime | runtime | ✘ | ✘ | ✔ | ✔ | ✘ | ✔ | ✔ | ✘ | ✘ | ✘ | ✔ | ✘ |
| **Policy Engines** | - | - | - | man/run | man/run | ✘ | ✘ | ✔ | ✘ | ✘ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| **Distributed System Technologies** | ✘ | ✘ | ✘ | manual | manual | manual | manual | ✘ | ✘ | ✔✘ | ✔ | ✔ | ✔ | ✘ | ✘ | ✔ | ✘ |
| **Patterns** | | | | | | | | | | | | | | | | | |
| **OOD Patterns** | - | ✔ | - | - | ✔ | - | - | - | - | ✔ | - | - | - | - | - | - | - |
| **Workflow Patterns** | ✔ | ✔ | ✘ | ✘ | ✔ | ✘ | ✘ | ✘ | - | ✔ | ✔ | ✔ | ✔ | - | - | - | - |
| **Enterprise Application Patterns** | - | ✔ | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| **Enterprise Integration Patterns** | - | - | - | ✔ | ✔ | ✘ | ✘ | ✘ | - | - | - | - | - | - | - | - | - |
| **Policy Driven Development** | ✔ | ✔ | manual | runtime | runtime | ✔ | ✔ | ✔ | ✘ | ✔ | ✔ | ✔ | ✔ | ✘ | ✘ | ✘ | ✘ |

## Key

| Symbol | Meaning |
|---|---|
| - | does not apply |
| ✘ | "no" or "unavailable" |
| ✔ | "yes" or "available" |
| manual | "available" by manual refactoring |
| static | "available" at compile time |
| runtime | "available" without recompiling or redeploying |
| man/run | manual or runtime, depending on system |

Table 2. Summary of Existing Work, Continued

| **Legend** | |
|---|---|
| **Workflow Specification** | |
| **A** | Separate Workflows |
| **B** | Workflow Interface |
| **C** | Contract Enforcement |
| **Workflow Injection** | |
| **D** | Specified Site |
| **E** | Workflow Selection |
| **F** | Choice on choice |
| **G** | Composite Choice |
| **H** | Centralized Concerns |
| **I** | Visualization |
| **State Management** | |
| **J** | Workflow Unique |
| **K** | Instance Shared |
| **L** | Global |
| **M** | Customized Lifecycle |
| **Modeling** | |
| **N** | Verification |
| **O** | Model Checking |
| **Deployment** | |
| **P** | Secure Deployment |
| **Q** | Consistent Deployment |

### 2.7.1 Workflow Specification

While most contributions allow the articulation of a workflow with a corresponding call interface (though not a semantic interface), OOP provides contract enforcement via types and encoded assertions, and AOP enables enforcement as a separate concern [149]. SQL's constraint injection is also a form of contract enforcement.

Contract enforcement exists in practical programming systems and frameworks as a means for harmonizing workflow activities *a priori* so as to avoid bugs by detecting obvious semantic incompatibilities. It most often relies on the compile-time or link-time evaluation of declared program meta-information, though it also appears in runtime implementations as JavaScript [150] function calls and UDDI [151] searches, both of which rely on function signatures as expressions of contracts.

Achieving a similar degree of certainty regarding runtime workflow injection requires that the contracts be available (at runtime) for each injectable workflow as well as source-target interaction onto which a workflow can be injected. Given that, a means must exist for comparing these contracts before an injected workflow can execute. While this has been demonstrated at the function signature level (e.g., JavaScript and UDDI, which includes verifying messages based on type), it has not been demonstrated for interactions occurring in a runtime context and depending on workflow or application state (as would be the case for a complex protocol).

### 2.7.2 Workflow Injection

While the computational models, methodologies, and mechanisms enable choice (including a decision and alternate workflows) to be composed onto a base workflow, most solutions require that the injection site be determined manually, and that the choice be explicitly coded prior to application testing, release, and deployment. Given this manual orientation, they also support explicitly coding the composition of non-base concerns, including the composition of multiple choices at a single site. However, they also suffer from scattering and entanglement when workflows represent crosscutting concerns.

AOP differs by allowing the injection site to be chosen automatically (via pointcut), thereby enabling the concentration of crosscutting concerns as separable modules, which avoids scattering and entanglement. Similarly, execution frameworks (e.g., ESBs and SQL engines) enable the concentration of crosscutting concerns as separable modules, though they can be injected into workflows at runtime (via interceptors, transformations, and dynamic routing that implement IoC and DI). Policy engines allow the concentration of decision criteria as separable modules, but require that the injection site and alternative workflows be provided manually at the coding phase. xESB is exceptional in that it does not require the manual injection of policy and workflows, though it is limited in the types of workflows it can inject. All three approaches provide little support for composing choice onto injected workflows or composing multiple choices that affect a common site.

Because both manual and automatic workflow composition breed complexity that can result in increased maintenance costs and failure to meet stakeholder requirements, much attention has been given to workflow visualizations appropriate for each programming methodology and mechanism. Particularly, because AOP separates composed workflows from base workflows, visualization support is needed so that programmers can readily appreciate the effects of composition [152].

While OOD patterns, workflow patterns, EA patterns, and EI patterns offer guidance regarding the essential qualities of choice, the forms that choice can take, and how choice can effect various control and data flows, they are largely silent regarding specific mechanisms useful in the early or late binding of choice. This is true, also, of IoC and DI, which bind workflows before execution time, and offer little guidance regarding the binding of workflows into *executing* workflows.

Note, however, that IoC and DI do provide the bases for injecting policy evaluation at each stage of workflow execution as a concern that crosscuts all workflow activities. Chapter 4 and Chapter 5 demonstrate how this can be parlayed into runtime-defined choice that can result in the injection of workflows implementing requirements not known at the time of application authorship or deployment.

Achieving the injection of requirements (as workflows) into a running system requires that workflow interactions be exposed for reflection at runtime, that workflow interactions support the injection of requirements from multiple sources, that choice between injected workflows be made based on runtime state, and that requirements must be composable onto composed requirements. No single or combination of contributions meets these challenges, though various contributions offer insights into a solution.

### 2.7.3 State Management

Given that a requirement may decompose into a number of workflows injected upon multiple locations within a single or multiple base workflows, workflow coordination requires that associated state have a lifecycle matching the workflow duration, multiple workflow durations (i.e., a session), or across all workflows. Additionally, to the extent that workflows can be injected onto oblivious base workflows at runtime, state must also be defined and injected at the same time.

In the algebras, methodologies, mechanisms, and patterns surveyed, state is most often explicitly declared at program authorship time, and its injection has the same limitations and drawbacks (including contributing to entanglement and scattering) as manual workflow injection. AOP, SQL, and policy engines enable state injection at runtime (coincident with workflow injection), though they do not address state management on a workflow or session basis because they do not model or track such information.

No single or combination of contributions enables the injection of workflow-based state onto oblivious workflows, though various contributions provide inspiration for modeling state having other lifecycles.

### 2.7.4 Modeling

The value of verifying and model checking injectable workflows (as proxies for requirements) is to determine that workflows implement requirements alone or in combination with other workflows, and that they do not disturb the implementation of uninvolved requirements.

Extensive work has been done with programming methodologies in verifying that manually written programs meet articulated requirements and that their code guarantees important properties (e.g., livelock avoidance, state avoidance). Indeed, computational models exist, in part, to demonstrate how properties are maintained or guarantees are broken. While execution frameworks are not exploited in this way, declarative models (e.g., UML [55]) and policies evaluated by policy engines are (as in Section 2.4.6 and [153]), though primarily in the security domain, which focuses on consistency, completeness, and correctness. Additionally, unit testing and model checking techniques are applied directly against code produced under various programming methodologies.

Generally, verification and model checking are applied to workflows at the modeling, design, and authorship stages, resulting in iterations over those stages until the workflows achieve the desired properties.

Insofar as a composed requirement (via composition of one or more workflows) represents an incremental change to base workflows, applying such techniques requires an understanding of both the base and injected workflows at one of multiple levels (requirement, modeling, or coding) with assurances that applying such techniques at abstraction levels (i.e., requirement or modeling) ultimately confers guarantees at the coding level.

The contributions I surveyed contemplate verification and model checking at the modeling or coding level in the context of an entire system, and compliment the monolithic application delivery model. For distributed systems containing components that may not be available for such verification and model checking, the monolithic approach may be too costly if it is possible at all.

To the extent that a requirement represents a partial system behavior, the workflow(s) that implement it can be independently verified or model checked, though no contributions are situated to do this at runtime either at the requirement, model, or code level.

### 2.7.5 Deployment

While computational model expressions are generally not deployed as executable applications, means exist for deploying all other kinds of applications completely and authoritatively. However, where components of an application (e.g., DLLs, policies, POJOs, constraints, and stored procedures) can be delivered separately, only ad-hoc means provide guarantees that all such components are consistent with the deployed application and with each other. Policies evaluated by policy engines have been packaged as atomic modules when they relate to each other via shared state, and are therefore consistent by design. Even so, the execution of one policy at one time may set state that is semantically different than is expected by a policy deployed and executed at another time – this is an unsolved problem.

### 2.7.6 Assessment

The dominant programming paradigms involve binding requirements early in the design and coding process, and take the form of manually selecting workflow injection sites, decision, and workflows. Additionally, such binding may lead to scattered and entangled code. While AOP solutions avoid this, they still rely on early-bound aspects to achieve this and provide little facility for scaling applications.

Prominent programming and enterprise application patterns are agnostic as to early binding. Execution frameworks enable late binding, but fail to support injection of workflows onto injected workflows, the injection of multiple workflows onto a single site, and the maintenance of state across the execution of a workflow. While existing policy engines enable late binding of decision criteria, most rely on early bound injection site and workflow alternatives.

While early binding of requirements enables a significant testing phase prior to deployment, binding requirements at runtime doesn't involve any pre-deployment testing phase. While significant visualization, verification, and model checking support would greatly benefit the fidelity and reliability of late-bound applications, such support is underdeveloped for the execution frameworks that support such applications.

Finally, while policy engines enable secure deployment of decision criteria and (to a degree) support the deployment of consistent policies, no solution deploys site selection, alternate workflows, and decisions in a secure and consistent manner.

For reference, a quick list of additions needed relative to existing contributions (i.e., a gap list) includes:

- Identification of policy (including decision leading to selection amongst alternative workflows) injection site at runtime
- Injection of policy at runtime
- Tracking policy -centric state to allow coordination of multiple policies injected on a base workflow
- Composition of multiple policies onto a single injection site
- Enabling composed workflows to act as base workflows for other compositions
- Verification that an injected policy is interface- and semantically-compatible with its injection site
- Incremental testing and proofs that policies implement requirements
- Enabling a consistent relationship between state and policy across policy deployments

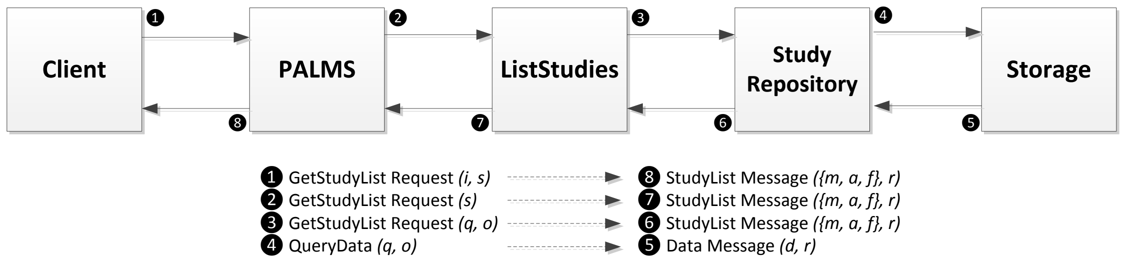
The PDD contributions described in this thesis (per Section 1.4) address gaps pertaining to workflow specification, workflow injection, and state management at the foundational level and as a case study. In Chapter 4, I describe the means by which policy can be injected into existing workflows at runtime. Fundamentally, PDD leverages an equivalence between workflow activities and services, where workflows are cast in terms of SOAs whose interactions can be identified and intercepted at runtime. PDD exploits this by injecting policy evaluation and state propagation services that crosscut all service interactions. The result is to provide an opportunity to fill each of the gaps identified above. Chapter 5 demonstrates the implementation of PDD in the context of a real world system.

Chapter 7 describes how the PDD approach fills the gaps identified in this chapter. As discussed in Sections 7.5, 7.4.4, and 7.4.6, PDD does not address gaps pertaining to modeling and deployment – these gaps must be filled before late binding can be reliably supported in production environments, and are left to future work.

# CHAPTER 3

# A RUNNING EXAMPLE – PALMS' GETSTUDYLIST WORKFLOW

The concept of the injection of choice can be demonstrated and discussed using a single interaction between two workflow activities. However, as a motivating example, Figure 6 presents a realistic workflow (called *GetStudyList*) that implements a stakeholder requirement end-to-end in the PALMS case study (described in Chapter 5), and which I will use throughout the remainder of this dissertation.



Client
PALMS
ListStudies
Study Repository
Storage
❶ GetStudyList Request (i, s) → ❽ StudyList Message ({m, a, f}, r)
❷ GetStudyList Request (s) → ❼ StudyList Message ({m, a, f}, r)
❸ GetStudyList Request (q, o) → ❻ StudyList Message ({m, a, f}, r)
❹ QueryData (q, o) → ❺ Data Message (d, r)

Figure 6. GetStudyList Generic Workflow

(As described in Chapter 5, the PALMS case study is an example of a cyberinfrastructure that serves diverse stakeholder communities having different interests, and whose requirements change over time. The GetStudyList example is representative of all PALMS workflows, any of which could be similarly subject to evolving stakeholder requirements, as addressed in this dissertation. Finally, while GetStudyList represents PALMS workflows, it equally represents other types of workflows and workflow patterns as described in Section 2.5.2.)

As an example of a generic workflow (as described in Section 2.1.1), GetStudyList shows processing and information flow. It shows five activities (e.g., services) that interact in a request/reply pattern, where each activity sends a message to another activity, which in turn replies with a message.

GetStudyList provides context to demonstrate the realization of several different types of additional stakeholder requirements, including:

- Various forms of access control
- Data stream filtering
- System of Systems integration
- Composition of choice on choice

It also provides context for a discussion of requirements that affect multiple interactions and share state.

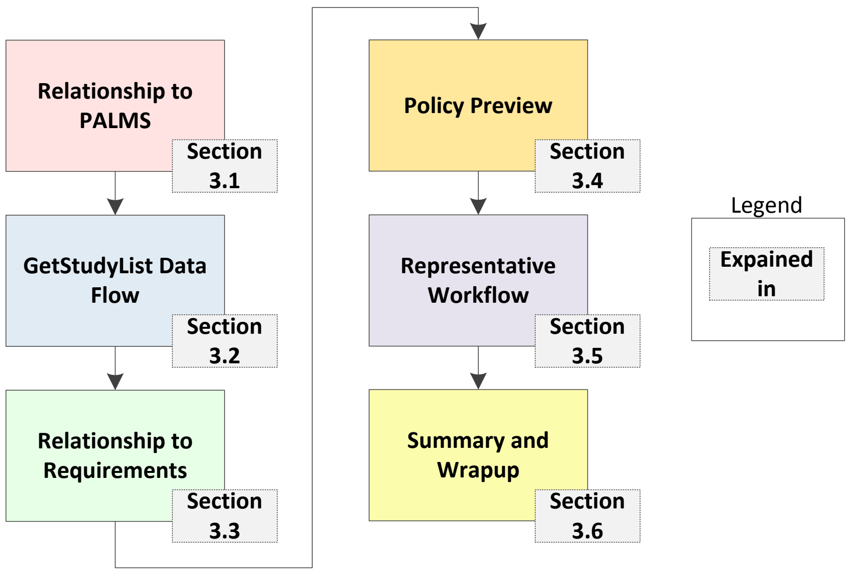


Figure 7. Chapter 3 Flow

As shown in Figure 7, this chapter describes the GetStudyList workflow in terms of its relationship to PALMS (Section 3.1), its data flow (Section 3.2), how it can be affected by emergent requirements (Section 3.3), how policy can be used to effect emergent requirements (Section 3.4), and how it relates to other kinds of workflows (Section 3.5).

## 3.1 Relationship to PALMS

The objective of this workflow is to return a list of studies to a client, which may be a Web browser acting on behalf of a user (represented by the Client activity). A study is an abstraction central to the PALMS application -- it is a research activity that organizes observational and other data, and is described by attributes (as metadata) in Table 3.

Table 3. PALMS Study Components

| Value | Meaning |
|---|---|
| **studyID** | GUID unique to the study |
| **studyName** | textual description of study |
| **groupName** | name of collection of roles associated with study's access control rights |
| **attributesXML** | collection of Client-defined metadata |
| **accessSchemaXML** | schema describing Client-defined metadata |
| **formsXML** | Client data structures enabling metadata display and maintenance |
| **primarySubjectAttr** | name of data containing study subject ID in subject repository |

Conceptually, the Client retrieves a list of studies for display to a user. The Client allows the user to select a study, which then enables the Client to view or modify observational and other data associated with the study.

PALMS maintains a number of abstractions similar to a study, and there are similar workflows involving each.

To accomplish this, the Client activity interacts with the PALMS activity, which encapsulates all PALMS capabilities – in this example, it exposes the study list retrieval capability, but in a fleshed out system, it exposes additional capabilities. In general, PALMS either invokes a workflow appropriate to the request (e.g., ListStudies, described below) or rejects a client's request. This pattern is repeated for each downstream activity.

The ListStudies activity represents an abstraction that returns a list of existing studies (if $s = null$) or a particular study $s$ (if $s \neq null$). It fetches study information through interaction with the Study Repository activity (described below). It qualifies the scope and format of the study list and packages the resulting information for return to the Client.

The StudyRepository activity represents an abstraction that maintains a collection of studies, and is capable of adding, modifying, or removing a study, its attributes, or its observational and other data. The StudyRepository abstraction defines and enforces semantics for each study attribute, including how each attribute relates to other attributes. It implements study attribute storage by interacting with the Storage activity (described below), and returns study attributes as received from Storage.

Finally, the Storage activity stores a collection of related elements so they can be retrieved as a unit (e.g., as a row in a table). The Storage abstraction does not maintain any semantics for any element, and is reused by other PALMS workflows that store and retrieve related elements.

In this example, each activity represents an abstraction that transforms input messages into output messages, and can be decomposed as combinations of internal computation and interactions with some number of other activities. As such, an activity can be implemented in any number of ways, so long as it interacts appropriately with activities that interact with it. (These concepts are elaborated upon more precisely and robustly in Chapter 4.) For example, while the StudyRepository activity leverages the Storage activity to persist study attributes, an alternative implementation might translate attributes returned by Storage, assemble attributes from multiple sources, or calculate or synthesize them based on some criteria.

## 3.2 GetStudyList Data Flow

In Figure 6, by convention, arrows going from left to right indicate requests, and arrows going from right to left indicate replies. Requests and replies are tagged by a number (e.g., ❶) that indicates their sequence in the workflow, and which matches a message tag in the legend. The associated message entry contains a textual description and lists values contained in the message, as listed in Table 4.

Table 4. GetStudyList Message Contents

| Element | Meaning | Comment |
|---|---|---|
| ***i*** | Identity credential | X.509 certificate for requesting user |
| ***s*** | Study ID | Primary key for study metadata |
| ***o*** | Option collection | Formatting directives for result |
| ***q*** | Query statement | Predicate describing data to be fetched |
| ***{}*** | Collection of tuples | Container for list of metadata tuples |
| ***m*** | Study metadata | Study ID, name, group, etc |
| ***a*** | AccessSchema | Study accessShema |
| ***f*** | Form | Study GUI form |
| ***r*** | Result | Error text if request was rejected |

While the particular contents and semantics of each message and its elements is highly relevant to the interactions in which they are exchanged, they are not important to this discussion, except to recognize that each interaction is defined in terms of messages it exchanges, as described in Chapter 4.

## 3.3 Relationship to Requirements

Each activity represents an abstraction that adds value to a data flow in some way. The activities in the example were created to add this value responsive to abstractions present in a decomposition of high level user requirements. Consequently, an interaction pattern represents such an abstraction.

The sample workflow arose from a PALMS requirement that the user be able to see a list of available studies. Each activity interacts with at most one other activity only because this simple pattern satisfies the PALMS requirement.

## 3.4 Policy Preview

As explained more fully in Chapter 4 and Chapter 5, PDD addresses new and changed requirements as modifications to the base workflows that reflect the existing requirements. For example, consider two new requirements:

1) If the user (represented by $i$) is European, use a study repository located in the EU; otherwise, use a local (US) repository. In any case, if the system is in test mode, use a test repository.
2) Return only studies the user has permission to view

Requirement 1 addresses a hypothetical preference of European users that private data be stored on European servers, subject to European privacy laws. It also addresses system testing. Requirement 2 is a form of access control.

A back-of-a-napkin illustration of responsive policies using informal notation is shown in Figure 8 as follows:

A policy that implements Requirement 1 would apply to the StudyRepository-Storage interaction (❹-❺), and would choose amongst different Storage activities, depending on a user attribute or a system environment variable.

A policy that implements Requirement 2 would apply to ListStudies-StudyRepository interaction, and could be implemented in two ways, given a set $X$ containing the studies the user is allowed to view. On ❸, query $q$ could be constrained (via rewriting) to fetch only studies $x \in X$, or on ❻, the tuple list could be filtered to contain only studies $m.s \in X$.

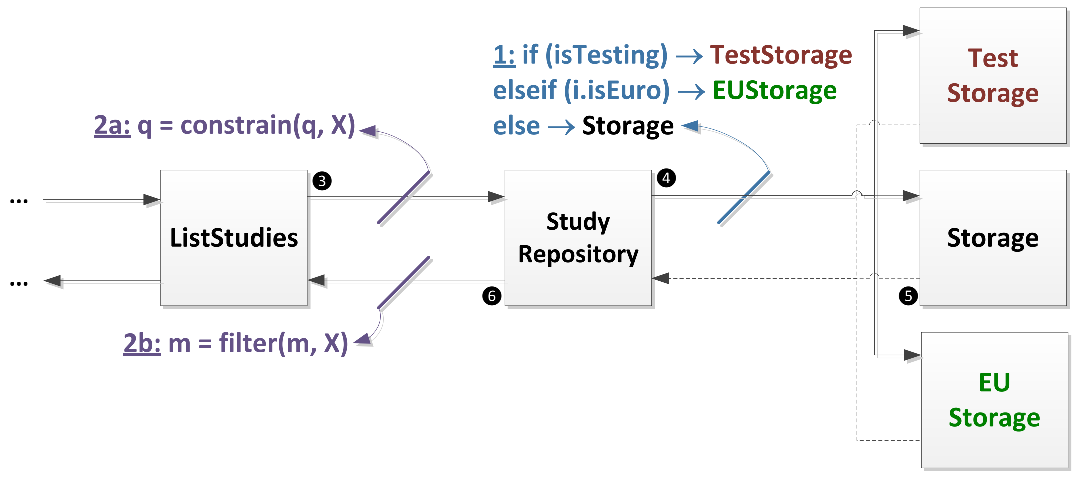


Figure 8. GetStudyList Generic Workflow (with sample policies injected)

Both policies assume the availability of the user identity $i$ in the affected interactions. A close inspection of Figure 6 reveals that $i$ is not directly available to those interactions. As explained in Chapter 4 and Chapter 5, such values represent context that crosscuts base workflows, and are accessible to policies through a context support system. Additionally, system environment variables and attributes related to the user $i$ are themselves instances of separate concerns exposed via domain-specific policy libraries that support those concerns – such libraries expose functions that calculate and return $X$, access and return system environment variables, or implement other functionality that supports a policy expression.

## 3.5 Representative Workflow

While the motivating example provides a means for the discussion of injectable choice, it is representative of a wider class of workflows to which the discussion also applies. As depicted, GetStudyList is structured as a series of request/reply interactions, but my discussion applies to other interaction patterns, as well. For example, an activity could interact with multiple activities in parallel. Similarly, a modified request/reply interaction could involve three activities, where one activity makes a request of a second activity, and the second activity delegates the response to a third activity.

The possible variations in relationships between activities are given in [70]. Note that this list includes a conditional relationship ("Multi-choice") where an activity could interact with any of several activities, but chooses a subset based on some criterion. This workflow relationship corresponds to the policy injection that PDD seeks to leverage.

## 3.6 Summary

In this chapter, I presented the GetStudyList workflow, which I will reference throughout subsequent chapters. It illustrates properties key in the discussion of PDD, including data flow relationships between activities, and activities that represent transformational abstractions that can be further decomposed. It hints at how policy injection can be used to implement new or changed requirements without perturbing a base workflow. It also hints at a system that supports state required for the composition of complex, orthogonal concerns on workflows.

In Chapter 4, I lay a theoretical foundation for the discussion of activities and their interactions, where an interaction between two activities is understood independent of other interactions, and is characterized by sequences of responses and replies. It also describes how an activity can be decomposed into a workflow that implements the interaction pattern the activity supports.

In Chapter 5, I present a real world case study that illustrates the application of the PDD foundation in a running system (i.e., PALMS), making extensive use of the GetStudyList workflow as it exists on a running system.

# CHAPTER 4

# A FOUNDATION FOR POLICY COMPOSED ON WORKFLOW

PDD policies enable the realization of stakeholder requirements as concerns that crosscut a base workflow. In general, PDD defines a policy as a decision that is injected into a base workflow, and may lead to the composition of a new workflow onto the base. The decision can incorporate a number of factors, including data flows (or histories of data flows) managed by the base or other workflows, or the state of the application or external system. The composition is achieved by applying the composed workflow to a data or control flow as in Table 5.

Table 5. Types of Composition

| Flow | Type | Action | Example |
|---|---|---|---|
| **Data** | Tap | Operate on copy of data flow | Auditing |
| | Filtering | Modify data flow | Augment/decimate data |
| **Control** | Combination | Add a new workflow | Obligation enforcement |
| | Replacement | Replace an existing workflow | Access control as Allow/Deny decisions |

For convenience, I refer to the injection of a decision that may lead to the composition of a new workflow onto a base workflow as *policy injection*.

The implementation of a requirement may involve the injection of several different policies onto one or more workflows. While policies can act independently (as might occur when one policy encodes a data flow, and the other policy decodes it), policies can also coordinate via shared state.

A major consequence of policy injection is the opportunity to drive the integration of separate concerns (including crosscutting concerns and feature sets) into a system at runtime, thereby enabling faster and more flexible system integration (including System of Systems integration).

Furthermore, workflows composed via policy can, themselves, be considered as base workflows eligible for policy injection -- PDD mechanisms enable policies on policies, thereby enabling the composition of requirements on requirements. Finally, because policy evaluation is, itself, a workflow, PDD mechanisms enable policies to affect the policy evaluation process itself. Consequently, these mechanisms lead to flexible ways of combining and evaluating multiple policies, possibly provided by multiple stakeholders across multiple domains, all oblivious to each other.

Realizing this vision requires solving challenges not met by in the existing systems described in Chapter 2, specifically:

- a clear definition of a policy
- a means to locate where in a workflow to inject a policy
- a means for composing a *single* policy onto a workflow interaction (including onto workflows contributed by policies)
- a means for composing *multiple* policies onto a single workflow interaction
- a means for maintaining policy state separate from state maintained for other policies
- a means to verify the suitability of policies relative to requirements and existing policies
- policy languages that encourage and leverage stakeholder participation in policy authorship

In this chapter, I answer these challenges at a foundational level, and present a real world case study implementation in Chapter 5, followed by an evaluation of how PDD met these challenges in Chapter 7.

Briefly, this chapter articulates several of the major contributions of my dissertation: foundations for policy definition, injection of policy into a workflow, and composition of multiple policies at a single injection point. It also presents an architecture for maintaining workflow-based policy state as part of a larger context system. This foundational work motivates and underpins the PALMS case study described in Chapter 5.

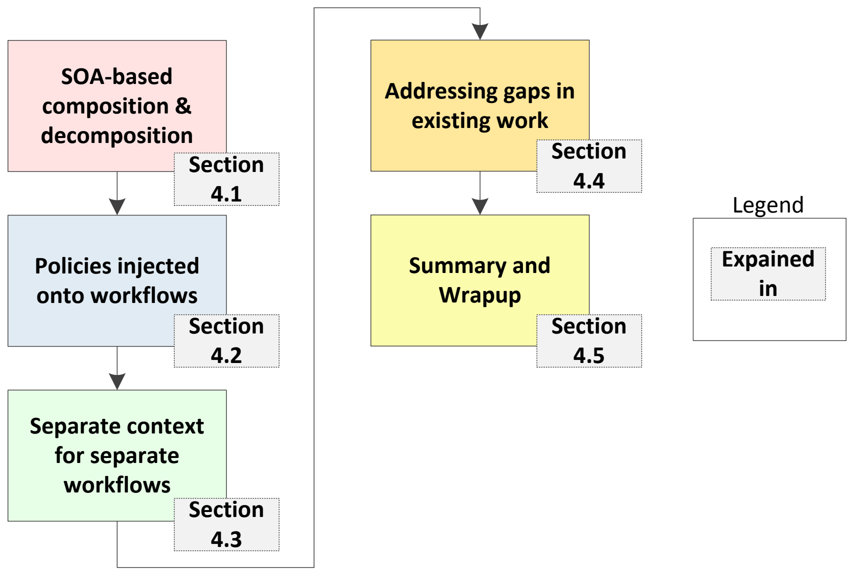

SOA-based composition & decomposition
Section 4.1
Policies injected onto workflows
Section 4.2
Separate context for separate workflows
Section 4.3
Addressing gaps in existing work
Section 4.4
Summary and Wrapup
Section 4.5
Legend
Expained in

Figure 9. Chapter 4 Flow

As shown in Figure 9, I first describe basic service orchestration concepts, starting with definitions of service, service interactions, service contracts, and service composition, all framed under a Rich Services SOA model (in Section 4.1). Next, I explain the mechanisms by which a crosscutting concern (represented as policy evaluation) is injected onto a base workflow (in Section 4.2). Finally, I demonstrate mechanisms that enable workflows to maintain state separate from other workflows (in Section 4.3).

Note that π-calculus (described in Section 2.2.3) is a process algebra that describes workflows in terms of message-based process interactions and aligns well with models developed in this chapter. However, requirements modeled by π-calculus (and executable languages based on it) are bound at authorship time, and injecting emergent requirements (as policies) at runtime is left unaddressed. Relative to π-calculus, the contributions of PDD include a means to locate where in a process to inject policy, a foundation for composition of policy onto a process interaction, a foundation for maintaining independent context for composed workflows, and a foundation for realizing requirements using a stakeholder-centric language instead of a process-centric language. From a π-calculus perspective, this enables the integration of separate concerns without incurring entanglement and scattering, which, in turn, enables the maintenance and analysis of separate concerns separately. From the perspective of executable languages based on π-calculus, it also represents requirement integration at runtime, where stakeholders can be active participants in realizing requirements.

## 4.1 A SOA Approach to Policy-based Workflow Composition

Given an equivalence between workflows and service orchestrations, I frame the foundational discussion of PDD and its mechanisms in terms of a Service Oriented Architecture (SOA), particularly the Rich Services [154] (RS) architectural blueprint (as described in Section 4.1.1), which aligns well with the PDD concepts of workflows, workflow element decomposition, and workflow injection. Using RS as a backdrop, I explain an interface-centric foundation for service interactions (in Section 4.1.2), which I leverage to define service composition (in Section 4.1.3) and decomposition (in Section 4.1.4) leading to a foundational basis for policy evaluation via message interception (in Section 4.1.5).

In subsequent sections (4.2 and 4.3), I define PDD policies and use these mechanisms to show how policies can be injected into base workflows at runtime, thereby enabling late binding of stakeholder requirements.

### 4.1.1 Rich Services

Rich Services [154] (RS) is a SOA-based architectural blueprint (i.e., a specialized SOA) that aligns well with the PDD concepts of workflows, workflow element decomposition, and workflow injection. I chose a SOA-based approach over other approaches (e.g., AOSD or pure OOP) because of how it aligns with many objectives of building large scale systems capable of servicing multiple evolving stakeholder communities and because it addresses multiple development and application concerns. Particularly, SOA's message-based service interactions enable interface-based component reuse (via loose coupling and late binding); inherent reusability, scalability, and distributability (via message routing); and the opportunity to constrain and augment workflows (via message interception), while retaining the freedom to use other approaches to build services themselves.

As a SOA, the RS blueprint inherits these SOA properties. It can represent either a logical model (as in Section 5.3), a deployment model (as in Section 5.4), or both (given a mapping from a logical RS to a deployment RS).

In the abstract, a Rich Service is a service that transforms one or more streams of input messages into one or more streams of output messages, and consists of a service interface and an orchestration of loosely coupled (sub-) Rich Services comprising orchestrations that implement the service interface. Since Rich Services can be built from other Rich Services, they form a service hierarchy conforming to a Composite pattern [33] (as briefly described in Appendix C).

Critically, Rich Services includes the concept of Rich Infrastructure Services (RISs), which enables the injection of service processing that implements crosscutting concerns (e.g., failure detection and mitigation [155], encryption [156], auditing, and access control, all defined in terms of policies injected responsive to stakeholder requirements).

A Rich Service can model functionality at either the logical level (relating interactions and hierarchies involving services in the abstract) or at the physical level (relating realized services and the communications channels that connect them). A common use of Rich Services is to model at both levels, with a deployment mapping that translates logical modes to physical models (as described in Sections 5.3 and 5.4).

This section briefly describes the structure and semantics of the Rich Services blueprint, which creates a robust context for discussing policy injection that leads to the rapid realization of stakeholder requirements.

#### 4.1.1.1 *Rich Service Structure*

As shown in Figure 10, a Rich Service contains five major components: the Service/Data Connector (SDC) ①, a collection of Rich Application Services ② and Rich Infrastructure Services (RISs) ③, a message transport ④, and a message router ⑤.

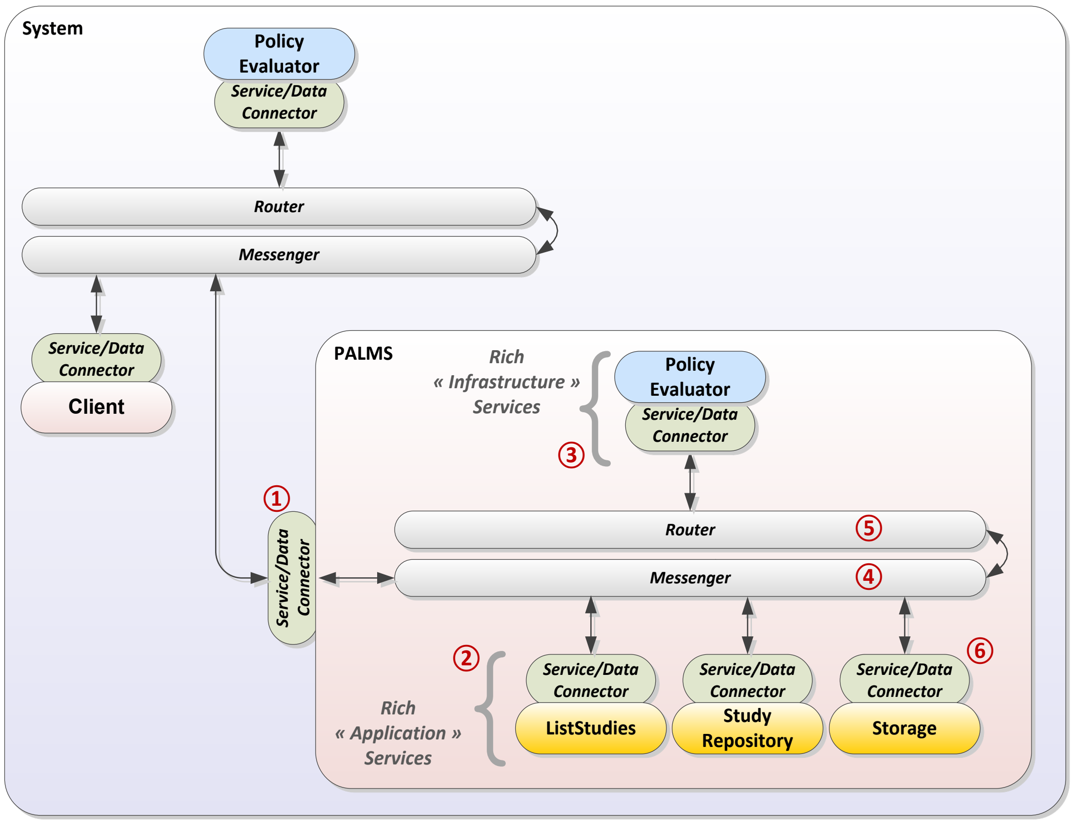


Figure 10. A Typical Rich Service

The SDC defines the RS's input streams and output streams, representing a definition of the RS's service interface (as a Bridge pattern [33] at the logical level, and a Messaging Gateway pattern [135] at the physical level). An SDC can be modeled in a number of ways, including as explicit mappings between input and output values (in the λ-calculus tradition described in Section 2.2.4), as lists of imported and exported function definitions [156], and as Message Sequence Charts.

A Rich Application Service (RAS) is a service that implements business rules and application processing supporting a dominant decomposition, and itself is modeled as a Rich Service. As such, it implements a service interface and can be implemented as a service orchestration or as an atomic action. Additionally, a RAS' SDC can perform translation services between the inputs presented to the RAS and the workflow that implements the RAS, and similarly between the workflow and the RAS outputs. In this respect, the RAS' SDC implements a Mediator pattern [33].

A Rich Infrastructure Service (RIS) is a service that implements behaviors that crosscut RAS orchestrations, and are therefore injected into those orchestrations in one or more place. Structurally, it is identical to a RAS and can be modeled as a Rich Service. For example, a requirement to encrypt a message travelling between a source and target service may be implemented as two RISs (an encryption RIS and a decryption RIS, not shown) where one RIS is injected to encrypt a message sent by the source service, and the other RIS is injected to decrypt a message received by the target service. (Under PDD, the RIS feature is used to evaluate policies that may lead to workflow composition, as described throughout this chapter.)

The message transport transfers messages between services, where the messages are formatted in some way compatible with the SDC of the source and target service.

The router is coupled to the message transport, and implements a RAS-to-RAS service interaction by accepting a message from a source service and routing it to the target service, as an example of the Mediator pattern [33]. It can also enable crosscutting processing by interposing a RIS into the interaction, thereby creating a RAS-to-RIS-to-RAS interaction, which leads to policy injection.

Considering that both RASs and RISs can be Rich Services, each with SDCs ⑥, RASs, RISs, and message routing of their own, they complete the Composite pattern.

For a RS, the existence of an SDC signals a decomposition implemented by an encapsulated orchestration, where either a single service or multiple services may be orchestrated to implement a workflow. For simple services, the SDC may be trivial (or non-existent), thereby leaving the service functionality to be implemented by the service itself.

The Rich Service shown in Figure 10 models the GetStudyList motivating example described in Chapter 3. It represents three levels of hierarchy:

- An interaction between the Client and PALMS services (within the System model)
- The decomposition of the PALMS service as an orchestration of the ListStudies, StudyRepository, and Storage services
- The possible decomposition of the Client, ListStudies, StudyRepository, and Storage services into their own service orchestrations (not shown)

Additionally, it shows two (possibly different) Policy Evaluator RISs. Within the PALMS RS, the RIS represents the possibility of policy injection on interactions between the PALMS SDC, ListStudies, StudyRepository, and Storage services. Within the System model, the RIS represents the possibility of policy injection on interactions between the Client and PALMS services.

It does not show the actual service interactions depicted in Chapter 3 – they would be encoded in the routing tables or rules implemented within the router. Likewise, it does not show the interfaces defined by each of the SDCs – they would be specified separately as attributes of each SDC.

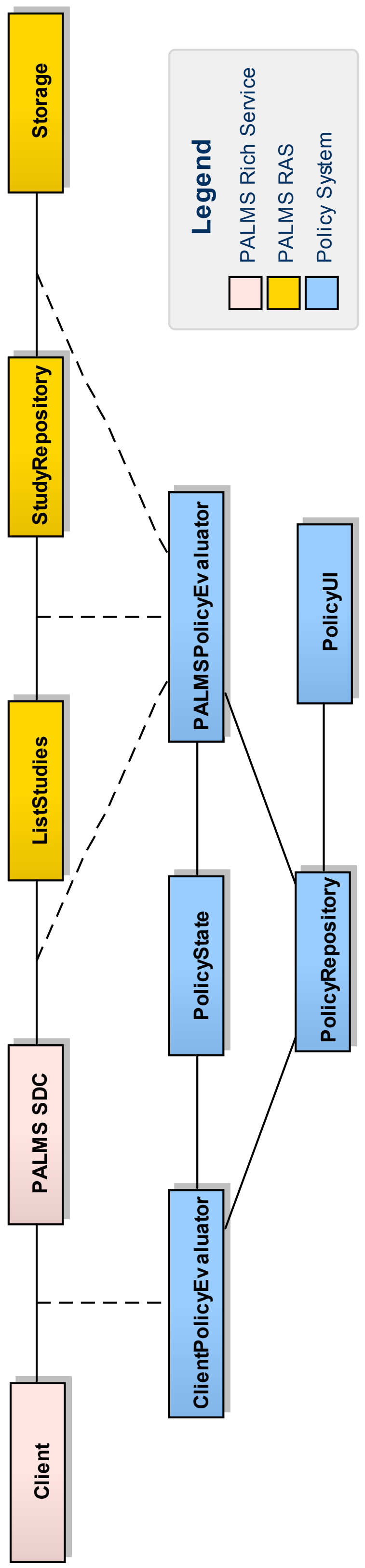
Note that RASs and RISs can maintain state that drives service behavior over time. Whereas such state is generally encapsulated within a service, sharing state amongst services can be modeled by a common shared service, such as Storage represents to the ListStudies and StudyRepository services.

#### *4.1.1.2 System of Systems (SoS) Composition using Rich Services*

From the perspective of the Rich Service in Figure 10, the Policy Evaluator RISs at the System and PALMS level are independent. However, given that RISs can share state, they can be modeled from another viewpoint as integrated, thereby implementing a single concern. As such, they can be modeled as RASs within a separate Rich Service-based policy support system. This represents a System of Systems (SoS) composition as described below.

Figure 11 presents the System and PALMS services and their relationship to Policy System services using a UML class diagram for brevity. It shows interaction relationships as encoded at the Rich Services router level. The classes along the top represent System and PALMS RAS services (from Figure 10), which are related by service interaction relationships. The second tier (including ClientPolicyEvaluator and PALMSPolicyEvaluator) represents System and PALMS RIS services (from Figure 10), which are related to the RAS interactions by association classes that model interception. The second tier and third tier combine to represent interacting RASs in the Policy System application, in which policy is stored in a policy repository, is defined by a policy UI, and is evaluated by PolicyEvaluator classes. The PolicyEvaluator services share state via a common policy state RAS. As such, this demonstrates the use of RIS injection to merge two independent applications as a SoS integration via service composition.

Figure 11. RIS Injection Achieves System of Systems Integration


Client
PALMS SDC
ListStudies
StudyRepository
Storage
ClientPolicyEvaluator
PolicyState
PALMSPolicyEvaluator
PolicyRepository
PolicyUI
Legend
PALMS Rich Service
PALMS RAS
Policy System

While the Policy System is an example of a complex application, and its integration with the base application is a high value integration, the same principles apply to the injection of other concerns into a base workflow – injected concerns can represent large, small, or trivial systems, each of which can be independent of each other or of the base application. In subsequent sections, I position policies and collection of policies as representatives (or actual implementations) of large, small, or trivial systems.

I explain the criteria and mechanisms for this composition in the following sections.

### 4.1.2 Service Definition

In this section, I describe how Rich Services (aka *services*) logically interact with each other, notwithstanding mediation by the message router. Specifically, I define how messages are exchanged and the meaning of message exchange, which form the basis for interception that enables policy injection. My definition of message exchange is based on Streams, as defined in [157] and [117] (as component refinement).

I define a relation implemented by Rich Service $RS$ as $f: \vec{I} \rightarrow \wp(\vec{O})$; $I$ and $O$ are disjoint sets of names of directed channels, where a channel contains a message $M$ at time $t$, called a *channel valuation* at time $t$. Relative to $RS$, $I$ represents input channels, $O$ represents output channels, and $I \rhd O$ defines the syntactic I/O interface of the service, represented by the SDC (as shown in Figure 12).

I define $C = I \cup O$ as the collection of names of input and output channels for a Rich Service. For a set of channel names $X \subseteq C$ and a set of messages $M$, I define $\tilde{X} \triangleq (X \to M^*)$, a set of time-ordered valuations of the channels named in $X$. Finally, I define $\vec{X} \triangleq \tilde{X}^\infty$, where $\vec{X}$ is the *infinite valuation* or *history* of the channels named in $X$ [158]. Therefore, $\vec{I}$ denotes the channel history for input channels, $\vec{O}$ represents the channel history for output channels, and $\vec{C}$ represents the channel history for both input and output channels.

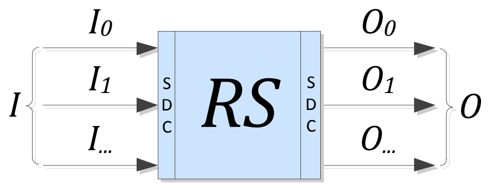


Figure 12. Rich Service with Input and Output Channels

I define a *service contract* as the set of channel valuations for which $\vec{I} \to \wp(\vec{O})$ in the context of $RS$ where $\vec{O}$ can be generated by $RS$ given input $\vec{I}$. $RS$ is said to *fulfill* a service contract if it maps $\vec{I}$ to $\vec{O}$ as defined in the service contract.

(Note that this view of Stream-based semantic compatibility is more restrictive than the semantic-free view present in Petri Nets presented in Section 2.2.2 -- it leads to an understanding of compatibility between services, including which services can substitute for other services. This view of service contracts is complimentary to the Design by Contract [159] (DbC) view, where a service is defined by pre-conditions, post-conditions, and invariants. It is also complimentary to Interface Oriented Design [160] (IOD), which advocates the separation of interface from implementation. Rich Services defines interfaces more broadly as message streams, and defines correct execution in terms of valid output streams reactive to valid input streams, which enables service interactions to be framed in terms of protocols and guarantees.)

Finally, I define $RS$ as a relation $f : \vec{I} \rightarrow \wp(\vec{O})$, which produces the power set of $\vec{O}$ because $RS$ can execute non-deterministically.

Note that this definition allows $f$ broad discretion in choosing how to map $I$ to $O$; $f$ itself determines which input channels to read, when to read them, which output channels to write, what to write onto them, and when to write them. This discretion applies to both RASs and RISs.

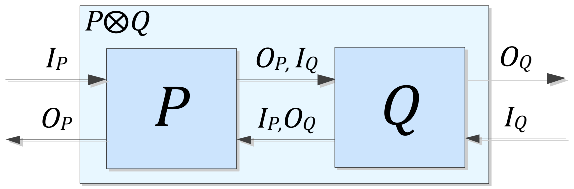


Figure 13. Composing Two Services P and Q

Note that the execution of $f$ may incorporate choice in its computation. CSP [113] differentiates internal and external choice, where internal choice leads to non-determinism, and external choice relies on external state. When modeling external choice, external state is accessed via one or more channels constituting $I$.

### 4.1.3 Service Composition

Two services, $P$ and $Q$ are said to interact when at least one output channel of $P$ is connected to an input channel of $Q$, or vice versa, and I use the notation $< P, Q >$ to represent the interaction of $P$ and $Q$. More precisely, I define a *service composition* between the two services as $P \otimes Q$ where some of $Q$'s input channels (called $I_{Q_h}$) are connected to some of $P$'s output channels, and vice versa, as shown in Figure 13. Such channels are called *hidden channels* $H_{P \otimes Q}$, and include $I_{Q_h} = O_p, I_q$ and $I_{P_h} = I_p, O_q$ in the figure. The input and output channels of $P \otimes Q$ are all of the input and output channels of $P$ and $Q$ except those in $H_{P \otimes Q}$. The service contract defined for hidden channels is called the *hidden channel service contract*.

Note that the $P \otimes Q$ definition allows the composition broad discretion similar to $RS$ defined above, and channel valuations and channel history for the composition is defined similarly to $RS$. Channels not connecting the two services are free to connect with other services.

In Figure 10, the StudyRepository and Storage services might be composed to realize retrieval from a repository.

### 4.1.4 Service Decomposition

I define decomposition of a Rich Service $RS_D$ as an orchestration of sub-services to achieve the functionality of $RS_D$, given as $RS_D: \vec{I} \rightarrow \wp(\vec{O})$ (from above). A service orchestration can result from the conditional or unconditional interactions of serially and concurrently executing services, thereby implementing the workflows described in [70]. $RS_D$ also denotes the collection of sub-services that interact (via composition) to implement its functionality:

$$RS_D = \bigotimes_i RS_{D_i}$$

where $RS_{D_0}$ is the SDC for $RS_D$; the SDC initiates (and may terminate) the orchestration. The orchestration consists of the interactions $< P, Q >: P, Q \in RS_D$ where the service contracts for each service $P$ and $Q$ are fulfilled.

The orchestration is driven by a collection of services $R$ that perform routing functions that compose sub-services. For each sub-service $P$ and $Q$ paired in a composition, $R$ contains a service $R_{PQ}$ where the composition is implemented as $\mathrm{P} \otimes \mathrm{R}_{\mathrm{PQ}} \otimes \mathrm{Q}$ where $\mathrm{R}_{\mathrm{PQ}}$ mediates between $\mathrm{P}$ and $\mathrm{Q}$ (as shown in Figure 14). In this example, $RS_D = \{\mathrm{P}, \mathrm{R}_{\mathrm{PQ}}, \mathrm{Q}\}$.

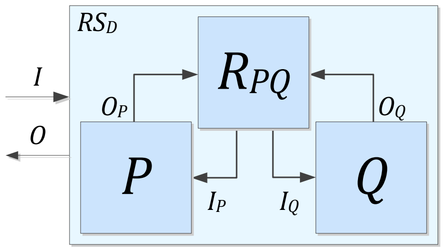


Figure 14. Rich Services Routing Service (with sub-services; SDC not shown)

In Figure 10, the composition of the Study Repository and Storage services is mediated by a routing function in the message router ⑤, which contains $R$.

### 4.1.5 Message Interception

$R_{PQ}$ must fulfill the hidden service contract for $P \otimes Q$ in its interactions with both $P$ and $Q$, and can do so simply by acting as a pass-through for their hidden channels. It is also free to interact with a different service $W$ instead of $Q$, provided that $W$ fulfills $Q$'s service contract.

Beyond this, $R_{PQ}$ is free to interact with RISs provided it continues to fulfill its own service contracts. Considering that a RIS is a Rich Service that implements a workflow, a routing function interacting with a RIS equates to composing a workflow onto the base workflow represented by $P \otimes Q$.

In Figure 10, a router function is defined for each possible pair of interacting services (including the PALMS SDC and each of the three RASs). Each router function may communicate with the Policy Evaluator RIS (or any other defined RIS), which can conditionally execute any service (either synchronously or asynchronously), and may replace target service with an alternative (e.g., a service that returns an access control error), provided the alternative fulfills the service contract in effect between the source service and the original target service.

### 4.1.6 Services and Workflows

Under Rich Services, workflow actions are represented as Rich Application Services (RAS), and data flowing between actions are represented as messages. A workflow (called a *base* or *target* workflow) is implemented as a service orchestration by routing messages between RASs.

Crosscutting concerns are implemented by intercepting such messages and routing them to Rich Interface Services (RISs), which implement *injected* workflows that process the messages.

Rich Services (including both RISs and RASs) can be decomposed into orchestrations of finer grained RASs representing finer grained abstractions. Accordingly, Rich Services enables workflow injection at each level of this abstraction hierarchy, as evidenced in Section 4.1.1.2

## 4.2 Policies

Under PDD, a *policy* is a decision that affects the data flowing between services or the operations performed on the data. Conceptually, it leverages the duality between service orientation and data flow orientation. Under a service-centric view, a service represents an action consuming input channels and producing output channels; under a channel-centric view, input channels beget output channels via a transform implemented as a service.

### 4.2.1 Policies and Workflows

To understand how policies can cause an alternate workflow to be composed onto a base workflow, consider a simple service interaction consisting of a one-way exchange:

$$P \xrightarrow{m:T} Q$$

where $P$ is a source service, $Q$ is a target service, $m$ is a message of type $T$, and the service interface for $Q$ supports a service interaction $< P, Q >$that accepts a message of type $T$. A refinement of this relationship allows the choice of an alternate target service as shown below and in Figure 15:

$$P \xrightarrow{m:T} [\Pi(\pi(\mathrm{P}, Q, K)]$$

where $\Pi$ is a policy evaluator service that returns $S$, a service having a service interface that includes $Q$'s service interface (relative to $P$), and may, in fact, be $Q$ itself. $\pi$ is a policy expression that is evaluated by $\Pi$ (by calling internal evaluator service $\Gamma$), subject to some context $K$, and returns $S$. The [] notation dereferences the result of the $\Pi$ service, thereby specifying the actual interaction target. Because $P$ now interacts with $[S]$, I say that policy $\pi$ on $(P, Q, K)$ results in the substitution of $[S]$ for $Q$ in the service interaction. The context $K$ is the information $\pi$ references in its policy evaluation – its content is implementation specific, but at least includes the message $m$ (as $I_{Q_h}$ defined in Section 4.1.3) and may include application, environment, workflow, or other state per Section 4.3.

Note that for $[S] \neq Q$, the policy evaluator is iterated on the interaction $< P, [S] >$, thus allowing the evaluation of policy defined on $P \rightarrow [S]$.

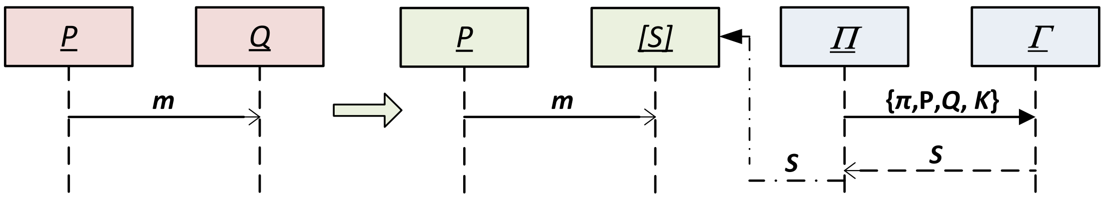


Figure 15. Service Refinement for Alternate Choice

$\Gamma$ has wide latitude in choosing an appropriate target service $S$ so long as the target fulfills $Q$'s service contract, it is not constrained as to its activity or effect. Exploiting this allows such policies to inject services that execute workflows implementing application features as separate concerns maintaining their own state. Further, policies are also free to coordinate with each other through shared state independent of the workflows on which they are composed. For example, a control policy that sets state can combine with a filter policy that tests state and executes a workflow, thereby enforcing an obligation. While state is conceptually present in channel histories, it is more conveniently accessed according to Section 4.3.

Below are examples using my own notation, where if $X$ is a service, then $X: S_0 \rightarrow S_1$ denotes replacing $X$ with an orchestration that invokes $S_0$ followed by $S_1$, and $X: S_0 \,||\, S_1$ denotes replacing $X$ with two services $S_0$ and $S_1$ both of which receive $X$'s message and execute concurrently.

Below, message $m$ is routed to $F$, which transforms $m$ into $m_1$, which is routed to $Q$. $F$ acts as a filter transformation.

$$P \xrightarrow{m:T} X \text{ where } X: F \xrightarrow{m_1:T} Q$$

Below, message $m$ is routed to $B$, a workflow that implements one or more *obligations* [161] (defined as altering or acting upon persistent state, but not altering message $m$) before $m$ is routed to $Q$.

$$P \xrightarrow{m:T} X \text{ where } X: B \xrightarrow{m:T} Q$$

Below, message $m$ is routed to both $Q$ and $B$, which execute concurrently.

$$P \overset{m:T}{\longrightarrow} X \text{ where } X: (Q||B)$$

An inventory of workflow configurations is given in [70].

### 4.2.2 Compound Service Interactions

While the one-way interaction is simple, policy can result in workflow substitution in more complex interactions following the same principles. An important example is a request/reply interaction on $< P, Q >$ below:

$$P \overset{q:T_q}{\longrightarrow} Q \text{ followed by } Q \overset{a:T_a}{\longrightarrow} P$$

where $q$ is a query message, $a$ is an answer message, and $a$ is correlated with a particular $q$. Both $q$ and $a$ play the role of message $m$ above. In a request/reply interaction, a replacement service $S$ fulfills $Q$'s service contract if it returns an $a$ to a particular instance of $P$, just as $Q$ would.

A policy that transforms $q$ before invoking $Q$ is called a *pre-filter*, and a *post-filter* is a policy that transforms $a$ before returning it to $P$. A policy that replaces $Q$ with some service $S$ is called a *control policy*. All three policy types can apply to a single service interaction, either alone or in combination. Using Figure 6 as an example, a pre-filter policy might qualify the query (on ❹), a post-filter policy might remove undesired data (on ❺), and a control policy (on ❹, after a pre-filter policy) might prohibit data access for uncredentialed users.

In the general case of multiple interactions between $P$ and $Q$, both $P$ and $Q$ must fulfill the service contract governing the interactions, and the contract is bound to the specific instances of $P$ and $Q$ in play. The service contract can be discovered in various ways, including via a service registry. Under the simplifying assumption of a request-reply interaction, the registry maintains $T_a$, the type of reply due $P$ from $Q$, and $S$ must return a reply of type $T_a$.

Note that it is not guaranteed that a replacement service $S$ can function or produce full benefit under the service contract defined by $P$. For example, suppose $S$ must return a result $a$ of type $T_a$ not acceptable to $P$. This fundamental mismatch can properly be recognized as a new requirement on the base workflow – the requirement can be captured and the base workflows can be evolved in an orderly manner. Recognizing the mismatch can be done at runtime (if typed messages are exchanged and typing is checked) or at design time (given service modeling or model checking support [162] that ties in with the policy language), and is beyond the scope of this dissertation, though is considered further in Section 7.8.1.

### 4.2.3 Control Policies

Leveraging the service-centric view, a *control policy* $cp$ calculates a service $W_{cp}$ to replace service $Q$ (in $P \otimes Q$) such that $W_{cp}$ fulfills $Q$'s service contract (and could be $Q$ itself). $W_{cp}$ is calculated by $cp_{PQ}(P, Q, K)$, $P$ is the source service, and $Q$ is the default target service, and $K$ is the workflow context (defined in Section 4.3, and including inputs $I_{Q_h}$ as message $m$). The corresponding router interaction (per Section 4.1.4) is $P \otimes R_{PQ} \otimes W_{cp_{PQ}}$ where $R_{PQ}$ is composed with the calculated service $W_{cp_{PQ}}$ instead of the default service $Q$.

A control policy has wide latitude in choosing $W_{cp}$, and it usually consists of *control policy expression* $cpe_{PQ}(P, Q, K)$ that acts as a decision function by choosing between various pre-identified candidate $W_{cp}$ services -- the $cpe$ is also free to create and return a novel $W_{cp}$ on the fly. The $cpe$ can reference any criteria, including the input channel history $\vec{I}_{Q_h}$, the $P \otimes Q$ channel history $\vec{C}_{PQ}$, or the channel history for other interactions in the system.

A control policy is authored by a stakeholder or policy programmer, and is inserted into a policy repository indexed by the service interaction $< P, Q >$ to which it applies. It represents a means by which new workflows can be composed into a base workflow pursuant to emerging stakeholder requirements.

Note that a valid control policy may return a service $W_{cp}$ that incorporates a replacement service *composed* with the default service $Q$. The composition could be serial, parallel, or complex, sufficient to implement the intent of the policy. Furthermore, service $Q$ may itself fulfill its service contract, or fulfillment can be delegated to another component of the composition.

### 4.2.4 Filter Policies

Leveraging the channel-centric view, a *filter policy* $fp$ calculates a service $W_{fp}$ to intercept $Q$'s inputs $I_{Q_h}$ (as message $m$) and replace them with other values $I'_{Q_h}$ that are valid inputs to $Q$. $fp$ is a function $fp_{PQ}(P,Q,K)$ whose arguments are the same as for $cp$. The corresponding router interaction is $P \otimes R_{PQ} \otimes W_{fp_{PQ}} \otimes Q$.

Similar to a control policy, a filter policy acts as a decision function that has wide latitude in choosing $W_{fp}$, which consists of a filter policy expression $fpe_{PQ}(P,Q,K)$ that either effects the channel value transformation or simply returns $I_{Q_h}$ without any transformation.

A *filter policy expression* can reference the same criteria as a control policy expression, and is authored and stored in the same manner. It represents a means by which data flows can be constrained, augmented, inspected, or replaced consistent with emerging stakeholder requirements.

The router function $R_{PQ}$, executes both control and filter policies for an interaction, and injects the resulting workflows. When both a control and filter policy exist for an interaction, the filter policy is executed before the control policy, and the input channel is transformed before the target service is executed. In Figure 14, a control policy replaces the service $Q$, while a filter policy replaces the *hidden channel* $I_{Q_h}$.

### 4.2.5 A Simple Policy Evaluation Service

A simple example of policy evaluation is a control policy (Section 4.2.3) that makes an Allow/Deny decision involving the $< StudyRepository, Storage >$ interaction in the GetStudyList workflow in Chapter 3. The objective is to evaluate available information to determine whether to allow the Storage interaction or to return an error message instead. In this example, I make a simplifying assumption that a user credential is available in the request message ❹, and that the Allow/Deny decision is based on it – Figure 6 doesn't show this, and Section 4.3 describes a more likely source for user credential information.

Figure 16 is a UML sequence diagram that shows services in a policy evaluation as described in Section 4.2.1. It assumes that the $P \otimes Q$ composition (representing $StudyRepository \otimes \mathrm{Storage}$) adheres to a request/reply pattern (as described in Section 4.2.2), and that the $P$ service contract supports an error return. The request message $I_{Q_h}$ (included in context $K$) contains the user credential. A control policy $\pi$ is associated with the interaction $< P, Q >$ in a policy repository, and it evaluates $I_{Q_h}$ to return either the default service $Q$ (for the Allow case) or a replacement service $E$ (for the Deny case), which in turn returns an error.

In more detail, the router function $R_{PQ}$ delegates the policy evaluation to the policy evaluator RIS $\Pi$, which implements a three step workflow: it fetches the control policy $cp$ (i.e., $\pi$), executes it , and returns the workflow $W_{cp}$ (i.e., $Q$ or $E$). The $cpe$ is expressed in a language (e.g., XQuery) that can be interpreted by the evaluator service $\Gamma$ (not shown here, but described in Section 4.2.1).

An expression could be as simple as "$\text{if}\ (I_{Q_h}.\text{user} = \text{"bob"})\ \text{then}\ Q\ \text{else}\ E$", where a user identity is fetched from the input channel and is compared to a static string.

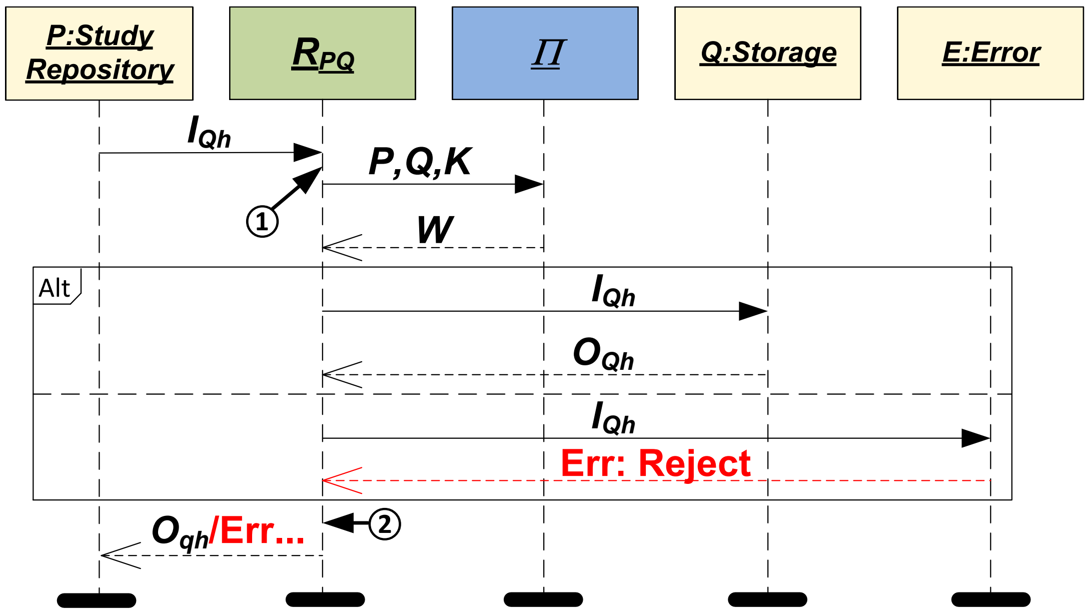


Figure 16. Simple Allow/Deny Policy Evaluation

To fetch and evaluate a pre-filter policy (not shown in Figure 16), $\Pi$ can be called at ① to fetch and evaluate the policy $fp$ and return $W_{fp}$. If $fp$ exists, $W_{fp}$ is executed to transform $I_{Q_h}$ before executing the access control policy.

Note that apart from the request interaction between $P$ and $Q$ (or $E$), the reply interaction between $Q$ (or $E$) and $P$ is eligible for interception by a post-filter policy. (It is not eligible for interception by a control policy because under a request/reply pattern, the service contract for $P$ requires a response, which the router function $R_{PQ}$ must guarantee.) If the post-filter exists, $O_{Q_h}$ is transformed and fed to $I_{P_h}$, which $P$ consumes. Figure 16 shows the router function relaying the result $O_{Q_h}$ to $P$, but does not show a call to $\Pi$ (at ②) to return $W_{fp}$ or the transformation of $O_{Q_h}$ before relaying it to $P$.

In this example, applying a pre-filter to the inbound message $I_{Q_h}$ is useful for altering or qualifying a query request to the Storage service. Applying a filter to the outbound interaction $O_{Q_h}$ is useful for altering, augmenting, or decimating the study collection returned by the Storage service. While pre- and post-filters can be used to enforce access control in this way, they can also be used to enforce HIPAA-style requirements that call for altering or augmenting data.

Because the user credentials may not be contained in an outbound channel (which contains a study collection, in this example), a post-filter would rely on access to credential using means described in Section 4.3.

### 4.2.6 Feature Injection and Obligations

Note that a control expression $cpe$ has wide latitude regarding the process it uses to determine which service $W_{cp}$ it returns. The expression may call services and instigate separate workflows so long as the expression returns a service that fulfills the service contract of the default service. Filter expressions $fpe$ have similarly wide latitude provided they return an appropriate (or no) service. Particularly, a feature can be composed into an interaction by a control or filter expression by interacting with a service that implements it. The feature service may or may not terminate before the completion of the expression. Similarly, a feature may be composed into services returned by control and filter expressions.

For example, a feature that produces an audit stream may synchronously or asynchronously log the contents of an interaction message. There are several ways to implement such a feature, including:

- within a control expression, invoke the audit service and return the default service $Q$ (or any other appropriate service)
- within the service returned by a control expression, invoke the audit service and then the default service $Q$ (or any other appropriate service)
- within a filter expression, invoke the audit service and return no service (or an appropriate other service)
- within the service returned by a filter expression, invoke the audit service and then a filter service (including a passthru filter)

An independent workflow maintains state pertinent to meeting its requirements. However, the policy decision ($cpe$ or $fpe$) that returns the workflow can, itself, maintain state via SIVs and IVs as in Section 4.3. Such state can be accessed in future control and filter decision functions, or can constitute parameters to an injected workflow. Policies and groups of policies that maintain and consume state are, themselves, injectable features in that they implement a discrete requirement set independent of their relationship to base workflows. An important use of policies-as-applications is in implementing *obligations* [161], which constitute actions to be executed in the future based on past decisions, and in creating a System of Systems (SoS) as described in Section 4.1.1.2.

### 4.2.7 Policy Composition

A service interaction may be subject to multiple filter or control policies, particularly when multiple independent stakeholders each propose requirements that affect the interaction. Synthesis of multiple policies cannot be done naively, as a synthesis that accounts for the intent of all stakeholders is itself a matter of application design, not mechanics.

For example, given inputs $I_{Q_h}$, filter policy $fp$ returns service $W_{fp}$, which produces replacement inputs $I'_{Q_h}$ (per Section 4.2.4), as does service $W'_{fp}$ returned by policy $fp'$. Casting a service as a function, $I'_{Q_h} = W_{fp}(I_{Q_h})$, two filter services $W_{fp}$ and $W'_{fp}$ maintain the commutative property under composition if $W'_{fp}\big(W_{fp}(I_{Q_h})\big) = W_{fp}\big(W'_{fp}(I_{Q_h})\big)$. Logically, commuting filter services can be executed in any order, and so serial application results in a logically consistent result, and a composition policy that executes one filter after another would be appropriate.

However, factors such as the administrative or security domain that supplied the policy may dictate a hierarchical approach to correctness, where if $fp$ returns a service filter, $fp'$ should not be executed at all. For example, if $fp'$ represents a default filter, and $fp$ represents a user-supplied filter, the user filter should replace the default filter. Other approaches to correctness are possible, too.

With non-commuting filters, dependency relationships (based on either external state or the contents of $I_{Q_h}$) are violated, and composing such filters requires a means for determining execution order and compatibility. For example, filter services $W_{fp}$ and $W'_{fp}$ must be ordered if $W_{fp}$ deletes information on which $W'_{fp}$ relies. They are incompatible if each deletes information on which the other relies. Additionally, such filters are subject to the additional administration and security that apply to commuting filters.

Determining whether filters commute (either logically, administratively, or securely) requires deep inspection of filter code and an understanding of the environment in which the filters are deployed, and are beyond the scope of this dissertation.

Composition of control policies poses similar considerations. Section 4.2.3 describes the combination of control policies as serial, parallel, or complex compositions. While control policies don't transform messages in flight, they can modify system state, and can therefore be order dependent or incompatible. Additionally, administrative and security concerns may drive ordering and compatibility decisions. For example, given control policies $cp$ and $cp'$ representing access control decisions contributed by different domains, each policy can generate its own error message if their access control criteria are not met. While serially composing these policies allows resource access only if both sets of criteria are met, the particular error message returned may be a function of which policy is executed first, and would follow scenarios similar to those for filter policies.

In general, composition policies address the combination of filter or control policies as policies on policies. A composition policy $\chi$ is a policy that combines policies consistent with the service interaction to which those policies apply individually, and consistent with the application's administrative and security concerns. Composition policies exist separately for control policies, pre-filter policies, and post-filter policies.

For each interaction $< P, Q >$, the policy evaluator executes a separate four stage workflow for pre-filter, control, and post-filter policies. In general:

- retrieve a collection $PE_{PQ}$ of policy expressions $pe_{PQ}$ from one or more policy repositories
- retrieve a *composition policy expression* $\chi_{\mathrm{PQ}}$ from a composition policy repository
- evaluate $W_{PQ} = \chi_{PQ}(PE_{PQ}, P, Q, K)$
- return $W_{PQ}$

For composing control policies, if $|PE_{PQ}| = 0$, $Q$ is returned. For composing filter policies, if $|PE_{PQ}| = 0$, nothing is returned. If $\chi_{\mathrm{PQ}}$ is undefined and $|PE_{PQ}| > 0$, an error occurs. (For control composition policies, $W_{PQ}$ is a control service, and for filter composition policies, $W_{PQ}$ is a filter service.)

Analogous to the Simple Policy case in Section 4.2.5, $\chi_{\mathrm{PQ}}$ is the composition policy expression specific to the $\mathrm{P} \otimes \mathrm{Q}$ composition, and has wide latitude in evaluating members of $PE_{PQ}$ and combining the resulting workflows.

An example of a simple $\chi$ policy is a multi-expression Allow/Deny control policy that Allows if all policy expressions Allow, or chooses an error to return if at least one policy expression Denies. It calculates a collection $W$ of workflows generated by members of $PE$: $W = \{w : \forall pe_{PQ} \in PE_{PQ},\ w = pe_{PQ}(P, Q, K)\}$, and returns $Q$ if $\forall w \in W, w = Q$; else $w \in W, w \neq Q$ (where the choice of the particular $w$ is determined by the composition policy).

While the router function $R_{PQ}$ executes services based on filter and control policies, it calls the policy evaluator RIS $\Pi_{\chi}$ to calculate filter and control compositions. ($\Pi_{\chi}$ is not the simple RIS $\Pi$ defined in Section 4.2.1, though it serves the same function. Instead, $\Pi_{\chi}$ executes the composition calculation workflow defined in this section, which in turn calls the simple RIS $\Pi$.)

## 4.3 Context System

While the injection of a policy into a workflow is a critical contribution to the rapid realization of stakeholder requirements, the ability to *coordinate* policies injected into a single interaction or multiple interactions is key to realizing requirements that produce or depend on state, especially *within* a workflow (as exemplified by the Work Study Flag in the SOARS Bursar workflow in Chapter 2). A key contribution of my dissertation is a design for a Context System, which enables composed workflows to maintain private or shared state either during or beyond the execution of a target workflow. Ultimately, this capability enables a collection of policies to act as a separate, composable application in its own right (assuming appropriate restrictions on access to policy state from other policies), thereby powering System of

Systems integration. Recognizing that a Context System can be implemented in a number of ways, this section presents a design that harmonizes with the policy foundations laid in Section 4.2 – a concrete implementation is presented as part of the PALMS case study in Chapter 5.

In order to map $\vec{I}$ to $\vec{O}$ per Section 4.1.2, a service may rely on its own state, state maintained by other services, or the channel history for any service. (Arguably, except for services' initialization states, the state of any service derives from its channel history, and discrete state is a convenience that captures a part of channel history for later use. A service's access to external state can be considered a channel into or out of a service that manages the external state.) When composing a policy onto a base workflow, a key challenge is in correlating the policy service state with the base workflow instance to which it applies, then extinguishing the state when the base workflow is complete. The Context system meets these challenges by creating and managing state containers tied to workflow instances (observing their lifecycles) and providing composed policies with access to that state.

As in Section 4.2.1, a policy $\pi$ is a service that returns a service $W$ responsive to a set of inputs $K$. Relative to a base workflow, policy $\pi$ represents a separate concern equivalent to a requirement composed on a set of base requirements, and its choice of $W$ is based on its own state independent of state maintained by the base orchestration. Furthermore, two policies $\pi_0$ and $\pi_1$ may share state $s_{\{\pi_0,\pi_1\}} \in K$ in selecting their respective services $W_0$ and $W_1$.

When policies $\pi_0$ and $\pi_1$ cooperate, one depending on $s_{\{\pi_0,\pi_1\}}$ generated by the other, they form a separate workflow whose data flow is mediated by $s_{\{\pi_0,\pi_1\}}$.

Each state $s_{\{\pi_0,\pi_1\}}$ is associated with a state collection called a *SIV* (service interaction-related values) or *IV* (independent values), and a context $K$ can contain numerous SIVs, IVs, and AEVs (application and environment-related values) as shown in Figure 17.

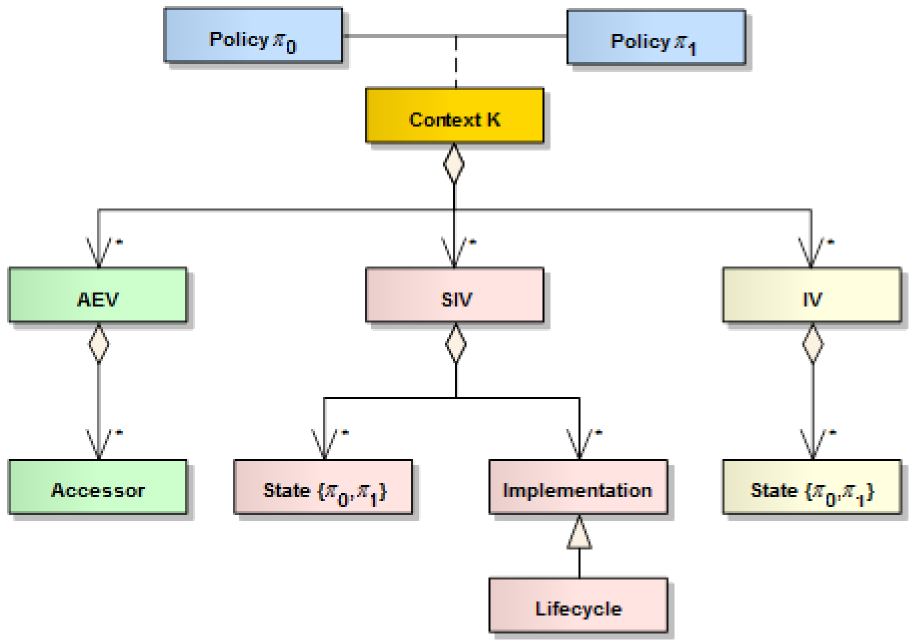


Figure 17. Context Containing AEVs, SIVs, and IVs

A SIV is a container of uniquely named states (e.g., $s_{\{\pi_0,\pi_1\}}$) that have similar lifecycles or other implementation characteristics – SIV contents may be produced or consumed by one or more services (e.g., policies) composed onto the same or different base workflow instances. I define $\Sigma$ as the set of all SIVs (i.e., $SIV_i$) where

$$\Sigma = \biguplus_i SIV_i$$

and $SIV_i$ contains a set of states $s$ common to one or more policies. (Note that the $\uplus$ operator specifies a *disjoint union*, which is a union that maintains identical sets as distinct members.)

IVs are similar to SIVs, except their lifecycles are determined explicitly by the services that create and use them. AEVs include application and system states, and have lifecycles independent of workflows.

Whereas service composition supports the separation of concerns into base concerns and policy-injected concerns, execution contexts support this separation by enabling independent data flows between services implementing those concerns. In this discussion, I consider SIVs as carriers for data exchanged between services that implement separate concerns composed into a base workflow. Three prominent PALMS SIVs are interaction messages $m$, Workflow SIVs, and Session SIVs.

A message $m$ carries data shared by two service instances. The data is created by the source service, and is consumed (or destroyed) by the target service, as described in Section 4.2. (Note that for request/reply interactions, the request message $q$ – as described in Section 4.2.2 – is accessible in the reply interaction as a SIV element.)

A Workflow SIV carries data that applies to a single instance of a base workflow. The data is created and accessed by concerns (e.g., policy) composed onto the workflow, and is destroyed when the workflow is complete. Data for each concern is named by convention to be unique to the concern, so as not to interfere with another concern's data.

A Session SIV is similar to a Workflow SIV, except that its data is created and accessed by concerns composed onto multiple instances of a workflow or onto multiple workflows. Its duration is determined by some external criteria, such as the lifecycle of a client-based workflow.

Other SIVs are possible, as demonstrated in Section 5.5.4.

Consistent with SOA principles, service and policy execution can occur within a distributed computing system, with different services executing on different platforms, and possibly including services running in different processes. Consequently, SIVs must be available to services and policies wherever they execute, and cannot depend on values available in a common memory space. SIVs may contain actual data elements (if the data lifecycle spans only a service interaction – for example, message $m$) or a reference to a data container (for data that spans service interactions). References are globally unique (GUIDs) resolved in a globally available Context Infrastructure Service (CIS), which itself can be distributed, and access to which can be regulated by policy.

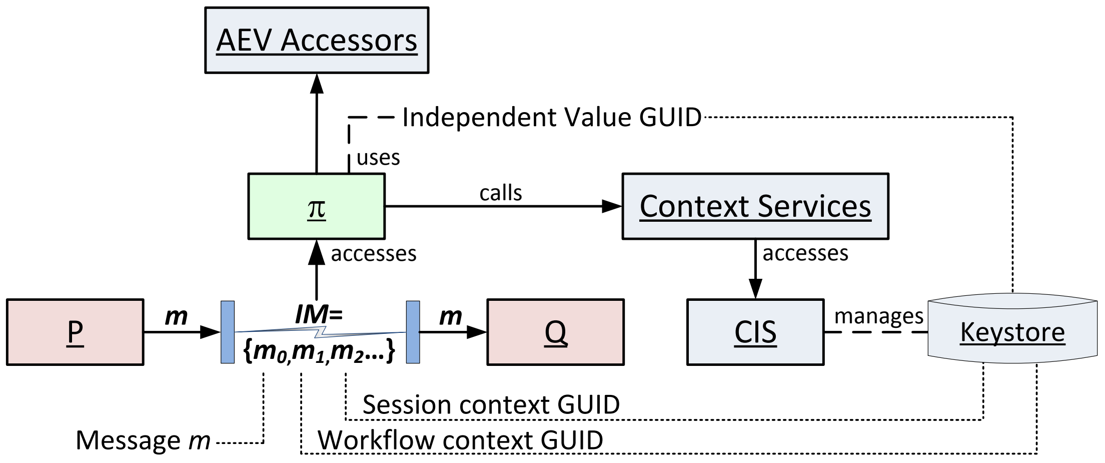


Figure 18. Context Management

When multiple instances of a base workflow exist, SIVs must be properly correlated with instances of service interactions. The context system maintains the association between a base workflow and its SIV values by composing them (or their references) into the workflow service orchestration's message flow as shown in Figure 18. Each interaction message $m$ in $P \overset{m}{\rightarrow} Q$ is replaced with an *interservice message* $IM$, which is a composition of all SIVs relevant to an orchestration: $P \overset{IM}{\rightarrow} Q$, where $IM$ exists for the sole purpose of conveying an interaction message along with its context (as inspired by π-calculus, per Section 2.2.3) from a source service to a target service as formally defined:

$$IM = \biguplus_{i} m_i$$

$$m_0 = m$$
$$m_i = SIV_{i-1}\, for\ i > 0$$

$IM$ is created immediately when $P$ sends $m$, and $m$ is extracted from $IM$ immediately before $Q$ is invoked, thereby fulfilling the service contracts between $P$ and $Q$. (In a SOA context, this substitution can be performed via interceptor services injected onto each service interaction, as demonstrated in Section 5.5.4.) I use $m_0$ to refer to the service interaction message $m$ wrapped in an $IM$, and each $m_i$ is a distinct, different SIV that represents a collection of states. (A message $m$ is also a collection of states, though with a lifecycle lasting for only a service interaction, and therefore is also a SIV.)

Using Context Services (as in Figure 18), a policy or feature (represented by $\pi$) can maintain its own state in a SIV, and can control the lifecycle of the state subject to the SIV lifecycle. For example, a value (e.g., a counter or Boolean) set by a policy in a Workflow SIV can be tested by the policy later in the workflow, and is automatically deallocated at the end of the workflow. Similarly, a session value can be set in a Session SIV, can be tested across multiple workflows sharing the session, and is automatically deallocated when the session is terminated.

(While Figure 18 portrays the conceptual relationships between a pair of existing services, injected policy, and context management, Figure 33 (explained in Section 5.5.4.1) shows a set of components, control flows, and data flows that implement these relationships in the PALMS case study.)

Common examples of SIV values include user credentials, channel histories for important workflow services, a service history for a particular thread, and so on. Particularly, features such as failure detection/mitigation and information assurance can use histories to reason about the state of an application. Additionally, features can be implemented by a combination of policies (as in Section 4.2.6), where some policies save important channel values during a workflow, and other policies act upon them.

Unlike SIVs, AEVs are state defined and maintained by the application or the environment, and are not associated with a particular workflow. AEVs are accessible by policies and features through interaction with application and environment services. Examples of AEVs include values in application repositories, the system time of day, and application analytics. Note that policies can be defined on interactions with application and environment services, thereby creating security and additional value added processing connected with accessing AEVs.

Note that while Workflow and Session SIVs establish context relative to workflow-oriented lifecycles, IVs allow policies to establish persistent state independent of these lifecycles by interacting directly with the CIS (via Context Services) to store and retrieve values. Such policies must choose context GUIDs that do not conflict with GUIDs that could be chosen for SIVs, and these GUIDs must be communicated amongst the policies via out-of-band means.

## 4.4 Addressing Gaps Identified in Existing Choice Mechanisms

In the sections above, I addressed many of the gaps identified in Section 2.7.6 (and summarized in Table 6). The service and composition definitions in Sections 4.1.2 and 4.1.3 describe the basic mechanisms underlying the injection of a crosscutting concern on an interaction between two services, while Section 4.2 describes the particular mechanisms of policy injection as relates to control flows, data flows, and policy compositions. Consequently, they demonstrate means by which choice can be late-bound to a workflow without requiring workflow recoding and re-release, thereby enabling the runtime injection of workflows that implement stakeholder requirements. Section 4.3 describes a context system that allows the implementation of crosscutting concerns to be stateful, thereby lifting policy injection to a System of Systems composition. This chapter does not address policy verification or testing of policies and injected concerns – these topics are considered further in Sections 7.5 and 7.8.

Table 6. Gaps Addressed Foundationally

| Section | Gap |
|---|---|
| **4.2.2** | Identification of policy injection site at runtime |
| **4.2** | Injection of policy at runtime |
| **4.3** | Tracking workflow-based policy -centric state |
| **4.2.7** | Composition of multiple policies onto a single injection site |
| **4.1.3** | Enabling composition onto injected workflow |
| | Verification of interface and semantic compatibility between policy and base workflow |
| | Incremental testing and proofs that policies implement requirements |
| | Enabling a consistent relationship between state and policy across policy deployments |

## 4.5 Summary

In this chapter, I have explained the principles behind policy-oriented service composition, where a policy evaluation concern crosscuts all service interactions in a Service Oriented Architecture. This composition occurs at the service level, thereby enabling the policy-defined injection of workflows representing application-level crosscutting concerns.

As groundwork, I described a formal relationship between services, and extended the relationship to describe general composition as interception. Next, I defined the conditional injection of workflows onto base workflows (mediated by a policy evaluation interceptor), and described control and filter policies that enable the transformation of both data flows and control flows. I also described a strategy (as composition) for integrating multiple policies defined on the same service interaction. Finally, I described how policies can maintain and access state, thereby creating injectable concerns that are in fact systems in their own right. The ability to define the injection of a simple workflow, of stateful workflows, or of systems of state-sharing workflows onto base workflows parallels the composition of requirements onto a set of base requirements, and is therefore a suitable implementation of both simple and complex requirements onto existing applications.

In defining policy evaluation as the processing of collections of policies mapped to service interactions at runtime, I demonstrate how such requirements can be injected into an application at runtime, thereby avoiding the deployment delays and risks endemic in traditional software engineering practices. Finally, by couching policy expressions in terms of DSLs, I invite a collaboration between programmers (who are familiar with base workflows) and stakeholders (who are familiar with requirements) in quickly and reliably realizing new and changing requirements.

Based on this perspective, the story of policy injection consequently rises to a story of System of Systems composition, where policy injection can be used to compose systems onto both base systems and onto each other. From the viewpoint of any one system, policy injection presents a linkage point between systems, as will be demonstrated by the injection of the policy system itself and other systems in Chapter 5.

In Chapter 5, I present the PALMS-CI case study, which demonstrates these principles with real world implementations that are evaluated in Chapter 6. In Chapter 7, I present comparisons to existing systems that either share goals with PDD, or implement portions of PDD using different approaches. Additionally, I make a case for the new role of Policy Programmer, which combines and extends the roles of programmer and stakeholder described in this section. Finally, Chapter 7 also describes the successes and shortcomings of both PDD foundations and implementations relative to gaps identified in Chapter 2.

## 4.6 Acknowledgments

Chapter 4, in part, is a reprint of material as appeared in 3 papers:

1) A paper currently being prepared for submission for publication of the material. B. Demchak, C. Farcas, E. Farcas, I. Krüger. The dissertation author was a co-investigator and co-author of this material.

2) B. Demchak and I. Krüger, “Policy Driven Development: Flexible Policy Insertion for Large Scale Systems,” in 2012 IEEE International Symposium on

Policies for Distributed Systems and Networks, Chapel Hill. IEEE Computer Society, Jul. 2012, pp. 17-24. The dissertation author was the primary investigator and author of the text used in this chapter.

3) B. Demchak, J. Kerr, F. Raab, K. Patrick, and I. H. Krüger, "PALMS: A Modern Coevolution of Community and Computing Using Policy Driven Development," in 45th Hawaii International Conference on System Sciences (HICSS), Maui, Hawaii. Jan. 2012. The dissertation author was the primary investigator and author of the text used in this chapter.

# CHAPTER 5

# POLICY IN THE PALMS CYBERINFRASTRUCTURE – A CASE STUDY

In Chapter 4, I described the core principles that enable the conditional composition of workflows at runtime via executable policy. In this chapter, I describe how I implemented those principles in the PALMS Cyberinfrastructure (PALMS-CI), a large scale system designed to support exposure biology research while incorporating requirements from multiple independent stakeholder communities.

The PALMS-CI currently serves a growing, worldwide community of researchers, successfully meets a number of important requirements, and is evolving to capture more requirements, based on policies injected onto base workflows. In demonstrating the implementation of PDD, it shows how PDD can be used to improve evolvability along two important dimensions: workflow maintenance costs and timely realization of requirements responsive to new and changed stakeholder requirements.

In this chapter (as shown in Figure 19), I begin by explaining the basic PALMS-CI requirements, both in terms of the stakeholder community it serves and technical design drivers (in Sections 5.1 and 5.2). Next, I outline the PALMS-CI design process and rationale (in Sections 5.3 and 5.4).

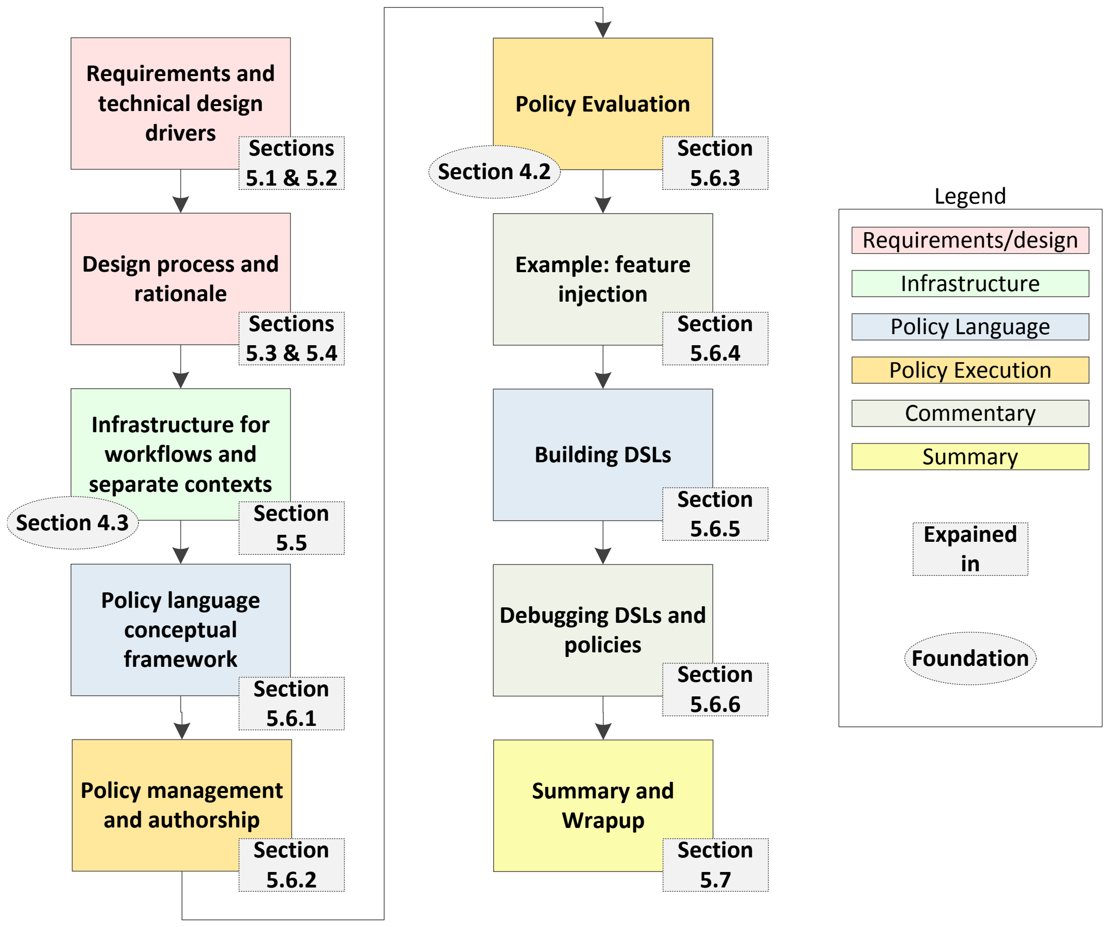


Figure 19. Chapter 5 Flow

Section 5.5 explains the design of the infrastructure that supports workflows and maintains separate contexts for composed workflows, which realizes context foundations laid in Section 4.3. Section 5.6.1 describes the conceptual framework on with PALMS policy languages are built, and Section 5.6.2 discusses how policies are managed and authored. Section 5.6.3 describes the mechanics of policy evaluation, which realize the injection and policy definition foundations laid in Section 4.2.

Section 5.6.4 gives an example of how PALMS policies can be used to inject novel features (as separate and crosscutting concerns) not addressed (or originally conceived of) in PALMS' base workflows.

Section 5.6.5 describes how to build and use DSLs as policy languages within PALMS; Section 5.6.6 addresses developing and debugging policies and DSLs that support them.

Chapter 6 follows up with an evaluation of how well the PALMS-CI's policy system meets these goals. It shows that PALMS' policy implementation meets the goals of PDD by successfully enabling policy articulation and injection leading to the realization of stakeholder requirements at runtime. It demonstrates that this injection occurs at an acceptable cost in many cases. It also identifies costly cases that give insights into future evolution paths.

Chapter 7 compares the PDD approach and its PALMS-CI implementation to existing theoretical and practical systems, and gives additional perspective on the approach and implementation, including how PDD succeeds or falls short in addressing the gaps identified in Chapter 2.

Note that the GetStudyList motivating example presented in Chapter 3 is used throughout this dissertation to demonstrate PDD principles, and is a simplified subset of the PALMS-CI discussed in this chapter. Simplification notwithstanding, GetStudyList is similar to each workflow implemented in the PALMS-CI, and fairly represents other types of workflows and workflow patterns as described in Section 2.5.2.

Note that while this chapter discusses PDD as applied to building and evolving a cyberinfrastructure (defined in Section 1.1), it applies equally well to other classes of large scale systems in the general category of Ultra-Large-Scale Systems (ULS) [163] [164] [165], including grid systems [166], Systems of Systems, and distributed cyber-physical systems [30] (including automotive and aeronautical systems). Such systems have a number of requirements in common, including a need to address emergent requirements from continuously changing stakeholder populations operating under multiple policy domains. Additionally, they require continuous evolution and development while maintaining high availability and quality of service. Each type of system has different development and deployment constraints, and each would benefit critically from the user-controlled evolution and online modification capabilities at the heart of PDD.

## 5.1 The PALMS Project and the PALMS-CI

The PALMS (Physical Activity Location Measurement System) Project [167] was chartered by the National Institutes of Health (NIH) to study exposure biology (EB) questions answerable using geo-tagged data collected from biosensors worn by targeted populations. The PALMS-CI was created at the University of California, San Diego, to serve a then-unformed EB community, while accounting for differences between study data, analysis, and personnel organization, and facilitating NIH, HIPAA, and other policy objectives.

The PALMS-CI realizes these requirements, and currently serves a growing community of over 40 research groups worldwide, manages over 170

EB studies with over 150GB of data, and offers 24/7 availability. It supports 81 top level workflows (each implementing a PALMS API call, and all similar to the GetStudyList example presented in Chapter 3), 440 services, 440 service interaction message definitions (expressed as Java classes), all of which is contained in 2070 Java files. Figure 20 shows the community growth over time, beginning in 2010. Both the number of users and study groups (corresponding to one or more studies managed by the same group of users) has grown over time. It also shows that data under management has grown faster – the plot shows the size of compressed backup data, which grows as the logarithm of the actual data size.

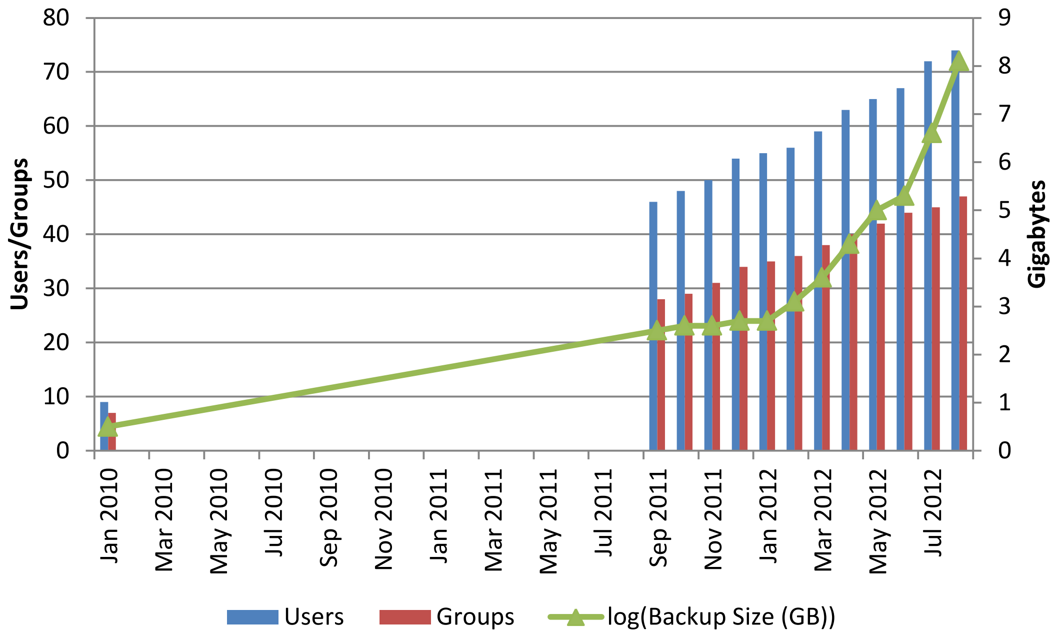


Figure 20. PALMS Community Growth over Time

From the outset, major PALMS-CI challenges included the ability to customize access to data and application features on a per-study basis, and to add new features quickly and reliably. Examples of per-study customizations include access control policies (e.g., allowing research assistants to add study participants or sensor data, but not delete either) and sharing policies (e.g., enabling a guest to view only anonymized sensor data instead of a raw data stream). Examples of new features include auditing and data provenance tracking added to the CI's basic data access workflows.

Given the PALMS project's tight funding and time constraints, I conceived the PALMS-CI as a highly evolvable, but complex system of systems (SoS), where different concerns are modeled and implemented separately, and are composed into a functioning system conditionally and incrementally without imperiling existing functionality. Furthermore, for the PALMS-CI to remain agile and responsive to changing stakeholder requirements, it was essential that decisions regarding which concerns to compose, which other concerns to compose them with, and the conditions and parameters of the composition be made at runtime.

To meet this challenge, I employed the emerging Policy Driven Development technique, which leverages Rich Services (described in Section 5.3) and the Rich Services Development Process (described in Section 5.2) to create a highly responsive service oriented architecture (SOA) and corresponding implementation. I chose a SOA-based approach over existing approaches (e.g., AOSD or pure OOP) because SOA's message-based service interactions enabled interface-based component reuse (via loose coupling and late binding), inherent scalability and distributability (via message routing), and the opportunity to constrain and augment workflows (via message interception), while retaining the freedom to use other approaches to build components themselves.

I developed PDD to leverage SOA's workflow and message interception features to define and inject new and unanticipated concerns without endangering existing CI functionality, while remaining highly responsive to stakeholder requirements and maintaining maximum PALMS-CI availability. (As described in Chapter 2, other existing solutions were inappropriate.) Using the PALMS-CI, I demonstrate how PDD simplifies and improves access control decisions, simplifies policy problems (such as Separation of Duties) relative to other solutions, enables feature composition, and in the process facilitates system evolution, which is the key to simultaneously satisfying the requirements of multiple stakeholder communities.

### 5.1.1 Exposure Biologists – the Core PALMS Community

In 2007, the NIH began funding the Genes, Environment and Health Initiative (GEI), having two main components [168]. The Genetics Program was chartered to analyze "genetic variation in groups of patients with specific illnesses". The Exposure Biology (EB) program is chartered "to produce and validate new methods for monitoring environmental exposures that interact with genetic variation to result in human diseases", and includes the PALMS project [5] at the University of California, San Diego (UCSD). PALMS' mission is to develop an integrated suite of hardware (e.g., sensor devices), software, and database solutions that support real-time capture and subsequent analyses of data on physical activity energy expenditure from a geospatial perspective. PALMS is intended to help answer questions such as the percentage of a person's energy expenditure that occurs at various locations, and while moving between locations.

Historically, the EB community has consisted of a number of independently operating principal investigators (PIs) attempting to answer questions relating to human disease as a function of environmental exposures, diet, physical activity, psychological stress, and addictive substances. Like PALMS investigators, they select (or create) their own sensor hardware, software, and database systems to support the capture and analysis of their own data.

## 5.2 The Basic PALMS Requirements

In 2008, PALMS investigators realized that their own data capture, storage, analysis, and visualization requirements and workflows were conceptually similar to those of many other EB researchers, and that economies of scale weighed in favor of creating a web-based system that could perform such functions for EB projects similar to PALMS, thereby delivering significant technical, economic, and collaborative benefits to the EB community. This was particularly important given the emergence of small person-worn Global Positioning System (GPS) devices and a new field of precise location and activity measurement. Leveraging existing knowledge of GPS processing, the PALMS-CI aimed to provide many benefits, including:

- Elimination of redundant programming, debugging, and maintenance of common data acquisition, analysis, and visualization code
- Centralized, secure, scalable and highly available data management (including backups)
- Computational resources scalable to large datasets and complex analysis
- Sharing of processing protocols, allowing standardized comparison and more rapid scientific advances
- Logging of data manipulations and analyses
- Standardized interfacing to external packages (e.g., Microsoft Excel, Google Earth, and ESRI's ArcGIS modeling tools)
- Streamlined discovery and sharing of observation data and results

PALMS investigators postulated that the organization and workflow of each EB researchers' data collection and processing was substantially similar (as shown in Figure 21, and as validated at a number of international workshops and presentations to EB community members):

- PIs and research teams manage multiple studies
- Studies include collection of time-stamped data from multiple sensors, where the data must be stored securely according to Institutional Review Board (IRB) ethical guidelines
- A study incorporates a collection of subjects, where each subject wears one or more sensor device (e.g., a GPS unit, an accelerometer, a heartbeat monitor), each of which produces a stream of time-tagged observations
- A study also incorporates one or more calculations that filter observation data; correlates observations with time and location; and produces result sets containing inferred trips, bouts, physical activity levels, and other parameters of exposure
- Over the course of a study, subjects may be added, sensor observations may be captured and uploaded, and calculations may be run at any time. The ultimate result of a study may be a calculation

result set or a product that an external package creates from a calculation result set

- Data are aggregated and summarized
- Primary and secondary data analyses are performed by the original study investigators and other investigators now and, ideally, in the future
- Information assurance (IA) [28] is implicit

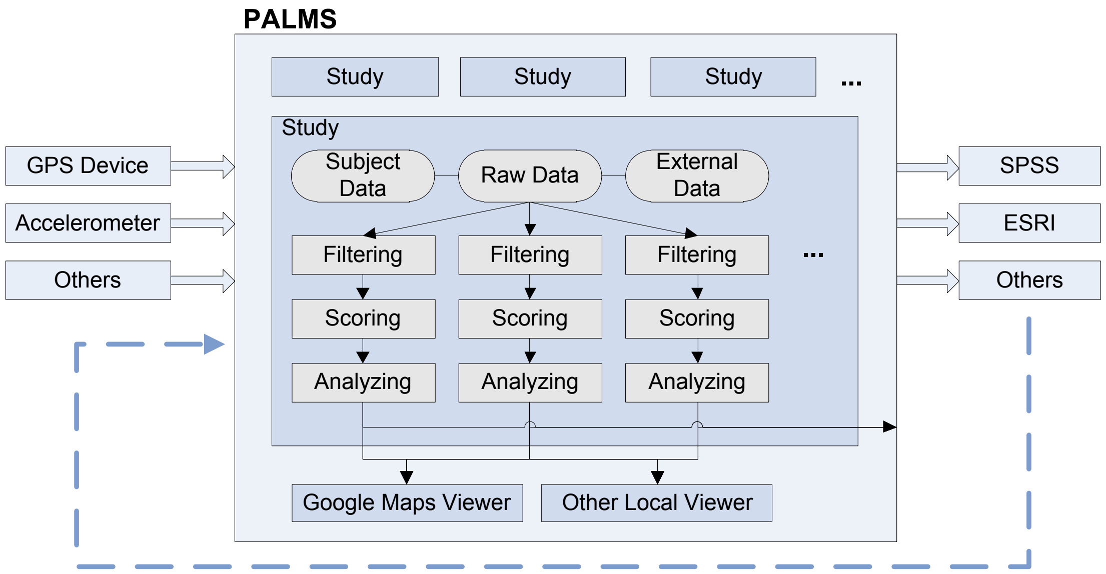


Figure 21. PALMS Studies – Structure and Flow

Additionally, the following key differences between studies were recognized:

- Demographic characteristics and geographic location of study populations
- Study aims and outcomes
- The intensity, frequency, and duration of data collection, with different devices deployed according to feasibility and research question
- Subject information collected (e.g., sex, age, home location, work location, etc.)
- Calculations, parameters, and thresholds used during analysis
- Particular roles, privileges, and responsibilities of research personnel (e.g., PIs, research assistants, guests, etc.)

PALMS investigators defined two challenges in building the PALMS-CI:

- create a technical design that could leverage the common data organization and workflows to deliver the benefits listed above, yet enable study customization that accounted for differences between study data, analysis, and personnel organization
- attract a community of investigators interested in both advancing PALMS-CI capabilities and collaborating to define and leverage data, analyses, and visualizations beneficial to the overall EB mission

A key insight derived from these propositions is that for the PALMS-CI to be viable, the requirements of all stakeholders (including system operators, data producers, and data consumers) must be simultaneously and continuously met, both in the technical and the community governance domains – otherwise, a disenfranchised stakeholder may opt-out, thereby weakening the entire PALMS community. In traditional system development, the lag between the discovery of a new stakeholder requirement and its enactment in a delivered system is often several months, which increases pressure on stakeholders to opt-out. PALMS-CI must quickly respond to requirement changes, and must do so without becoming unstable or compromising usage by other stakeholders.

To meet the challenge of community formation, we gathered an expert advisory board; recruited willing, early adopter-investigators worldwide as initial users; hosted the annual International Users Conferences (two, so far); partnered with key organizations (e.g., GPS Health Research Network [169] and caBIG [170]); and generated instructional collateral useful in creating relationships with prospective collaborators and their respective Institutional Review Boards (IRBs).

## 5.3 The PALMS Logical Rich Service

We designed and currently maintain the PALMS-CI using the Rich Services Development Process (RSDP) [47], which is a multi-stage, end-to-end software engineering process ranging from requirement elicitation to physical network deployment. The RSDP is compatible with agile development methodologies, leverages Model Driven Architecture (MDA) [171] and Model Driven Engineering (MDE) techniques, and is geared specifically toward producing hierarchically decomposable Rich Services. It produces a clean separation between logical and deployment models, where a logical model depicts relationships between logical entities, a deployment model depicts relationships between physical entities, and a deployment mapping can derive a deployment model from a logical model.

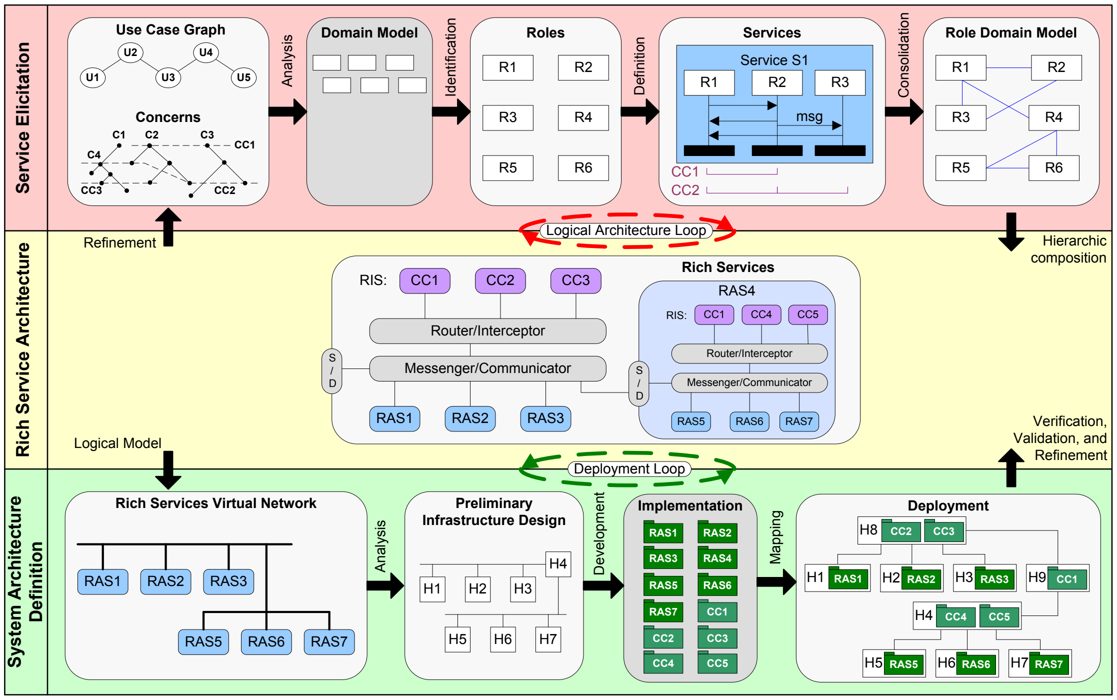


Figure 22. Rich Services Development Process

The RSDP is structured into three phases, each of which is subdivided, as shown in Figure 22. The major phases include:

- **Service Elicitation** gathers use cases, extrapolates crosscutting concerns, constructs a domain model, identifies service roles, and then defines services and workflows
- **Rich Service Architecture** articulates a Rich Service that incorporates those services and workflows
- **System Architecture Definition** creates a service hierarchy, defines a corresponding virtual network, and then maps the virtual network onto a set of physical networks and compute engines

While each phase (and sub-phase) is sequential and depends on a previous phase, RSDP allows and encourages the re-execution of earlier phases (in spiral fashion [46]) as new requirements come to light in later phases.

During the PALMS-CI development, I followed RSDP by first soliciting user stories (as text), refining them into use cases (organized as pages in an Excel spreadsheet with traceability implemented as inter-page links), and eliciting additional requirements by using low resolution user interface mockups.

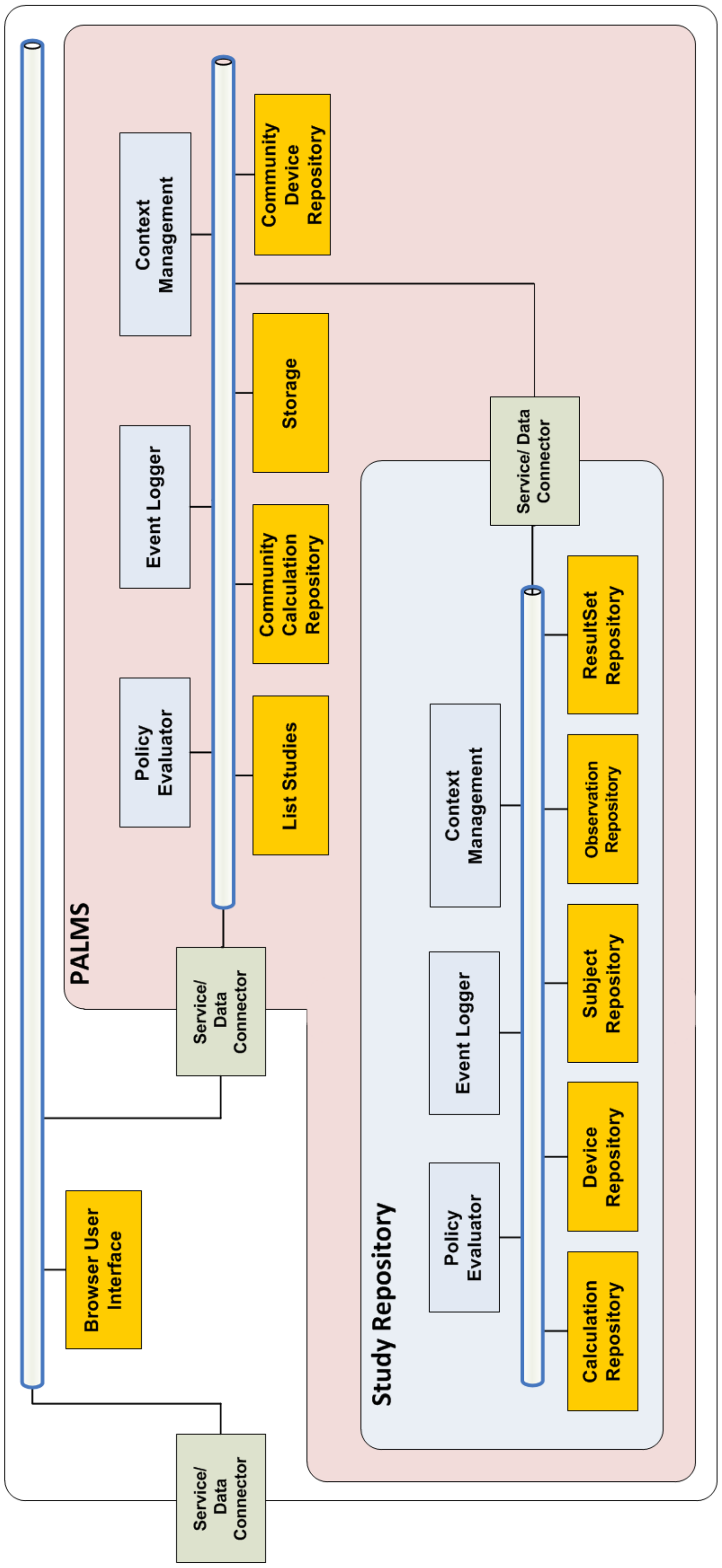
Per RSDP, I segregated requirements into two groups:

- those that described data storage, data transfer, and data analysis (so-called "base workflow" requirements)
- those that bore on decisions and workflow options (crosscutting concerns as so-called "policy" requirements).

Data flow requirements were modeled using standard RSDP – as domain models (expressed as UML [40] class and sequence diagrams using Enterprise Architect [172]), and then as roles and services, which were factored into a candidate Rich Service. Policy requirements were set aside as candidates for future policy definition and execution, as described in Section 5.6. (These substantially involved access control decisions, but also included other crosscutting concerns such as logging, auditing, provenance tracking, and the mapping of calculation execution to available processors.)

A simplified version of the resulting logical Rich Service is presented in Figure 23. It shows the relationship between a Browser User Interface and a PALMS Service, which is decomposed into a PALMS system layer containing services representing a repository of studies and of community-authored calculation and sensor device functions that can be shared amongst studies. The Study Repository service is further decomposed into sub-services representing a number of repositories, including calculations and devices actually used in the study, a study’s participants and observations, and data analysis results.

Figure 23. PALMS-CI Logical Rich Service


PALMS
Service/ Data Connector
Browser User Interface
Service/ Data Connector
Policy Evaluator
Event Logger
Context Management
List Studies
Community Calculation Repository
Storage
Community Device Repository
Study Repository
Service/ Data Connector
Policy Evaluator
Event Logger
Context Management
Calculation Repository
Device Repository
Subject Repository
Observation Repository
ResultSet Repository

Critically, the policy execution function is represented as a RIS capable of intercepting interactions between any and all services. It is discussed in Section 5.6.

Finally, I used the logical Rich Service to generate a deployment Rich Service by assigning the User Interface to a PC, the PALMS Service to a server, and defining the message bus between them as the Internet as described in Section 5.4

In the case of the PALMS-CI, the RSDP approach resulted in a clear definition of services and basic workflows, separate from a pool of crosscutting requirements that can be composed onto the workflows by using policies.

As a result of basing the PALMS-CI on Rich Services, it was highly evolvable from the outset. Separate concerns were implemented as peers in a distributed System of Systems (SoS), as described in Sections 5.6 and 5.6.4. They could be conditionally composed into other workflows, and were primed to accommodate the injection of unanticipated foreign workflows. Additionally, the PALMS-CI included a set of seed policies that inject known crosscutting concerns, such as access control decisions.

Note that while the PALMS-CI supports the PALMS project, and the PALMS Service is a component of the PALMS-CI, going forward, I use "PALMS" as shorthand and rely on the reader to infer either "PALMS project", "PALMS-CI", or the "PALMS service" from the context. The majority of PALMS references are to the PALMS service. When an inference would be ambiguous, I use the longhand form.

Relative to the PALMS service, the Browser User Interface functions as a Client, and is referred to as such in workflow depictions such as Figure 6 and Figure 29. Similarly, the PALMS service is referred to as PALMS.

## 5.4 The PALMS Deployment Rich Service

While the logical Rich Service described in Section 5.3 describes service definitions, it does not specify the technologies used to implement or link the services, nor how the services are deployed. Under RSDP, applying a deployment mapping function to the logical RS yields a deployment model in which links and service deployments are specified, as shown in Figure 24.

Consistent with the logical architecture (Figure 23), the PALMS-CI deployment is implemented in two major parts: the Browser User Interface (UI) service and the PALMS service.

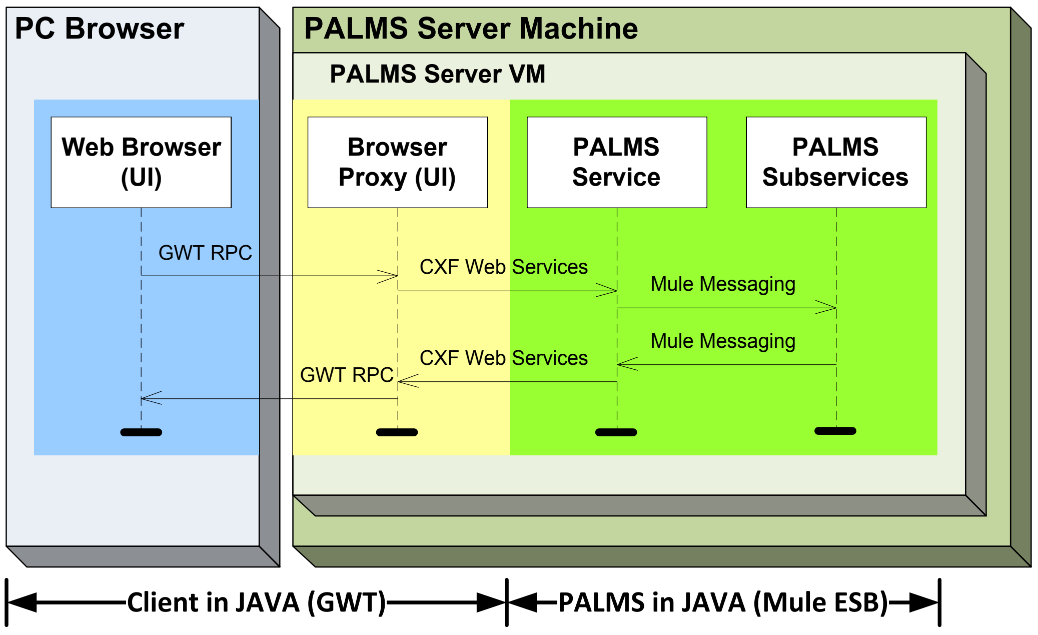


Figure 24. PALMS Deployment

The Browser User Interface service maps to a largely browser-based application that serves as the PALMS Community's main interface to PALMS. In Figure 24, it is represented by the "Web Browser (UI)", which displays and manages screen content, and "Browser Proxy (UI)", which functions as the Browser User Interface's Service Data Connector (SDC) by proxying the PALMS service. It is implemented using the Google Web Toolkit [173], a toolkit specially built and optimized for building complex, performant browser-based applications, and which leverages a large collection of pre-existing user interface widgets. It was chosen to provide a rich interface for community members, while enabling PALMS developers to efficiently add new features. While the code for both types of Browser User Interface services resides on the PALMS Server Machine, the screen management services execute on a PC browser, and the proxy services execute on the PALMS Server Machine.

Communications between these services uses GWT's proprietary remote procedure call protocols across the Internet.

An example of a Browser User Interface screen is shown in Figure 25. It shows a graphical representation of activity and location data collected for a study.

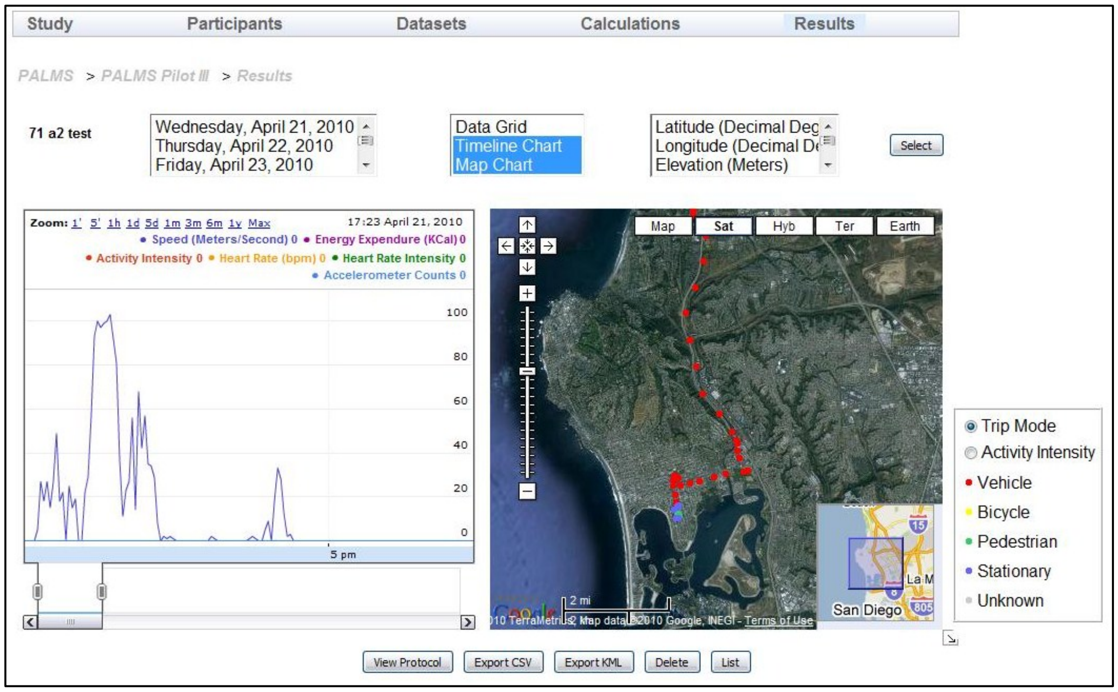


Figure 25. PALMS User Interface

PALMS (represented in Figure 24 as "PALMS Service" and "PALMS Subservices") is written in Java and leverages the Mule Enterprise Service Bus (ESB) [174], which was chosen because it provides message transmission, message routing and interception, and service execution features that closely match the relationships modeled in a logical Rich Service framework. Therefore, the deployment mapping function for the PALMS service is one-to-one, with each PALMS logical service being implemented by an actual Java-coded Mule-hosted service. PALMS executes on the PALMS Server machine on top of an Apache Tomcat server v6.0.20, and stores its data using a local MySQL v5.0.77-4 DBMS. The Browse User Interface proxy services communicate with PALMS using Apache CXF Web Services protocol [175].

The PALMS Server runs Red Hat Linux, and is hosted as a VMware virtual machine (VM) at UCSD under a high availability, secure infrastructure that includes an intrusion detection system, automatic multilevel backup, and automatic live migration. The VM uses 8GB RAM and one 2.0GHz Xeon processor. User ID authentication services are provided by caBIG's Dorian ID Provider [176], which verifies PALMS users' credentials, and returns an X.509 certificate [177], which in turn, PALMS uses as proof of user identity. caBIG's Grouper [178] implementation is used to securely store and manage role, user group, permission, and access control list (ACL) information.

## 5.5 The PALMS-CI Design

As described in Sections 5.3 and 5.4, the PALMS-CI consists of a user interface component and a PALMS-CI service component. While PDD can be used in the design and realization of both components, it was used primarily in creating the PALMS service, which implements basic workflows leveraged by the user interface. As such, the PALMS service presents a service interface (defined by its Service/Data Connector) that is agnostic as to caller, and can be reused to provide PALMS functionality in other contexts.

While PALMS embodies many design decisions that attend to the diverse issues of production worthy services, I describe PALMS in terms of service interface and workflow considerations that influence or leverage PDD. A complete description is beyond the scope of this dissertation.

Most importantly, in Chapter 6, I describe PALMS from the perspective of workflow maintenance costs and timely delivery responsive to stakeholder requirements, where stakeholders represent multiple, independent domains. In this section, I lay a foundation for describing PALMS' PDD implementation by describing key PALMS-CI workflows, including key workflow activities, and the information exchanged between workflow activities. Section 5.5.1 describes repositories, which are key PALMS workflow activities, and themselves demonstrate opportunities to add value using policy injection. Section 5.5.2 describes the messages exchanged between the Browser UI and PALMS, whereas Section 5.5.3 describes message exchanged within PALMS. Finally, Section 5.5.4 describes high level details of an implementation that enables and supports policy injection. The policy evaluation system itself is described in Section 5.6.

### 5.5.1 Repositories

All PALMS functions are expressed as workflows that either manipulate or rely on state information either maintained by PALMS or available to it. When data is intended to persist across workflows, one or more workflow activities perform operations that maintain the persistence abstraction. They have wide latitude in implementing the abstraction, including calling on encapsulated workflows according to a Mediator pattern [33].

Each such workflow activity represents a transformation of input data (e.g., a query) to output data (e.g., a row set), and can maintain the persistence abstraction by:

- fetching from persistent (database) store
- direct calculation
- returning pre-initialized data
- redirecting the operation to another workflow activity
- caching and memoization
- other mechanisms
- any combination of the above

PALMS implements the persistence abstraction by using a repository model, which is described in [103] and [134] (as Data Mapper). According to [103], the advantages of repositories include:

- presenting a simple model for obtaining persistent objects and managing their lifecycle
- decoupling application and domain design from persistence technology or multiple data sources

Particularly, because a repository combines a computational and storage model, it allows flexibility in maintaining a persistent data abstraction, even as underlying implementation requirements (e.g., speed and physical location) and dependency assumptions (e.g., underlying database table definitions) evolve. Though it's possible to achieve this flexibility at the database level through the use of stored procedures, lifting the persistence abstraction to the workflow activity level enables access to calculations, data sources, and workflow-based state generally unavailable at the database layer. Furthermore, traditional structuring of stored procedures (as either monoliths or hierarchies) presents little opportunity for runtime evolution via mechanisms such as policy injection because of their tight coupling to the database engine and other stored procedures. Repositories implemented as workflow activities loosely coupled with other workflow activities encourage policy injection. Finally, while databases generally present data in terms of low level types and inter-table links, repositories are unconstrained in this way, and can define and manipulate data along other dimensions, including standard or custom ontologies and logic systems.

On the other hand, in a repository model, the data and its semantics are not invested completely (if at all) in the storage system. Because a repository encapsulates its calculations and storage mechanisms, external assumptions about underlying storage are often invalid or can easily become invalid. Consequently, repository data can be accessed only through a repository interface, and combining data models from multiple repositories must be done explicitly by repository clients (at a cost of development time, execution time, and memory). By contrast, a database-centric model presents a data abstraction that encompasses multiple data models (as groups of related tables), and combining models is simple and efficient using facilities (e.g., join) of the database management system.

I chose the repository abstraction (over a database-centric model) for its flexibility in implementing data models isolated from other data models, thereby reducing possible hidden dependencies between data models and leveraging loose coupling to improve PALMS' responsiveness to evolving requirements affecting base workflows. Additionally, for repositories realized by encapsulated workflows (e.g., via access to external storage systems or other repositories), the service interactions implementing those workflows present additional opportunities for policy injection, thereby improving prospects for quickly responding to evolving stakeholder requirements as they arise.

PALMS maintains repositories that track system-wide resources such as calculations (PALMSCalculation, for data analysis), device profiles (PALMSDevice, which define a data acquisition device), and schema (PALMSSchema, which defines data ontologies). It also maintains a repository containing study metadata (PALMSStudy), and a hierarchy of repositories that tracks various study-specific attributes and data as shown in Figure 26. They include:

- **Calculation** – collection of PALMSCalculations available to analyze study data
- **Protocol** – collection of parameter sets associated with execution of calculations
- **Device** – collection of PALMSDevices that can contribute study observation data
- **Dispatch** – collection of execute-ready calculations and their parameters
- **Forms** – collection of UI screen layouts for displaying and entering repository contents
- **Schema** – collection of metadata definitions, including those for subject data
- **ResultSet** – collection of metadata describing result data generated by calculations
- **Result** – collection of result data generated by calculations
- **ObservationSet** – collection of metadata describing observation data contributed by devices
- **Observation** – collection of observation data contributed by devices
- **Subject** – collection of data for each study subject

(Note that the repositories shown in Figure 23 model the PALMS-CI, the model is simplified and does not show all repositories. The list in this section is complete.)

While each repository supports the abstraction of storing, fetching, and managing a collection of objects, its strategy for implementing the persistence abstraction varies according to the relationship of its data to other PALMS data and the performance requirements and logistics of accessing it.

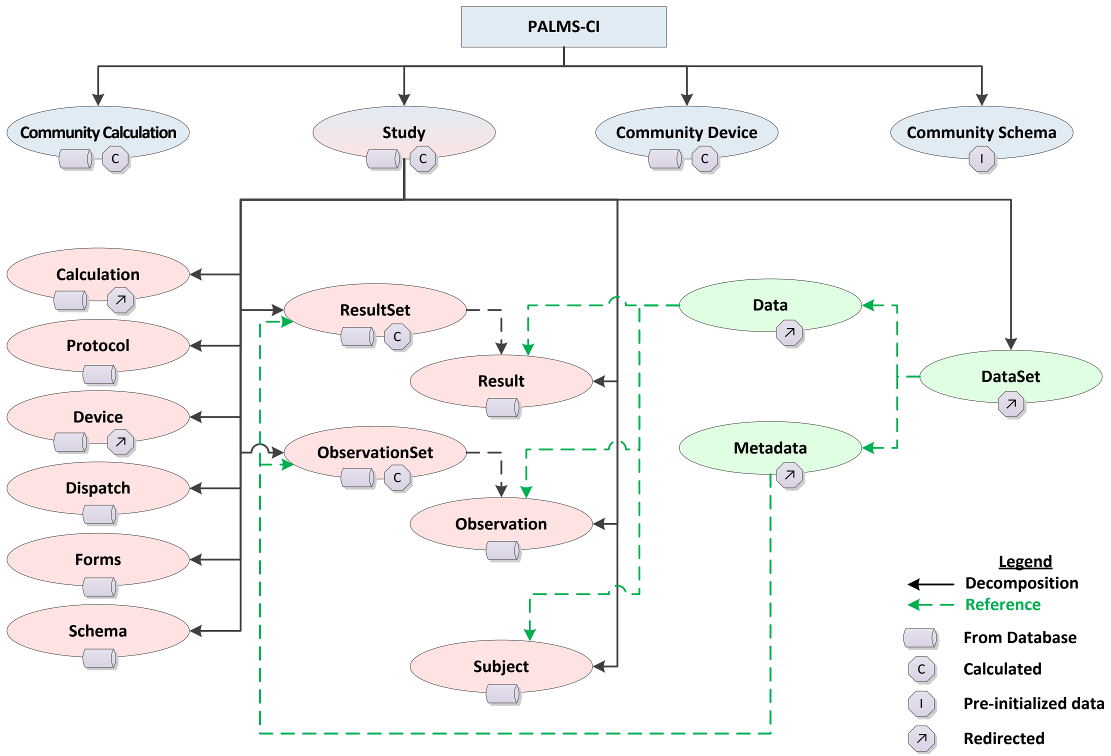


Figure 26. PALMS-CI Repositories

Many study repositories represent tabular data stored as rows in a database, and data persistence is realized via interactions with a storage service that implements a tabular data abstraction by using a DBMS, as shown in Figure 6 and Figure 29. In Figure 26, these are marked by ▭ and include the subject, observation, result, schema, forms, dispatch, and protocol repositories.

Repositories marked by ▭ C realize data persistence both by interacting with a storage service and by inline transformation and synthesis. They include the system level calculation, study, and device repositories, and the study level result set and observation set repositories.

Repositories marked by ▭ ↗ realize data persistence both by interacting with both a storage service and other repositories. For example, the study calculation repository data model combines data in its own study-based calculation table with data maintained in the PALMSCalculation repository.

Repositories marked by I realize data persistence by accessing pre-defined and compiled-in data.

The DataSet repository (marked by ↗) virtualizes data repositories that can act as inputs or outputs for analysis calculations. It defines a name space that allows calculations to access to the observation, result, and subject data repositories without differentiating between them. It implements this abstraction by delegating each repository function to an equivalent function in the workflow activity that implements the repository.

The Repository pattern creates a clean separation between a data abstraction it represents and the implementation of the abstraction. Interaction between a repository workflow activity and activities that implement it are fertile ground for policy-based workflow injection (including access control, data transformation, and new features) responsive to emerging stakeholder requirements. For example, a query message (such as defined in Section 5.5.3.1) can be edited to constrain, filter, or augment repository data by a policy injected between the repository and its storage activity as shown in Figure 27.

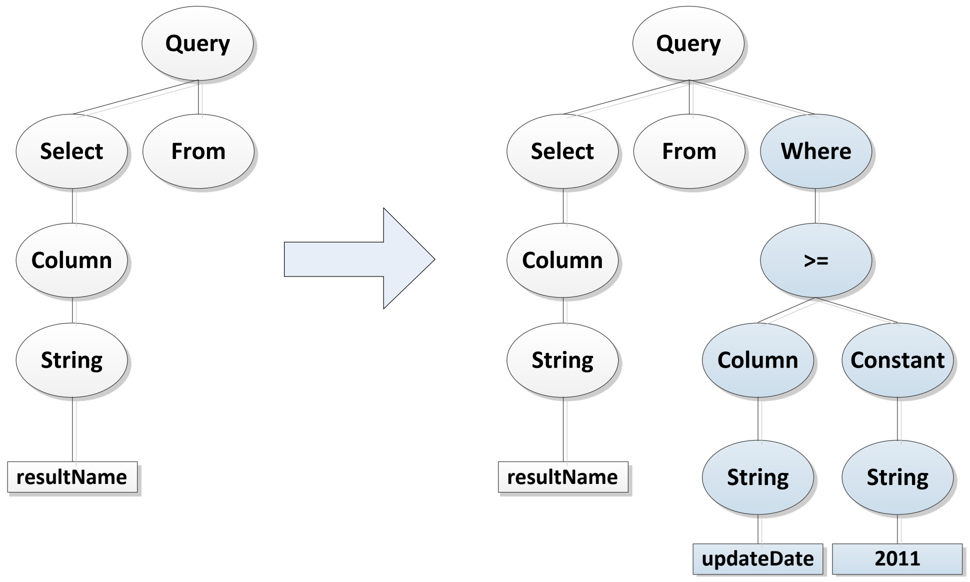


Figure 27. Query Edit Constrains Original Query to Post-2010 Results

## 5.5.2 Interface between Browser UI and PALMS-CI

Logically, the PALMS SDC exposes 72 functions, each of which decomposes to a service orchestration. Each function consumes an input channel (containing function parameters), and fills an output channel (representing function results), thereby implementing a request/reply pattern. The Mule ESB offers a number of protocols for realizing the input and output channels, including simple TCP/IP, simple HTTP, and Web Services. The PALMS SDC is implemented as a Web Service (using CXF [175]), which allows Java-style typing of input and output channels, SOAP-based marshaling and de-marshaling of channel values, and automatic dispatch of function requests to appropriate service orchestrations.

In order to preserve the option of using other protocols, PALMS' channel messages are defined on simple data types, including scalars (e.g., Booleans, integers, and floating point), byte arrays, Unicode strings, and untyped XML documents (as strings). Defining channel messages as a mixture of typed and untyped data leverages the benefits of both. A typed definition sets the structure of a message sufficiently to detect a gross mismatch in the service contracts fulfilled by interacting services. Untyped XML self-defines its structure, which can vary from interaction to interaction. This enables a dynamic definition within the context of the overall typed definition, but requires that interacting services verify untyped XML contents at runtime.

On input channels, a PALMS message includes untyped XML parameters to represent context-sensitive processing options and complex data whose structure and semantics are agreed upon by interacting services *a priori*, with missing values being assigned pre-defined defaults. For example, a data retrieval service may accept an untyped XML parameter that specifies an output format, where including the element `<format>csv</format>` results in comma-separated values, and including no `<format>` element defaults to tab-separated values. In this way, services are free to add functionality (to support different service interactions) without invalidating existing service interactions.

```
<document-root>
   <schema>
      <lat>
         <class>PALMS.Latitude</class>  <type>xsd:double</type>
      </lat>
      <lon>
         <class>PALMS.Longitude</class>  <type>xsd:double</type>
      </lon>
      <heartrate>
         <class>PALMS.HeartRate</class>  <type>xsd:int</type>
      </heartrate>
   </schema>
   <values>
      <row index='0'>
         <lat>32.868345</lat>  <lon>-117.235204</lon>
         <heartrate>72</heartrate>
      </row>
      <row index='1'>
         <lat>32.871255</lat>  <lon>-117.216044</lon>
         <heartrate>84</heartrate>
      </row>
   </values>
   <forms>
   </forms>
</document-root>
```

Figure 28. Sample Dataset

On output channels, a PALMS message includes untyped XML to convey datasets whose structure can change over time, including datasets defined uniquely for each PALMS study, and which are subject to revision by a study's PI over the lifetime of the study. Consequently, the XML document pairs a *data* sub-document with a *schema* definition sub-document that describes the data. (It also contains a *forms* sub-document, which contains user interface screen layout information.) For example, a simplified version of a dataset containing a GPS location and heart rate value might appear as in Figure 28. For purposes of illustration, each schema entry contains a data value name, an ontology identifier (`<class>`), and a data type (`<type>`) – an actual schema entry contains more information.

By convention, all input channel messages contain an X.509 certificate that securely provides the identity on behalf the calling service operates. All output channel messages contain a result structure that, if non-empty, gives the reason for the failure of a service function.

As a result of using Web Services protocols, input channel messages are automatically tagged with a PALMS function identifier, and output channel messages are automatically tagged with a document type unique to the responding PALMS function. Additionally, other message protocols (e.g., TCP/IP or HTTP) can easily include this information so as to achieve the same effect.

Consequently, interactions between the Browser UI and PALMS services bear sufficient information to qualify as good candidates for policy-based workflow injection responsive to emerging stakeholder requirements. (In fact, Section 5.5.4 describes how the X.509 certificate is captured from an inbound message and is exposed to downstream policies as a workflow variable.)

Note that the message bus that carries interactions between the Browser UI and PALMS is the Internet itself, which does not offer message interception capabilities directly. However, interception can be effected by proxies (e.g., a Web Service or proxy server) as intermediaries between the Browser UI and PALMS. Such proxies can execute policies consistent with those described in this dissertation, but are not implemented in the PALMS case study, and so are not discussed here.

### 5.5.3 Interface between PALMS-CI Internal Services

Each function exposed by the PALMS SDC is executed by an orchestration of services internal to PALMS. Each orchestration can involve interacting services, decomposed services, or both. Chapter 3 presents a logical workflow that represents a common PALMS function (GetStudyList), and a corresponding Rich Service is presented in Figure 10. A more robust version of the Rich Service is presented in Figure 23, which serves as a model for the actual PALMS-CI and grounds the discussion of the implementation of the GetStudyList workflow.

The $< Client, PALMS >$ interaction ❶/❽ is described in Section 5.5.1 (as the interaction between the Browser User Interface and the PALMS server). The $< PALMS, ListStudies >$ interaction ❷/❼ is implemented by the PALMS Service/Data Connector (SDC) operating as a dispatcher to the encapsulated List Studies service – the SDC serves as a dispatcher for all PALMS function requests, and mediates all function replies. The $< ListStudies, StudyRepository >$ interaction ❸/❻ is implemented as a peer-level request/reply interaction between the List Studies and Study Repository services. Finally, the $< StudyRepository, Storage >$ interaction ❹/❺ is implemented by a peer-level request/reply interaction between the Study Repository and Storage services.

Within PALMS, messages are passed to peer-level services or to encapsulated service orchestrations (through an SDC) via the Mule message bus and are subject to interception via Mule's router provisioned with interceptor functions. Messages are implemented as Java classes, with a different class for each PALMS internal service interaction. Services are implemented as POJOs (i.e., Plain Old Java Objects), though a POJO can house several services, each of which accepts a message of a different Java class. Mule directs a message to a POJO, and then automatically pairs the message with a service that accepts it.

Note that while an SDC presents a service interface to peer services, it also mediates interactions between an encapsulated service and the SDC's peer services – this essentially defines and mediates the external services accessible to encapsulated services. An example of this is the execution of PALMS' ListResults function, which ultimately accesses the ResultSet Repository (as a sub-service of the Study Repository), as shown in Figure 29.

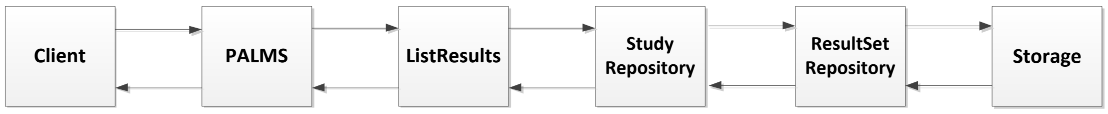


Figure 29. The ListResults Workflow

The ListResults workflow involves activities similar to the GetStudyList workflow, except that it contains an interaction between the StudyRepository and ResultRepository activities. As shown in Figure 23, the ResultRepository service is encapsulated in the StudyRepository service and is therefore accessible only through the StudyRepository SDC. As a subservice of the

StudyRepository service, it has no direct access to the Storage service. Instead, Storage access is mediated by the StudyRepository SDC, which exposes a Storage service to the ResultRepository service, and interacts with the actual Storage service on behalf of the ResultRepository.

The Mule messaging system allows the interception of all messages, starting with messages inbound to the PALMS service, and including interactions between peer services, and between SDCs and sub-services. Consequently, policy injected between peer services can implement an abstraction (e.g., failure management) on a service orchestration distinct from an abstraction on an encapsulated orchestration. When such policies coordinate (as described in Section 7.4.1), the composition exemplifies a System of Systems.

#### *5.5.3.1 PALMS Internal Message Contents*

A key criterion on which policy can be decided or enacted is the message $m$ exchanged between interacting services. In this section, I present a general description of messages exchanged between services internal to PALMS and discuss how these messages contain data conveniently accessible to policies.

Internal services exchange messages as instances of Java classes via Mule's VM protocol, where each interacting service pair exchanges a message of a class unique to the pair. This uniqueness is not a requirement of the message exchange, but serves the administrative purpose of verifying that service pairs exchange messages consistent with their channel definitions. To achieve this, the PALMS relies on the Mule behavior of passing a message only to a service whose Java function signature matches the message class.

Additionally, to preserve the option of scalability in a distributed environment via a marshaled protocol (e.g., JMS [179]), I constrained message classes to contain only base types (and compositions on base types), similar to Section 5.5.1[2]. Particularly, references (e.g., pointers) to service-local data not contained in the message (and therefore not resolvable by a policy) are avoided.

By convention, PALMS services interact in a request/reply pattern, where each request generates a single reply. A reply message contains data responsive to the request, and a `PALMSResult` structure indicating the reason for the service failure, if the request failed. Figure 30 shows a simplified version of the request and reply messages involved in the $< ListStudies, StudyRepository >$ interactions ❸/❻ in Figure 6.

[2] An experimental version of the PALMS-CI leverages this to distribute PALMS' data analysis calculations to virtual machines in Amazon's Elastic Cloud service.

```
public class GetStudyListMessage {
  private String studyID;
  public GetStudyListMessage(String studyID)
    { this.studyID = studyID; }
  public String getStudyID() { return studyID; }
}

public class PALMSResult {
  private String error;
  public PALMSResult(String error) { this.error = error; }
  public String getError() { return error; }
}

public class GetStudyListResult extends PALMSResult {
  private AttributeCollection studies;
  public GetStudyListResult(AttributeCollection studies)
    { this.studies = studies; }
  public AttributeCollection getStudies() { return studies; }
}
```

Figure 30. Sample Dataset

##### *5.5.3.1.1 AttributeCollections*

Note that the `GetStudyListResult` message contains an `AttributeCollection`, which is the structure used by PALMS to represent untyped, hierarchically composed data. An `AttributeCollection` maps attribute names to values, where the values can be scalars or `AttributeCollections` (thereby implementing a Composite pattern). `AttributeCollections` encode mandatory values, optional values, and structures such as row sets and data set queries as shown in Figure 31. An `AttributeCollection` maps conveniently to a number of forms, including an XML document, which can be easily accessed, parsed, and returned by policies.

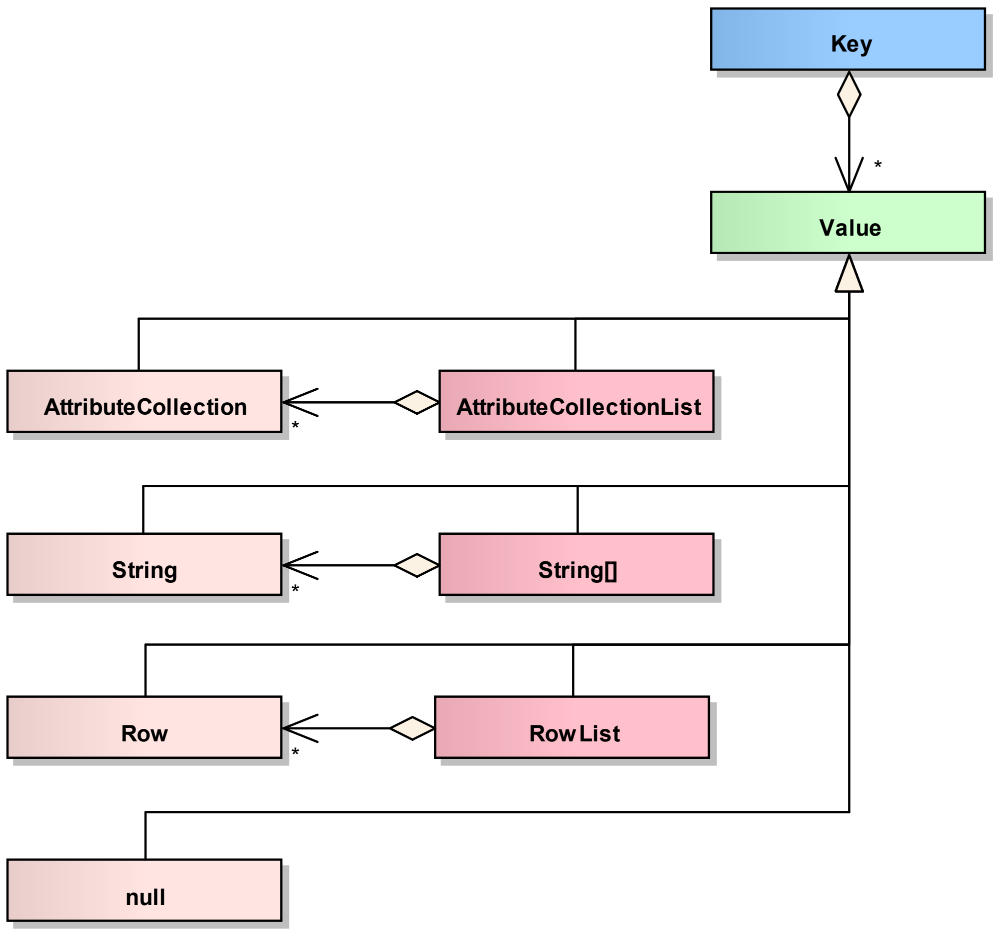


Figure 31. AttributeCollection Model

#### *5.5.3.1.2 SQLTupleTrees*

PALMS models each repository as a list of independent rows conforming to a repository-specific data schema, as described in Section 5.5.1. Optionally, a repository service can process queries expressed in an SQL-like language that can be encoded equivalently either as XML or as a PALMS-CI `SQLTupleTree` structure, and is therefore amenable to examination and manipulation by an injectable policy. The XML form is passed to PALMS (by a PALMS Client) and is converted to the `SQLTupleTree` form before being passed to the target repository. Commonly, a query can be built using a Hibernate-like Criteria API [180] [181] adapted to meet PALMS repository constraints [182].

An example is a query on the ResultSet repository (referenced in Figure 29), which contains metadata describing the results of an analysis of study data, and is shown in Figure 32. The example returns the result name and update time of results whose name begins with "Test".

**SQL**

```
select resultName,
     (date_format(updateDate, '%Y-%m-%d% %h:%i:%s' ) ) as updated
   from
   where resultName like "Test%"
```

**XML**

```
<QUERY>
  <SELECT>
    <COLUMN><STRING>resultName</STRING></COLUMN>
    <AS>
      <FUNCTION>
        <STRING>date_format</STRING>
        <COLUMN><STRING>updateDate</STRING></COLUMN>
        <CONSTANT><STRING>%Y-%m-%d% %h:%i:%s</STRING></CONSTANT>
      </FUNCTION>
      <COLUMN><STRING>updated</STRING></COLUMN>
    </AS>
  </SELECT>
  <FROM></FROM>
  <WHERE>
    <LIKE>
      <COLUMN><STRING>resultName</STRING></COLUMN>
      <CONSTANT><STRING>Test%</STRING></CONSTANT>
    </LIKE>
  </WHERE>
</QUERY>
```

**SQLTupleTree**

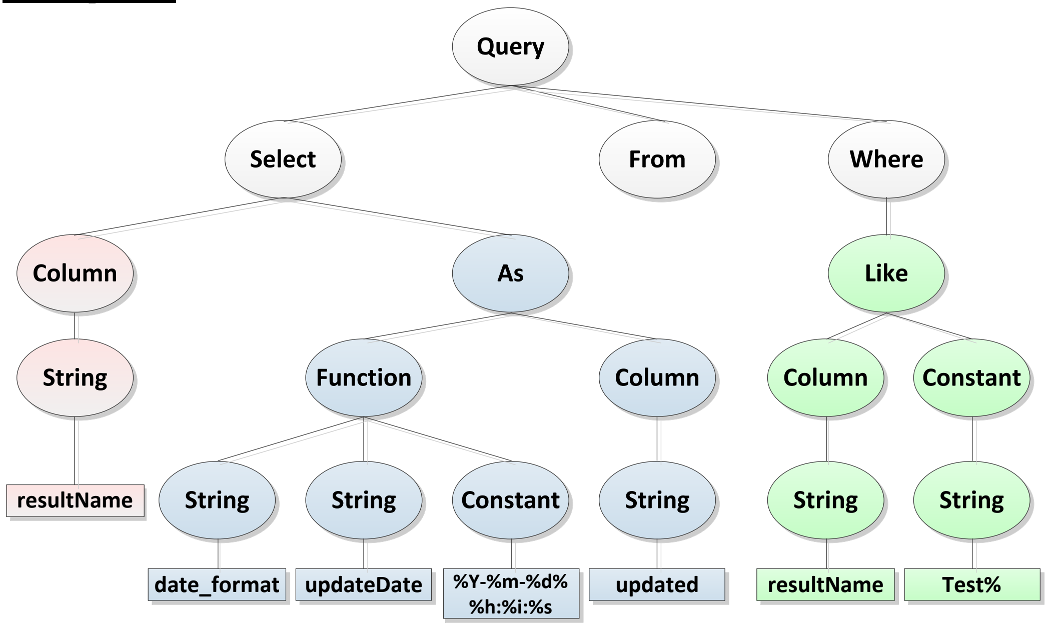


Figure 32. Sample Query

Note that because the query is directed to a particular repository (which represents an abstraction of a single table), the `From` clause is omitted or ignored in all forms.

The `SQLTupleTree` is represented by a package of Java classes that instantiate tree nodes and can return an equivalent SQL and XML expression. Both the `SQLTupleTree` and XML forms enable queries to be evaluated and manipulated by a policy without incurring the time and space penalties of parsing the SQL form. Furthermore, by representing a query in this form, PALMS focuses on defining the data set to be returned, independent of a particular SQL implementation. As such, it can exclude unsupported (and unsupportable) SQL language features available in robust SQL implementations.

### 5.5.4 Workflow Implementation

Under Rich Services, a workflow is implemented as an orchestration of service interactions, where a service may, itself, decompose into an orchestration of services. As described in Sections 5.3 and 5.4, the Mule ESB facilitates the execution and interaction of PALMS services via messages, and enables the interception of those messages by a policy evaluator service. As described in Section 4.2.1, policy evaluation requires that the identities of an interaction's source and target services (called *Service Tracking*) be known so that appropriate composition, control, and filter policies can be selected for evaluation. However, Mule does not provide this basic workflow support, nor does it implement the abstraction of an *interservice message* $IM$, as described in Section 4.3, both of which are necessary to implement workflow composition.

In this section, I describe the mechanics of PALMS service interactions beyond the message passing and routing that Mule provides. First, I describe the design of $IM$s and service tracking under PALMS, including how PALMS leverages Mule features to implement them. Next, I describe higher level workflow support facilities layered onto them, including support for data flows for composed policies. (Note that the examples in this section are actual PALMS interactions, but simplified or renamed to improve ready understanding.)

The key challenges are in exposing the service interaction endpoints to the policy evaluator, and enabling composed policy services to store state correlated with a workflow instance (via $IM$s).

### *5.5.4.1 Basic Interservice Messages (IM) and Service Tracking*

PALMS incorporates basic workflow support services as crosscutting concerns composed upon all service interactions (and which ultimately mediate the higher level crosscutting concerns represented by injected policy). From a Rich Service perspective, these relationships (shown conceptually in Figure 18 and described in Section 4.3) are modeled and implemented as RISs, which provide the $IM$ and service tracking abstractions, where service tracking is provided as a SIV in an $IM$:

- **SIV Pack/Unpack RIS** implements the $IM$ abstraction. It intercepts all messages incoming-to and outgoing-from a service and consists of two sub-services. **SIV Pack** intercepts an outgoing interaction message $m$, and then creates an $IM$ out of $m$ and all other available SIVs. **SIV Unpack** intercepts an incoming message $IM$, extracts its $m$ and other SIVs, and makes them available to other RISs and the target service itself.
- **Service Tracking RIS** implements the service tracking abstraction. It intercepts a message $m$ incoming-to a service and records its source and target service in the Service Tracking SIV.
- **Context RIS** implements the SIV abstraction (as Context Services, per Section 4.3). It is called by the SIV Pack/Unpack and Service Tracking RISs to manage SIVs associated with a service invocation. When a SIV containing a reference is accessed by a RIS or RAS, the Context RIS resolves the reference via interaction with the standalone Context Infrastructure Service (CIS). (Note that for performance reasons, the Service Tracking SIV contains a tuple $\{source, target\}$ instead of a tuple reference – no interaction with CIS is needed to resolve a Tracking Service SIV.)

Figure 33 shows a simplified schematic of the sequencing of RIS interceptor services relative to interactions between the Client, PALMS, and ListStudies activities of the ListStudies workflow.

As shown on the left margin, messages travelling between the Client and Mule (on the Internet) are SOAP-encoded. Messages travelling to and from internal PALMS-CI services (i.e., $< PALMS, ListStudies >$) are encoded as interservice message $IM$s, and a message processed by interceptors associated with a PALMS-CI service are encoded as a service message $m$. Using $IM$ encoding between services enables all information pertinent to an interaction (i.e., SIVs, including $m$) to be carried between a source and target service regardless of where in a distributed or multi-process system each service executes. Fundamentally, the purpose of an $IM$ is to associate SIVs (including $m$) across process boundaries.

Mule guarantees that the execution of a service and its associated interceptors occurs within the same process. The Context RIS (not shown) leverages this to store SIVs (retrieved by SIV Unpack) as thread-local variables CX for the duration of service execution, thereby enabling ready access to SIVs by the target service and composed concerns (including the policy evaluator). When SIV Pack creates an $IM$, it calls the Context RIS to render current SIV copies (from CX) into the $IM$.

Focusing on the ListStudies Service, a message from the PALMS Service arrives as an $IM$, which is unpacked by the SIV Unpack RIS – the $IM$'s SIVs are stored in CX, and $m$ is forwarded to the Service Tracking RIS. While Mule reveals a message's target service (as the target queue name), it does not reveal the source service. Consequently, the Service Tracking RIS assumes that the previous target service is the next source service – it updates the Service

Tracking SIV in CX and forwards an unaltered message $m$ to the Policy Evaluator interceptor. Finally, policy is evaluated (as described in Section 5.6). Assuming the policy allows the ListStudies Service to be executed, it returns a reply $m$, which is processed by each RIS in reverse of the inbound order – this nesting of RISs corresponds to the nesting of abstractions represented by the RISs (i.e., Policy Evaluation builds on Service Tracking, which builds on $IM$s). Ultimately, the SIV Pack RIS returns a new $IM$ to the PALMS Service – it synthesizes the $IM$ from the response $m$ and the SIVs in CX thread-local variables.

A similar sequence is used for the execution of the PALMS Service. Note that the original source of a message bound for the PALMS Service is the Client Service, which exchanges its message $m$ in SOAP format, not as $IM$. To facilitate this, the SIV Pack/Unpack RIS includes additional subservices, Pre-SIV Pack and Pre-SIV Unpack, that perform services analogous to SIV Pack and SIV Unpack. Pre-SIV Pack creates an $IM$ from the SOAP-encoded $m$ and initializes each other SIV. Pre SIV Unpack encodes $m$ as SOAP and deallocates all other SIVs.

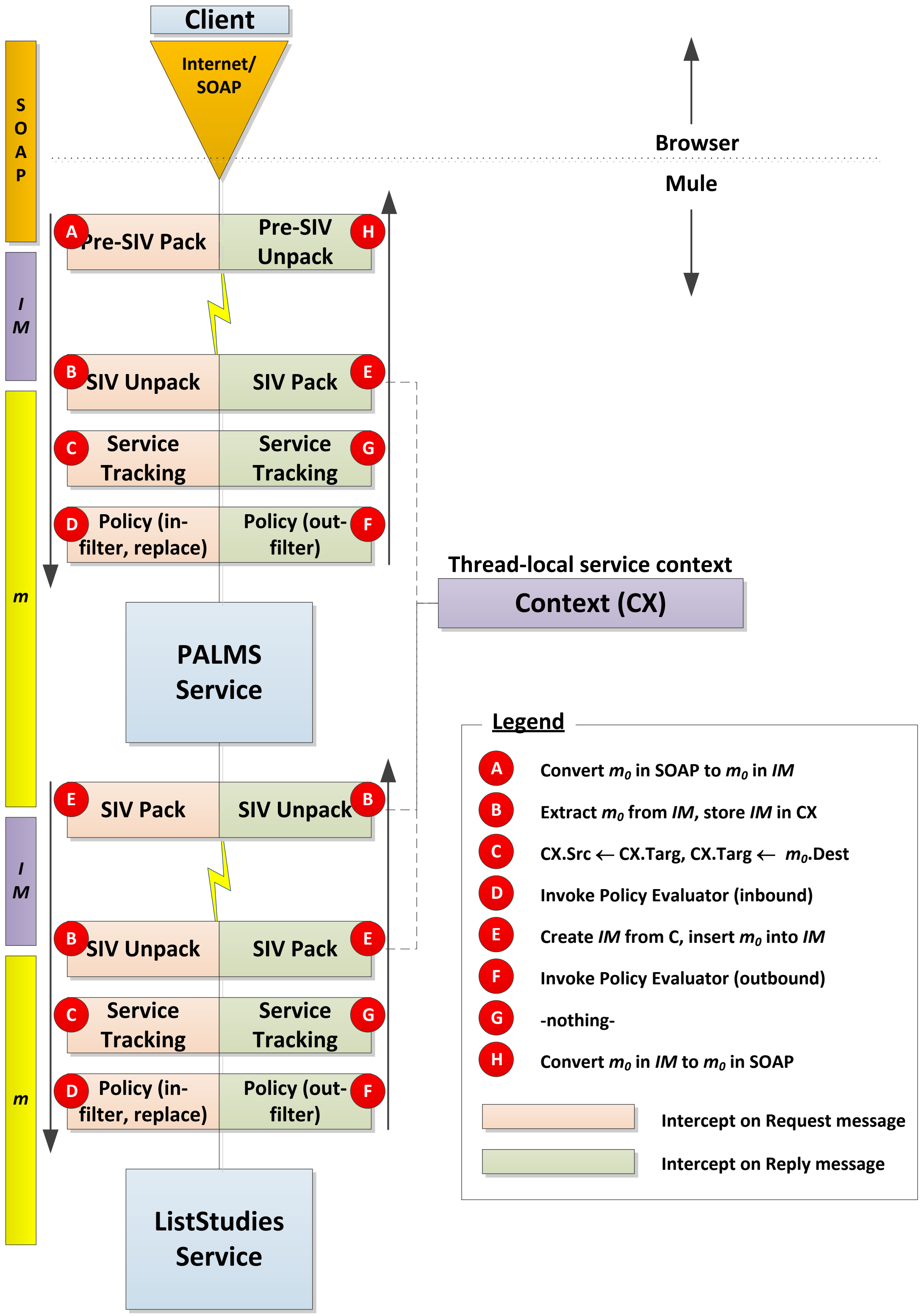

Client
Internet/ SOAP
S O A P
I M
m
I M
m
Browser
Mule
A Pre-SIV Pack
Pre-SIV Unpack H
B SIV Unpack
SIV Pack E
C Service Tracking
Service Tracking G
D Policy (in-filter, replace)
Policy (out-filter) F
PALMS Service
E SIV Pack
SIV Unpack B
B SIV Unpack
SIV Pack E
C Service Tracking
Service Tracking G
D Policy (in-filter, replace)
Policy (out-filter) F
ListStudies Service
Thread-local service context
Context (CX)
Legend
A Convert $m_0$ in SOAP to $m_0$ in IM
B Extract $m_0$ from IM, store IM in CX
C CX.Src ← CX.Targ, CX.Targ ← $m_0$.Dest
D Invoke Policy Evaluator (inbound)
E Create IM from C, insert $m_0$ into IM
F Invoke Policy Evaluator (outbound)
G -nothing-
H Convert $m_0$ in IM to $m_0$ in SOAP
Intercept on Request message
Intercept on Reply message

Figure 33. Simplified PALMS Interceptor Sequencing

Note that the objective of maintaining the source and target service in a SIV is to accurately identify an interaction $P \rightarrow Q$ to which a policy can be applied. While the Service Tracking RIS properly identifies $P$ as the source and $Q$ as the target, it fails when $P$ is decomposed into a service orchestration $P = (P_0 \rightarrow P_1)$. Semantically, the decomposition is encapsulated, and peer services should be oblivious to the decomposition – $P$ is the source service in $P \rightarrow Q$ whether or not $P$ is decomposed. However, the Service Tracking RIS naively records the last service of the decomposition ($P_1$ in $(P_0 \rightarrow P_1) \rightarrow Q$) as the source service for $P \rightarrow Q$ interactions, thus breaching the encapsulation and representing the interaction incorrectly. As a solution, PALMS recognizes the encapsulation by saving the Service Tracking SIV before a decomposition, and restoring it afterward.

#### *5.5.4.2 RAS and RIS Implementation in Mule*

All PALMS RASs are implemented as Mule POJOs that accept messages from a Mule message queue dedicated to it. A Mule configuration (in a Mule configuration file, palms.xml) associates a message queue with a POJO, and defines routing between a POJO (as a message source) and a message queue (as a message destination). Considering the bijective relationship between queues and service POJOs, queue and service names are effectively synonyms.

For example, in Figure 33, the queue associated with the ListStudies service might be defined via an `endpoint-identifier` element in PALMS' Mule configuration file:

```
<endpoint-identifier name="PALMS.ListStudies.queue"
                     value="vm://PALMS.ListStudies.queue" />
```

where the queue is named `PALMS.ListStudies.queue` (via the `name=` attribute), and is implemented using the Mule VM messaging protocol using the name `//PALMS.ListStudies.queue` (via the `value=` attribute).

From a Mule viewpoint, each PALMS service is defined by a tuple consisting of its queue, a POJO, and a list of interceptor services that pre-processes a message after it is removed from the queue and before it is passed to the POJO, and then post-process a message after it is returned from the POJO. For example, the ListStudies service might be defined via a `mule-descriptor` element:

```
<mule-descriptor name="PALMS ListStudies"
  implementation="org.palms.study.PALMSListStudies">
  <inbound-router>
    <endpoint address="PALMS.ListStudies.queue" synchronous="true" />
  </inbound-router>
  <interceptor className="PolicyInterceptors"/>
</mule-descriptor>
```

where the service is named `PALMS ListStudies` (via the `name=` attribute), and it is implemented using the `org.palms.study.PALMSListStudies` Java class (via the `implementation=` attribute). The `PALMS.ListStudies.queue` queue is bound to the POJO (via the `address=` attribute), and is configured for a request/reply pattern (via the `synchronous=` attribute). Finally, the `PolicyInterceptors` names the list of interceptors (via the `className=` attribute).

Mule defines an interceptor list as a sequence of Java classes to call in order, with the interaction message as a parameter. Nominally, each Java class can examine and change the message, then pass it to the next interceptor or the target service (if there are no more interceptors in the list). Each interceptor (or the target service) returns a message to the interceptor that preceded it. An interceptor can also change the message flow, thereby cancelling remaining interceptors and routing to a different target service. For example, an interceptor list associated with PALMS might be defined:

```
  <interceptor-stack name="PolicyInterceptors">
    <interceptor className="org.palms.SIVInterceptor"/>
    <interceptor className="org.palms.ServiceTrackingInterceptor"/>
    <interceptor className="org.palms.policy.PolicyEvalInterceptor"/>
  </interceptor-stack>
```

where the interceptors implement the SIV Pack/Unpack, Service Tracking, and Policy Evaluator RISs.

The interceptor list preceding the PALMS Service is slightly different, and accounts for the Pre-SIV Pack and Pre-SIV Unpack operations needed to bootstrap the PALMS service interactions described above:

```
<interceptor-stack name="PrePolicyInterceptors">
  <interceptor className="org.palms.PreSIVInterceptor"/>
  <interceptor className="org.palms.SIVInterceptor"/>
  <interceptor className="org.palms.ServiceTrackingInterceptor"/>
  <interceptor className="org.palms.policy.PolicyEvalInterceptor"/>
</interceptor-stack>
```

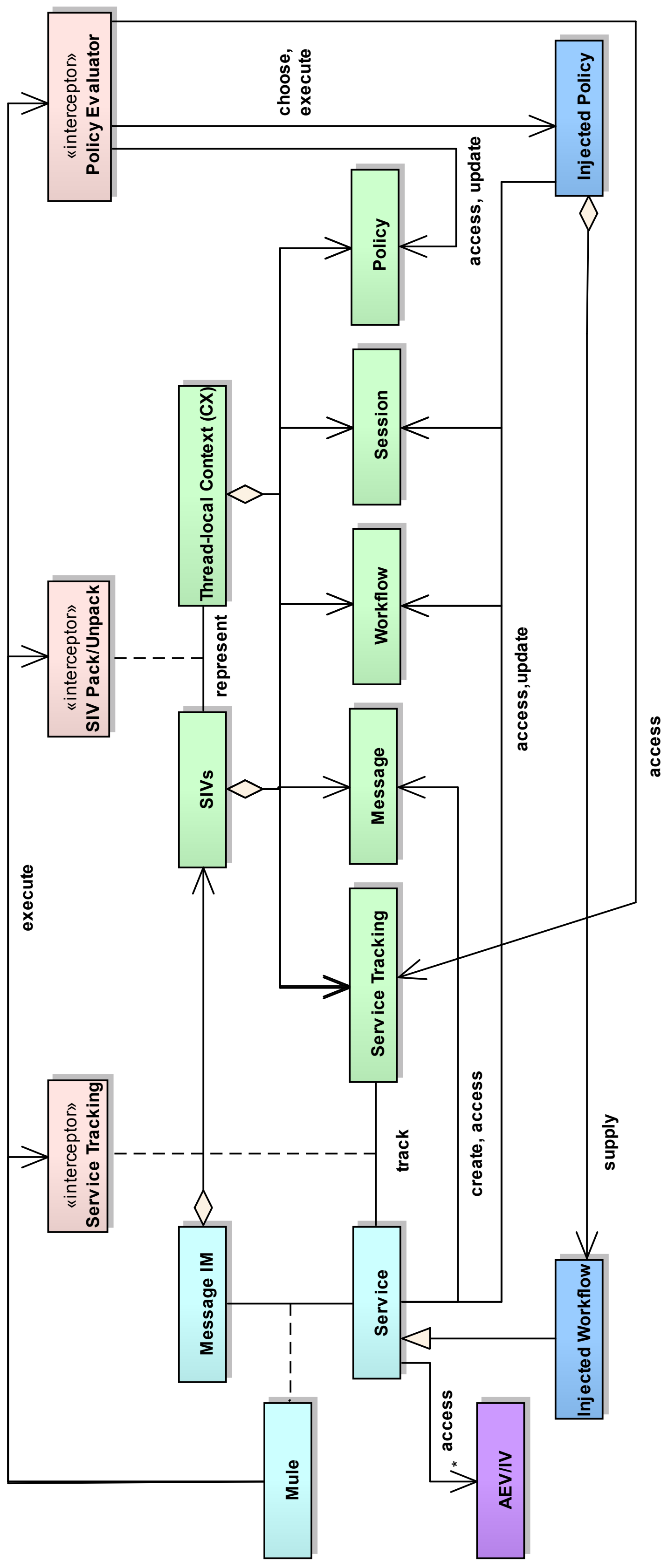
where the `PreSIVInterceptor` interceptor performs these functions, and is followed by the normal `PolicyInterceptors` stack. (`PreSIVInterceptor` can be grouped with the standard `PolicyInterceptors` because Mule guarantees that Web Services processing is performed in the same process as the PALMS Service.) Note that the actual `PrePolicyInterceptors` list contains additional interceptors, as described in Section 5.5.4.3.

Figure 34 depicts the structural relationship between entities in PALMS' workflow system from a Mule perspective. Mule realizes a correspondence between services and messages, thereby executing service interactions. Each interaction is subject to intervention by interceptors per Figure 33, where the SIV Pack/Unpack functions establish the relationship between SIVs (as part of an $IM$) and the thread-local Context (CX) where polices can store and access independent state. Service tracking functions maintain source/target service information, which the Policy Evaluator uses to determine policy appropriate for the interaction.

Note that Figure 34 does not show either the Context services or the Context Infrastructure Services (CIS). Context services manage SIV access and are accessible to all services and interceptors (including policies). Context Infrastructure Services provide GUID-based persistent store (as shown in Figure 18) and are accessible to all services and interceptors via Context services.

Note that while the message $m$ is strictly speaking a SIV, Mule enables services to access $m$ as an argument passed to the service's POJO – services do not access $m$ through Context services is would be the case for other SIVs.

Figure 34. Mule Interservice Message and Service Tracking Implementation


«interceptor»
Policy Evaluator
«interceptor»
SIV Pack/Unpack
«interceptor»
Service Tracking
execute
choose, execute
Injected Policy
access, update
Policy
Session
Workflow
Message
Service Tracking
Thread-local Context (CX)
represent
SIVs
access,update
access
track
create, access
supply
Message IM
Service
Injected Workflow
Mule
access
*
AEV/IV

#### *5.5.4.3 Workflow and Other SIVs*

While a message $m$ represents a data flow between services implementing a base workflow, other SIVs represent state associated with all concerns (e.g., RISs and policies) composed on a workflow instance. A SIV is implemented as a named `AttributeCollection` (per Section 5.5.3.1) that stores key-value pairs and is transported in an $IM$ – if a SIV does not exist when a key-value pair is stored into it, the SIV is automatically created and added to the $IM$'s SIV list. A SIV is destroyed at the end of a workflow, though data it references may not be destroyed – the Workflow and Session SIVs (described below) are examples of different data lifecycles. A concern can access key-value pairs (including creating, reading, storing, deleting, and cloning them) via Context services.

The values maintained in SIVs constitute channels into or out of a service (including a RIS or policy), and are therefore part of the service interface. Consequently, the $IM$ and SIV implementation constitute fundamental support for the workflow composition.

PALMS maintains a number of SIVs (shown in Figure 34):

- **Service Tracking SIV** supports tracking of source and target services by identifying the current service interaction. The tracking service updates it on every message exchange (per Figure 33), and the Policy Evaluator service queries it to determine applicable policies.
- **Workflow SIV** supports policies composed onto a base workflow by the Policy Evaluator. Key-value pairs are stored in the Context Infrastructure Service (CIS), subordinated to a globally unique identifier (GUID) assigned to the workflow and stored in the SIV. One or more policies can use Context system functions to store and retrieve key-value pairs

that persist for the lifetime of the workflow. At the end of the workflow, the GUID and all subordinated key-value pairs are destroyed.

- **Session SIV** supports policies similarly to a Workflow SIV, except that the CIS-backed GUID is created and destroyed explicitly by request of the Client. Consequently, session key-values are accessible to multiple instances of the same or different workflows.
- **Policy SIV** supports the Policy Evaluator, which maintains state that avoids infinite recursion due to the execution of policy while a policy is being executed. It has the same lifecycle as a Workflow SIV.

For example, a Service Tracking SIV contains a tuple $\{source, target\}$ identifying the endpoint services in a service interaction. For the interaction $< PALMS, ListStudies >$, the service tracking tuple would be:

```
{PALMS.ServiceDataConnector.palmsGetStudyList,
   vm://PALMS.ListStudies.queue}
```

For the interaction $< ListStudies, StudyRepository >$:

```
{vm://PALMS.ListStudies.queue,
  vm://PALMS.StudyRepository.queue?method=list}
```

(As described in Section 5.5.4.2, PALMS services are identified by their queue names. However, the source service in the $< PALMS, ListStudies >$ interaction begins with a Web Services entry point, which is not a queue, and is identified by the SDC function name instead. Additionally, the StudyRepository service is qualified by a targeted subservice `list`.)

As SIVs maintain state information for separate concerns, channel content (and channel history) is determinative for such state. SIVs can be used to maintain this information in a number of ways.

For example, while policy decisions are often made based on channel contents (as represented by the current message $m$ or other SIVs), legitimate decisions can also encompass an interaction's channel history, which can be represented as a list of messages $m$ exchanged on an the interaction. Such a list would be indexed by the interaction's $\{source, target\}$ tuple. Maintaining the list as a component of the Workflow SIV would capture only history relating to the current workflow, and maintaining it as an IV would capture history across all workflows. Channel history can be used to implement temporal logic policy predicates such as "if this request was made twice in the last 5 minutes".

Similarly, a channel *trace* can be implemented as a member of the Workflow SIV by accumulating a list of $\{source, target\}$ tuples and messages $m$ for each interaction throughout a workflow. A trace can be used to implement predicates based on control or data flows leading up to an interaction.

Maintaining channel history or traces is often not cost effective because of the storage, bandwidth, or time it requires, particularly because such information must be stored in the CIS so as to be available to any service regardless of the process in which it executes. As an optimization, policies can extract values from messages and store them in the CIS (via a Workflow SIV, Session SIV) or as an IV depending on the longevity required. An example of this is the Multiple Session Separation of Duties (MSoD) solution presented in Section 7.4.1.

The Policy Evaluator (described in Section 5.6) depends on channel history collected in this way. By convention, a message presented to the PALMS Service (from a Client) must contain a credential identifying the user. An interceptor (`CredentialInterceptor`, not shown in Figure 33 or Figure 34) called before the PALMS Service invocation extracts the credential and stores it as a Workflow SIV value available for subsequent policy evaluation.

The PALMS model for the Workflow SIV lifecycle requires an event that unambiguously signals termination of a workflow, after which the SIV's CIS GUID and referents are deallocated. In a serially executing workflow, the workflow is considered complete (and deallocation can occur) when it returns a result to the caller (after the SIV Pre-Unpack service in Figure 33). However, for workflows that spawn long-lived threads, concurrency and termination detection issues arise. Particularly, while concurrent access to CIS values are serialized via message-based protocol, PALMS has no locking and transaction system that would enable workflows with embedded concurrency to operate safely when multiple, coordinated values must be stored. Additionally, PALMS has no means of monitoring the creation or termination of an internal thread, where the creation and termination should delay the deallocation of a Workflow SIV until all such threads are terminated.

#### *5.5.4.4 New SIVs*

While an SIV groups state that has similar lifecycles, SIVs can group state based on other criteria, including security, where one SIV can be transmitted (in an $IM$) in the clear, and another must be transmitted under encryption. Similarly, access to one SIV's values may be unconstrained, while access to another SIV's values must be policy-protected (by policy composed onto interactions with the CIS service).

For example, the Workflow and Policy SIVs both maintain state that persists until the workflow ends. However, state in the Policy SIV bears on the reliability of the Policy Evaluator, while state in the Workflow SIV bears on data flows for policy-defined concerns. Mixing both states in the same SIV risks accidental or purposeful state corruption.

To create a new SIV, lifecycle and all other SIV properties must be addressed. Creating a SIV is a lazy process – a new SIV is created if a concern calls Context services to store a value in the SIV, but the SIV has not been created. Deallocating a SIV is performed by an interceptor in the `PrePolicyInterceptors` list, which executes at workflow termination. As Context services are also responsible for transitioning an SIV from thread-local storage to an $IM$, Context services are a natural encryption point. Currently, Context services are a self-contained, monolithic library service. Implementing such SIV transformations arbitrarily would require re-implementing Context services as an orchestration or plugin architecture.

## 5.6 PALMS' Policy System

The policy system is designed as a separate application, and consists of services implementing policy authorship, a policy repository, and a policy evaluator modeled as a Rich Service as shown in Figure 35. The Authoring

System service interacts with the Policy Repository to store and deploy policies, and the Policy Evaluator service interacts with the Policy Repository to retrieve policies for execution. Policies in the Policy Repository apply to service interactions tracked in the Interaction Repository.

As described in Section 5.5.4.1, the Policy Evaluator service is composed upon PALMS workflows by intercepting every service interaction, determining whether one or more policy applies to the interaction, and then executing any policies that do. Figure 35 depicts this by showing the Policy Evaluator service acting both as a RIS in the PALMS Rich Services and a RAS in the Policy System Rich Service. Considering that policy execution is, itself, a separate concern relative to PALMS base workflows, it is natural to model it as a loosely coupled composable service that is defined, developed, and maintained separately from the PALMS service. Integrating services using this RIS-RAS pattern enables the creation of a System of Systems that relies on the composed service observing the service contracts in force at the target service injection points.

As described in Section 4.2.1, the policy evaluator inherently observes all such service contracts, and delegates the contract fulfillment requirement to services returned by policy evaluation.

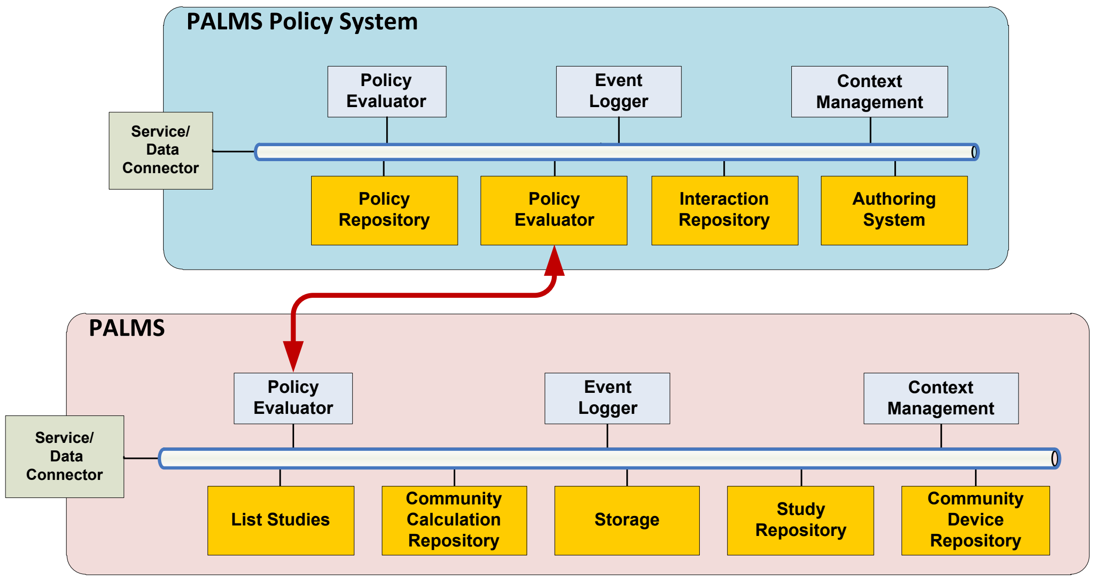


Figure 35. PALMS Policy System Composed onto PALMS Service

In this section, I describe the policy evaluation system, beginning with a description of the basic policy language, followed by a motivating example, a description of policy expression dependencies, the interaction and policy repositories, the authoring system, and the implementation of control, filter, and composition policies.

### 5.6.1 PALMS Basic Policy Language

The discussion in Section 4.2 is agnostic as to the form or language of a policy expression, so long as the result of its execution is a service $W$ that fulfills the service contracts defined for the service interaction to which the policy applies. In order to make decisions, a policy must have access to the contexts described in Section 5.5.4, though to serve the purpose of rapid realization of stakeholder requirements, policies must also be easily correlated with requirements, and must be injectable at runtime with minimum ceremony.

In PALMS, policy expressions are written as XQuery [183] expression, and are executed by the Saxon XQuery processor [184]. XQuery was chosen as the base policy language for a number of reasons:

- Assuming that service interaction messages $m$ (expressed as Java objects), are easily fungible with XML documents (via XStream libraries [185]), the XQuery language and its close cousins, XPath and XSLT, can easily and efficiently interpret, filter, or transform messages in flight
- XQuery expressions can draw on library functions written either as XQuery functions or Java code, including domain-specific libraries and Context service functions
- XQuery can be written and executed as plain text, thereby avoiding compiler and runtime system dependencies in the authoring and repository services – and XQuery can be just-in-time-compiled
- XQuery expressions can be succinct enough to encode domain-level abstractions (as a DSL) without incurring undue notational overhead

The choice of XQuery as a language for policy expression does not address policy execution speed or secure, complete, and consistent deployment. Additionally, it does not result in any guarantees regarding important properties of coordinated policies, such as correctness, completeness, consistency, safety, and liveness. Execution speed is addressed in Section 6.3, deployment is addressed in Section 7.4.4, and policy guarantees are addressed in Section 7.5.

#### *5.6.1.1 Policy Example*

In PALMS, access control decisions drove the design of an access control DSL based on XQuery expressions (as described in Section 5.6.1). Access control requirements were specified by PALMS stakeholders as a spreadsheet that correlated desired decisions with particular workflow functions. From these requirements, I abstracted a set of primitives that would filter messages $m$ (as pre-filters and post-filters) or produce the desired allow/deny decisions (as control policies). Both filtering and control decisions depend on the user's identity (carried in an X.509 credential issued by the caBIG ID Provider [176] to the Client), as correlated with a hierarchically defined role-based (RBAC [186]) and access control list-based (ACL) taxonomy maintained in Grouper [178] by stakeholders themselves, described in Section 5.6.1.3.

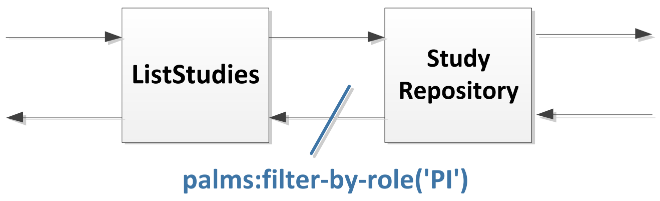


Figure 36. Example of Post-Filter Policy Injection

An example of a post-filter can be applied to a list of studies returned by the Study Repository to the ListStudies workflow activity, as excerpted in

Figure 36. The list is described in Section 3.1 and is returned from the Study Repository as a message $m$. The post-filter returns a new $m$ containing only studies for which the current user is the principal investigator (i.e., the user is contained in the study's 'PI' role).

The filter policy is an XQuery expression `palms:filter-by-role('PI')` where `palms:` is an XQuery namespace, and the `filter-by-role('PI')` function calls an XQuery library function that accesses the current message $m$, finds the study list, and eliminates all studies where the current user is not listed as one of the study's PIs. In this process, the policy accesses four different contexts: the current message $m$, the Workflow SIV (to determine the current user's identity, described in Section 5.5.4.3), application context (i.e., an AIV, to determine the location of the PI role in the RBAC hierarchy), and the RBAC hierarchy itself.

An example of a control policy can be applied to prevent uncredentialed users from accessing the Study Repository in the ListStudies workflow, as excerpted in Figure 37. A decision is injected on the interaction between the ListStudies and Study Repository workflow activities. Per Section 4.2.3, the decision returns either the Study Repository service or the Return Error service, which is then executed and completes the interaction with the ListStudies service.

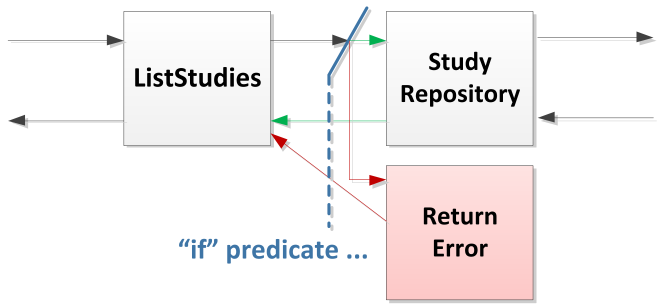


Figure 37. Example of Access Control Policy Injection

The control policy is an XQuery expression such as:

```
if (palms:subject-in-study-role('PI') then ()
else palms:control-error('Invalid role for this operation')
```

The `subject-in-study-role` function calls an XQuery library function that returns `true` if the current user is listed in the RBAC study hierarchy as one of the study's PIs. If so, the interaction with the Study Repository workflow activity is continued. Otherwise, an interaction with the Control Error workflow activity is selected, and that activity returns an error result.

In this section, I describe control policies, filter policies, composition policies, and the various XQuery library functions the policies can call. I also show how policies can be used to inject features into a base workflow to create a System of Systems, and how to create a DSL that is implemented as an XQuery library.

#### *5.6.1.2 Policy Expression Dependencies*

The simplest control policy returns the default service ($Q$, as described in Section 4.2.3), and the simplest filter policy returns the message $m$ that was passed to it. Policies that implement richer requirements base their decisions and calculations on information from a number of sources, depicted as accessible or updateable to Services in Figure 34. Particularly:

- the contents of the interaction message $m$
- the identity of the user on whose behalf the workflow is executed, which may be:
  - passed as a parameter to the workflow and captured in the Workflow SIV per Section 5.5.4.3
  - obtained from a repository (to support deferred workflow execution)
- state created and maintained in the Workflow, Session, or other SIVs for and by policy expressions
- state created and maintained as an IV per Section 5.5.4.3
- state available as an AIV via service interactions with base application services or services presented by other concerns, or via system calls

The PALMS Policy support XQuery library provides access to each source via XQuery function calls, as described in Section 5.6.5.

#### *5.6.1.3 Identity, RBAC,and ACLs*

Many types of access control and filter policies rely on either a user's identity, a role within a virtual organization, or permissions associated with either the identity or the role as applied to particular resources. Such policies include access control (i.e., resulting in an allow/deny verdict), resource allocation and scheduling, auditing, provenance, and information filtering. These decisions rely on an authenticated identity and a repository that maintains a mapping between identities and roles and permissions. Traditionally, these concerns are called authentication and authorization, though authorization has a narrow connotation relative to the broader capabilities represented by policy injection, where a combination of identity, role, and permission can be used to inject features or determine parameters to composed workflows.

Typical predicates evaluated in PALMS policies are framed relative to a group of studies (called a *study group*) that contains the PALMS study involved in the current workflow. A predicate may test whether a user is associated with a particular study group role, or whether a user has particular permissions as a result of the user having a particular role.

PALMS roles and permissions are easiest to configure relative to a study group, not an individual study. Typical predicates include:

- Does the user hold the “PI” role?
- Does the user hold any (or all) of the “PI,RA,Guest” roles?
- Does the user hold a role (e.g., “Collaborator”) relative to a user (e.g., bdemchak)?
- Does the user hold the “addStudy” permission?
- Does the user hold any (or all) of the “addStudy,viewStudy” permissions?

PALMS implements credential and authorization services as separate Rich Services in support of policy evaluation as shown in Figure 38. Credential services accept an X.509 credential, validate the credential and cache the result, and render embedded identity information (e.g., an X.500 Distinguished Name such as `/O=UCSD/OU=LOA1/OU=Dorian/CN=bdemchak`) for use in policy expressions. Authorization services return the mapping between roles, permissions, and user identities stored in caBIG’s Grouper system, which is maintained by caBIG's GAARDS-UI user interface. They return the mapping in an XML form augmented with information that assists in policy calculations.

Note that PALMS' credential and authorization services do not generate an X.509 credential. Such credentials are generated on behalf of clients of the PALMS service (via calls to caBIG Dorian APIs), and are included in all interactions with the PALMS SDC (as described in Section 5.5.2). The credential is saved as state in the Workflow SIV (as described in Section 5.5.4.3), where it is available to policies executing under the Policy Evaluator service.

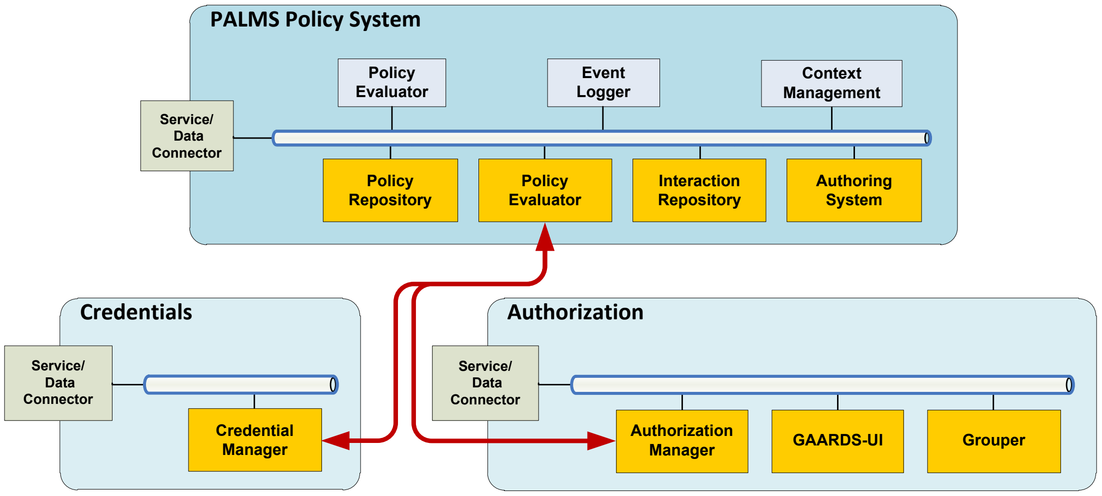


Figure 38. PALMS Credentials and Authorization Services

#### 5.6.1.3.1 *Roles and Permissions under Grouper*

Grouper is a general use Internet2 repository that organizes users and permissions into groups. Under Grouper, groups can include other groups (by reference), and can be hierarchically defined. Grouper enables credentialed access and maintenance of group nodes, thereby enabling group management by individuals on behalf of *virtual organizations* (VOs), examples of which include staff and participants in a PALMS study or associates of an investigator.

PALMS leverages Grouper (as shown in Figure 40) to enable decisions based on a user's membership in a role in a study group or VO. Additionally, decisions can be based on the user having permissions associated with one or more roles, and on the user being included in an access control list (ACL) for a study object (e.g., a Result Set).

Roles for study groups are organized under the `StudiesBranch`, where each study group can have a number of roles (e.g., `PI`, `Manager`, `Helper`, `Guest`), and a user can be a member of any role (as exemplified in Figure 39). The roles for a particular study group are defined by PALMS administrators and users according to the requirements for studies in that group. Additionally, roles can have sub-roles as organized in a Composite pattern [33] (e.g., under `Helper`, sub-roles `Day` and `Night` for shift workers) as required to define policy predicates with fine granularity.

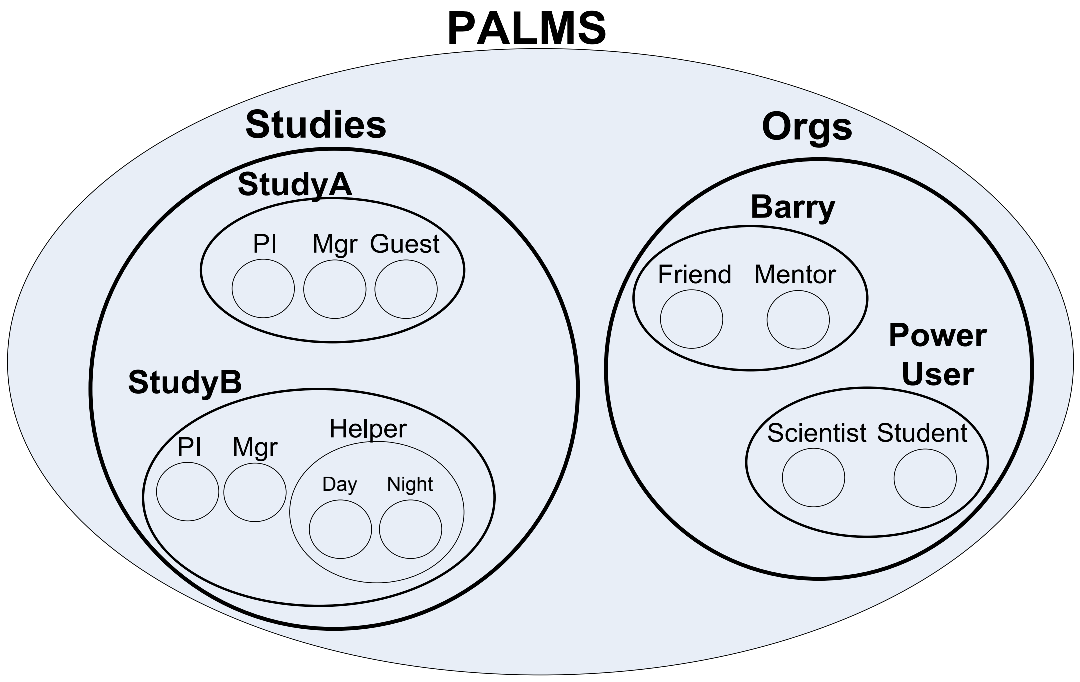


Figure 39. PALMS Role Ontology

As a matter of convention, a user is considered to have a particular role if it is associated with that role or any of its subroles. For example, a test for a user having the `Helper` role returns positive if the user actually holds the `Helper:Day` role, though the converse is not true. This enables fine grained role predicates.

Roles for virtual organizations are organized under the `OrganizationsBranch`, where an organization is simply a group of stakeholders having similar interests or common relationships. As with study groups, organization roles are organized into a hierarchy following the Composite pattern. Examples of organizations include `PowerUser` and `Barry`, where

`PowerUser` may include `Scientist` and `Student` roles, and `Barry` may include `Mentor` and `Friend` roles.

The focus of study group roles is to enable, constrain, or qualify activities performed in the course of conducting a study (e.g., deleting data). The focus of organization roles is to enable activities outside of a study's operations. For example, organizational roles can be used to implement various data sharing models, including a coarse model (i.e., people who can view data vs those that can't) or a concentric model (i.e., people who can view a constrained version of the data, people who can view full data, and people who can view and modify data).

Roles can be associated with lists of permissions, where a *permission* is a token assigned a meaning important to a policy decision (e.g., `p_AddStudy`). Permissions are defined within permission lists under the PermissionsBranch (in Figure 40), and permission lists can contain permission lists according to a Composite pattern. A predicate that tests for a user having a permission essentially tests whether any of the user's roles (e.g., relative to the current study's study group or relative to any organization to which the user belongs) are associated with the permission.

Writing policy predicates that test for permissions enables flexibility in structuring and maintaining role complex hierarchies for study groups and organizations without affecting existing policies. However, the tradeoff is complexity in maintaining the association between roles and permissions.

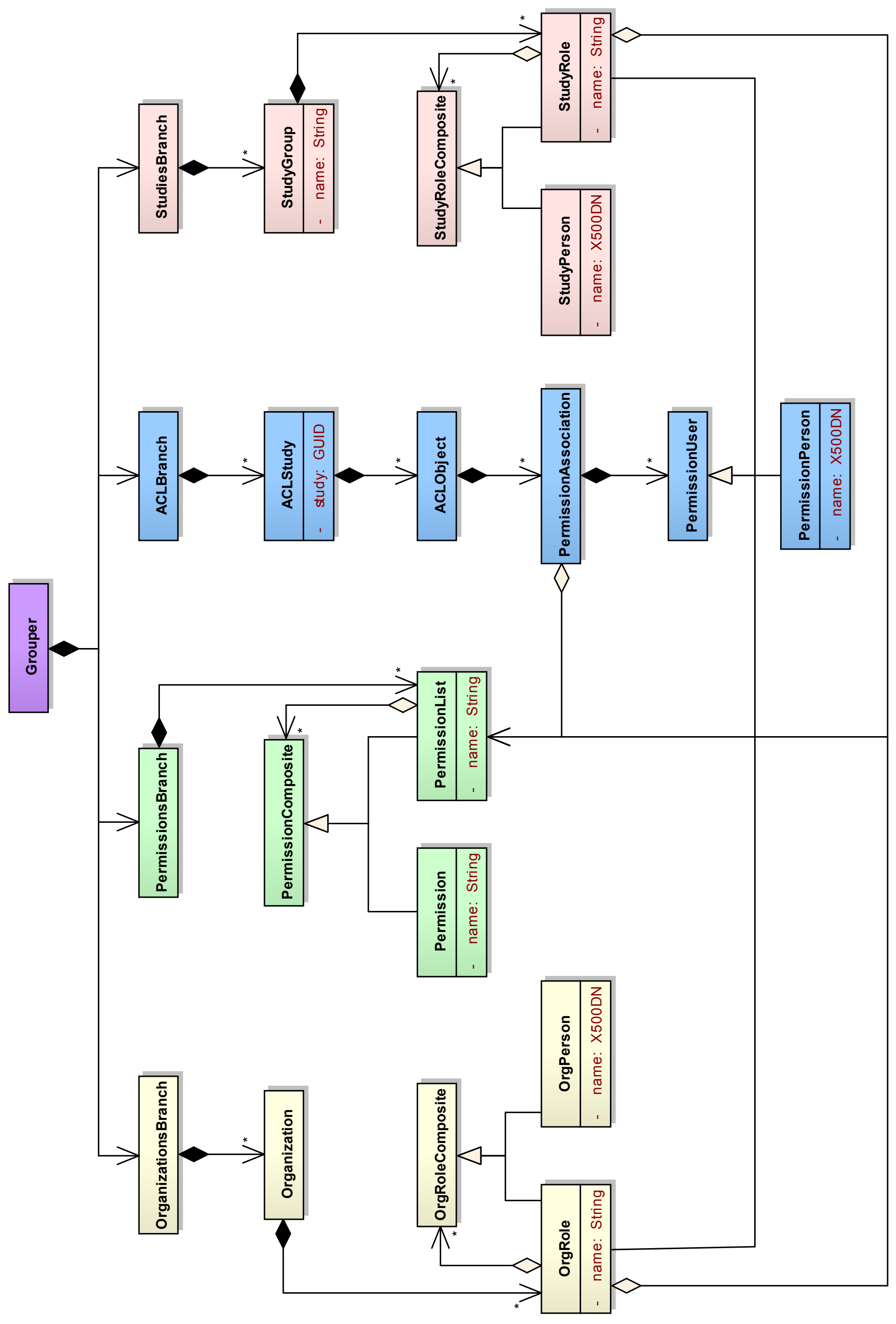
For example, the control policy example in Section 5.6.1.1 makes a decision (`subject-in-study-role('PI')`) based on whether the current user is a member of the `PI` role for the study group containing the current study. An alternative would be a decision based on a permission (e.g., `p_ListStudies`) using an expression such as (`'p_ListStudies' = permissions-for-subject-in-study-role()`), where `permissions-for-subject-in-study-role()` returns a list of permissions associated with the current user's roles in the current study's study group[3].

Permission-based decisions enable the flexible assignment of permissions to roles without hard-coding role names in policy expressions, and they simplify policy expressions by allowing a decision based on the user's having *any* of the roles associated with a permission, instead of having to enumerate these roles.

Currently, PALMS provides no means for users to delegate their roles or permissions to other users.

[3] Note that the XQuery '=' operator returns true if a string is contained in a *list* of strings.

Figure 40. Simplified PALMS Grouper Tree


Grouper
StudiesBranch
StudyGroup
- name: String
StudyRoleComposite
StudyRole
- name: String
StudyPerson
- name: X500DN
ACLBranch
ACLStudy
- study: GUID
ACLObject
PermissionAssociation
PermissionUser
PermissionPerson
- name: X500DN
PermissionsBranch
PermissionComposite
PermissionList
- name: String
Permission
- name: String
OrganizationsBranch
Organization
OrgRoleComposite
OrgRole
- name: String
OrgPerson
- name: X500DN

Finally, individual users and groups of users can be assigned particular permissions relative to specific study objects (e.g., the ResultSet repository or a particular result tracked by that repository) by creating a `PermissionAssociation` under the `ACLBranch` (of Figure 40). A predicate that tests for a user having a permission evaluates ACLs (in addition to permissions granted in the `StudiesBranch` and `OrganizationsBranch`) if the predicate identifies a study and object.

PALMS' use of the Grouper database combines roles as defined by standard RBAC with permissions of a standard ACL model, thereby enabling choice and flexibility in maintaining role hierarchies and in tailoring policy predicates to match stakeholder requirements. To that end, it improves on the standard RBAC model by enabling role hierarchies and combining RBAC-based permissions with ACL-style permissions.

#### *5.6.1.3.2 Role and Permission Namespace*

A role or permission identified in a predicate names a node under a main branch of the Grouper tree (e.g., `StudiesBranch`). In general, a node's full name is formed by a ":" followed by the names of all ancestor nodes, separated by ":".

For example, a role `Helper` in a study group `StudyB` would have the full name `:StudyB:Helper` and would be contained under the `StudiesBranch`. Sub-roles `Day` and `Night` for `Helper` would have the full names `:StudyB:Helper:Day` and `:StudyB:Helper:Night`. As a shorthand, a study group role can be interpreted relative to the study group of the current study by leaving off the initial ":", the study group name, and its ":" separator. For example, if the current study's study group is `StudyB`, the full names above could be abbreviated `Helper`, `Helper:Day` and `Helper:Night`. Thus, with the simple predicate example given in Section 5.6.1.1, the `PI` role is interpreted relative to the study group for the current study.

As a matter of convention, a user holding a role assumes all permissions associated with that role, and for all ancestor roles. For example, a user associated with a `Helper` role holds permissions associated with the `Helper` role, but not the `Day` subrole. A user associated with the `Day` subrole holds `Day` permissions and `Helper` permissions, but not `Night` permissions. This enables the differentiation of roles and permissions by sub-role, thereby enabling fine grained role and permission predicates.

When predicates involve checking for multiple roles (i.e., *any* roles or *all* roles) in a list, the roles can be enumerated in a comma-separated list. For example: `Helper:Day, Helper:Night`.

Identical rules hold for predicates on roles under the `OrganizationsBranch`, except all names must be full names (e.g., `:Barry:Collaborators`), as there is no default organization node in any context.

For permissions, all names must be full names under the `PermissionsBranch`, formed as ":" followed by a ":"-separated list of permission list names, and ending with the name of a permission. This allows the disambiguation of identical permission names used in different permission hierarchies. Examples of permission names include `:StudyPermissions:Admin:addStudy` and `:StudyPermissions:viewStudy`. By convention, associating a permission list with a role gives the role all permissions present in ancestor lists. Using the example above, associating the `:StudyPermissions:Admin` permission list with a study group's `PI` role gives PI users the `addStudy` and `viewStudy` permissions.

### 5.6.2 Policy Repositories and Authorship

As described in Section 5.6 (and shown in Figure 31), the policy system contains the Interaction Repository, a Policy Repository, and an Authoring System. The Interaction Repository implements the abstraction of service interaction identification described in Section 4.2.2. The Policy Repository implements the policy abstractions described in Sections 4.2.3, 4.2.4, and 4.2.6. The Authoring System allows policies to be defined and maintained.

This section describes the implementation of each of these abstractions specific to PALMS-CI use cases. Section 5.6.2.1 describes how the Interaction Repository maintains service interaction information tailored to the implementation of the PALMS-CI on the Mule ESB. Section 5.6.2.2 explains how the Policy Repository implements control, filter, and composition policies responsive to the PALMS concept of study groups. Finally, Section 5.6.2.3 describes PALMS' simple system for editing and staging policy information.

#### *5.6.2.1 The Interaction Repository*

The Interaction Repository represents a collection of service interaction definitions, which identify candidates for policy injection. An interaction definition is a tuple consisting of:

- **Location** is a unique, human-readable name by which policies can identify their injection site
- **Source Service** is endpoint that emits a message intercepted by a control or pre-filter policy – for a post-filter, it is the endpoint that receives the message
- **Target Service** is endpoint that receives a message intercepted by a control or pre-filter policy – for a post-filter, it is the endpoint that emits the message
- **Return Message Type** hints the type of the reply message expected by the Source Service in a request/reply interaction

The Interaction Repository accesses interaction tuples stored in the interactions.properties property file in tag format. A sample interaction tuple is:

```
Interaction1.location = Get Study List (Repository)
Interaction1.sourceService = vm://PALMS.ListStudies.queue
Interaction1.targetService = vm://PALMS.StudyRepository.queue
Interaction1.returnMessage = org.palms.messages.GetStudyListResult
```

The Interaction Repository reads and caches the property file at system startup time, and it presents a service that returns an interaction that matches both a Source Service and Target Service key.

While the interaction tuple identifies service endpoints (as Mule queue names), it does not describe either the input or output channels, including their type, content, or semantics. Furthermore, considering that the interaction itself may be part of a larger protocol, the channel contents may change depending on the state of the protocol. Channel contents are defined and enforced by services themselves, and form a service contract that can be complex, and the description of which is not addressed in this dissertation, but is considered further in Section 7.8.1.

In PALMS, interactions appear once in a workflow, so all protocols are simple request/reply interactions. Nevertheless, a workflow can occur more than once in the context of a higher level workflow. When policy is injected into a workflow, the policy itself can keep state that enables it to make decisions appropriate to the channel definitions for the interaction at the time the policy executes, per Section 5.5.4.3.

Given that PALMS interactions typically follow a request/reply pattern, the Return Message Type enables a control policy to return a service appropriate for multiple interactions. The service uses the `Return Message Type` to determine the type of response that can fulfill the interaction with the Source Service. Using the example in Section 5.5.3.1 (and as shown in Figure 37), if the Return Message Type is `GetStudyListResult`, the workflow returned by the control policy must return an instance of `GetStudyListResult` to satisfy the interaction service contract.

An error handler is an example of a workflow the can be returned by a control policy. As a practical matter, most PALMS messages extend the `PALMSResult` class, and are defined to contain only `PALMSResult` members when an error occurs. An error handler service returns a response of type `Return Message Type`, with the `PALMSResult` members set.

Note that if the service returned by a control policy is unable to fulfill the service contract, it is likely due to a mismatch between the application requirements and the services available to fulfill them. Consequently, this is a signal that the base workflow must evolve before a requirement can be injected, or that the injected service may require helper services. These evolution and mitigation paths are beyond the scope of this dissertation, but are considered further in Section 7.8.3.

#### *5.6.2.2 The Policy Repository*

The Policy Repository represents a collection of policies, including control, filter, and composition policies, each of which identifies the service interaction to which it applies. Because PALMS requirements include the ability to impose separate policies on separate domains (studies, study groups, and the PALMS system as a whole), policies are grouped according to domain (as shown in Figure 41):

- **PALMS** policies apply to interactions in workflows that serve studies and those that maintain system-level resources, including devices and calculations.
- **Study Group** policies apply to interactions serving studies. Policies for a particular study group apply to all studies in the study group (as defined in Section 5.6.1.3.1).
- **Study** policies apply to interactions serving particular studies.
- **Composition** policies combine multiple PALMS, Study Group, and Study policies defined on a single interaction (per Section 4.2.7).

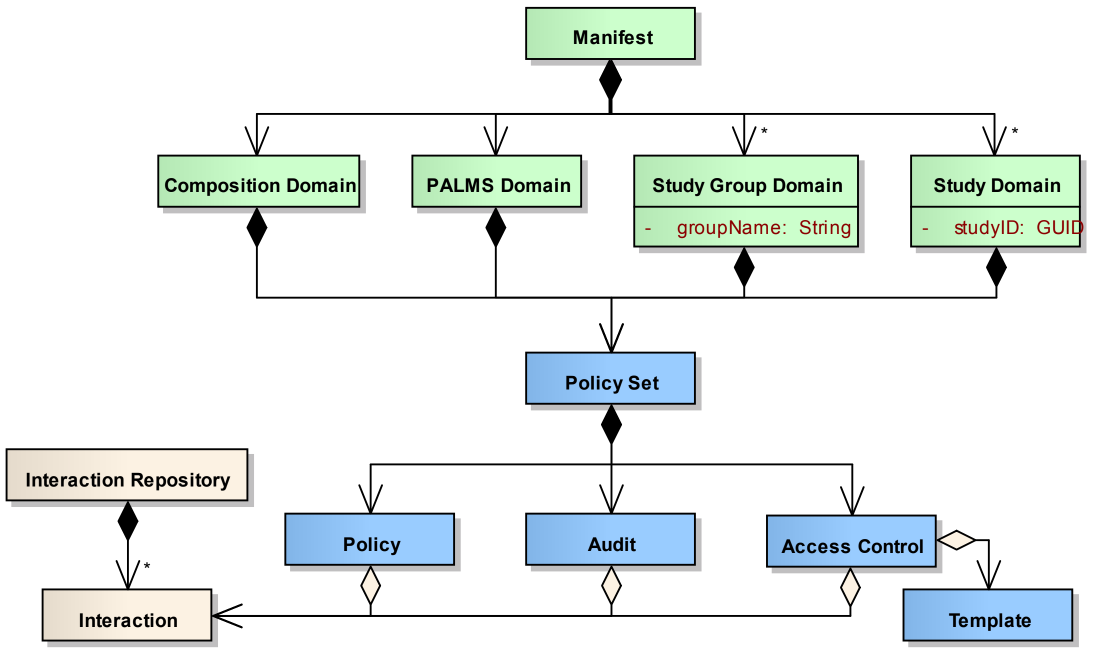


Figure 41. PALMS Policy Domains

Within each domain, policies are grouped as to purpose and form for maintenance convenience. A policy can be expressed as a standalone XQuery expression with metadata, or can reference a template (which contains the XQuery expression and metadata) and provide fillin parameters. Policy groupings include:

- **Access Control** includes control policies that implement access control
- **Audit** includes filter policies that implement auditing functions
- **Policy** includes any control and filter policies (for the PALMS, Study Group, and Study domains) or control and filter composition policies (for the Composition domain)

Note that the list of policy domains and the grouping of policies reflects the circumstances under which policies are executed or maintained, and therefore address stakeholder concerns expressed by PALMS system administrators. They have no effect on policy meaning or execution mechanics, which are described in Sections 4.2 and 5.6.3.

All policies are stored in property files in tag file format. Policy files are indexed in a manifest property file, also in tag file format, where the index keys are domain and category, and the Study Group and Study domains are sub-indexed by the name or GUID of particular study groups and studies. A sample manifest is:

```
palms.palms.accesscontrol = palms.accesscontrol.properties
palms.palms.policy = palms.policy.properties
palms.palms.template = palms.template.properties
palms.palms.audit = palms.audit.properties

studygroup.TestGroup.accesscontrol = \
  TestGroup.accesscontrol.properties
studygroup.TestGroup.policy = TestGroup.policy.properties
studygroup.TestGroup.template = TestGroup.template.properties
studygroup.TestGroup.audit = TestGroup.audit.properties
studygroup.CWPHS.accesscontrol = CWPHS.accesscontrol.properties
studygroup.CWPHS.policy = CWPHS.policy.properties
studygroup.CWPHS.template = CWPHS.template.properties
studygroup.CWPHS.audit = CWPHS.audit.properties

study.d182a31a-2003-4aa7-8d60-622ecd.accesscontrol = \
  d182a31a-2003-4aa7-8d60-622ecd.accesscontrol.properties
study.d182a31a-2003-4aa7-8d60-622ecd.policy = \
  d182a31a-2003-4aa7-8d60-622ecd.policy.properties
study.d182a31a-2003-4aa7-8d60-622ecd.template = \
  d182a31a-2003-4aa7-8d60-622ecd.template.properties
study.d182a31a-2003-4aa7-8d60-622ecd.audit = \
  d182a31a-2003-4aa7-8d60-622ecd.audit.properties
study.86e822ca-f21f-4834-a5f4-8522f7.accesscontrol = \
  86e822ca-f21f-4834-a5f4-8522f7.accesscontrol.properties
study.86e822ca-f21f-4834-a5f4-8522f7.policy = \
  86e822ca-f21f-4834-a5f4-8522f7.policy.properties
study.86e822ca-f21f-4834-a5f4-8522f7.template = \
  86e822ca-f21f-4834-a5f4-8522f7.template.properties
study.86e822ca-f21f-4834-a5f4-8522f7.audit = \
  86e822ca-f21f-4834-a5f4-8522f7.audit.properties

composition.composition.accesscontrol = \
  composition.accesscontrol.properties
composition.composition.policy = composition.policy.properties
composition.composition.template = composition.template.properties
composition.composition.audit = composition.audit.properties
```

The PALMS policy domains correspond to groups of stakeholders that can contribute policies for evaluation, including system administrators (PALMS), principle investigators (Study Group and Study), and PALMS programmers (Composition). Domains exist as a consequence of being identified in the Manifest, being tracked and returned by the Policy Repository, and being selected by Composition policies (as described in Section 5.6.2.2.4). Enabling PALMS to evaluate policies submitted by arbitrary groups of stakeholders is feasible by upgrading the Manifest parsing and repository tracking, and changing applicable Composition policies.

#### 5.6.2.2.1 ***Control Policies***

A control policy is a triple consisting of a policy name, an XQuery expression, and the name of the interaction to which it applies. As described in Section 4.2.3, the XQuery expression returns the service to execute. By convention, returning an empty service selects the interaction's default target service.

A control policy can be expressed in one of three formats, each of which responds to different authorship intent. In the simplest and most general format, the XQuery expression is specified directly (e.g., a control policy injected into the $< ListStudies, StudyRepository >$ interaction (in Figure 37)):

```
GetStudyListPolicy.location = Get Study List (Repository)
GetStudyListPolicy.capability = Get a study list (control)
GetStudyListPolicy.controlExpression = \
  if (palms:subject-in-study-role('PI') then () \
  else palms:control-error('Invalid role for this operation')
```

The `.location` matches the name of an interaction managed by the Interaction Repository (per Section 5.6.2.1). The `.capability` names the policy for reference in policy execution logs and in error messages. The `.controlExpression` is the XQuery expression to execute – in this example, it uses an XQuery library function to determine whether the current user has the `PI` role for the study group containing the current study. If so, it returns an empty service, indicating that the target service is the interaction's target service. Otherwise, it returns a service calculated by the `control-error` XQuery library function, which returns an error result. The `.controlExpression` can return any service that fulfills the interaction source service's contract, and can be based on any calculation.

A more complex policy might involve a compound predicate, an XQuery FLWR construct, an entire XQuery function body, or XQuery function declarations and usage. An example of a compound access control predicate is:

```
if ((palms:subject-in-study-roles('Researcher,PI')
     or (palms:subject-in-org-role(':Barry:Collaborator:Close'))
   and palms:has-acl-permission('p_peek_list'))) then ()
else palms:control-error
       ('Insufficient permissions for this operation.')
```

where access is granted if the current user has the `p_peek_list` permission as a consequence of any roles it holds, and the user either holds the `Researcher` and/or `PI` roles for the current study or is one of Barry's close collaborators.

Note that it is possible to define policies on interactions that result from a control policy evaluation. In the examples above, the policies are defined on the $< ListStudies, StudyRepository >$ interaction, but may result in a $< ListStudies, PolicyError >$ interaction if the `control-error()` function returns a reference to the PolicyError service (as in Section 5.6.3.2). Such a policy would be defined directly on the $< ListStudies, PolicyError >$ interaction.

For convenience, a second control policy format assumes that the alternate workflow returns an error (as in the example above), and involves breaking the control expression into a predicate and an error without manually forming a complete XQuery expression:

```
GetStudyListPolicy.location = Get Study List (Repository)
GetStudyListPolicy.capability = Get a study list (control)
GetStudyListPolicy.controlExpression = \
  palms:subject-in-study-role('PI')
GetStudyListPolicy.controlErrorMessage = \
  'Invalid role for this operation'
```

The complete error expression is formed by the Policy Repository by combining the template's `.controlErrorMessage` with the literal value "`palms:control-error`", and the entire control expression is formed by combining the `.controlExpression` predicate with the literals "`if`" and "`then () else`". In this example, the result is the `.controlExpression` in the triple in the example above.

For access control policies, a third form of a control policy is a template reference, where the template may be shared amongst several access control policies and configured via fillin parameters. The template facility is provided as a convenience supporting optimized production of access control policies – once a template exists, it can to generate an access control policy by supplying parameter values appropriate for the policy.

An example of the template-based access control policy (which generates the policy above) is:

```
GetStudyListPolicy.templateReference = GetStudyListTemplate
GetStudyListPolicy.templateParameter.RoleList = PI
```

The `.templateReference` refers to a group of attributes in the template (i.e., the `GetStudyListTemplate` group). The `.templateParameter` entries identify key names and values for substitution into the template's control expression (i.e., `RoleList` is the key, and `PI` is the value). There can be multiple `.templateParameter` entries if the template accepts multiple substitutions.

The accompanying template could be:

```
GetStudyListTemplate.location = Get Study List (Repository)
GetStudyListTemplate.capability = Get a study list (control)
GetStudyListTemplate.controlErrorMessage = \
  Invalid role for this operation
GetStudyListTemplate.controlExpression = \
  palms:subject-in-study-role ('%RoleList%')
GetStudyListTemplate.paramList = RoleList
```

The policy's `.location` and `.capability` attributes are fetched directly from the template's `.location` and `.capability` attributes.

The control expression predicate is formed by inserting parameter values listed in the reference's `.templateParameter` (e.g., `RoleList = PI`) into the template's `.controlExpression` (e.g., at `%RoleList%`). The template's `.paramList` enumerates the list of required parameters. As in the examples above, if the `.controlErrorMessage` is present, the full control expression is formed by combining the `.controlExpression` with the

`.controlErrorMessage`. Otherwise, the `.controlExpression` is assumed to contain the entire control expression.

Policies that don't fit the access control template mechanism can be defined as a standard control policy triple. The use of access control templates does not preclude the processing of other templates for other purposes, but such template processing would first need to be added to PALMS.

#### 5.6.2.2.2 *Filter Policies*

A filter policy is a four element tuple consisting of a policy name, an XQuery expression, the name of the interaction to which it applies, and filter library support. As described in Section 4.2.4, a filter policy returns a service that transforms an input message $m$ into a new message $m'$. Under PALMS, the filter expression also executes the filter, thereby effecting the message transformation – the filter expression is free to return the original message $m$ if no transformation is appropriate. An example of an output filter policy injected into the $< ListStudies, StudyRepository >$ interaction (in Figure 36) is:

```
GetStudyListPolicy.location = Get Study List (Repository)
GetStudyListPolicy.capability = Get a Study List (filter)
GetStudyListPolicy.outFilterExpression = \
  palms:(:NameSpace-~:)(:~-NameSpace:)filter-by-role('PI')
GetStudyListPolicy.outFilterSupportFile = FilterRemove.support.xq
GetStudyListPolicy.outFilterSupportParameter.1 = studies/studyList
GetStudyListPolicy.outFilterSupportParameter.2 = studyRow/rowValue
```

The `.location` matches the name of an interaction managed by the Interaction Repository (per Section 5.6.2.1). The `.capability` names the policy for reference in policy execution logs and in error messages. The `.outFilterExpression` is the XQuery expression to execute – in this example, it uses the `filter-by-role()` function, which is implemented as a support function (described below) in an XQuery library; it eliminates all studies where the current user is not listed as one of the study's PIs.

(The `(:NameSpace-~:)(:~-NameSpace:)` construct is described further in Section 5.6.2.2.4, and can be ignored in the discussion of filter policies.)

Note that the examples in this section apply to post-filters, as indicated by each of the attribute names starting with `.outFilter`. The discussion applies equally well to pre-filters, whose attribute names start with `.inFilter` instead.

Note that a filter can be defined as a template reference in a manner analogous to template references described in Section 5.6.2.2.1.

The `filter-by-role()` function assumes that the message $m$ has been transformed from its normal Java object form to an XML form. It then uses XQuery operators and PALMS XQuery library functions to:

- find each study in the XML document
- find the study group associated with the study
- determine whether the current user's X.500 identity is contained in any of the roles listed in the role parameter (i.e., PI in this example)
- drop any studies where the user does not have one or more roles
- return a new XML document containing studies that were not dropped

The XML document produced by the function is transformed to its normal Java object form, and forwarded to the interaction's target service. A simplified example of an XML document corresponding to the Java object produced by the Study Repository list function is:

```
<org.palms.ListStudyResult>
  <studies>
    <studyList>
      <studyRow>
        <rowValue>
          <entry>
            <string>studyid</string>
            <string>Study1A2B3C</string>
          </entry>
          <entry>
            <string>groupname</string>
            <string>TestGroup</string>
          </entry>
        </rowValue>
      </studyRow>
    </studyList>
  </studies>
</org.palms.ListStudyResult>
```

where a `<studyRow>` element exists for each study.

A filter function such as `filter-by-role()` is defined in a filter support file named by the `.outFilterSupportFile` attribute. The support file contains all of the XQuery support needed to define and execute the function. The `filter-by-role()` function is parameterized to be flexible regarding where in the XML document it looks for a list of study elements, and where to find a study group within a study element. Such parameters are considered part of the support file definition, and are expressed as `.outFilterSupportParameter` attributes. In the `filter-by-role()` example, `.outFilterSupportParameter.1` indicates the element containing a study list (e.g., `studies/studyList`), and

.outFilterSupportParameter.2 indicates the sub-element containing the study group (e.g., studyRow/rowValue). A given filter function may have its own support file and parameter set.

Note that the process of parameterizing a filter depends on the XQuery language support provided by the XQuery processor. For PALMS, the XQuery processor is Saxon, which provides different language support depending on the license purchased. Paid versions of Saxon provide functional language programming constructs that enable function customization at runtime. The free version is used by PALMS, and such constructs are not available. PALMS overcomes this by rewriting the function immediately before execution, with parameter values substituted directly into the function text. A simplified version of the filter-by-role() function (shown without other supporting functions) is an example of this:

```
  declare function palms:filter-by-role($Roles as xs:string) as
node() {
    palms:substitute($Message/*[1]/(:1-~:)study list param(:~-1:),
       palms:select-children-by-role(
         $Message/*[1]/(:1-~:)study list param(:~-1:)/*,
         "groupname",
         $Role))
 };
```

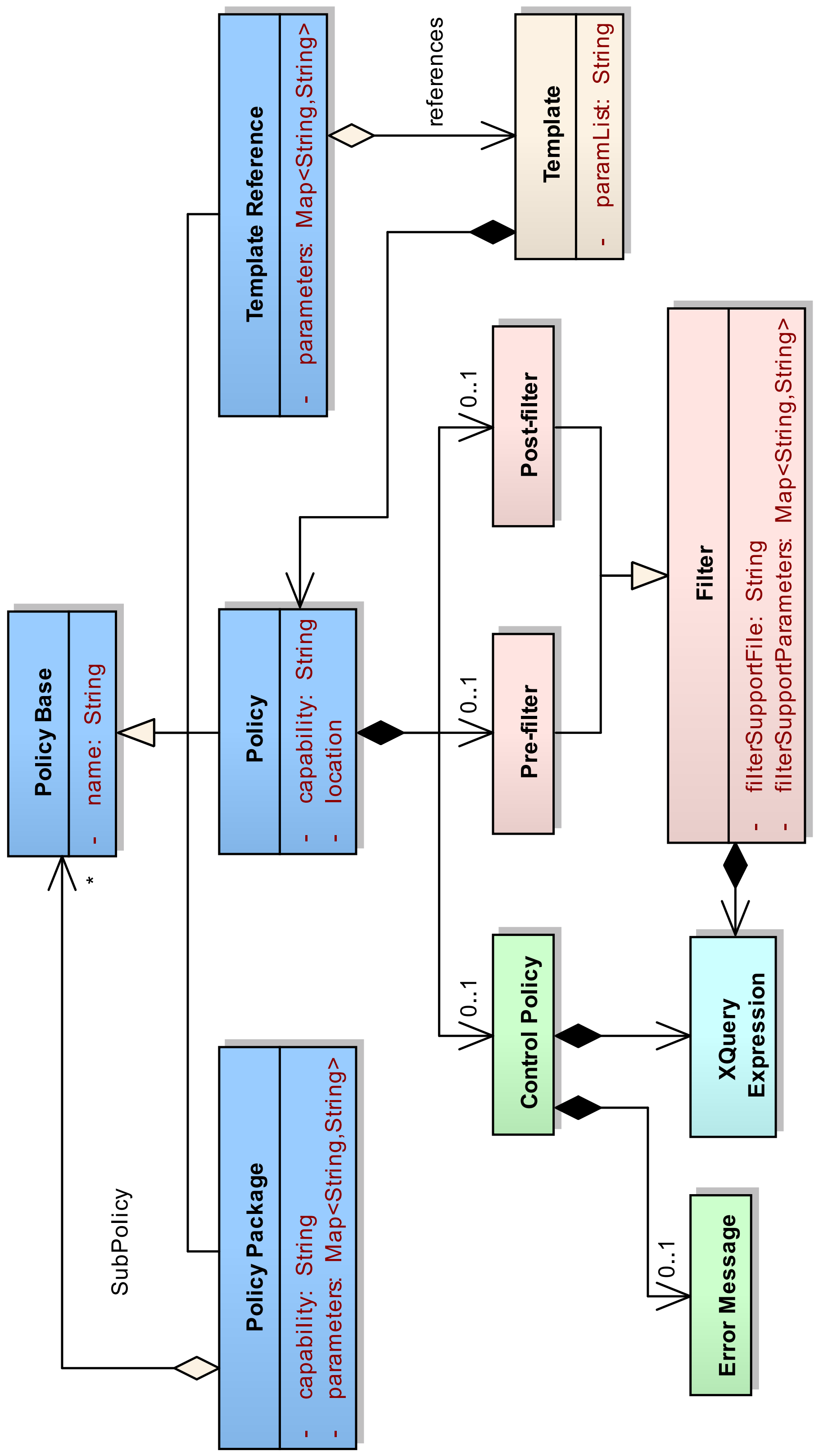
The substitution site for parameter 1 (i.e., `.outFilterSupportParameter.1`) is bracketed by `(:1-~:)…(:~-1:)`; the site for parameter 2 is bracketed by `(:2-~:)…(:~-2:)` and so on. The result of the substitution for the example (with substitution sites in bold) above would be:

```
  declare function palms:filter-by-role($Roles as xs:string) as
node() {
    palms:substitute($Message/*[1]/studies/studyList,
       palms:select-children-by-role(
         $Message/*[1]/studies/studyList/*,
         "groupname",
         $Role))
 };
```

#### *5.6.2.2.3 Policy Packages*

When a requirement calls for the implementation of more than one policy on one or more service interactions (as described in Section 7.4.1), defining each policy independently is effective, but fails to establish that the policies are related. This leads to either external documentation, which atrophies, or policy maintenance errors. PALMS addresses this in a primitive way by grouping related policies together as a policy package. Each policy in a package is called a sub-policy and is named distinctly from other sub-policies in the package. But for their grouping with other sub-policies, a sub-policy is a complete control or filter policy as shown in Figure 42.

Figure 42. Policy Package


Template Reference
- parameters: Map<String,String>
references
Template
- paramList: String
Policy Base
- name: String
*
Policy
- capability: String
- location
0..1
Post-filter
0..1
Pre-filter
Filter
- filterSupportFile: String
- filterSupportParameters: Map<String,String>
0..1
Control Policy
XQuery Expression
0..1
Error Message
Policy Package
- capability: String
- parameters: Map<String,String>
SubPolicy

For example, combining the control and post-filter policies from Sections 5.6.2.2.1 and 5.6.2.2.2 would result in the following:

```
GetStudyListPolicy.capability = Get a study list (package)
GetStudyList.templateParameter.RoleList = PI

GetStudyListPolicy.subpolicy.p1.location = \
  Get Study List (Repository)
GetStudyListPolicy.subpolicy.p1.capability = \
  Get a study list (control)
GetStudyListPolicy.subpolicy.p1.controlExpression = \
  if (palms:subject-in-study-role(%RoleList%) then () \
  else palms:control-error('Invalid role for this operation')

GetStudyListPolicy.subpolicy.p2.location = \
  Get Study List (Repository)
GetStudyListPolicy.subpolicy.p2.capability = \
  Get a study list (filter)
GetStudyListPolicy.subpolicy.p2.outFilterExpression = \
  palms:(:NameSpace-~:)(:~-NameSpace:)filter-by-role(%RoleList%)
GetStudyListPolicy.subpolicy.p2.outFilterSupportFile = \
  FilterRemove.support.xq
GetStudyListPolicy.subpolicy.p2.outFilterSupportParameter.1 = \
  studies/studyList
GetStudyListPolicy.subpolicy.p2.outFilterSupportParameter.2 = \
  studyRow/rowValue
```

Specifically, the `GetStudyListPolicy.capability` names the policy package, and each sub-policy (e.g., `p1` and `p2`) can be injected onto a different interaction, named by its own `.location` attribute. Note that the `GetStudyList.templateParameter` attribute provides fillin values (e.g., `RoleList = PI`) useful in maintaining consistency in sub-policies – they are named and used in the same way as the template parameters described in Section 5.6.2.2.1.

Policy packages can contain policy packages according to a Composite pattern. When a fillin value is evaluated, the `.templateParameter` attribute for the closest parent policy package is used – it overrides all ancestor `.templateParameter` attributes.

#### *5.6.2.2.4 Composition Policies*

A composite policy is a tuple consisting of between two and eight elements, including a policy name and the name of the interaction to which it applies, as with control and filter policies described in Sections 5.6.2.2.1 and 5.6.2.2.2. As described in Section 4.2.7, a composition policy determines the effective policy when multiple policies are defined on the same service interaction. A separate composition policy is defined for control, pre-filter, and post-filter policies.

For control policies, a composition policy's `.controlCompositionExpression` attribute defines the composition function (as an XQuery expression), and its `.controlCompositionSupportFile` attribute identifies the library file (if any) containing the composition function and/or the XQuery functions that support it.

Similarly, the `.inFilterCompositionExpression` and `.outFilterCompositionExpression` attributes defines the composition functions for pre-filters and post-filters, and the `.inFilterCompositionSupportFile` and `.outFilterCompositionSupportFile` attributes identify the library file supporting functions.

An example of a composition policy specification is shown below. It defines composition policies for control, pre-filter, and post-filter policies:

```
Comp1.location = Get Study List (Repository)
Comp1.capability = Get a study list
Comp1.controlCompositionExpression = local:compose-unanimous()
Comp1.controlCompositionSupportFile = ControlComposition.support.xq
Comp1.inFilterCompositionExpression = local:compose-all()
Comp1.inFilterCompositionSupportFile = \
  InFilterComposition.support.xq
Comp1.outFilterCompositionExpression = local:compose-all()
Comp1.outFilterCompositionSupportFile = \
  OutFilterComposition.support.xq
```

The parameters for a control composition policy are defined by the control policies associated with the service interaction. For example, if there are two control policies bound to the `Get Study List (Repository)` interaction, the control expressions for each policy would be parameters to the control composition expression.

As with filter policies (described in Section 5.6.2.2.2), the Saxon XQuery processor used by PALMS does not support functional programming, so control expressions cannot be passed directly as parameters. Instead, before a composition function is executed, its supporting library is rewritten to contain each control expression. The control composition then:

- chooses which control policy to execute
- executes the control policy
- returns the target service the control policy calculates

A similar process is undertaken for filter composition functions. However, because a filter support file itself is customized by rewriting it with filter parameters (as described in Section 5.6.2.2.2), the customized filter is included in the rewritten filter composition support library. Different instances of filter support are distinguished by assigning a unique local namespace identifier (e.g., `N1-`, `N2-`, etc) to each instance's global identifiers (such as helper function names) and references (e.g., calls to helper functions). Consequently, filter functions (including filter expressions in filter policies) contain markers `(:NameSpace-~:)(:~-NameSpace:)` indicating sites for this namespace insertion.

For example, the full `.outFilterExpression` associated with the GetStudyListPolicy is:

```
palms:(:NameSpace-~:)(:~-NameSpace:)filter-by-role('PI')
```

and the full `filter-by-role()` function is:

```
declare function palms:(:NameSpace-~:)(:~-NameSpace:)filter-by-role
                ($Roles as xs:string) as node() {
  palms:(:NameSpace-~:)(:~-NameSpace:)substitute(
     $Message/*[1]/(:1-~:)study list param(:~-1:),
     palms:(:NameSpace-~:)(:~-NameSpace:)select-children-by-role(
       $Message/*[1]/(:1-~:)study list param(:~-1:)/*,
       "groupname",
       $Role))
};
```

The rewritten expression is:

```
palms:N1-filter-by-role('PI')
```

and the rewritten function is:

```
  declare function palms:N1-filter-by-role($Roles as xs:string) as
node() {
    palms:N1-substitute($Message/*[1]/(:1-~:)study list param(:~-1:),
       palms:N1-select-children-by-role(
         $Message/*[1]/(:1-~:)study list param(:~-1:)/*,
         "groupname",
         $Role))
 };
```

Note that if control or filter policies are defined on a service interaction, but without a complimentary composition policy or without a corresponding Interaction Repository entry, the Policy Evaluator will report an error.

A simplified example of a control composition policy (`compose-unanimous()`, shown below) composes two control policies, where if one policy returns a replacement service, the second policy is not evaluated. If either policy returns a replacement service, the service will be used instead of the base workflow service.

```
declare variable $Policies as element()* :=
 <policies>
  <policy name="(:Name1-~:)(:~-Name1:)" valid="
     (:Valid1-~:)(:~-Valid1:)"/>
  <policy name="(:Name2-~:)(:~-Name2:)" valid="
     (:Valid2-~:)(:~-Valid2:)"/>
 </policies>;

declare function local:policy-names() as xs:string* {
  for $policy in $Policies/policy
      return if ($policy/@valid = "") then $policy/@name else ()
};

declare function local:policy1() as item()* {
  (:Text1-~:)'undefined1'(:~-Text1:)
};

declare function local:policy2() as item()* {
  (:Text2-~:)'undefined2'(:~-Text2:)
};

declare function local:exec-policy($PolicyName as xs:string)
    as item()* {
  if ($PolicyName = "(:Name1-~:)(:~-Name1:)")
    then local:policy1()
    else if ($PolicyName = "(:Name2-~:)(:~-Name2:)")
      then local:policy2()
      else ()
};

declare function local:execute-policy-sequence(
                   $PolicyNames as xs:string*,
                   $ReturnVal as item()*) as item()* {
  if (fn:empty($PolicyNames) or fn:count($ReturnVal) > 0)
    then $ReturnVal
    else local:execute-policy-sequence(
      fn:remove($PolicyNames, 1),
                local:exec-policy($PolicyNames[1]))
};

declare function local:compose-unanimous() as item()* {
  local:execute-policy-sequence(local:policy-names(), ())
};
```

As described above, the control composition policy is formed by inserting each control policy via a rewriting operation, where markers in the form of pre-defined comments indicate insertion sites. Comments of the form `(:Name1-~:)(:~-Name1:)` indicate sites in which a policy name is inserted;

comments of the form `(:Valid1--~:)(:~--Valid1:)` indicate sites that mark valid substitutions; and comments of the form `(:Text1--~:)(:~--Text1:)` indicate sites in which policy text is inserted.

The `compose-unanimous()` function uses the `policy-names()` function to collect a list of valid control policies, then uses the `execute-policy-sequence()` function to choose which policies to evaluate, and finally, uses the `exec-policy()` to evaluate a control policy. The sample composition policy accommodates only two control policies – actual composition policies can accommodate any fixed number of policies (by adding more insertion sites). Filter composition policies use similar techniques.

#### *5.6.2.3 The Authoring System*

PALMS policies are stored in a Policy directory and are organized as text-based property files per Section 5.6.2.2. They are read and cached by the Policy Repository service when the PALMS Policy System (described in Section 5.6) starts, and the cache is periodically refreshed to capture changes in the policy files. Policies read from this directory are expected to be syntactically well formed, and if they are not, the PALMS system shuts down. Well-formedness means that each property (attribute) can be parsed, and template references can be resolved to templates actually defined.

Policy files are mirrored in a Policy_Staging directory, and are edited there either with a standard text file editor or with a GUI utility. A policy author can use a text file editor to replace a staged copy. The user can then execute a maintenance utility that invokes a Policy Repository service that verifies well-formedness of all staged files. The service either returns an error or copies staged files to the Policy directory, where the Policy Repository can read them and put them into effect.

PALMS exposes services that allow a GUI utility (under development) to read and update template-based access control policies (as described in Section 5.6.2.2.1), thereby allowing the GUI utility to display and assign roles to access control checks on pre-defined service interactions. The GUI can examine the template to determine its description (via its `.capabilities` attribute) and the parameters it requires (via its `.paramList` attribute). Based on these, the GUI utility presents access control as a relationship between user identities and roles in the context of each interaction. This allows stakeholders to configure policies without being exposed to the underlying policy language. As such, the GUI is a high level DSL.

Policy development and debugging are discussed in Section 5.6.6.

Authorship of interaction tuples (in the interactions.properties file described in Section 5.6.2.1) follows similar lines, where a text editor is used to add, change, or delete interaction definitions, which are cached in the Interaction Repository. The Interaction Repository's interaction cache is periodically refreshed to capture interaction tuple changes.

### 5.6.3 Policy Evaluator

The PALMS Policy Evaluator is a service injected into all PALMS service interactions, as described in Section 5.5.4.1. It assumes a request/reply service interaction where $P$ is the source service, $Q$ is the target service, and messages $q$ and $a$ exchanged between $P$ and $Q$ (as in Section 4.2.2):

$$P \overset{q}{\rightarrow} Q \text{ followed by } Q \overset{a}{\rightarrow} P$$

The reply interaction is optional, though most PALMS interactions require it. (Messages $q$ and $a$ are instances of message $m$ from previous discussions.)

The Policy Evaluator intercepts $q$ (outbound from $P$) and returns $a$ (inbound to $P$) intuitively as follows:

$$a = postFilter(control(preFilter(q))$$

The Policy Evaluator's internal workflow consists of three main stages (as shown in Figure 43):

- marshaling the policy execution context
- fetching the policies for the current interaction
- executing the policies.

The marshaling phase interacts with various PALMS and composed services to collect information needed to fetch and execute policies. The Interaction Repository (Section 5.6.2.1) returns the service contract that $Q$ must fulfill. The Credentials Repository (Section 5.6.1.3) returns the user's identity (for use in policy decisions), and the Authorizations Repository (Section 5.6.1.3) returns the mapping between user identity, roles, and permissions. The Study Repository returns the study group associated with the study ID contained in message $q$.

Fetching policies depends not only on the current interaction, but on the study and study group associated with the interaction message $q$ – the Policy Repository (Section 5.6.2.2) returns pre-filter, control, and post-filter policies pertaining to the PALMS, study group, and study domains, and also returns composition policies pertaining to the current service interaction.

Finally, the XQuery processor is called to execute the pre-filter, control, and post-filter policies in order, thereby transforming message $q$ to message $a$, and returning $a$ to $P$. Filter and control policy evaluation are described in Sections 5.6.3.1 and 5.6.3.2. Note that such policies are evaluated only in the context of a composition policy, which evaluates any and all policies appropriate for an interaction as described in Section 5.6.2.2.4. In this section, filter and control policy evaluation is assumed to be within the context of an appropriate composition policy. Note that if a policy is defined on an interaction, PALMS requires that a corresponding composition policy be defined on the interaction, too – if there is less or more, PALMS signals an error.

Because the Policy Evaluator is injected into every PALMS interaction, it is possible that Policy Evaluator recursion could result during either the marshaling activities or policy execution. Policy Evaluation during the marshaling phase is an opportunity for infinite recursion – a Policy Evaluation interacts with a repository, and the interaction results in a second Policy Evaluation, which attempts the same repository interaction, thereby triggering the infinite recursion. The Policy Evaluator prevents this by setting a lock in the Policy SIV (described in Section 5.5.4.3), and checking the lock immediately upon entry. If the lock is set, service $Q$ is executed (and message $\alpha$ is returned) without any policy intervention – essentially, policies cannot be applied to repository interactions conducted by the Policy Evaluator.

A Policy Evaluator execution consequent to policy execution is desirable, as this amounts to policy on injected concerns. This does not result in infinite recursion because an atomic (non-decomposed) service will eventually be injected.

Note that Figure 43 shows marshaling operations performed in parallel – no marshaling operation depends on another marshaling operation. In fact, PALMS' Policy Evaluator performs these operations serially as a coding convenience. To execute in parallel (or to enable Policy Evaluator to function in a multi-threaded workflow), the Policy SIV lock would have to be upgraded to a semaphore.

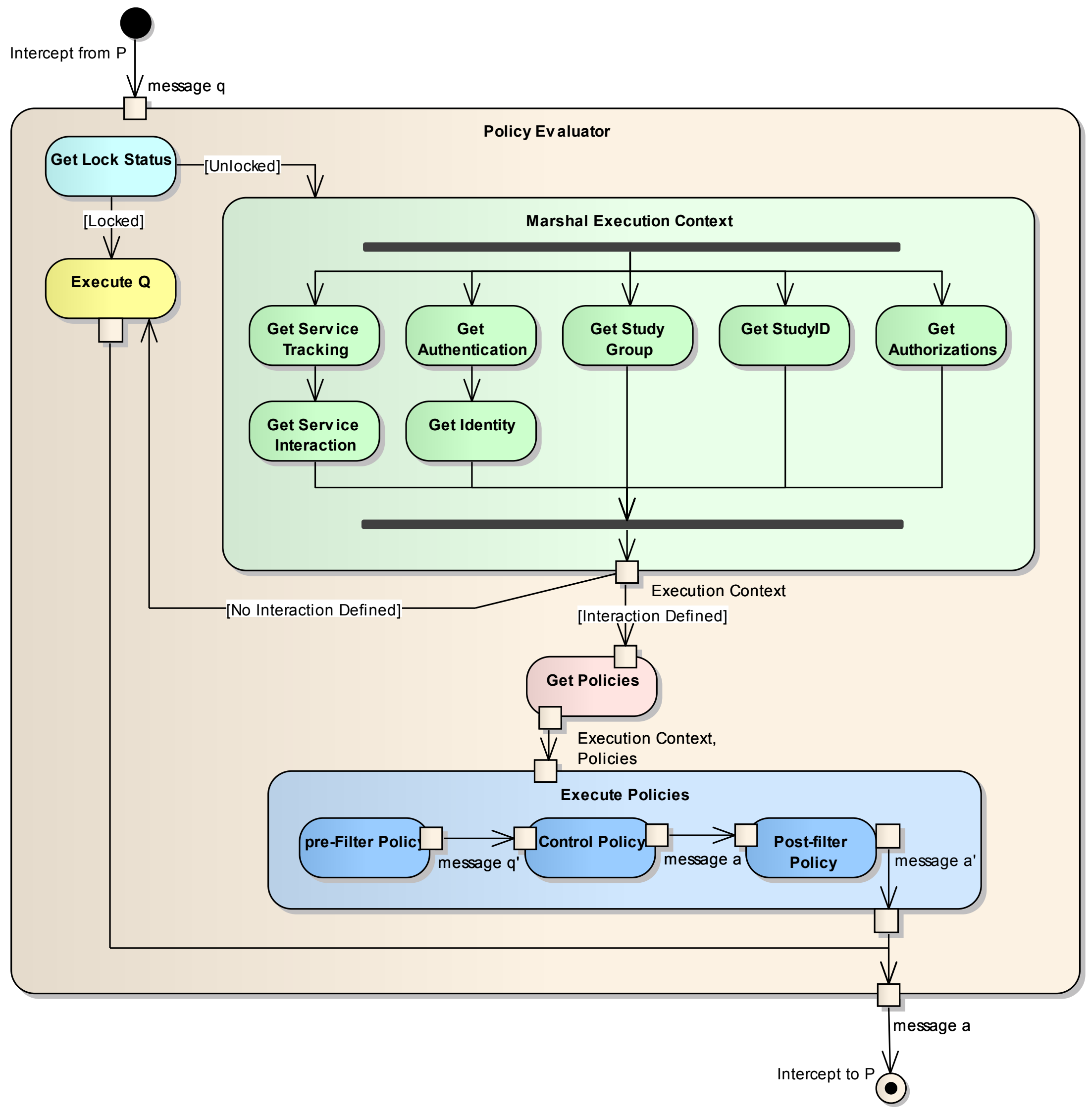


Figure 43. PALMS Policy Evaluator

### *5.6.3.1 Evaluation of Filter Policies*

As described in Section 5.6.2.2.2, a pre-filter is evaluated in the same way as a post-filter, with the difference being that a pre-filter transforms a message $q$ passed to a service, and the post-filter transforms a message $a$ returned from a service. In this section, the description of PALMS' filter processing applies equally to pre- and post-filters, which are composed using pre- and post-filter composition policies (as in Section 5.6.2.2.4).

For a given service interaction, if a filter composition policy is defined, all filter policies for the interaction are processed by the filter composition policy to determine the effective filter expression. Considering that the Mule ESB transports Java objects, and PALMS services process Java objects, a filter expression transforms one Java object to another. The transformation occurs as shown in Figure 44 (for pre-filters).

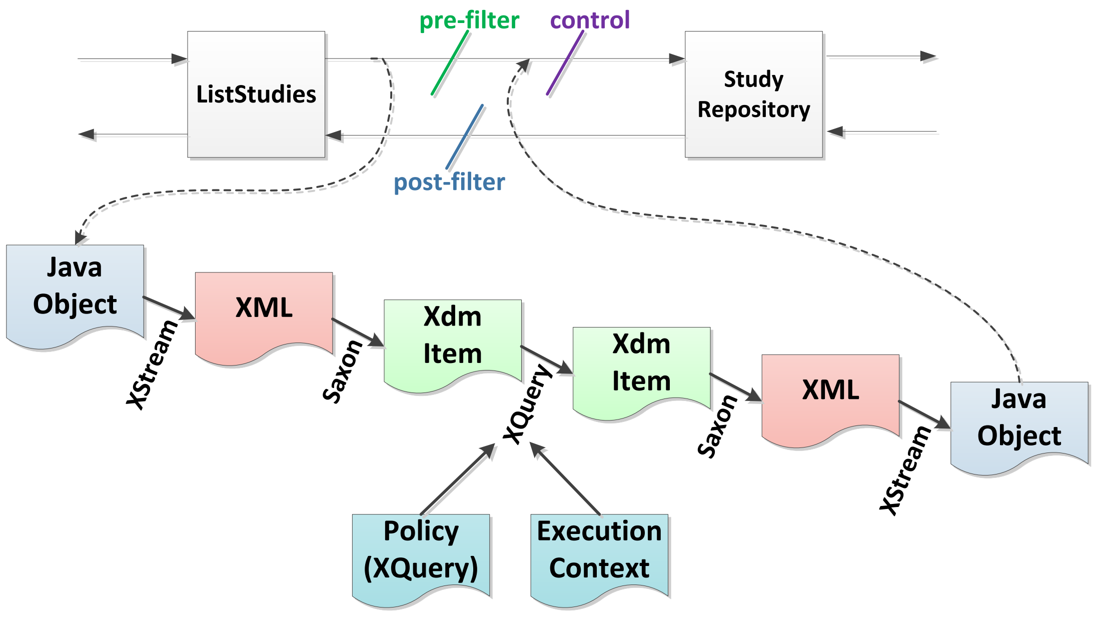


Figure 44. Filter Evaluation Sequence

The XStream Java library [185] is used to convert the Java object to an XML document, and the Saxon Java library is used to convert the XML document to an internal Saxon format. The converted message and the execution context are assigned to global variables in a Saxon context object, and the policy expression is then evaluated by the Saxon XQuery parser. While a null policy simply returns the converted message, a more substantial policy (e.g. as in Section 5.6.2.2.2) returns a transformation of the converted message, possibly using the execution context during the transformation. Finally, the returned message is converted back to XML, and then back to a Java object.

#### *5.6.3.2 Evaluating Control Policies*

Unlike a filter (which returns a message that is forwarded to a service), a control policy returns the service to be executed next (per Section 5.6.2.2.1). While the evaluation sequence for a control policy (as shown in Figure 45) is similar to the sequence for a filter policy, a control policy returns an array of Saxon-formatted values that may represent a workflow, and may be configured in three ways:

- **Null or empty** – indicates that the default service $Q$ should be executed
- **Single string value** – indicates the name of the Mule queue corresponding to the service $S$ that replaces $Q$
    - **Optionally, an XML document** contains a new message $q'$ to pass to the replacement service $S$ instead of the normal interaction message $q$, and is realized as a Java object using the Saxon and XStream functions used during filter evaluation (as in Figure 44)

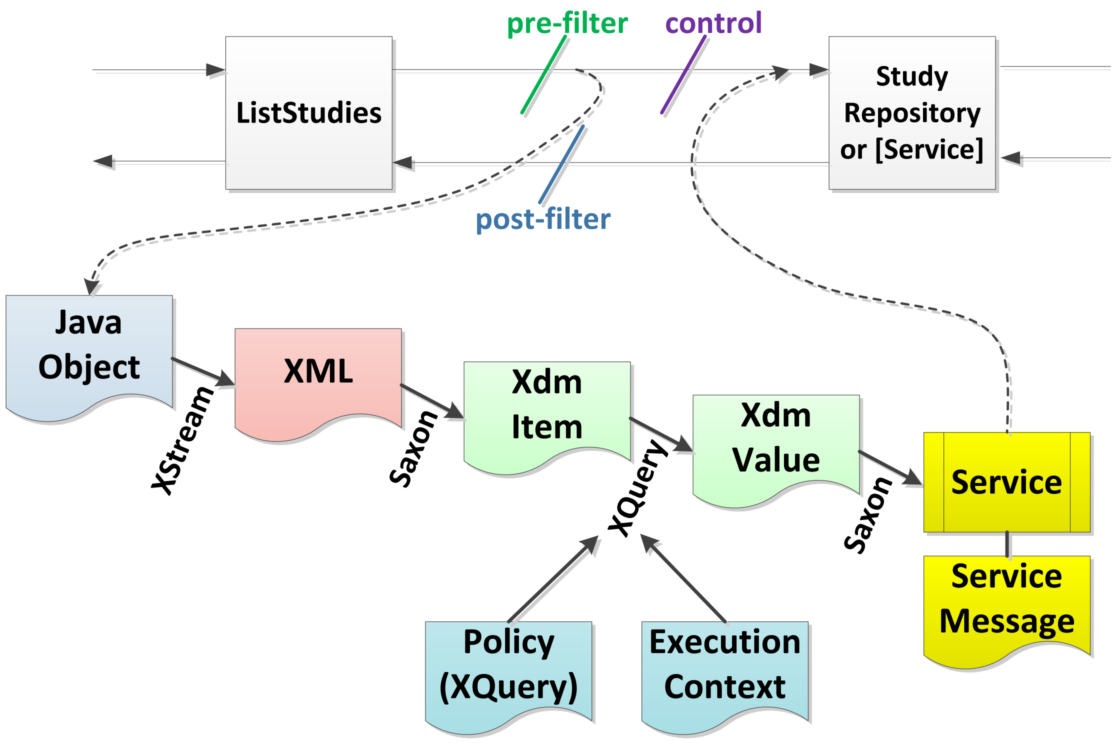


Figure 45. Control Evaluation Sequence

Returning the optional XML document is functionally equivalent to returning a dynamically constructed workflow that decomposes into two services: a transformation of message $q$ to $q'$, followed by service $S$.

Note that Section 4.2.3 calls for a control policy to return a workflow. Because under Rich Services, a service itself can decompose into a workflow, a policy can be said to inject either a service or, equivalently, a workflow. Within this definition, there are no limitations on the calculation that can be performed by the control policy decision expression, or on the activity of the injected workflow. Furthermore, the injected workflow can be parameterized or customized by the control policy, so long as it realizes the original service contract.

The common case under PALMS is for an existing service to represent and implement the workflow as a service decomposition – this case is implemented by returning a Mule queue name. The uncommon case of returning a dynamically created workflow is represented by the Policy Evaluator constructing the workflow based on returning both the service and a service message. Other workflow constructions are possible, though not implemented in PALMS.

For example, consider the control policy in Section 5.6.2.2.1:

```
if (palms:subject-in-study-role('PI') then () \
else palms:control-error('Invalid role for this operation')
```

The `()` service corresponds to specifying that the default service $Q$ be executed.

The `palms:control-error()` function returns a Mule queue name and replacement message as follows:

```
("Policy.Error.queue",
 <org.palms.messages.PALMSResult>
    <error>{$ErrorText}</error>
 </org.palms.messages.PALMSResult>
)
```

where `Policy.Error.queue` corresponds to the PolicyError service, which returns the replacement message. The replacement message in this example is the `org.palms.messages.PALMSResult` XML document, which contains the text of the error (derived from the `control-error()` parameter `$ErrorText`).

Note that the dynamic construction of the error message represents the myriad processing opportunities available to a replacement service such as `control-error()`. Other possibilities include customizing the type of the XML document according to the interaction specification, logging the error, and performing failure detection and/or remediation.

### 5.6.4 Feature Injection

PALMS' base workflows focus on data storage and retrieval, and do not entangle orthogonal concerns (i.e., features) such as auditing, provenance tracking, failure management, and information assurance issues. Given that the requirements that define such features are often fluid and are managed by diverse stakeholder communities having different interests, such features are natural candidates for policy-based injection as described in Section 4.2.6. Stakeholders define the policies, which in turn express which information is captured and under what conditions.

As an example, I cast an auditing concern as an independent application comprising acquisition, storage, and visualization sub-concerns as shown in Figure 46 (with an abbreviation of the policy system shown in Figure 35). Injection of the acquisition sub-concern into PALMS base workflows creates a System of Systems where both PALMS and the audit application can evolve independently.

The audit system consists of a listener service that is invoked by a policy expression (described in Section 5.6.5.3), a repository for audit information, and an audit information visualizer. The listener stores audit information by interacting with the repository service. The visualizer is invoked to view and analyze audit information, and interacts with the repository service. Given that the audit services participate in a workflow, their interactions are subject to composition of yet other concerns, such as encryption, failure management, and data flow augmentation (e.g., time stamping).

Note that a real world auditing system would store audit information in a secure, tamperproof store, and may include commercial visualizers such as Crystal Reports [187]. The PALMS audit system stores audit information in local tab-separated text files, which can be viewed and analyzed in Excel.

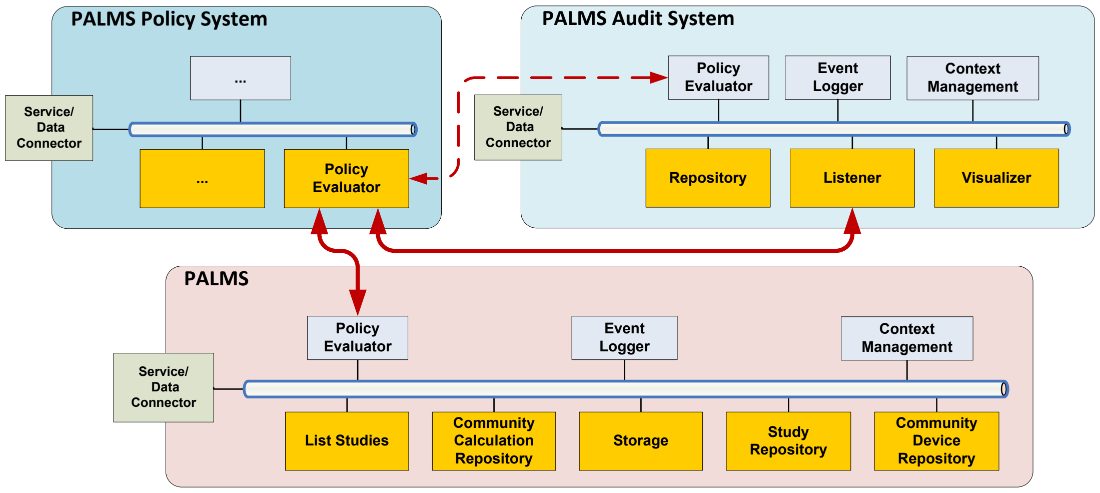


Figure 46. PALMS Audit System Composed onto PALMS Service via Policy

An example of feature injection is implemented as a post-filter on the $< ListStudies, StudyRepository >$ interaction:

```
AuditLow.location = Get Study List (Repository)
AuditLow.outFilterSupportFile = FilterAudit.support.xq
AuditLow.outFilterExpression =
  palms:(:NameSpace-~:)(:~-NameSpace:)audit(
    "AuditID1",
    ("event", "success"),
    ("user", xf:get-workflow-user()))
AuditLow.capability = Audit Study List (low level)
```

As a filter, the .outFilterSupportFile attribute names the XQuery file that contains functions that define an audit Domain Specific Language (see Section 5.6.5); the .outFilterExpression attribute specifies a filter expression that returns the current message $q$, which is supplied as a member of the policy execution context.

Ultimately, the `audit()` function relies on interactions with the separate Listener service. Such feature injection is facilitated by the `call-service()` function described in Section 5.6.5.3.

As an independent application, the audit system maintains state pertinent to meeting its requirements, and control or data flow within the application can be workflow-dependent if the CIS workflow GUID is passed to it. However, the policy decision that results in an interaction with the audit system can, itself, maintain state via IVs as in Section 5.6.5.1. Such state can affect either future decisions, can constitute part of the information exchanged with the audit system, or can affect other policy decisions and workflows derived from them.

Policies and groups of policies that maintain and consume state are, themselves, injectable features in that they implement a discrete requirement set, which may or may not be further implemented by injection of workflows as demonstrated in Section 7.4.1.

### 5.6.5 PALMS Domain Specific Languages (DSLs)

While the PALMS policy system enables the composition of control and filter policies on service interactions, the policy expressions (described as $\pi$ in Section 4.2.1) themselves reflect the intentions of the policy writers. An important insight of PDD is to enable policy phrasing in terms of languages congruent with the domain concepts understood by the writers, and which can therefore be easily, accurately, and repeatably authorable. To achieve this, PDD promotes the use of Domain Specific Languages [35] (DSLs) to compose policy expressions, where DSLs can be tailored by and for a stakeholder community interested in expressing policy.

The XQuery language supports the concept of DSLs by allowing:

- parameterized definition of functions using basic data structures, including strings, arrays, collections, and structured types such as XML
- combination of functions in logical expressions
- control structures, including looping, encapsulation, and decomposition
- composition of functions to create higher abstractions
- automatic parallelization of expression evaluation
- access to external information sources and transformations

PALMS includes a number of DSLs useful in expressing policy in different domains, thereby catering to interests of diverse stakeholder groups that have been important to PALMS so far. Additionally, XQuery enables the improvement of existing languages and the definition of new languages simply through declaring new XQuery functions. The current list of PALMS' DSLs reflect the interests of PIs (for controlling access to study data, results, and calculations), PALMS operators (for combining or prioritizing policies submitted by stakeholders), and PALMS administrators (for tracking resource access and billing) – additional DSLs (or evolution of existing DSLs) are possible and are encouraged so as to address the requirements of new or existing stakeholders. They include:

- **Access Control** (described in Sections 5.6.1.1, 5.6.2.2.1, and 5.6.2.2.2) allows a determination of whether a user holds a role, a list of roles, or any in a list of roles within a study or virtual organization. It also allows similar calculations based on permissions based on such memberships.
- **Policy Composition** (described in Section 5.6.2.2.4) allows the combination of multiple control and filter policies defined on a single interaction.
- **Audit** (described in Section 5.6.4) allows injection of audit tracing on a single interaction.
- **Feature Injection** (described in Section 5.6.4) allows injection of arbitrary services on a single interaction.

The design of the Access Control DSL followed a traditional development process, where I marshaled access control requirements and use cases, then abstracted primitives that filter data flows and produce desired allow/deny decisions as described in Section 5.6.1.1. Each other DSL was designed using the same technique, while making incremental modifications and upgrades to DSLs to reflect emerging requirements.

In particular, major functions available in each DSL are listed in Table 7.

Table 7. Major DSL Functions

| Domain | Function | Parameters | Return |
|---|---|---|---|
| **Access Control** | subject-in-any-study-roles<br>subject-in-all-study-roles<br>subject-in-any-org-roles<br>subject-in-all-org-roles | $role-list | boolean |
| | subject-in-study-role<br>subject-in-user-role | $role | boolean |
| | has-acl-permission | $object, $permission-list | boolean |
| | permissions-for-subject-in-study-role | | permission-list |
| | permissions-for-subject-in-user-role | $role | permission-list |
| | filter-by-attribute | $name, $value | message |
| | filter-by-any-role | $role-list | message |
| | filter-by-role | $role | message |
| | control-error | $queue, message | $queue, message |
| **Policy Composition** | compose-unanimous<br>compose-override | | message |
| | compose-all<br>compose-preemptive<br>compose-hierarchical | | $queue, message |
| **Audit** | audit | $auditID {,(name, value)}* | message |
| | audit-if | $boolean-condition, $true-param-list, $false-param-list | param-list |
| **Feature Injection** | call-service | $queue, $message | message |

Section 5.6.3.2 presents an example of the use of an Access Control control function that returns a Boolean value (which determines a service to return) and the `control-error()` function. Section 5.6.2.2.2 presents an Access

Control filter function that returns a message. The `has-acl-permissions()` function returns a Boolean value if a user holds one or more permissions pertaining to a particular study object. Similar functions exist for determining whether the user has one or more permissions as a result of roles it holds.

Section 5.6.2.2.4 describes the `compose-unanimous()` control composition policy, which evaluates each control policy, and returns the default service if no control policy overrides it. Similarly, the `compose-override()` policy performs a `compose-unanimous()` composition on PALMS-level policies (if any exist), or study group-level policies (if no PALMS-level policies exist), or study-level policies if no PALMS-level or study group-level policies exist.

For filter composition policies, the `compose-all()` policy evaluates each filter, one after the other. The `compose-hierarchical()` policy evaluates the PALMS-level filters, then the study group-level filters, and study-level filters in order. The `compose-preemptive()` policy evaluates filters using the same rules as the `compose-override()` control policy.

The Audit functions and the Feature Composition function are described in Section 5.6.5.3.

### *5.6.5.1 XQuery Library Functions*

A number of helper functions are available for authoring DSL functions (including the DSL functions themselves), and policy authors are free to add helper functions as appropriate. A sample of existing helper functions are listed in Table 8.

Table 8. DSL Helper Functions

| Topic | Function | Parameters | Return |
|---|---|---|---|
| **Current Message** | cur-name() | $role-list | Java object class |
| | cur-elements() | $xml-element-name | XML document |
| | cur-value() | $xml-element-name | value of element |
| **Inbound Message** | inbound-name() | $role-list | Java object class |
| | inbound-elements() | $xml-element-name | value of element |
| | inbound-value() | $xml-element-name | value of element |
| **SIV** | get-workflow() | $element | value of element |
| | get-workflow-user() | | X.500 user name |
| **AEV** | get-study-list() | $study-id | XML document |
| **IV** | create-iv-id() | | GUID |
| | clone-iv-id() | $context-id | GUID |
| | drop-iv-id() | $context-id | true |
| | get-iv-value() | $context-id, $key | value of element |
| | get-iv-values() | $context-id | XML document |
| | set-iv-value() | $context-id, $key, $value | true |
| | delete-iv-value() | | true |

All helper functions service control, filter, and composition policies. Each policy can define its own helper functions, and persistent common functions can be added to the XQuery libraries described in Section 5.6.5.2. Additionally, helper functions can call other helper functions and DSL functions. Notably, the `call-service()` function can be used to create AEV functions by interacting with pre-existing services in the PALMS or other Rich Services.

The Current Message functions return values from the message exchanged in interaction for which the policy is being evaluated. For a pre-filter or control policy, this is the message q (interactions ❶/❹ in Figure 6), and for a post-filter policy, this is the message $a$ (interactions ❺/❽). The Inbound Message functions always refer to message $q$ regardless of the policy.

The SIV, AEV, and IV functions set and return context values as described in Section 5.5.4.

Note that there are no functions that alter the contents of a message. Using standard XQuery, an XML document cannot be altered – instead, a new document containing the desired alterations must be generated. An XQuery function can leverage the `auth-utils` functions described in Section 5.6.5.2 for this.

#### *5.6.5.2 XQuery Policy Support Libraries*

Filter and composition policy definitions can refer to XQuery support libraries that are available only for the duration of the policy evaluation (as described in Sections 5.6.2.2.2 and 5.6.2.2.4). Other XQuery support libraries are available to all policies at all times. PALMS' XQuery libraries are arranged as a layered architecture, where top-level libraries leverage low-level libraries as in Figure 47.

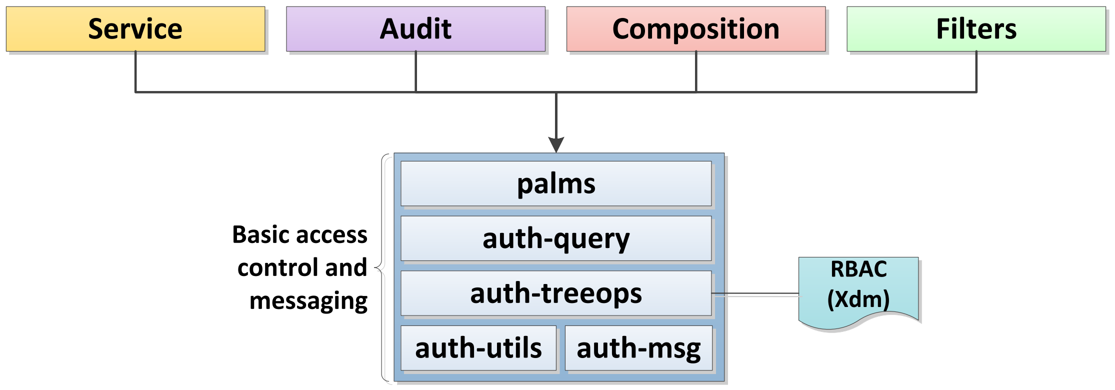


Figure 47. PALMS XQuery Library Hierarchy

The `auth-utils` and `auth-msg` libraries contain utility functions that apply to XML documents in general, and PALMS messages in particular. The `auth-tree` library contains functions that query the PALMS Grouper tree for role and permission inclusion (described in Section 5.6.1.3.1), and the `auth-query` library contains functions that support access control decisions (described in Section 5.6.5) by calling `auth-tree` functions. The palms library provides interface functions for access control functions by calling `auth-query` functions.

Other DSL support functions reside in individual libraries, and they call utility functions in the `palms` stack.

#### 5.6.5.3 *Specialized DSLs*

The Audit DSL is an example of a specialized policy language that supports injection of the independent audit concern. DSLs supported under PALMS' XQuery execution model are expressed as named XQuery functions that transform parameters into a result, with the function possibly having side effects. To create the Audit DSL, I modeled the roles and relationships supporting the audit abstraction (using a UML class diagram as shown in Figure 48), then expressed the audit operations as XQuery functions.

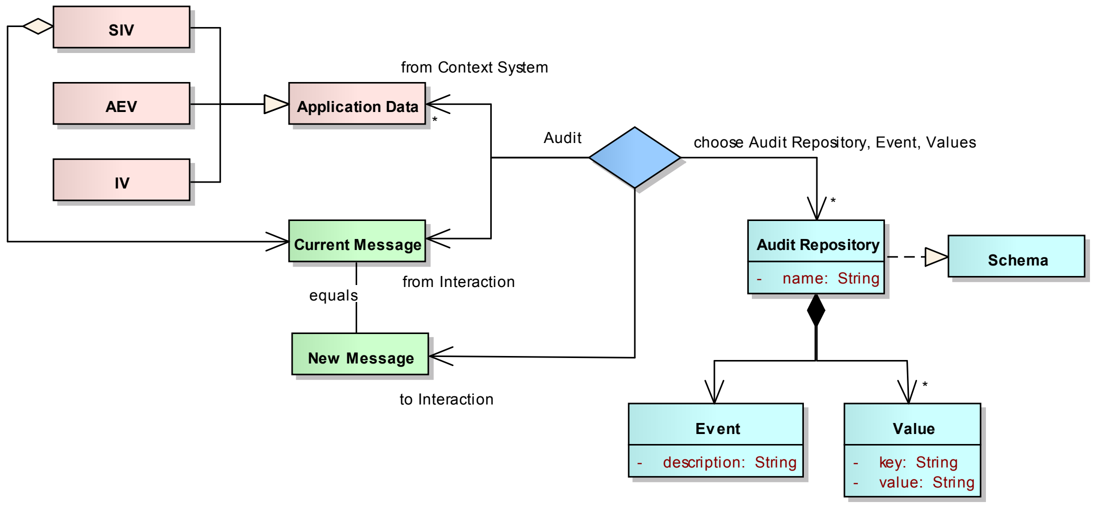


Figure 48. PALMS Audit Operators

Abstractly, the audit operator generates an audit event based on values available from the context system (described in Section 5.5.4), including the interaction message $m$. It chooses an audit repository, an event type, and the contents of a tuple forming the event description. Tuple elements are key-value pairs. Finally, it returns a new message $m'$, which is propagated through the service interaction. For an audit operation, $m' = m$.

I posited that each repository would have its own schema. From implementation experience, I posited that the choice of repositories could be fixed for a given audit operation, and that choice determined the types of events and tuple values. However, the choice of actual event and tuple values depended on the system state at the time of audit.

Consequently, the audit operators in the DSL are `audit()` and `audit-if()` where `audit()` writes a tuple to a repository, and `audit-if()` chooses the event and tuple to write. The XQuery function declarations are:

```
declare function palms:audit($AuditRepository as xs:string,
                   $KeyValueList as xs:string*) as node() {

declare function palms:audit-if($Chooser as xs:boolean,
                   $TrueReturn as xs:string*,
                   $FalseReturn as xs:string*) as xs:string*
```

A simple example of an audit DSL expression is:

```
palms:audit("AuditGetList",
            palms:audit-if(palms:cur-value("error") = "",
                           (("event", "success"),
                            ("user", xf:get-workflow-user()),
                           (("event", "failure"),
                            ("error", palms:cur-value("error"))))))
```

The hypothetical `AuditGetList` audit repository is defined to maintain an event type and a single key-value pair describing the event. The actual event and key-value pair depend on whether the interaction message $m$ contains an error. If so, the error is fetched from the message and logged. If not, the user's name is fetched from the Workflow SIV and logged. Note that the message contents and user name are fetched using context library functions described in Section 5.6.5.1. Using combinations of SIV, AEV, and IV functions (including message functions), the `audit()` function can log application, environment, and policy state in addition to message contents.

While the `audit-if()` function is defined completely using XQuery, the `audit()` function interacts with a service representing the audit system listener (described in Section 5.6.4). As shown below, the `audit()` parameters are packaged as an XML document and passed to the `call-service()` function, which converts the XML document to a Java object (using the XStream Java library) and sends it to the audit listener service POJO using the Mule ESB (either synchronously or asynchronously):

```
let $audit := element org.palms.audit.AuditRequest {
  element auditRepository {$AuditRepository},
  element keyValuePairs {palms:make-key-value-list($KeyValueList)}}

xf:call-service("vm://Audit.AuditRecord.queue", $audit)
```

An example of a policy defined using the audit DSL is presented in Section 5.6.4, where an `audit()` call is used to implement feature injection. Note that the audit DSL is implemented in the `FilterAudit.support.xq` XQuery support file, which is referenced in the audit policy definition.

While the Audit DSL reflects simple requirements, and is implemented using simple XQuery function definitions, more complex DSLs can be modeled and created using the same techniques. For example, a DSL for provenance tracking would take a similar form, including identifying source data, passing the data to an independent service, and resuming the base workflow. Section 7.4.1 presents an example of an MSoD policy realized as a DSL.

Note that while XQuery (and therefore policy expressions) can include complex predicates, control flows, and data flows, such complexity works against the easy, reliable, and unambiguous expression of requirements. A DSL can be used to encapsulate such complexity, when complex relationships can be encoded as higher level abstractions. This greatly reduces the need for complex policy expressions, and shifts policy debugging load to the DSL author, as described in Section 5.6.5.4.

As the requirements driving a DSL change, the DSL can change, too. In the simplest scenario, additional functions can be made available in the XQuery support library without affecting existing functionality. The XQuery language itself supports type overloading, thereby enabling a degree of evolution without changing function names. PALMS does not currently support any versioning system that would correlate a policy expression with a version of supporting XQuery library.

#### *5.6.5.4 Policy Support Development and Debugging*

While the PALMS system development and debugging support are provided by a combination of Eclipse [188] and the Java log4j library [189], there is no equivalent system for XQuery-based DSL development.

DSL support libraries can be developed in vitro using the oXygen XML Editor IDE [190] and a suite of test cases and test results. In vivo development is supported by logging using XQuery calls to `audit()` functions as described in Section 5.6.5.3. Note that the Saxon XQuery engine freely prunes execution paths that do not contribute to a result. Consequently, in vivo debugging within an XQuery function may yield results that shed insufficient light on internal function execution (and at first may be confusing and inscrutable).

Note that under PALMS, replacing an XQuery library with an updated version automatically results in PALMS using the new version, thereby contributing to online experimentation.

### 5.6.6 Policy Development and Debugging

Policies are authored by policy programmers who are aware of the domain to which a policy applies, and the DSLs available for the expression of that policy. When a DSL supports a policy well, it provides parameter-driven XQuery functions that succinctly express the policy, and its implementation is vetted as in Section 5.6.5.4. To the extent this is true, policy debugging focuses on the complete and correct expression of requirements using DSL functionality, including correct composition of such policies onto the base workflow. Additionally, policies can be written without DSL support simply by writing an appropriate XQuery expression.

The interests of a policy programmer are, themselves, a crosscutting concern addressable in the policy domain. While PALMS provides no explicit policy debugging DSL, existing DSL functions can be used for that purpose:

- Calls to the `audit()` and `call-service()` functions (described in Section 5.6.5.3)can be incorporated into a DSL support library to shed light on DSL execution. They can also be incorporated into a policy expression itself (as shown below). Particularly, the `call-service()` function can be used to call custom debugging facilities by invoking external services using programmer-supplied data.
- Audit policies can be injected into interactions between a policy expression and the Context system (described in Section 5.5.4) to track SIV and IV variables.

Additionally, the Policy Evaluator (described in Section 5.6.3) records a log of all policies evaluated, including their parameters and result.

Examples of coupling a policy expression with an `audit()` or `call-service()` function call include:

```
palms:subject-in-study-role('PI') and
  palms:audit('TestLog', 'message', $Message)

palms:subject-in-study-role('PI') and
  xf:call-service("vm://Testing.queue", $Message)
```

Note that the use of DSLs or raw XQuery expression generally relies on knowledge of the structure and semantics of the message(s) exchanged during the interaction. While the Interaction Repository (described in Section 5.6.2.1) defines the interaction endpoints and the type of message $a$ expected by the source service in a request/reply interaction, it does not describe the message $q$ sent by the source service to the target service – a more robust interaction description would include this information. Regardless, coordinating (request or reply) message structure and semantics with policies that depend on them is not addressed in this dissertation, but is considered further in Section 7.8.1.

A simple example of a possible mismatch between a message and policy occurs when using the `audit()` function to capture information from the reply message of a $< ListStudies, StudyRepository >$ exchange (as described in Section 5.6.2.2.2). The `org.palms.ListStudyResult` message contains a `studies` element but contains no `device` element. Attempting to inject an audit policy referencing a non-existent `device` element would result in recording a blank value, without detecting an error either during policy authorship, policy injection, or policy execution:

```
palms:audit('TestLog', 'deviceName', palms:cur-value('device'))
```

In order to inject a policy that matches the structure and semantics of an interaction message, a policy programmer must know the structure and semantics ahead of time.

## 5.7 Summary

In this chapter, I presented the PALMS-CI as a case study demonstrating an implementation of the principles explained in Chapter 4, and which currently serves a growing worldwide user base. It addresses a number of practical issues, starting with the choice of service implementation (i.e., ESBs, particularly Mule), conventions on message passing and message content, support for user-based access control requirements (as an example that can be applied to other requirement domains), and the mechanics of maintaining workflow context in systems that can be scaled and distributed across platforms.

I chose the Rich Services blueprint as a modeling framework for the PALMS-CI, which allowed the expression of workflows in terms of service interfaces, orchestrations, and decompositions, and allowed the expression of policy evaluation as an interceptor-based infrastructure service. Rich Services closely aligns with the vision of cyberinfrastructures as System of Systems that realize stakeholder requirements as a composition of partial behaviors. I described the implementation of context in the policy system as a relationship between workflows, infrastructure services, and interservice messaging, all within the Rich Services paradigm.

Based on this, I described the purpose and operation of the Policy Repository, which is a key entity in the management of policy, independent of policy definition and actual policies. It makes the runtime connection between collections of control and filter policies, policies that compose them, and the service interactions onto which they are composed. As the Policy Repository is conceptually agnostic as to the base policy language (e.g., XQuery), it serves as a conduit connecting the policy authorship process with the policy evaluation process.

This chapter also examined the conceptualization and use of DSLs tailored to particular requirement sets (given an XQuery substrate), including the mechanics of authoring such policies, on one hand, and executing them, on the other. It described how new DSLs can be created and deployed to address new and emerging stakeholder concerns, all responsive to stakeholder requirements at runtime.

The PALMS-CI demonstrates how ubiquitous policy evaluation combined with state management at the service, workflow, application, and other levels can enable the conceptualization of requirements as systems that can then be combined with base workflows into a System of Systems integration (e.g., the Audit system). This concept is further developed in Section 7.4.1.

Finally, Chapter 6 presents an evaluation of the use of PDD in PALMS-CI case study, particularly demonstrating that policy injection occurs at an acceptable cost in many cases, while identifying costly cases that give insights into future PALMS-CI evolution paths.

Chapter 7 compares PDD and its PALMS-CI implementation to alternative approaches and implementations. It explains how PDD fulfills the gaps identified in Chapter 2, and also describes outstanding issues.

## 5.8 Acknowledgments

Chapter 5, in part, is a reprint of material as appeared in 3 papers:

1) B. Demchak and I. Krüger, "Policy Driven Development: Flexible Policy Insertion for Large Scale Systems," in 2012 IEEE International Symposium on Policies for Distributed Systems and Networks, Chapel Hill. IEEE Computer Society, Jul. 2012, pp. 17-24. The dissertation author was the primary investigator and author of the text used in this chapter.

2) B. Demchak, J. Kerr, F. Raab, K. Patrick, and I. H. Krüger, "PALMS: A Modern Coevolution of Community and Computing Using Policy Driven Development," in 45th Hawaii International Conference on System Sciences (HICSS), Maui, Hawaii. Jan. 2012. The dissertation author was the primary investigator and author of the text used in this chapter.

3) B. Demchak, C. Farcas, E. Farcas, and I. Krüger, "The Treasure Map for Rich Services," in Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration (IRI), Las Vegas, USA. IEEE, Aug. 2007, pp. 400-405. The dissertation author was a co-investigator and co-author of this material.

# CHAPTER 6

# PALMS' EXPERIENCE WITH PDD

Policy Driven Development has proven effective in improving the evolvability of PALMS relative to both its early versions and established policy execution systems. PDD itself is not specific to PALMS, and can be applied to a broad spectrum of scenarios where simultaneously satisfying evolving requirements of multiple stakeholders quickly is crucial (e.g., OOI-CI [191], CitiSense [192], and CYCORE [193]).

PDD can be evaluated along dimensions that include:

- execution speed
- contributions to evolvability
    - development time
    - stakeholder ease of use
    - deployment time
- policy expressability
- scalability
- security

The execution speed of policy evaluation determines the hardware requirements and network topology needed to support PDD – or, alternately, the workflows on which policy can be economically evaluated on a single processor. To the extent policy evaluation takes *any* time, the pool of workflows that are candidates for policy injection is diminished. However, the execution cost of policy injection must be weighed against the costs and risks of the traditional regimen of application redeployment (as exemplified in the SOARS motivating example from Chapter 2), which are multidimensional, but include stakeholder dissatisfaction (as their requirements go unmet until a redeployment occurs). Application redeployment costs and risks and the cost of stakeholder disaffection are difficult to quantify, and are not the focus of this dissertation. Consequently, my evaluation of execution speed focuses on where and why bottlenecks in PDD occur (as in Section 6.3). Briefly, it shows that in PALMS, delays caused by the PDD policy mechanisms per-se are minimal and likely imperceptible, but can become noticeable depending primarily on the size and complexity of the data set contained in a service interaction message, and secondarily on the number of policy evaluations performed on a service interaction. These observations (summarized in Section 6.3.7) do not invalidate the premises of PDD, but may lead to evolution of the implementation of PDD within PALMS as described in Section 7.4.5. Nevertheless, bottlenecks due to policy evaluation are in line with those in comparable policy evaluation systems, and I defer a discussion of those bottlenecks to Section 7.4.2.

Though a major premise of this dissertation pertains to evolvability improvements, evolvability metrics relating to policy injection and concern separation [32] are poorly developed. As a proxy, I choose to evaluate PDD-related evolvability improvements in terms of development time and stakeholder ease of use (as in Sections 6.1 and 6.2).

The overall chapter flow is shown in Figure 49.

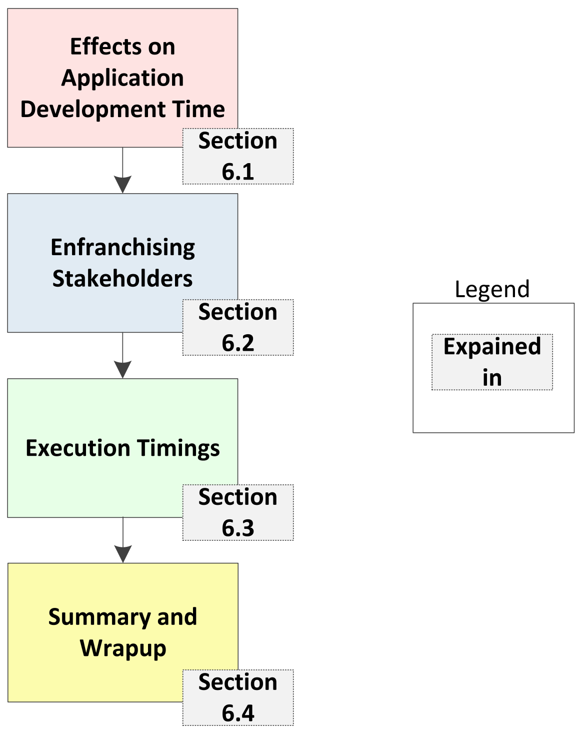


Figure 49. Chapter 6 Flow

The short deployment time of policies in the PALMS-CI (due to ease of policy authorship and quick physical deployment) compares favorably to times required to the long release cycles of traditional development techniques (even including agile techniques, which are defined by their comparatively short iterations). However, the larger view of deployment time includes testing and verification, which are often built into traditional techniques, but are not yet present in PDD or the PALMS-CI. Consequently, no meaningful comparison can be made in this regard. However, considering that PALMS' DSLs rest on the development of XQuery libraries, and base and composed workflows rely on SOA-related techniques, comparisons to between PDD (as PALMS) and traditional techniques can and have been made in Chapter 7.

An important evaluation would be the likelihood of delivering an application that fulfills stakeholder requirements based on the ease of creating, validating, and deploying a policy using one of PALMS' DSL-based policy languages versus using existing policy languages or versus inline coding. Such comparisons are far ranging and require more experience in exposing workflows to policy programmers and creating DSLs for their use – this is beyond the scope of this dissertation. However, a direct comparison between a PALMS DSL and policy expressed under a leading policy evaluation system is appropriate, and is presented in Section 7.4.1.

The non-functional requirements of scalability and security can be critical to real-world deployments. To the extent that PDD is designed to be scalable, it inherits the scalability characteristics of the Rich Services architecture on which the base application is built. Similarly, the security characteristics of PDD are defined partially by the design of the base application. PDD was not designed to answer scalability or security criteria, per se, where other policy evaluation systems (e.g., PERMIS, per Section 7.4.1) were, and lessons can be taken from them. Consequently, I do not attempt to evaluate PDD or PALMS on these dimensions.

## 6.1 Development Time

Initial implementations of PALMS had no explicit policy subsystem. Consequently, access control decisions were coded directly into base workflows, which is a common access control paradigm in many applications. The key information needed to make an access control decision was the user credential (as described in Section 5.6.1.3). For early PALMS versions, the credential was passed as a parameter on all Web Services calls and then incorporated into all internal workflow messages so as to anticipate the possibility that any service might make an access control decision. PALMS messages are defined as Java classes, and so adding this information required changing all message classes to derive from a base class containing the credential. Additionally, it was also necessary to change all services to propagate the credential from inbound messages to outbound messages. To accomplish this, nearly 900 Java files were manually modified, requiring approximately 20 hours' work.

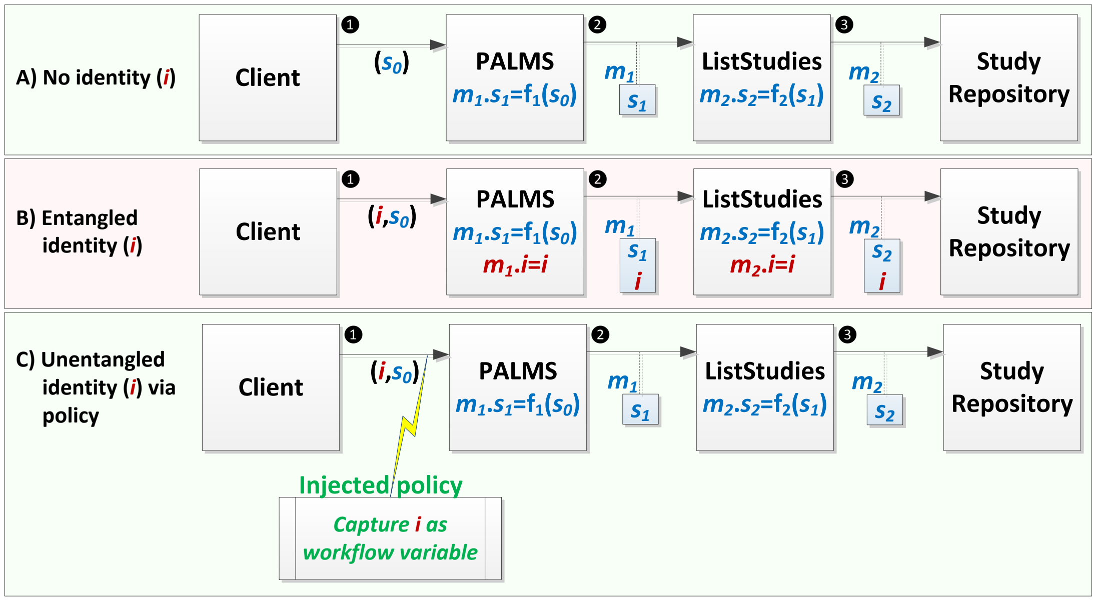


Figure 50. How Policy Use Avoids Entanglement

This situation is shown in Figure 50, which shows a subset of the GetStudyList workflow (from Chapter 3) where Java data flows are added to the diagram (in blue). Interaction ❶ shows the Client activity passing data $s_0$ as a Web Services parameter. The PALMS activity creates a message $m_1$ having value $s_1$ and uses it to interact with the ListStudies activity, which performs a similar sequence to interact with the StudyRepository activity. Workflow A is the original base workflow where identity credential $i$ is not passed. Workflow B represents the same workflow, but with the credential (in red) passed in ❶, then manually added to messages $m1$ and $m2$, and then manually propagated in the PALMS and ListStudies activities. The manual operations are costly in terms of time, and increase entanglement and scattering in the workflow code. Workflow C is described below.

Java type checking was *insufficient* to guarantee that such changes were complete and correct, as correctness relied *both* on subclassing *and* on the manual credential propagation code in each service. Completeness and correctness were verified by extending a pre-existing suite of external unit tests.

Subsequently, adding service tracking information (as described in Section 5.5.4) to each message required changing only the message base class, but still required manual modifications to all services to calculate and propagate the source and target service names across the workflow. Again, approximately 20 hours of work was required to manually update nearly 450 Java files, and completeness and correctness were verified via external unit tests. (Under Java, other implementation choices exist, but choice leads to other, equivalent, forms of entanglement and scattering.)

While these are classic refactoring exercises, their cost speaks to the need for separate context maintenance for separate concerns – in this case, policy evaluation, which depends on workflow-based contexts. This is shown in Workflow C, which shows how a credential is passed in ❶, then captured as a workflow variable, thus eliminating the need for manual message and activity changes downstream. A policy that makes use of the credential can fetch it using the context system (not shown).

Using the refactoring method, the programming exercise involved repetitions of the modify-compile-test cycle, followed by redeploying the application. Using the PDD policy method, the programming exercise involved repetitions of writing the capture policy, injecting it, and testing any $< Client, PALMS >$ interaction. Since the policy method is simpler and involves far fewer modifications, it is much faster and is reliable to the extent it does not modify the control or data flow. (See Section 7.5 for a discussion of PDD verification.)

Cost minimization is an important evolvability requirement, and evolution involves adding unanticipated concerns, which access control represented to the early PALMS design. The PDD context abstraction was added to address separation of context for separate concerns. Once the context system was implemented, re-implementing the credential and service tracking propagation took only an hour, and adding the custom Policy Context (described in Section 5.5.4.3) took another hour.

Philosophically, this approach leverages classic Aspect Oriented Programming (AOP) techniques [32] in separating the credential and service tracking concerns. However, PDD's service-oriented state maintenance and propagation techniques (as described in Section 5.5.4) – particularly the combination of interservice messages and a ubiquitous workflow-aware context system – are unique to PDD, and account for these productivity gains.

By using these mechanisms to achieve separation of concerns, PDD encourages practices that result in a clear time improvement over Java-based refactoring. Furthermore, by avoiding these concern entanglements, PDD encourages the decentralized implementation of such concerns (i.e., as small sets of composed workflows), thereby promoting reliable code evolution and encouraging proofs of correctness at both the base and composed workflow levels. Finally, because separate concerns are composed dynamically at runtime, large scale and time consuming release cycles can be reduced or avoided.

## 6.2 Stakeholder Ease of Use

As a test of the authorability of various types of policies, stakeholders (including both exposure biologists and programmers) were asked to author or maintain control policies using a general policy format, described as the *first format* in Section 5.6.2.2.1. Many stakeholders complained that this policy meta-language (including interaction references, XQuery expressions, and the tag file format in general) requires them to specify more information than is necessary to define or maintain a policy, and that because the comprehension and creation of new policies is overly complex, their use of the policy system would be rare. This lead to the insight that when a policy injected onto a particular interaction can be parameterized for use in different domains (e.g., the PALMS, study group, and or study domains differentiated in Section 5.6.2.2), a meta-language that requires the stakeholder to specify only a template and parameters reduces cognitive load on policy authors while enabling more uniform policy definition.

As shown in Figure 51, a single template can reference a particular interaction, and can combine with parameters provided by a template reference to form a complete policy. A single template can be reused by PALMS, Study Group, and Study policies composed onto the interaction.

For example, Section 5.6.2.2.1 initially describes the `Get a study list (control)` policy in a policy meta-language consisting of a triple containing a free-form XQuery expression:

```
GetStudyListPolicy.location = Get Study List (Repository)
GetStudyListPolicy.capability = Get a study list (control)
GetStudyListPolicy.controlExpression = \
  if (palms:subject-in-study-role('PI') then () \
  else palms:control-error('Invalid role for this operation')
```

It defines a simpler meta-language (i.e., the *third form*) that recasts the triple as a template (`GetStudyListTemplate`) referenced by a simpler policy:

```
GetStudyListPolicy.templateReference = GetStudyListTemplate
GetStudyListPolicy.templateParameter.RoleList = PI
```

Consequently, reliance on templates reduced the skillset (e.g., XQuery programming) required to create policy variants for different study groups and studies, thereby enabling stakeholders trained as programmers but untrained as policy authors to confidently write and insert such policies. They wrote and deployed twenty template-based access control policies within two hours. With the templates defined, stakeholders were able to experiment with different permissions (expressed as RoleList parameters in the example above) fluidly.

Stakeholder reactions to other aspects of PDD shed further light on how PDD can enfranchise stakeholders in the requirement enactment process; they are discussed further in Section 1.1.

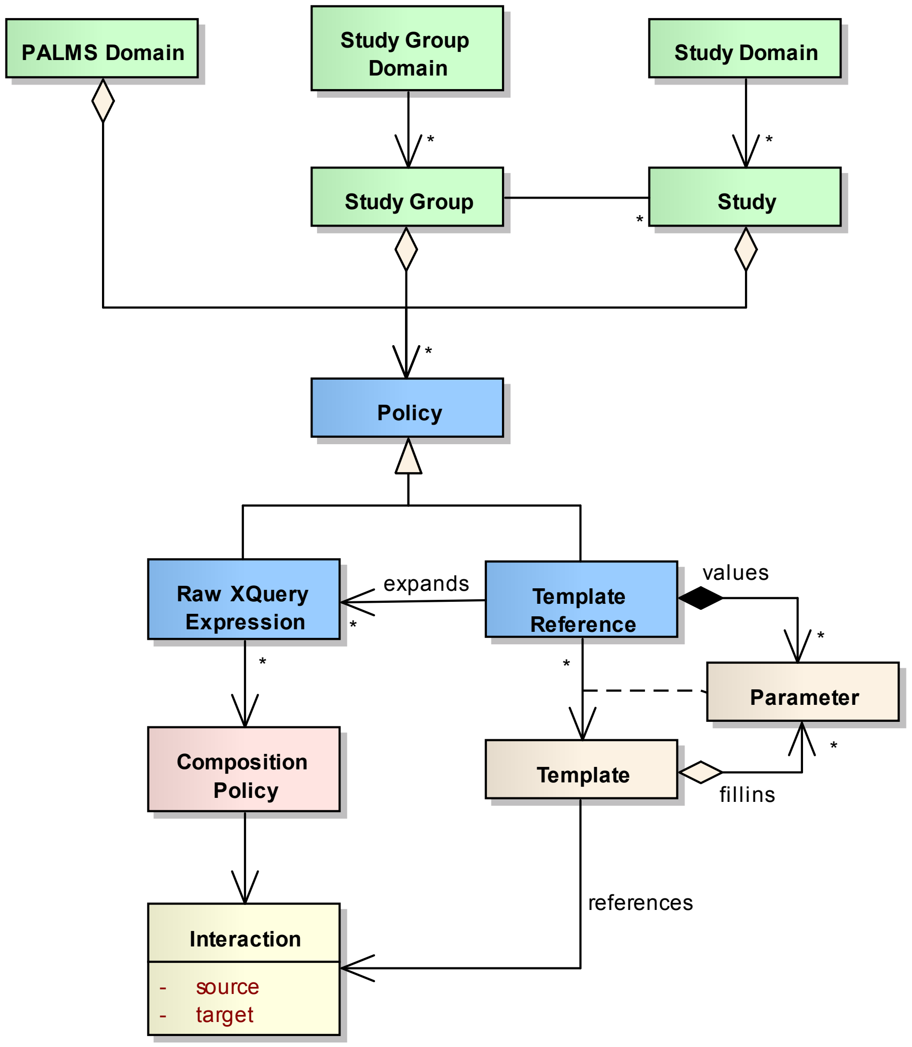


Figure 51. PALMS Policy Template Expansion

While the use of XQuery expressions presents an opportunity to align policies with stakeholder requirements quickly and easily (as described in Section 5.6.5), the policy meta-language itself is a DSL addressing the concerns of policy authorship, and the bifurcation of policy definition under a template-

and-reference model aligns PALMS' requirements for policy creation with the stakeholders' concept of policy authorship.

## 6.3 Execution time

As described in Section 5.3, the PALMS system comprises a browser-based GUI accessing the PALMS server through Web Services-based API calls. My evaluation of PDD focuses on the PALMS server, where policy injection applies, though with an eye toward the overall user experience, which derives from the combination of server execution time, Web Services marshaling and transmission time, and the speed of the GUI itself.

The objective of the execution time tests was to understand the contribution of policy execution to overall workflow execution times. I hypothesized that:

1. The contribution of the Policy Evaluation RIS to execution time is negligible when:
    a. an interaction *is not* contained in the Interaction Repository
    b. an interaction *is* contained in the Interaction Repository, but no policy is defined
2. The contribution of policy evaluation is small compared to the end-to-end workflow when:
    a. a control composition and control policy are defined
    b. a filter composition and filter policy are defined

Figure 6 shows the GetStudyList workflow, which is a common PALMS workflow that includes five activities: a browser (Client), a PALMS API entry point (PALMS), two stages of processing (ListStudies and StudyRepository), and a storage (Storage) activity. For calibration and context, I measured baseline end-to-end times through all activities, and for policy measurements, I measured the time for the interaction on which policy was injected. In order to evaluate my hypotheses, I created tests that started with simple scenarios and progressed in complexity:

- End-to-end measurement of trivial GetStudyList workflow with:
    - no Context interceptors and no Policy Evaluation
    - Context interceptors, but no Policy Evaluation
- Single service interaction with:
    - no Policy Evaluation
    - empty Interaction Repository (1a)
    - single interaction, with no policy defined (1b)
    - control composition policy with:
        - control policy (2a)
    - filter composition policy with:
        - null filter policy injected (2b)
    - filter-by-role() policy with:
        - no studies in list (2b)
        - one study in list, no studies filtered out (2b)
        - 97 studies in list, all studies filtered out (2b)
        - 97 studies in list, no studies filtered out (2b)

Each test is useful in evaluating the hypotheses, and tests that bear on particular hypotheses are identified as to the hypothesis they address.

The policy tests represent policies and scenarios that I expect to be typical in normal PALMS execution, and which, in fact, are typical at this time.

### 6.3.1 Test Platform and Circumstances

As shown in Figure 24, the PALMS server ran on a 64 bit VMware virtual machine under Tomcat 6.0.20 on Red Hat Centos 5.4 under vSphere 4.1 with 1.5GB non-shared RAM, a single core hyperthreaded processor, and a 1Gbps network connection. PALMS is written in Java, and ran on JVM v1.7.0_04-b20. While the physical server supports several other VMs, all were inactive during our tests, and the datacenter network and wide area network (serving the browser GUI) were unloaded.

Even with an unloaded physical server, virtual machine, datacenter network, and wide area network, variability is introduced into execution and transmission times by occasional service execution, incidental network traffic, and garbage collection. To mitigate these effects, I ran large numbers of tests and generally reported the median time – the average time was often skewed by a very small number of outliers. When a mode could be calculated, it was generally very close to the median, thereby validating this approach. Additionally, when showing test results on graphs, I remove outliers, and for easier understanding, I sort trials by their execution times and rename them using contiguous trial numbers (wherein I assume that each trial is an independent test).

In general, for each test, a test pass was executed before a second test pass, which was measured. This guaranteed that the Mule ESB had loaded all pertinent POJOs, Java had loaded all pertinent classes, the database had loaded and cached all pertinent data, and all other caches were primed.

Measuring execution taking place entirely on a VM risked inaccuracies originating with VMware's strategy of clock tick simulation as an average of ticks over real time [194]. To mitigate this risk, I measured only loops that iterated over code under test, where each loop ran several seconds. This afforded VMware the opportunity to accurately simulate clock ticks on average, and to obviate any small deficiency in the simulation at the end of the measured interval.

In any case, the objective of the execution time tests was to understand the contribution of policy execution to overall workflow execution times. Because test execution times were significant and had substantial differences compared to baseline times, sophisticated statistical analysis was unnecessary.

For tests involving a single service interaction, the $< PALMS, ListStudies >$ interaction was used, and measurements reflect times for a request/reply round trip. The $< PALMS, ListStudies >$ interaction is representative of all other PALMS service interactions. For the purpose of these tests, the ListStudies service was altered to return a constant result – it did not interact with other services to return the result. In all testing, light weight intermediary components (e.g., intermediate routers and Web Service stacks) and mocks (e.g., simulated database queries) were used to isolate the processing costs of interest, similar to the approach taken by [195].

### 6.3.2 End to End Tests

End-to-end tests were used to establish an overall context for interpreting the remaining tests. They measured the total execution time of the entire GetStudyList workflow, and differentiated client, network, and PALMS server contributions. No PALMS studies were defined, so all workflow services performed only minimal processing. Note that the client was a PC-based browser, but had only minimal code (as JUnit tests) and did not execute any GUI functions. As such, its execution time would be an order of magnitude less than a GUI browser client.

#### *6.3.2.1 End to End with No Interceptors*

In the first test, Mule was configured to exclude all interceptors (i.e., all RISs). As a result, no SIVs or interservice messages $IM$ (as described in Section 5.5.4) were created, used, or destroyed, and no service tracking or policy evaluation occurred. Only the PALMS SDC and individual services (comprising base workflows) were executed, and they all exchanged only raw interaction messages $m$.

Figure 52 shows the workflow execution time over 2,000 trials (sorted by App Time, with outliers removed) as follows:

- **App Time** (**63ms** median) indicates the number of milliseconds (measured at the PC browser) taken to send a GetStudyList request across the network, have it evaluated at the server (including any database query at the Storage service), and return a study list across the network. It included nominal browser processing, Web Services call overhead, message passing, and POJO invocation.
- **Latency** (**1.3ms** median) indicates the round trip time for a packet to travel between the PC and the PALMS VM. It was measured by WireShark as the time required to receive a TCP SYN/ACK reply to the TCP SYN packet used to establish a TCP/IP connection.
- **Server Time** (**7.2ms** median) indicates the time required by Mule to process the Web Services call and return (prior and after the PALMS service), and execute each GetStudyList workflow service. It was measured by Wireshark directly, and is reported including Latency.
- **In-App** (**55.2m**s median) indicates the time spent in the PC browser, calculated by subtracting Server Time (including Latency) from App Time.
- **In-Server** (**12%** median) indicates the ratio of Server Time and Latency to App Time, as a way of demonstrating the impact of PALMS server interactions on the overall responsiveness of the browser GUI.

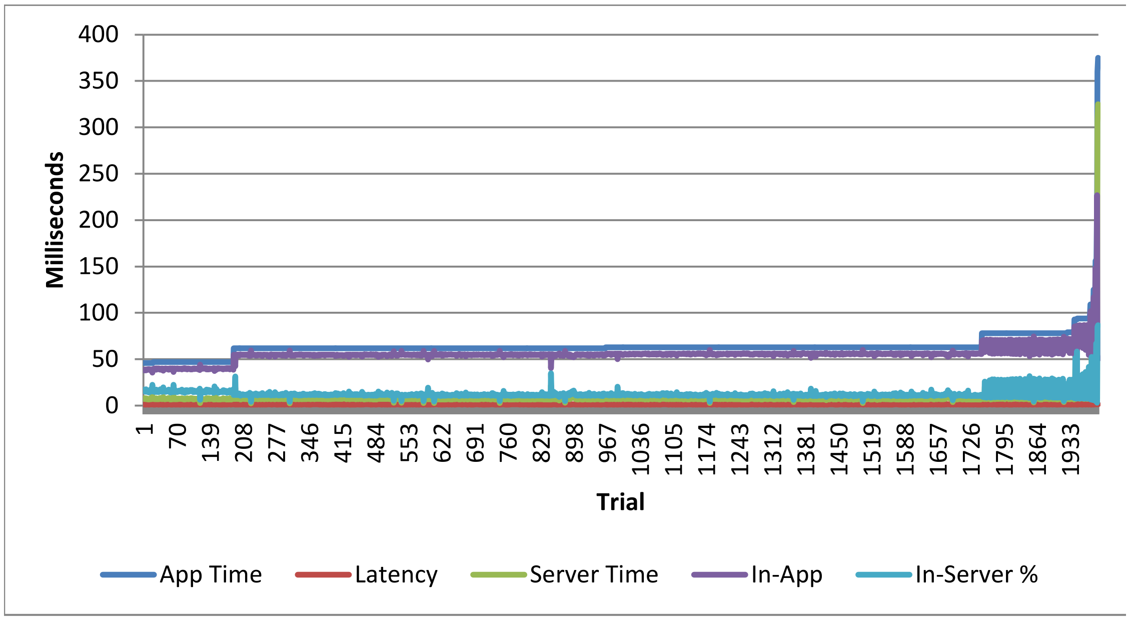


Figure 52. End-to-End Execution Time

Figure 53 shows the timeline for the baseline operations using median times, with emphasis on the time for the PALMS server portion of the GetStudyList workflow.

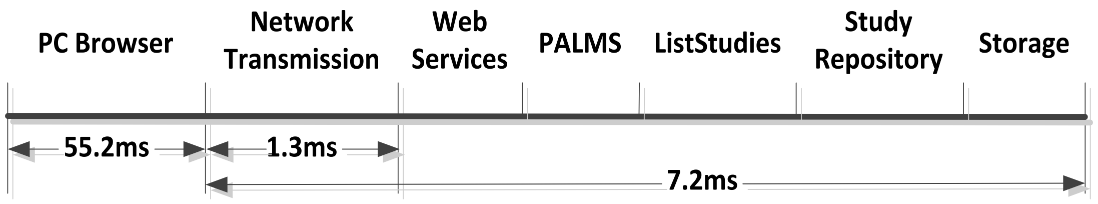


Figure 53. PALMS Execution Timeline without Context or Policy Evaluation

This test shows that the PALMS user experience is dominated by latencies within the PALMS Browser itself, and that the network latency and PALMS server execution combine to account for only a small portion of overall execution time. Considering that the GUI version of the PC browser is much slower than the PC-based JUnit test actually executed, the server contribution to a running PALMS system is tiny, as shown in Figure 54 (sorted by Server Time). Alternately, the PALMS server can be expected to service numerous PALMS clients simultaneously without contributing to user-perceived delay, and so minimizing server-side execution (including policy evaluation) is important.

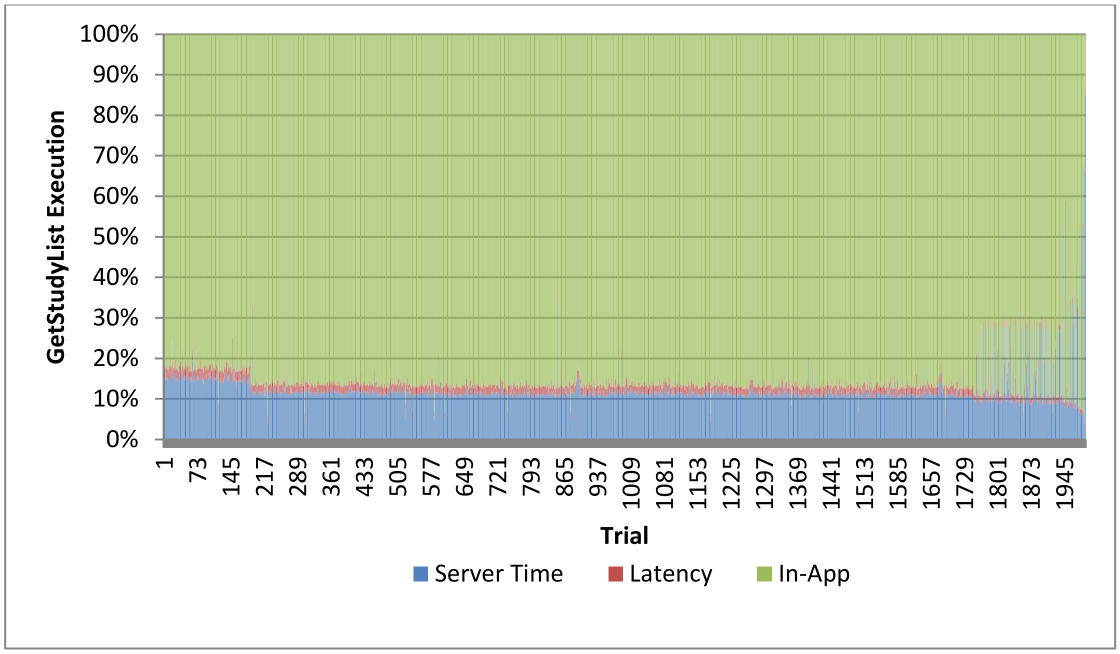


Figure 54. PALMS Server Contribution to Overall Execution

### *6.3.2.2 End to End with Context Interceptors*

In the second series of tests, Mule was configured to include all interceptors except the Policy Evaluator. As a result, SIVs were created, used, and destroyed, and interservice messages $IM$ were exchanged between services (as described in Section 5.5.4). In this configuration, PALMS operated with the entire policy system (including service tracking on each interaction) except for Policy Evaluation itself.

The first test measured the time taken to send a message from the Browser PC to the PALMS service and retrieve a reply, both with and without interceptors that create and destroy SIVs before and after the PALMS-Storage workflow executes (i.e., the `PrePolicyInterceptors`, less `PolicyEvalInterceptor`). The measurement was timed using WireShark, and includes network transmission time, the time to process the Web Services call and return, and the time to start a mock PALMS service.

Figure 55 shows the round trip time over 2,000 trials (sorted, with outliers removed) as follows:

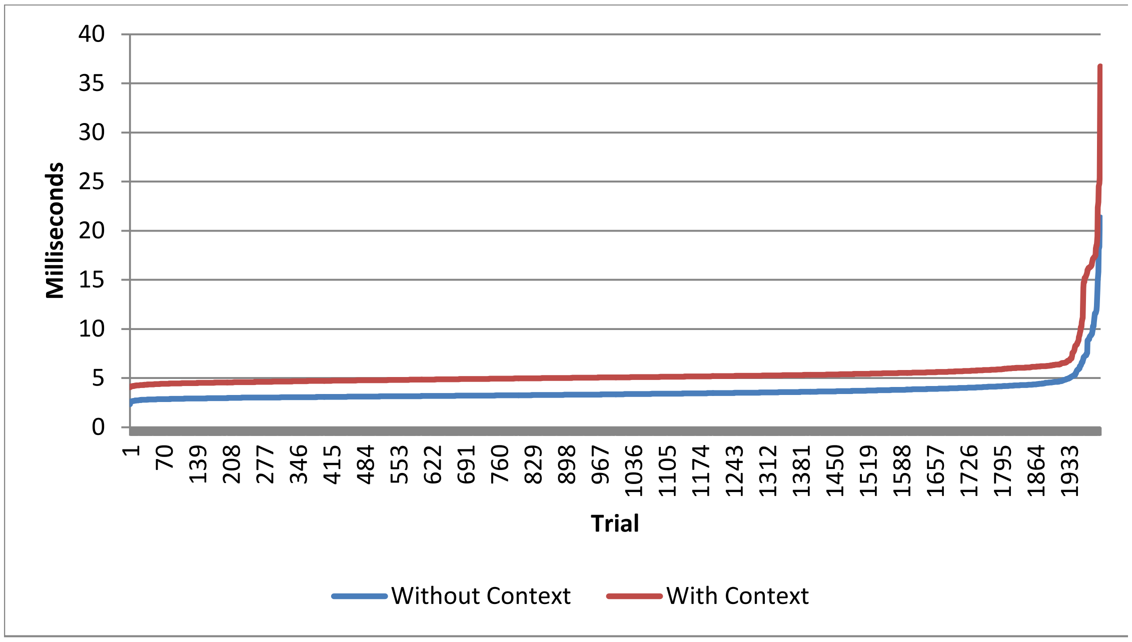


Figure 55. Network and Web Services Times with and without Context Interceptors

The second test measured the time added by interceptors that maintain service tracking and the abstraction of interservice messages $IM$ within the $< PALMS, Storage >$ workflow (i.e., the `PolicyInterceptors`, less

`PolicyEvalInterceptor`). It executed the ListStudies, StudyRepository, and Storage services, including the $< PALMS, ListStudies >$, $< LisStudies, StudyRepository >$, and $< StudyRepository, Storage >$ interactions. To address possible jitter issues with the VMware-simulated tick counter (as described in Section 6.3.1), each trial timed 2,000 executions of the workflow and reported the time divided by 2,000.

Figure 56 shows the round trip time over 50 trials (sorted) as follows:

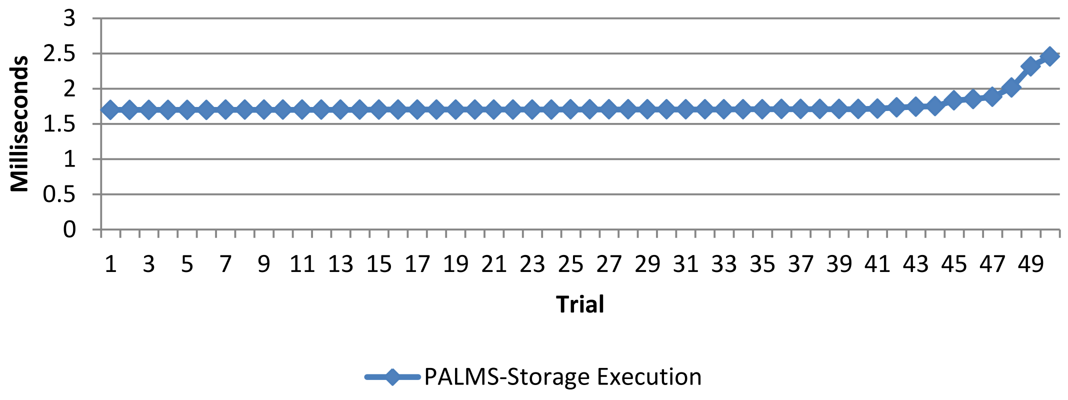


Figure 56. PALMS-Storage Execution with Context Interceptors

Figure 57 shows the timeline for round trips using median times. Executing the Context interceptors adds 1.72ms to the total GetStudyList workflow execution before the PALMS Mock, and 1.7ms after. Note that Figure 53 shows an end-to-end round trip time of 7.2ms, which includes the actual PALMS service and no context interceptors. Figure 57 shows a slightly different scenario, which takes 6.77ms and includes the PALMS Mock and context interceptors. The difference between the PALMS and PALMS Mock execution times accounts for this discrepancy. The main point, though, is to establish approximate baseline execution times for comparison against Policy Evaluation scenarios later in this section, and to show that the time overhead contributed by the Context interceptors is tiny.

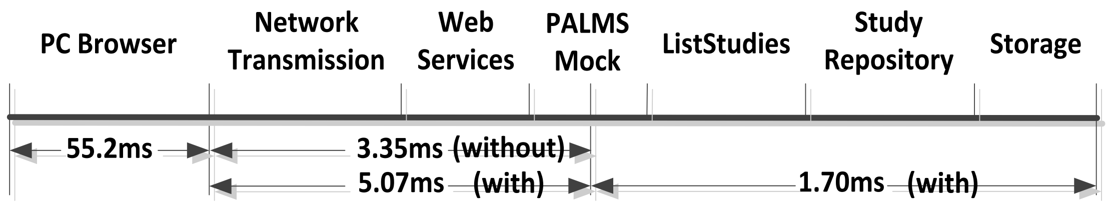


Figure 57. PALMS Execution Timeline with Context Interceptors

The two tests together provide a measurement of the overhead of the context system needed to support Policy Evaluation, without actually invoking Policy Evaluation functions. End-to-end, the cost is less than 2ms per round trip.

### 6.3.3 Single Interaction Baseline

As a baseline for evaluating the cost of policy evaluation for a single service interaction, I measured three scenarios using the $< PALMS, ListStudies >$ interaction:

- a logging interceptor (not part of normal PALMS)
- Context interceptors that implement the interservice messages $IM$ abstraction
- Policy interceptor that performs Policy Evaluation

Note that each scenario adds to the previous scenario. The logging interceptor makes a simple log4j call to log an interaction to a disk file. The Context interceptors are those measured in Section 6.3.2.2. The policy interceptor was executed with no interaction defined in the Interaction

Repository for the $< PALMS, ListStudies >$ interaction, so execution proceeds at the ListStudies service – this is the shortest path through the Policy Evaluator.

As in Section 6.3.2.2, for each scenario, I ran 50 trials of 2,000 interaction iterations each, and times are listed as the time for 2,000 iterations divided by 2,000. Figure 58 shows the round trip time for each scenario (sorted) as follows:

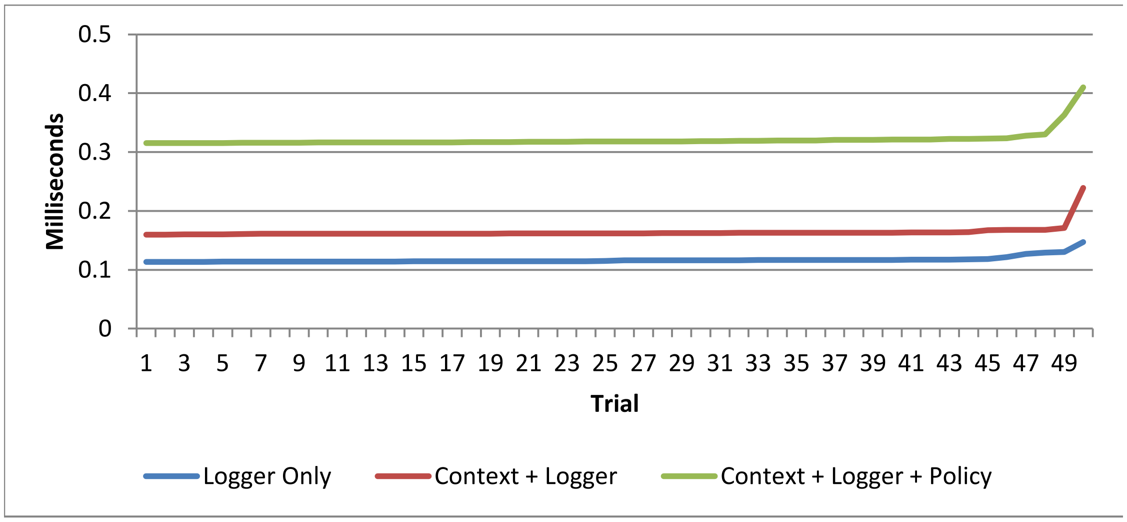


Figure 58. PALMS Interaction Baseline Times

Figure 59 shows the timeline for round trips using median times. Executing a round trip interaction intercepted by only the logger required 0.116ms. Adding the Context interceptors increased the interaction to 0.162ms, and adding the Policy Evaluator increased the interaction to 0.318ms.

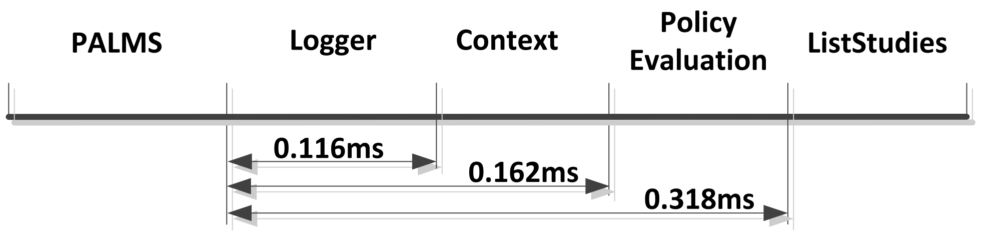


Figure 59. PALMS Execution Timeline with Context Interceptors

The incremental cost of the Context interceptors is 0.046ms, and the incremental cost of the Policy Evaluator interceptor is 0.156ms. The overall cost of the policy system (including the Context interceptors and a null Policy Evaluation) is 0.202ms per interaction.

### 6.3.4 Null Policy Baseline

As a baseline for evaluating the cost of different kinds of policies and data configurations, I measured two scenarios using the $< PALMS, ListStudies >$ interaction:

- an interaction for PALMS-ListStudies defined in the Interaction Repository
- a simple control composition policy defined on the interaction

Note that each scenario adds to the previous scenario. In addition to processing the logging, Context, and Policy Evaluator interceptors, the first scenario defines an Interaction Repository entry for the $< PALMS, ListStudies >$ interaction, which causes the Policy Evaluator to query the Policy Repository for policies defined on the interaction. In this scenario, there are none, so the Policy Evaluator exits and execution continues with the ListStudies service.

In the second scenario, a simple control composition policy (`compose-all()`) is defined, which causes the Policy Evaluator to check for control policies defined on the interaction. In this scenario, there are none, so the Policy Evaluator exits and execution continue with the ListStudies service.

As in Section 6.3.2.2, for each scenario, I ran 50 trials of 2,000 interaction iterations each, and times are listed as the time for 2,000 iterations divided by 2,000. Figure 60 shows the round trip time for each scenario (sorted) as follows:

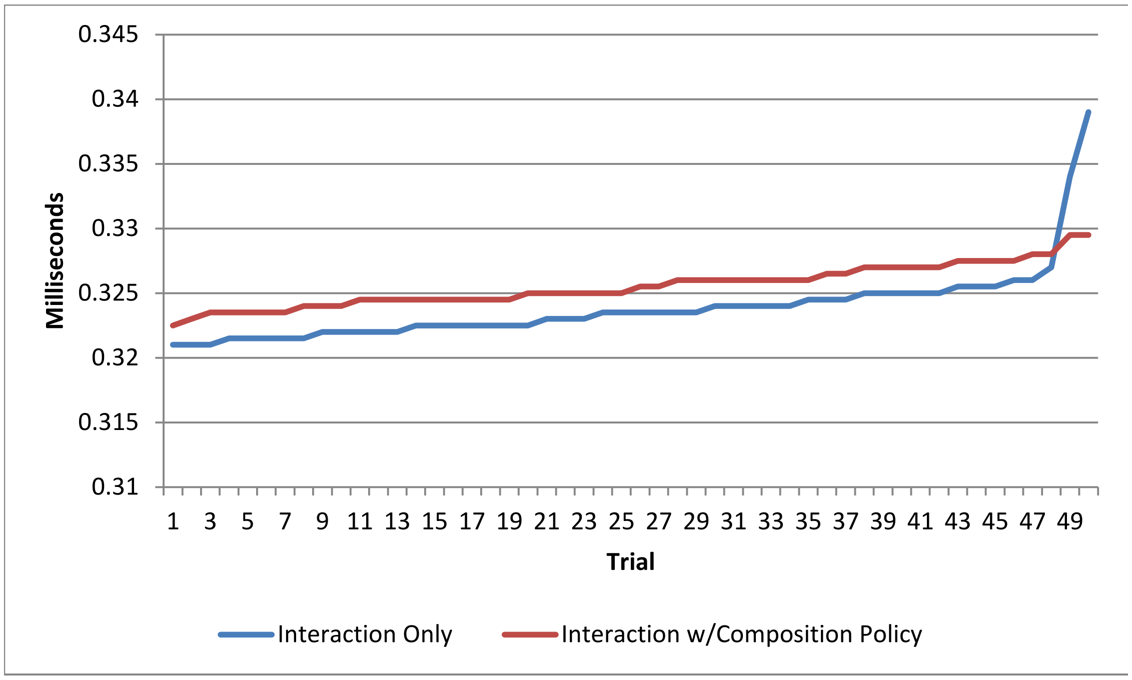


Figure 60. PALMS Policy Evaluation with No Policy

Figure 61 shows the timeline for round trips using median times. Executing a round trip interaction with an Interception Repository entry for the interaction required 0.323ms, and executing a round trip with a control composition policy but no control policy required 0.325ms.

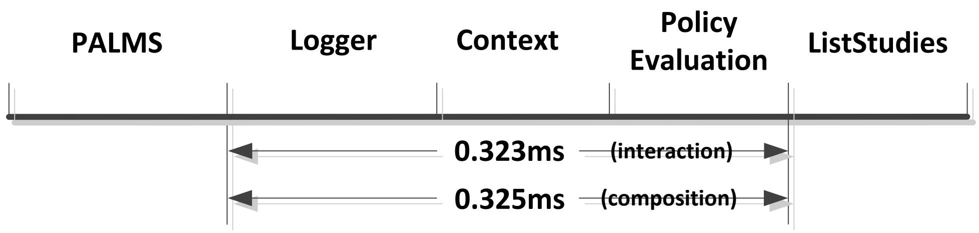


Figure 61. PALMS Execution Timeline with Interaction but No Policy

Given that the Context and Policy Evaluation interceptors are applied to each PALMS interaction, the cost for the Policy Evaluator to process an interaction in the Interaction Repository is 0.005ms, and the additional cost of processing a composition policy having no injected policies is 0.002ms.

### 6.3.5 Control Policy

Defining a control policy on an interaction results in the Policy Evaluator inserting the control policy into a control composition policy, then executing the control composition policy to determine a replacement service (if any), per Section 5.6.2.2.4. The process for preparing a message $m$ for evaluation, then evaluating the policy is described in Section 5.6.3.2.

I measured four scenarios using control policies composed onto the PALMS-ListStudies interaction:

- 1 control policy
- 2 control policies
- 5 control policies
- 10 control policies

The time for single policy evaluation demonstrates the cost of injecting the control policy into a composition policy, compiling the composition policy, preparing the current message $\mathrm{m}$ for evaluation, evaluating the composition policy expression, and interpreting the result. Note that whether a control policy is originally expressed as a raw XQuery expression or a template reference (as described in Section 5.5.1), the Policy Repository translates all control policies to raw XQuery expressions as the policies are loaded (as described in Section 6.2). Consequently, this translation does not contribute to times for policy evaluation during service interactions.

The time for multiple policy evaluation captures all of the costs for single policy evaluation, and includes the time needed to insert each policy into the composition policy, to compile the composition policy, and for the composition policy to evaluate each (as occurs with the `compose-unanimous()` composition policy described in Section 5.6.5). Preparing the current message $\mathrm{m}$ occurs only once for the entire composition policy. Injecting multiple control policies reflects the use case of multiple policies being injected by multiple stakeholders, possibly in different domains, and possibly oblivious to each other. In this test, the same control policy was injected, though in a realistic scenario, different policies would be injected.

Similar to Section 6.3.2.2, for each scenario, I ran 10 trials of 200 interaction iterations each, and times are listed as the time for 200 iterations divided by 200. The control policy was similar to the policy described in Section 5.6.3.2, where the policy returned the default service. Figure 62 shows the round trip time for each scenario (sorted) as follows:

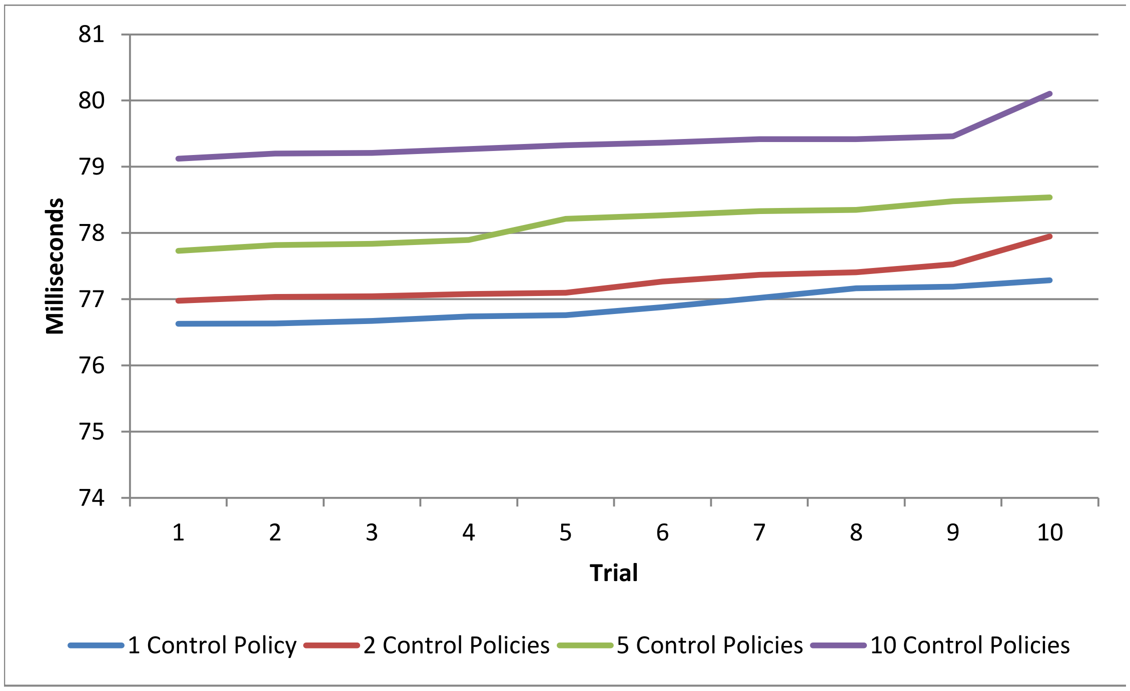


Figure 62. PALMS Control Policy Evaluation with No Policy

Figure 63 shows the timeline for round trips using median times. Executing a round trip interaction with 1, 2, 5, and 10 control policies required 76.82ms, 77.19ms, 78.24ms, and 79.35ms.

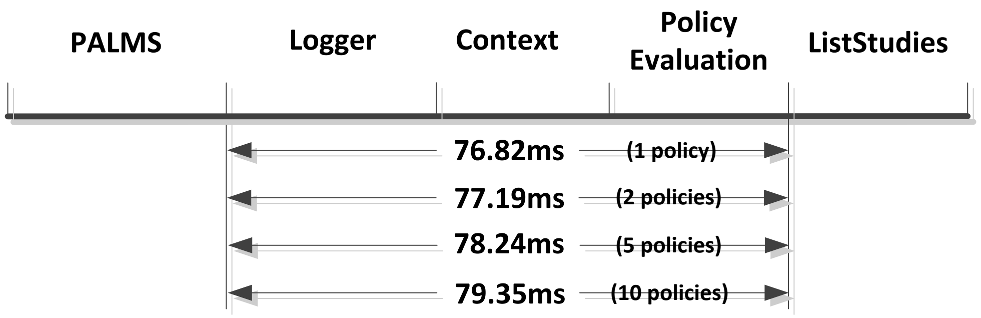


Figure 63. PALMS Execution Timeline with 1, 2, 5, and 10 Control Policies

Adding a single access control policy, the interaction time rose from the baseline 0.325ms (in Section 6.3.4) to 76.82ms. Adding and evaluating a second policy required an additional 0.365ms. The average additional cost for the fifth policy was 0.356ms, and the average additional cost for the tenth policy was 0.280ms. According to the definition of the `compose-unanimous()` composition policy, each copy of the policy would have been executed. Therefore, the additional cost accounts for both the additional compilation of the policy and its execution.

While the cost of an additional policy was very low, the cost of the first control policy was high, though still small relative to the time consumed in the PC browser.

### 6.3.6 Filter Policy

Defining a filter policy on an interaction results in the Policy Evaluator inserting the filter policy into a filter composition policy, then executing the filter composition policy to determine a filter service, execute it, and return a replacement message $m$ (if any), per Section 5.6.2.2.4. The process for preparing a message $m$ for evaluation, then evaluating the policy is described in Section 5.6.3.1.

I measured five scenarios using filter policies composed onto the reply phase of the $< ListStudies, StudyRepository >$ interaction – the filter policies are post-filters. Note that a different interaction is used for these scenarios as compared to the $< PALMS, ListStudies >$ examined in previous sections. The timing characteristics of this interaction are the same as for other interactions, but the message $m$ is appropriate for filtering, whereas the equivalent reply message on the $< PALMS, ListStudies >$ interaction has already been processed (by the ListStudies service) and would not be a good candidate for filtering.

In all scenarios, the filter operated on a list of studies whose cardinality was part of the scenario, and had an effect on the result. Each scenario built on the previous scenario, starting with a trivial filter on trivial data – it returned all data it is receives. The non-trivial filter was the `filter-by-role()` filter described in Section 5.6.2.2.2. In both cases, the composition filter policy was the `compose-all()` filter described in Section 5.6.5. The scenarios were:

- Passthru filter operating on a list of 0 studies
- Filter returns 0 of 0 studies
- Filter returns 1 of 1 study
- Filter returns 0 of 97 studies
- Filter returns 97 of 97 studies

The time for the passthru policy evaluation demonstrates the nominal cost of injecting the filter policy into a composition policy, preparing the current message $m$ for evaluation, evaluating the policy expression, and recasting the result as a new message $m$. Note that whether a filter policy is originally expressed as a raw XQuery expression or a template reference (as described in Section 5.5.1), the Policy Repository translates all filter policies to raw XQuery expressions as the policies are loaded (as described in Section 6.2). Consequently, this translation does not contribute to times for policy evaluation during service interactions.

Evaluation times for multiple filter policies are similar to those for multiple control policies, as the policy composition functions are similar.

However, evaluation times for a single filter policy depend on the number of elements in the input message, the complexity of the filter calculation, and the number of elements in the result message, as shown in Figure 64 and Figure 65. (Figure 65 shows all scenarios, and Figure 64 shows three scenarios that have lower execution times and are hard to differentiate in Figure 65.)

The "Passthru 0 of 0 studies" test establishes the baseline as a trivial input message, no filter calculation, and the same trivial message returned. The "Return 0 of 0 studies" test requires additional XQuery compilation time (for the non-trivial filter) and adds a loop over the (trivial) message. The "Return 1 of 1 studies" test adds message translation time for the single study entry on both the input and return messages, consistent with the evaluation sequence described in Section 5.6.3.1.

The "Return 0 of 97 studies" test accepts an input message listing 97 studies, and the filter criteria results in a return message reflecting rejection of all studies. The "Return 97 of 97 studies" test accepts the same message and returns a result message reflecting acceptance of all studies.

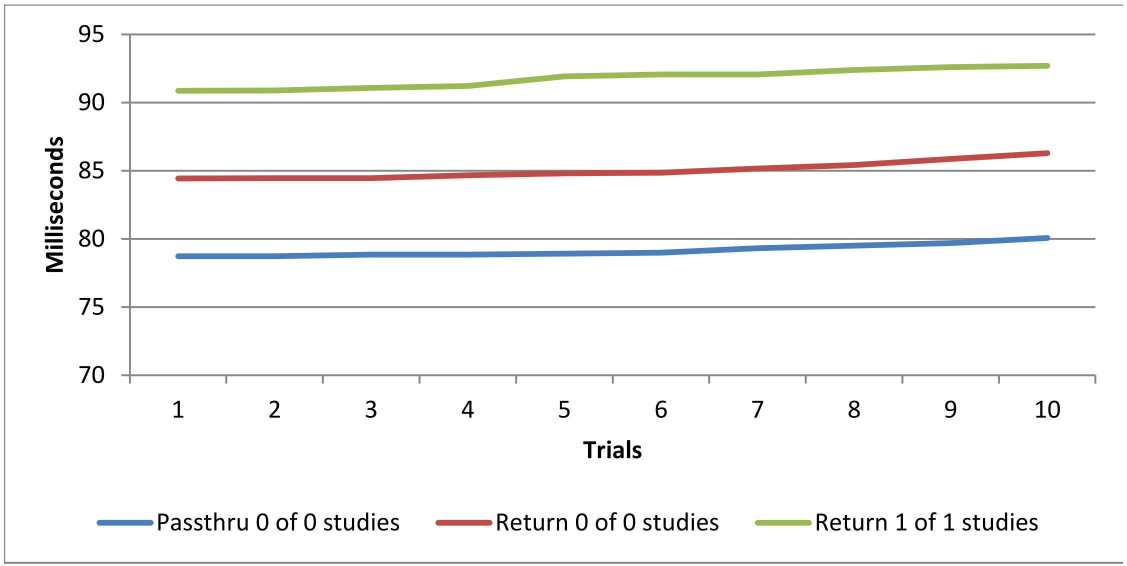


Figure 64. PALMS Filter Policy Evaluation with Few Elements

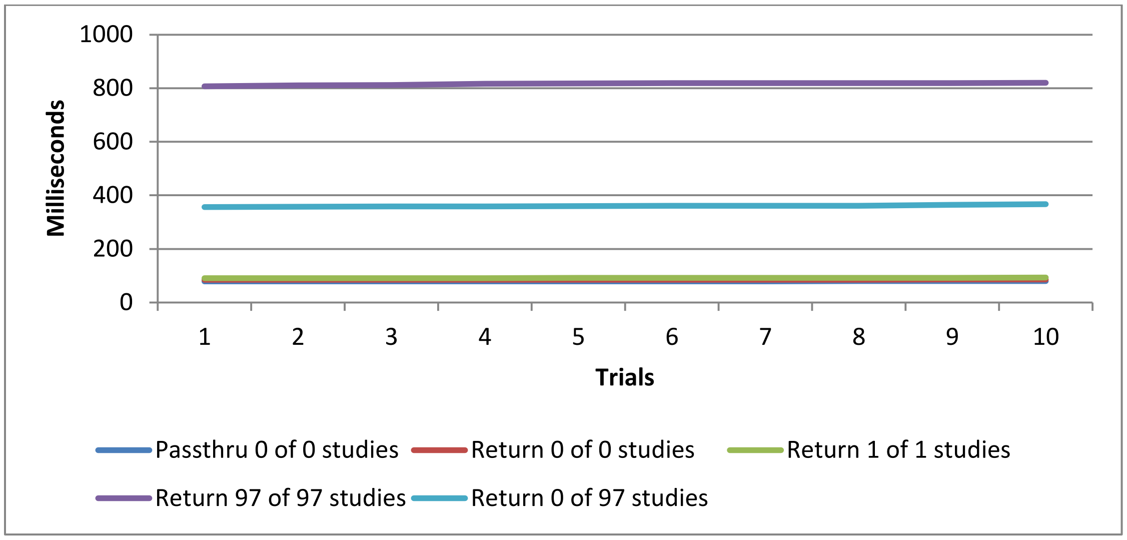


Figure 65. PALMS Filter Policy Evaluation with Many Elements

Figure 66 shows the timeline for round trips using median times.

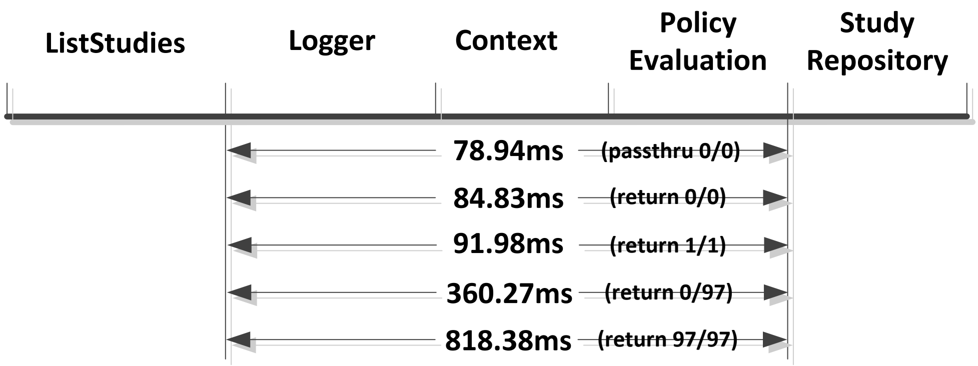


Figure 66. PALMS Execution Timeline with Filter Policies

There are three differences between the “Return 0 of 0 studies” and “Return 1 of 1 studies” scenarios (per Section 5.6.3.1):

- an input message must be translated from Java object to XML (via XStream) and then to Saxon native XDM format (via Saxon APIs)
- the filter must execute on the single study (which, for `filter-by-role()` means extracting the study group, using the user’s identity to discover a

role list relative to the study group, and then comparing the role list to the `filter-by-role()` parameter)

- a result message must be translated from Saxon native XDM format back to XML, and then back to a Java object

The additional time for both translations and the XQuery expression execution is 7.15ms.

Similar differences exist between the "Return 0 of 0 studies" and "Return 97 of 97 studies" scenarios, with the later requiring 7.56ms per study. This differs with the previous 7.15ms by 0.41ms or approximately 6%, for unknown reasons, possibly including coding efficiencies, non-linearities, and assumptions in either XStream, the Saxon API, or the Saxon XQuery engine.

Finally, the difference between the "Return 0 of 97 studies" and "Return 97 of 97 studies" scenarios is that for the former, the result message translation is trivial (because 0 studies are returned). By subtraction, the result message translation time is 4.72ms per study, and the combination of the input message translation and XQuery expression evaluation is 2.84ms per study.

Note that these timings depend heavily on the data being processed. For a study list (as discussed in Section 3.1), a study entry contains six elements, where three of the elements are XML and could be several kilobytes long. Messages containing a different number of fields, different field lengths, and structured fields may take more or less time to process by the XQuery engine and to render using XStream and Saxon APIs. However, the factors that determine relative execution times remain the same.

### 6.3.7 Execution Time Hypotheses

In evaluating the hypotheses posed in Section 6.3, I refer to the timings presented in this section to demonstrate that hypotheses 1.a and 1.b are supported, and hypotheses 2.a and 2.b are conditionally supported.

Section 6.3.2.1 establishes that the end-to-end execution time for a typical PALMS workflow (including a trivial PC browser application) is 63ms, approximately 55ms of which is in the PC browser application. (A practical PC browser application would take an order of magnitude longer.) Regardless, Section 6.3.2.2 demonstrates that the cost of the Context system at the start of a workflow is 1.72ms, and Section 6.3.3 demonstrates that the Context system interceptor support for each interaction costs 0.046ms, and that the cost of a Policy Evaluation for an interaction not registered in the Interaction Repository is 0.156ms.

With the cost of 2.01ms relative to a baseline workflow cost of 63ms, the Context system and Policy Evaluation system (absent any interaction or policy definitions) increases workflow execution costs by about 3%, which supports hypothesis 1.a.

Section 6.3.4 shows that interactions registered in the Interaction Repository and having a composition policy require 0.007ms longer to execute than interactions not registered in the Interaction Repository. Therefore, hypothesis 1.b is supported.

Section 6.3.5 shows that control policy evaluation adds between 76 and 80ms (or longer) to an interaction, depending on the size and complexity of the interaction message, the policies and their composition policy, and how many policies are evaluated. Relative to a baseline workflow time of 63ms, this is a significant cost, especially if a control policy is defined for more than one interaction in the workflow. However, a realistic PC browser likely requires an order of magnitude longer to execute, thereby rendering control policy execution time as insignificant. Notably, control policy execution times can become significant if they are embedded in loops or workflows must be executed on behalf of multiple users at the same time. Therefore, hypothesis 2.a is conditionally supported.

Section 6.3.6 shows that filter policy evaluation adds between 78 and 818ms (or longer) to an interaction, depending on the same factors as for control policies, with the size and complexity of the interaction message returned by the filter. Relative to the baseline workflow time of 63ms, this is a more significant cost, especially for large or complex interaction messages – pre-filters would likely incur costs at the low end of this range (because request messages $q$ are normally small and simple), and post-filters would likely incur costs at the high end of this range (because request messages $a$ can be large and complex, as in Section 6.3.6). Furthermore, as with control policies, these costs are magnified if the interaction is embedded within a loop or other workflow or must be executed on behalf of multiple users at the same time. However, in the case of PALMS, users infrequently execute post-filtered workflows and times allocable to the PALMS Browser GUI significantly exceed the cost of the post-filter execution. Therefore, hypothesis 2.b is conditionally supported, but is weaker than hypothesis 2.a.

## 6.4 Summary

In this chapter, I used the PALMS-CI case study to evaluate PDD along multiple dimensions, including development time, stakeholder enfranchisement, and execution time.

Through context composition mechanisms (in Section 6.1), I demonstrated a dramatic reduction in development time because PDD mechanisms enabled the separation of workflow-oriented concerns from base workflows (with commensurate decreases in maintainable code and complexity). Though the objective of PDD is to enable policy-based workflow composition, the reduction in workflow development time lends credence to PDD's context composition as an enabler of policy-based workflow composition.

In Section 6.2, I described the evaluation of policy language usage in constructing access control policies, and how the policy language syntax was evolved to satisfy stakeholder-perceived ease-of-use criteria. As a result, untrained stakeholders were able to write and inject simple access control policies. As a follow up, I discuss policy verification issues in Section 7.5.

Section 6.3 presents a number of tests that describe how policy execution times affect the overall execution of workflows, particularly how they vary with policy and interaction message size and complexity. The tests showed that for interactions on which no policies are associated, the time added to workflow execution is negligible. For control policy execution, the incremental cost is non-trivial but not significant so long as interaction messages remain small – a likely scenario, given that control policies are generally executed on request messages in a request/reply interaction. For filter policy execution, the incremental cost is non-trivial and can be significant for large and complex interaction messages. Using the same reasoning as for control policies, pre-filter policy execution is not likely to be significant. However, it was demonstrated to be significant for post-filter policies processing non-trivial interaction messages.

In Chapter 7, I compare PDD to systems and approaches that address a number of aspects of PDD, resulting in a wider perspective of how PDD succeeds and how it can evolve. Particularly, in Section 7.4.2, I describe how policy execution costs impact workflow execution in the xESB system, and offer contrasts and comparisons that put policy execution times into perspective. In Section 7.4.5, I discuss how changing the relationship between policy language, policy compilation, and interaction message structure can be altered to influence these costs.

# CHAPTER 7

# PDD AS COMPARED TO OTHER APPROACHES

In the preceding three chapters, I described a foundation of policy composition (as PDD in Chapter 4), its implementation in the PALMS-CI (Chapter 5), and important measures of its performance (Chapter 6). In this chapter, I summarize the approach and contributions of PDD, describe how PDD's approach fills gaps exposed in the existing solutions identified in Chapter 2, and compare and contrast PDD with other existing contributions that bear on issues either solved or unsolved under PDD. I round out the discussion by explaining my vision for a new role of policy programmer, describing how to use PDD to build large scale systems responsive to emergent stakeholder requirements, and exposing gaps remaining to be filled within PDD.

In this Chapter 2, I examined existing methodologies and technologies as contributions tuned to support *early* requirement binding but which may apply to a *late* binding paradigm, and described how each approach could contribute to late binding but somehow fell short. In the individual sections of this chapter, I revisit the PDD foundations and case study implementation as solutions to those shortcomings (as summarized in Table 9) and shown in Figure 67.

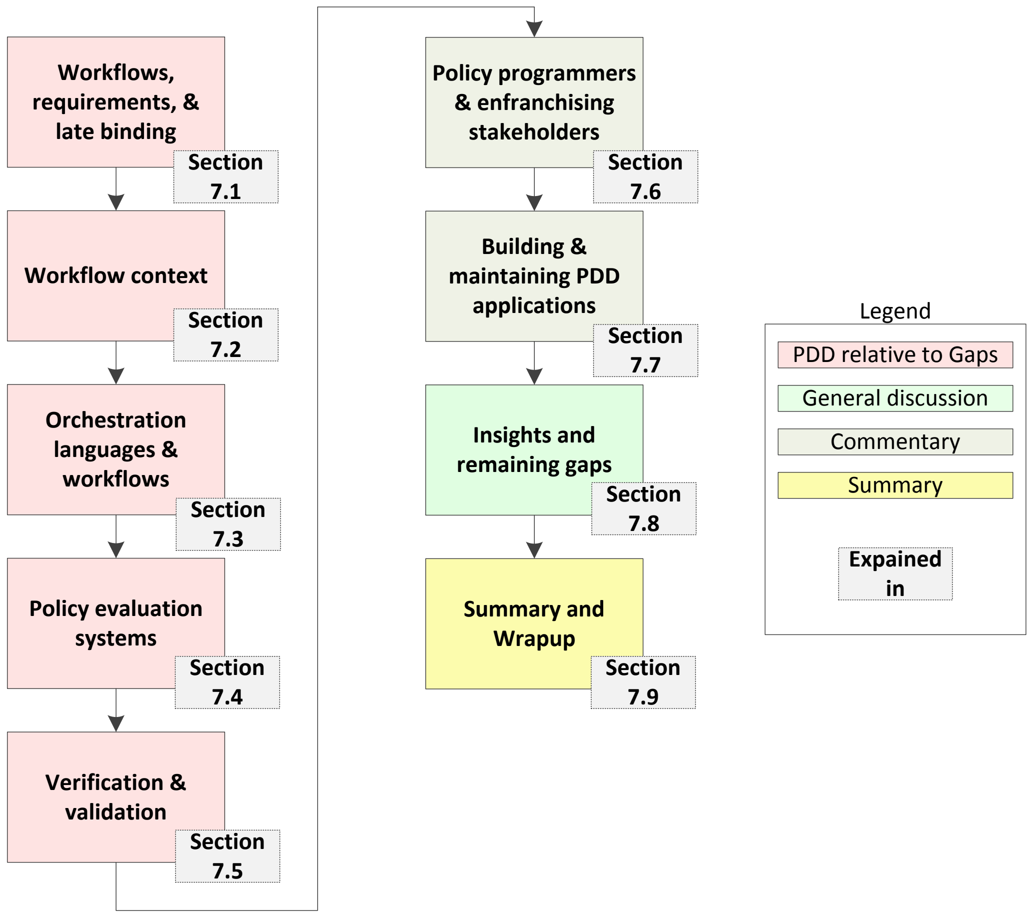
Table 9. Existing Contributions Addressed in PDD

| Topic | Discussion of Existing Work | Comparison to PDD |
|---|---|---|
| **Injecting Crosscutting Concerns** | 2.3.3<br>2.3.4 | 7.1 |
| **Workflow Context Management** | 2.4.5 | 7.2 |
| **Orchestration and Workflows** | 2.4.4 | 7.3 |
| **Policy Evaluation Systems** | 2.4.4 | 7.4.1<br>7.4.2<br>7.4.3 |
| **Policy Deployment** | 2.4.4.1 | 7.4.4 |
| **Policy Languages** | 2.4.4.1<br>2.4.4.4<br>2.4.6 | 7.4.5 |
| **Policy Versioning** | 2.4.4.1<br>2.4.4.2<br>2.4.4.3 | 7.4.6 |
| **Verification** | 2.4.6 | 7.5 |

In Section 7.6, I describe my work in developing the role of policy programmer (introduced in Section 1.3.3) and how it relates to traditional application development.

In Section 7.7, I explain processes and techniques useful in building or maintaining an application created under PDD principles.

Finally, in Section 7.8, I discuss both issues addressed and unaddressed by PDD, opportunities uncovered in the course of my PDD research, and insights useful in moving PDD forward.



Workflows, requirements, & late binding
Section 7.1
Workflow context
Section 7.2
Orchestration languages & workflows
Section 7.3
Policy evaluation systems
Section 7.4
Verification & validation
Section 7.5
Policy programmers & enfranchising stakeholders
Section 7.6
Building & maintaining PDD applications
Section 7.7
Insights and remaining gaps
Section 7.8
Summary and Wrapup
Section 7.9
Legend
PDD relative to Gaps
General discussion
Commentary
Summary
Expained in

Figure 67. Chapter 7 Flow

## 7.1 Workflows, Requirements, and Late Binding

PDD's focus on abstracting decisions at the workflow level reflect the assumptions that workflows are accurate models of stakeholder requirements and:

- stakeholder requirements are separable and composable
- individual requirements (and workflows) can be decomposed into sub-requirements (and sub-workflows),
- requirement changes or additions represent increments composed onto existing requirements

While these assumptions are substantially true and relevant, they can be shown false in various important circumstances – notably, workflows are a synthesis of structural relationships (between abstracted entities), data flow, and control flow, while most precisely representing only control flow. Changes to structural relationships and data flows sometimes have application repercussions beyond incremental modifications to workflows, though they can often be addressed through workflow modification.

Given that evolvability is a major application requirement and evolution is a major cost driver for long-lived, large scale applications that serve many stakeholder populations, it is incumbent on an application architecture to support multiple evolution paths. Explicitly enfranchising stakeholders in the change process is critical to system success, but is yet inadequately addressed at the architectural level (often via plugins and hooks constrained at system design time). SOAs rely on strong correlation between service orchestrations (as proxies for workflows) and stakeholder requirements. PDD's SOA foundations (particularly, leveraging Rich Services) and the DSL orientation of its policy expression strategy further enfranchises stakeholders by enabling them to drive the reuse of existing workflows quickly and accurately relative to existing programmer-centric strategies, with the highly granular workflows being most likely to recombine with other workflows – all responsive to emergent requirements not anticipated during the traditional design and programming process.

At its heart, PDD represents the lifting of a Strategy pattern (as briefly described in Appendix C) based on hitherto static or dynamic binding of workflows, and identifies a unique point on the continuum of binding behaviors. Whereas an embedded "if" statement represents a static (or "early") workflow binding, use of dynamic linked libraries and discoverable services (as in Web Services [196], IoC [34], and DI [139]) represents a dynamic (or "late") workflow binding. (Language-based polymorphism and Aspect Oriented point-cut-based code weaving represent other points close to "early" binding.)

With PDD's policy injection, the decisions that discriminate between alternate workflows are abstracted out, and are bound *and* evaluated at the *point of execution*. In the process, alternate workflows themselves are abstracted out, too, and such workflows can implement filters, alternate use cases, or independent features. This avoids creating systems made inflexible through unnecessarily premature choices and entangled workflows, as often occurs with hard-coded "if"s.

(As a point of reference, while dependency injection (DI) and policy injection share the concept of "injection", they differ in what is being injected and when. For DI, a workflow activity is created and bound to a workflow at a location determined at workflow authorship time. The workflow activity is realized before the workflow executes, and the choice of workflow activity is made without accessing state internal to the workflow. For policy injection, the workflow selection decision, the collection of selectable workflows, and the location in the target workflow are all determined at runtime using information available to the workflow as it executes.)

That said, PDD presents challenges that must be overcome before PDD can be widely used in large scale systems.

While policy injection improves evolvability when a stakeholder requirement maps a single decision (and alternate workflows) onto a single base workflow, complications arise when a requirement results in the injection of multiple sub-policies into various workflows (as in the MSoD example in Section 7.4.1), possibly involving alternate workflows that, themselves, only partially address the full requirement. From an authorship viewpoint, the complication arises in traceability between components of a requirement and sub-policies of a policy, and in the structural and semantic compatibility of information shared between sub-policies. From a verification viewpoint, the complication arises in demonstrating sub-policy data flow and dependency relationships congruent with their corresponding requirement components. These issues are not unique to PDD, and are at the core of large system maintenance. It is variously addressed by the FOSD community [197], and by techniques such as dependency visualization [198], type systems, requirement and goal decomposition and traceability, and verification (per Section 7.5).

In the following sections, I compare and contrast other approaches to workflow composition that address these issues, though from different viewpoints.

### 7.1.1 Aspect Oriented Software Design (AOSD)

PDD complements, leverages, and extends work already done by a number of investigators, particularly in the areas of AOSD. PDD draws inspiration from AOSD, which defines highly precise joins and point-cuts, enabling advice around or instead of almost any code in a baseline application. Point-cuts, however, amount to ad-hoc interfaces, which are easily broken by accident by oblivious application programmers – this results in brittle advice and advised code [104]. PDD's alternate workflows are analogous to AOSD's advice, and PDD allows interception only at the service interface, which has a clearly defined and infrequently changed protocol and semantics characterized by a service contract. Because workflow and policy writers can rely on service contracts, brittleness is reduced and oblivious service coding is encouraged.

Additionally, AOSD implements a form of message interception (enabling inbound and outbound filtering) via around-advising parameter-laden function calls or point-cuts at function entry and exit points. This function call mechanism does not gracefully distribute across distributed computing systems, whereas PDD's reliance on explicit message passing scales naturally in such situations.

While both AOSD and PDD encourage advice on concerns, the AOSD code weaving strategy discourages advice on advice except in the special circumstance of advice calling advisable functions. Under PDD, because of ubiquitous policy evaluation, composing workflows and features is encouraged, thereby improving evolvability of existing workflows and features.

Early AOSD work [32] suggests that workflows can be simplified by reducing entanglement using AOSD principles. Since then, various metrics have arisen to quantify the effects and other complexity-related code characteristics (e.g., Crosscutting Degree, Coupling on Method Call, etc) [199]. Such metrics are designed to drive workflow simplification and promote refactoring as aspects. As PDD rests on AOSD principles, PDD's focus on defining base and orthogonal workflows joined by policy produces applications that can be evaluated in the same way, though with tools extended to evaluate the PDD policy language (e.g., XQuery).

However, as a vehicle for the integration of both small and large scale components as Systems of Systems, PDD's injected workflows can be completely encapsulated (as with the Policy and Audit systems in Sections 5.6 and 5.6.4) or can share state with related workflows (as in Section 7.4.1). PDD discourages entanglement and promotes such encapsulation, and would naturally produce cohesive systems with low coupling. Whereas existing AOSD implementations discourage aspects on aspects, PDD encourages them, which can lead to complex workflow inter-relationships, including cyclic relationships, at the system level. Extensions of AOSD metrics would be helpful in assessing policy-based system complexity and maintainability, though defining such metrics would be challenging, as policies are free to choose amongst many workflows (or even create new workflows) based on dynamic conditions.

AOSD conceives of the issue of composing multple policies on a single interaction as *aspect interference* and provides both ad-hoc and model-level mechanisms for detecting and prioritizing interfering aspects [200] [201]. PDD addresses this through composition policies, which are specified and executed at runtime, responsive to both dynamic state and changing stakeholder population and relationships, as described in Sections 4.2.7 and 5.6.2.2.4. PDD makes no attempt to discover or resolve compositions of conflicting policies, and leaves this to policy programmers (in the short term) and future work (in the long term).

### 7.1.2 Policy-based Design

Policy-based Design (PBD) [202] defines a crosscutting concern in terms of abstractions reused throughout an application. It defines a policy class as a type-based interface representing a concern, and advocates that application designers and coders create and reference a policy class whenever they make design choices that can be deferred. While a policy class carries type information, the emphasis of its definition is behavioral. Therefore, a class (or application) is assembled out of policy class interfaces that represent orthogonal behavioral or structural aspects of an application design.

PDB's objective is to manage the combinatorial explosion of design choices encountered in the application design and code authorship process, and create an ecosystem that addresses new requirements as recombinations of new and existing policy implementations. By leveraging a collection of such policy classes and their implementations, an application can integrate and orchestrate a concern without committing to an implementation until compile time.

PDB relies on the C++ concepts of templates (generic programming) and multiple inheritance to compose type-safe policies into an application. For a given policy class, there may be several concrete classes, each of which represents a different set of characteristics and design choices in realizing the particular policy class abstraction. Templates enable the relatively safe type- and functional-parameterization of policy classes, and multiple inheritance exposes policy methods so they can be easily used during coding.

PDB attempts to recognize crosscutting concerns early in the design, coding, and re-engineering process. As such, it encourages building workflows that emphasize base concerns, and then composing policy abstractions into them. As with PDD, policy abstractions can implement filtering, control flows, and feature compositions. Similarly, as with PDD, PDB encourages the composition of one policy onto another, and on to base or composed features. Its contribution is to address evolvability at the coding and re-release level, but not to address emergent requirements at runtime or to enfranchise stakeholders directly. Additionally, it has no inherent contribution to large scale distributed systems capabilities beyond what is expressly coded into policy class instances.

## 7.2 Workflow Context

PDD contexts (described in Sections 4.3 and 5.5.4) focus on the availability of message-, workflow-, session-, and application-based state for workflow activities executing in a distributed system, which is not well addressed by other approaches. In subsequent sections, I compare and contrast PDD concepts of state and workflow management in popular and successful application architectural frameworks that represent different views of workflow context.

### 7.2.1 Struts

While composing context references and transporting them via interservice messages is unique to PDD, the lifecycles of such contexts are inspired by the Struts [126] system, which executes workflows on behalf of clients, similar to PDD, and is described in Section 2.4.5.1.

Whereas a Struts workflow is guaranteed to execute on a single computer in a single thread (thereby simplifying its request bean implementation), PDD has no such guarantee, which results in PDD maintaining workflow state in the thread-safe CIS as a service reachable by all computers. Consequently, PDD workflows are scalable across distributed systems, whereas Struts workflows are not.

Maintaining a Struts session bean depends on a session reference passed using an HTTP protocol between a client (browser) and the Struts system – via a cookie or URL rewriting. PDD does not assume an HTTP protocol or that the client is a web browser – while the PALMS implementation uses Web Services to communicate with clients, the Mule ESB on which it runs allows various protocols, including HTTP and others. Consequently, automatic session creation, passing, and deletion are not implemented in PALMS – sessions are passed and maintained under explicit request of a client. As a result, PDD sessions are not limited to representing users – they can represent any persistent context, including a user.

A Struts application bean is created by a workflow and can be accessed by any workflow for the duration of the application execution. A PDD IV has similar scope and function. PDD implements IVs using its independent, thread-safe CIS service, similar to a corresponding Struts service.

Unlike Struts, PDD supports the dynamic composition of a workflow onto a base workflow, and supports segregation of state for each workflow. Its CIS service allows storage and retrieval of key-value pairs for workflow contexts (and all other contexts), where key names can be dynamically generated by workflow activities (similar to the π-calculus approach described in Section 2.2.3). So long as a composed workflow activity generates a unique key name, it can store and retrieve workflow-scoped values without conflict with other composed workflows. Furthermore, PDD enables the creation of multiple workflow-based contexts (on behalf of multiple, separate concerns), which further segregates workflow-based state.

### 7.2.2 REST

A RESTful application (described in Section 2.4.5.2) executes a workflow where an interaction is implemented as message exchanged between a state-laden client and a server. Such an interaction can be modeled using the service concepts of Section 4.1.2, and includes both request-only and request/reply interactions.

REST can simulate PDD policy evaluation (per Section 4.1.3) by replacing every server with a policy evaluator proxy that wraps the server, evaluates policy, and then modifies a request or response, or directs a service request according to the policy. Additionally, REST can simulate PDD's IV-style variables, which PDD implements as REST-style calls to its CIS service.

However, to service a workflow injected by a policy evaluator proxy, workflow variables (or a CIS-style reference to them) must be included in all server requests (including propagation through downstream server requests), thereby accomplishing the objectives of PDD's interservice message feature (described in Section 4.3). This can be arranged in remote procedure call (RPC) proxies linked into RESTful clients and servers (to the extent that such code interacts via RPC proxies), but this would require relinking and redeploying RESTful client and server code, which may not be available or possible for an given application or set of remote services. For clients and servers that don't interact via RPC proxies or cannot be relinked and redeployed, the alternative is to encode the equivalent of interservice messages explicitly in the REST client and server code, followed by redeployment.

In sum, the policy evaluator proxy and the manual propagation of workflow variables impose burdensome requirements on REST style applications (just as they did on PALMS services before PDD was implemented in PALMS, as described in Section 6.1), making it difficult for them to react quickly to stakeholders' emergent requirements.

Note that REST servers are accessed via URL, which functions as the Internet version of a routing system. Consequently, from a routing viewpoint, replacement of a REST server can be a simple and low risk proposition – via changes to DNS or other mapping tables. However, to simulate interaction interception that could enable workflow injection, the routing system would need to somehow encode both the source *and* the target services so as to enable policies to associate with a source-target pairing (as in Section 4.1.5). Current DNS-based routing is source-agnostic.

Additionally, the process of generating and deploying a replacement server incurs the bottlenecks attendant to traditional programming disciplines, and can be time consuming and risky.

Under unmodified REST, implementation of diverse stakeholder requirements (e.g., access control, auditing, provenance tracking, quality of service, and failure management) represents entanglement of concerns application wide.

### 7.2.3 AJAX

An AJAX application (as described in Section 2.4.5.3) implements workflow state as a closure whose life cycle is limited to a single external service call. Maintaining workflow state across successive server interactions is performed via manual coding.

Considering that a closure implements a pairing of requestor state with reply data, a closure can be implemented under PDD as a workflow variable, where the source service explicitly stores the closure as a workflow SIV, and the target explicitly fetches and deletes it. Services can access SIV values using SIV access libraries.

Note that PDD-style policy under AJAX is possible, though with significant restrictions. PDD policies enable policy injection on interactions as characterized by a source and target service. Under AJAX, the client is always the source, and the server is always the target. Any finer distinction would require message-level information as a matter of convention, thereby imposing a bookkeeping burden on all requestors in the client, and on a message hook in the server. Given this, the servlet could be proxied by a policy evaluator service (as in Section 7.2.2), but access to SIVs common to client and server would require further study.

A browser-based client-side policy system is hampered by both the difficulty of intercepting interactions between functions in JavaScript, and the security requirement that such a client communicate only with a single server.

Note that these limitations apply to the GetStudyList workflow described in Chapter 3 – in the PALMS implementation, the Client has a REST-style AJAX relationship with the PALMS server. The bulk of my dissertation applies to workflows defined within the PALMS server (not the client) where interactions can be easily intercepted, service tracking can be readily implemented, and alternate services can be readily invoked.

## 7.3 Orchestration Languages and Workflow Systems

There are a number of orchestration languages and workflow management systems that address portions of the core PDD requirement space.

Scripting and orchestration languages (e.g., Groovy, BPEL, WS-CDL, WSCL, MSCs, UML and Orc) provide routing that includes decomposition (in addition to looping and other control flows), and provide scoped variables supporting global and workflow contexts, but not session context, message interception, or static policy-based workflow substitution. With no facility for aspect definition or injection, such languages encourage entanglement of concerns, which complicate the implementation of exception handling, access control, business rules, and feature composition.

In contrast, AO4BPEL [146] defines point-cuts on BPEL, where advice occurs at the service interface level, thus enabling modular maintenance of business rules composed into an orchestration at runtime. By composing aspects at the service interaction level, AO4BPEL avoids much of AOSD's brittleness. AO4BPEL relies on runtime uptake of static point-cut and advice specifications, thereby enabling a degree of dynamic composition, though only on service interactions identified by developers during the code authoring process. However, it has no mechanism for specifying point-cuts at runtime, and so provides limited opportunities for an application to react to stakeholders' emergent requirements via injectable policy.

Similarly, BPEL Business Rules Integration [203] (described in Section 2.4.4.2) extends BPEL by composing business rules onto service interactions, but without enabling the selection of rule injection points at runtime.

Workflow Management Systems [75] (WFMSs) assume centralized knowledge of workflows and available roles; WFMS performs centralized access control and scheduling of tasks. [204] distributes WFMS knowledge and decisions to create a mediator-free fabric based on task discovery protocols. Under PDD, knowledge of a workflow and access control policies exists at the site of policy injection, without any centralized or distributed control. WFMSs represent top-down workflow management, which PDD represents as bottom-up. As such, PDD's strategy encourages fluid policy contributions by multiple stakeholder groups.

WS-CDL [205] is a web services choreography language that defines rules that govern the ordering of messages exchanged during service interactions. It defines choreographies relative to a root choreography. At its heart, a choreography is a functional language (or ADL) describing the relationships to be imposed between services, including how service outputs and inputs are coordinated, and does not discriminate based on authority domains or other types of policy. As such, it defines the coordination of workflows, and presides over the coordination. Unlike PDD, it contains no provision for crosscutting concern or feature composition, scalability, or conditional composition.

## 7.4 Policy Evaluation Systems

At an abstract level, policy evaluation systems attempt to implement requirements as crosscutting concerns composed (via policy) upon workflows. They differ in the assumptions they make about scope of policy and its relationship to base workflows. This section describes how PDD represents different choices on these and other dimensions.

### 7.4.1 PERMIS

Historically, the dominant use of policy-enabled systems is in defining and enforcing access control, trust relationships, and data security. A common paradigm occurs in the application coding process, where a static decision is lifted to a dynamic decision through the use of a policy evaluation system (as discussed in Section 7.1.2).

As described in Section 2.4.4.1, the PERMIS policy languages express concepts important to security practitioners, but do not service other domains (e.g., failure management, data filtering, auditing, and intrusion detection). Additionally, their formulation in XML leverages standards that represent security and trust concerns, but neither the standards nor the XML encodings are easily understood by many communities (even with GUI support). While PDD's concept of DSL-oriented policy languages is more flexible and accessible to diverse communities, such DSLs have not yet been defined to replace, extend, or map to the standardized XML-oriented languages.

Solving the classic access control problem of Multisession Separation of Duties (MSoD)[4] [206] demonstrates PERMIS' and PDD's (as PALMS) contrasting policy statement and evaluation styles. Under MSoD, a user *cannot* exercise conflicting roles even if the user is a member of *both* roles. As shown in Figure 68, MSoD occurs in a shared context (represented by ObjectA), where a user attempting a particular operation must have a role that qualifies to execute the operation, and cannot have already executed a conflicting operation (as recorded in a hypothetical History log):

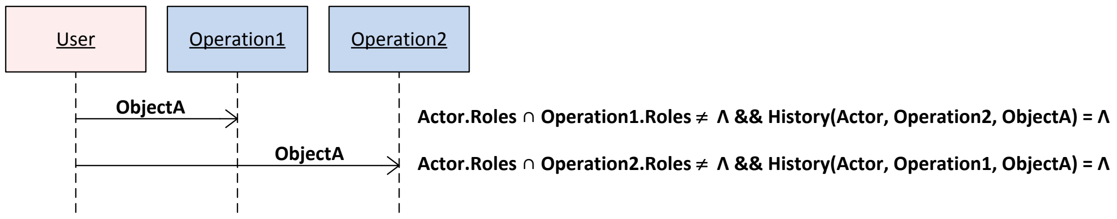

User
Operation1
Operation2
ObjectA
ObjectA
Actor.Roles ∩ Operation1.Roles ≠ Λ && History(Actor, Operation2, ObjectA) = Λ
Actor.Roles ∩ Operation2.Roles ≠ Λ && History(Actor, Operation1, ObjectA) = Λ

Figure 68. UML Activity Diagram of Multisession Separation of Duties

More generally, while role membership can be used to allow or deny a user access to an activity, relationships between roles themselves provide a separate layer of logic that extends and refines these constraints.

For example, in a bank, a user can fill multiple roles, including separate teller and auditor roles. Were access to a teller or audit function to be based solely on the roles a user holds, a user could be both a teller and an auditor in

[4]http://www.sans.edu/research/security-laboratory/article/it-separation-duties, http://www.niiapp.org/fileadmin/files/2010-05-20-procedure-policy-template.pdf, http://sec.cs.kent.ac.uk/permis/DevPlans.shtml#separation, http://www.cs.colostate.edu/~iray/research/dbsec08.pdf

the same bank branch (i.e., common context). This contradicts common financial checks and balances (i.e., Separation of Duties or SoD).

The example qualifies as “Multisession” because the teller workflow is completely separate in time from the independent auditor workflow (and may execute on a different computer) – state maintained only in a workflow context, a login session, or on a single computer is insufficient to enforce the Separation of Duties. The MSoD problem is further complicated when Virtual Organizations are in play – role sets defined in multiple, dynamic authority domains cannot be evaluated at any time except during actual workflow execution.

Both PERMIS and PDD solve the problem by maintaining application-level state outside of a workflow or session context. PERMIS introduces a “business context” (which tracks the events pertaining to a set of constrained activities such as might result in an MSoD conflict) and a “role constraint” (called multi-session mutually exclusive roles or MMER) that defines when, in a business context, the activation of a role is forbidden. A sample PERMIS MSoD policy is:

```
<MSoDPolicy BusinessContext="Branch=*, Period=!">
  <LastStep operation="CommitAudit"
            targetURI="http://audit.location.com/audit"/>
    <MMER ForbiddenCardinality = "2">
      <Role type="employee" value="Teller"/>
      <Role type="employee" value="Auditor"/>
  </MMER>
</MSoDPolicy>
```

The policy specifies that the business context applies to all bank branches (i.e., `Branch=*`) and all time periods (i.e., `Period=!`). It identifies the roles in play as `Teller` and `Auditor`, and it uses set logic to enforce that an employee cannot hold both roles (i.e., `ForbiddenCardinality = "2"`) in the business context. This policy assumes that a) the policy system tracks all events pertinent to all bank branches and time periods, b) it tags all events with the role that initiated the event, and c) an employee declares the role being exercised at some point (e.g., during signon). It asserts that for the business context, the history can be deleted once the `CommitAudit` event occurs. To support this policy, an application must identify when a user is exercising a role, and when the `CommitAudit` event occurs. Additionally, it must evaluate the policy immediately before the `CommitAudit` activity commences.

An analogous MSoD policy in PALMS might be that a research assistant adding participants to a study cannot be the one that deletes them. For consistency in this example, though, I use PDD policy principles to address the PERMIS MSoD banking scenario.

Under PDD, suppose a RBAC-oriented banking DSL that evaluates whether a user is a member of a role, and that the user can hold several roles simultaneously. To execute a workflow based on role membership alone (such as a Teller workflow), a plausible requirement could be phrased as `palms:user-in-role('Teller')`, and similarly `palms:user-in-role('Auditor')` for an Auditor workflow.

However, implementing the role conflict rules requires persistent state indexed by bank branch and role, and containing a list of users exercising a particular role. The list must be augmented when a user exercises the role, and must be checked when the user is about to exercise a conflicting role.

A hypothetical MSoD DSL might be useful in articulating MSoD policies in terms directly related to MSoD requirements. Assuming the branch is a value in a message $m$ (though it could be in a workflow or session context), the DSL might have the functions in Table 10.

Table 10. MSoD DSL Functions

| Function | Parameters | Return |
|---|---|---|
| **msod-valid** | $test-role, $add-role | boolean |
| **msod-clear** | | |

The `msod-valid()` function would fetch the bank branch from the current message $m$, check for the user having held the `$test-role` role at the branch, and if not, register the user as holding the `$add-role` role at the branch, and then continue the workflow. `msod-valid('Auditor', 'Teller')` would be used to guard a teller workflow, and `msod-valid('Teller', 'Auditor')` would be used to guard an auditor workflow.

The `msod-clear()` function would delete the branch's role lists at the end of the `CommitAudit` workflow.

A policy incorporating the `msod-valid()` function would be injected into an interaction that commences an MSoD-sensitive workflow:

```
        if (palms:MSoD-valid('Teller', 'Auditor')) then () \
          else palms:control-error('denied')
```

Similarly, a filter containing the `msod-clear()` function would be injected immediately after an audit workflow.

Using the library functions described in Section 5.6.5.1, an `msod-valid()` function would test whether the user was present in a role list for the branch, and would add the user to a role list for the branch. The following example shows how to use PALMS' Context system (CIS service) to store MSoD-related state as an IV, which persists across service interactions, workflows, and sessions:

```
  declare function msod-valid($test-role as xs:string,
                              $add-role as xs:string)
      as xs:boolean {
    let $msod-context := "MSOD Context"
    let $test-key := concat(palms:cur-value("branch"),
                            ".", $test-role)
    let $add-key := concat(palms:cur-value("branch"), ".", $add-role)
    let $cur-user := palms:get-workflow-user()
    let $users-in-test-role := get-iv-value($msod-context, $test-key)
    let $users-in-add-role := get-iv-value($msod-context, $add-key)
    return if ($cur-user = $users-in-test-role) then
      false
    else
      palms:set-iv-value($msod-context, $add-key,
                         ($users-in-add-role, $cur-user))
        }
```

Note that under PDD, the CIS stores key-value pairs, each of arbitrary composition. A suitable key for this event stream would be the name of the bank branch, and a `Teller` and an `Auditor` sub-key would contain a collection of users that have exercised the `Teller` or `Auditor` roles for the branch. By definition, the CIS resolves key-value pairs for policies executing a distributed system.

By avoiding PERMIS' XML and set logic constructs, a PDD policy more directly tracks the stakeholder's understanding of the issue, invites stakeholder interaction, and reduces the likelihood of conceptual or formulation errors – all of which contribute to efficient and effective system evolution. Because PDD supports DSLs via XQuery libraries, it extends this benefit to multiple stakeholder groups simultaneously.

Because XQuery has many features of a general programming language, the creation of complex policies under PDD is quick and simple. For example, if the MSoD rules were declared to be non-operational on weekends, a simple predicate could be added to the MSoD (or other) DSL language:

```
if (palms:is-weekend() \
    or palms:MSoD-valid('Teller', 'Auditor')) then () \
else palms:control-error('denied')
```

Relative to a base workflow, an MSoD policy represents a separate access control concern. The PERMIS authorization system abstracts the particular policy (as XML) but not its insertion into the base workflow. PERMIS acts as a Policy Decision Point (PDP), but not a Policy Evaluation Point (PEP). Therefore, invocation of a PERMIS policy requires that a base workflow be coded to call PERMIS and act on the policy result (both to guard a workflow and to signal a policy event, as in the example above). To add a new policy requires not only entanglement of the base workflow and the authorization concern, but also a separate deployment of the resulting code. As an alternative, PDD requires no modification of base workflow code, allows separate concerns to remain separate, and enables complex policies to be written and injected using a language that reflects the requirements of the concern.

While the MSoD calculation consists of a simple logic comparison, expressing more complex calculations follows the same pattern. Under the PERMIS approach, no facility exists to compose policy onto a complex calculation (e.g., making it inoperable on weekends). However, under PDD, such a policy could be composed onto complex calculations that themselves are expressed as workflows. In essence, PERMIS' emphasis on access control renders its composition shallow, while PDD's emphasis on feature injection results in potentially deep composition.

Note that while PERMIS does not address runtime policy injection, it presents an end-to-end secure policy execution solution, which encompasses transmission of principal attributes, secure policy storage and distribution, and role and attribute management in a distributed environment. Notably, PERMIS leverages distributed attribute and trust fabrics to service Virtual Organizations (VOs), where an organization can maintain attributes autonomously relative to other organizations, thereby granting or denying its members access to resources regulated by existing and new policies. PALMS achieves a similar effect by accessing role and permission definitions in Grouper, and phrasing policy in terms of those roles and permissions (as described in Section 5.6.1.3.1).

Under PALMS, the XQuery engine functions as the PDP, and a control policy returns a workflow to the Policy Evaluator, which then executes it. The policy performs part of the work of the PEP (i.e., selecting the workflow), and the Policy Evaluator performs the other part (i.e., executing the workflow). PERMIS does not have a good analog for deciding and executing filter policies.

### 7.4.2 xESB

The xESB system (described in Section 2.4.4.3) extends the ESB concept to include policy evaluation on every service interaction, where policies act to constrain behavior of workflows that incorporate the interactions. Constraints are expressed as rules and obligations based on [207] and [208], where rules implement a verdict as the combination of a decision and an action, and obligations maintain state upon which rules rely. xESB maintains state as counters, timers, and hashes, all globally declared and instantiated. xESB policies can simulate a session by using a hash variable indexed by user identity, though this assumes that multiple users with the same identity should share common state.

Insofar as xESB represents an extension to ESBs, neither xESB nor the ServiceMix ESB it extends has a concept of workflow. While it is conceivable that a workflow concept could be introduced into an ESB (given that PDD has accomplished this using the Mule ESB), this has not been done. Consequently, xESB's context system is simple and does not support the diverse scoping of PDD's SIVs.

Additionally, since xESB applications do not explicitly model workflow, xESB does not model workflow composition and decomposition (as supported in Section 5.5.3), and does not support workflow- and session-based state, it does not address System of Systems composition.

Examining alternate solutions to xESB example of a multisession video delivery requirement demonstrates xESB's and PDD's (as PALMS) contrasting execution time characteristics, policy statement, and evaluation styles. Similar to PDP, xESB injects policies into SOA service interactions directly at runtime, though it does so differently: it evaluates all policies on each service interaction. The comparison of xESB to PDP focuses on runtime performance, the policy language, and its suitability for composing complex requirements (as workflows) to create System of Systems architectures.

In [124], xESB was evaluated while running under Java on the Apache ServiceMix ESB v3.3 on a 32 bit 2.6GHz processor, and claims times of approximately 1ms per policy per service interaction to determine whether a policy applies to the interaction. No time was given for the execution of a policy action.

A comparison to the PALMS policy system would encompass porting the $< PALMS, ListStudies >$, $< ListStudies, StudyRepository >$,

$< StudyRepository, Storage >$ workflow to xESB, and then evaluating a rule collection for each of the three service interactions in the workflow. Supposing a robust rule set of ω xESB rules, each interaction would require approximately ω milliseconds (at 1ms per interaction), or 3ω milliseconds for a complete workflow execution. To match PALMS functionality, the ω rules must contain pre-filters, control policies, and post-filters. Considering that xESB has no construct that accomplishes PDD's composition policies, I make a simplifying assumption of implicit composition policies that evaluate all policies that apply to a particular interaction.

Per Section 6.3.2.2, the PALMS system adds 1.7ms for the entire test workflow, assuming each interaction is registered in the Interaction Repository but no interaction is actually associated with a policy. The cost for adding a pre-filter, control, or post-filter policy varies with the message data and the complexity of the policy, but (unlike xESB) is paid only when a policy is actually associated with an interaction. Per Sections 6.3.5 and 6.3.6, the time required to execute a single control policy or filter policy is between 76.82ms and 78.94ms, varying according to the size and complexity of the interaction message and the policy.

Considering that the PALMS tests were run on a much faster CPU than the xESB tests, it is fair to say only that the xESB cost per interaction is a fixed function of ω, and for interactions that have no policies, its execution time is comparable to PALMS'. However, for large policy collections, the xESB time can be an order of magnitude greater when interactions have few if any policies. Because [124] does not include execution time for policy actions, numerical comparisons to PALMS' policy execution time cannot be made. However, because xESB constrains the format of interaction messages to match its constrained policy language, it does not incur PALMS' overheads of message format conversion, XQuery expression compilation, and XQuery engine execution. Consequently, I expect that xESB policies would run much faster than PALMS' policies, assuming the policy requirement could be expressed in the xESB policy language.

On balance, more measurements are required to determine the time relationship between xESB and PALMS policies, and the answers would depend on the density of policies relative to service interactions, the size and complexity of interaction messages, and the complexity of policies themselves.

Additionally, the xESB policy language is tuned to address access and usage control requirements [209], where PALMS' policy language addressed workflow composition in general, which is a superset of access and usage control. Whereas PALMS posits DSLs that align with requirement statements of various stakeholder communities, the xESB policy language is structured for use by programmers writing code responsive to user requirements. An example of the xESB policy language addresses a video calling service whose requirement is that a "Silver" customer can call for at most three hours per month:

```
  default-action { allow; }

  // Total duration of video calls, in seconds
  hash videoDuration = 0;
  timer resetDuration = next month;

  obligation {
    if invocation
      when { resetDuration.fired }
      do {
        clear videoDuration;
        arm resetDuration fire next month;
      }
  }

  obligation {
    if response
      when { h "Type" equals "video-call" && h "Success" equals
"True"
              && h "Customer-Type" equals "Silver" }
        do { update-counter videoDuration[source] += h "Duration"; }
  }

  rule {
    if invocation
      when { h "Type" equals "video-call"
              && h "Customer-Type" equals "Silver"
              && videoDuration[source] > 10800 }
      do { block; }
  }
```

The policy is stated in three parts: two obligations and a rule. An obligation sets rule state – the first obligation executes before a video call service, and the second obligation executes after, thereby capturing the cumulative duration of video calls for a customer. The rule executes before the video call service, and checks to see whether the accumulated duration has exceeded three hours – if so, it waits until next month.

As shown in Section 7.4.1, an equivalent PALMS approach would be to define a DSL that abstracts these video accounting functions, and then used the DSL functions as policies injected onto appropriate service interactions. This has the immediate advantage of reducing the proliferation of accounting policy code, including variations on the policy, and eases the burden on domain-interested stakeholders in inspecting, verifying, and contributing to such policies.

Note that XQuery does not provide an equivalent of a scheduling or event sink function. However, the PALMS Feature Composition DSL (described in Section 5.6.5) exposes the `call-service()` function, which can be used to invoke a scheduling or event sink service capable of executing an XQuery policy or other workflow at an appropriate time.

xESB and PDP have generally similar performance – xESB's execution time depends on the value *each* policy lends to *each* service interaction, and PDP's execution time depends on the complexity of the message data and the query compilation time. However, PDP's DSL-oriented policy language approach invites stakeholder participation in the requirement implementation process, where xESB's does not. While xESB's policy language was formulated with an eye towards the validation and verification of policies, PDP's policy approach does not emphasize this – instead, it leverages validated DSL support libraries and DSL-simplified policy statements to achieve reliability ad-hoc. (It would be possible to specify a PDP DSL similar to xESB's policy language, and therefore similarly verifiable. However, this would defeat the goal of readability and stakeholder participation.)

While xESB's constraints on message format and content simplify and accelerate policy processing, PDP has fewer constraints, thereby enabling policies applied to services that aren't conceived under the xESB messaging model. PDP also enables injection of policies from multiple stakeholders (through composition policies) and enables the composition of requirements onto requirements, both of which enable the construction and scaling of cyberinfrastructures as large Systems of Systems.

### 7.4.3 Ponder2

In contrast to PDD's workflow-oriented model, Ponder2 (as described in Section 2.4.4.4) applications are implemented as a network of connected components. To the extent that the two abstractions accomplish similar goals, workflows represent a lifting of the connected component model by enabling the execution of multiple interactions under a common context. Consequently, in addition to its limited facility for runtime policy and workflow injection (as described in Section 2.4.4.4), PonderTalk offers no state lifecycle that implements the concept of SIVs (as described in Section 4.3).

### 7.4.4 Policy Deployment

PDD does not define the practical aspects of policy, user, and attribute management, and so is not an end-to-end solution in the PERMIS sense (as described in Section 7.4.1). The PALMS case study does demonstrate the use of Grouper as a securely managed attribute store and the use of X.509 certificates to prove user identity. However, PALMS does not claim the robustness claimed by PERMIS – it does not have either a secure policy or attribute distribution capability. Under PALMS, policies are imported and activated periodically (e.g., once a minute) from files placed in a policy staging directory, which is secured by the underlying operating system.

Secure and robust policy distribution systems and methodologies exist and can be adopted or leveraged by applications such as PALMS as the need arises: [210] provides a general-purpose policy deployment and execution model that is independent of underlying policy enforcement mechanisms. It is agnostic as to the type or language of a policy, and focuses on policy instantiation, distribution, enabling/disabling, unloading, evaluation, and deletion of policies in a distributed environment.

### 7.4.5 Policy Languages

A central focus of the policy community is access control, usage control, trust relationships, and digital rights management, as exemplified by the Ponder language [130] (Section 7.4.3), xESB's usage control language

[124] (Section 7.4.2), OSL [207] (for obligations), PERMIS' policy languages (Section 7.4.1), SPL [211] (for complex constraints), EPAL [212] (for privacy), and others as enumerated in [213].

Generally, each of these languages seeks to articulate and implement relationships in a particular domain. Their design approach is minimalist so as to enable arguments for sufficiency and completeness, and to position them to allow validation and verification of the properties of domain interest. Minimalism also serves implementability and tends to control execution overhead partially because of their simplicity, and sometimes at the expense of a rich dataflow between entities. These approaches generally produce languages that declaratively express requirements directly as mathematical, logical, and relational concepts, and are intended for use by programmers with those skills. Such languages are often inaccessible to stakeholders in other domains, even when expressing relationships from those domains – the PERMIS MSoD policy in Section 7.4.1 is an example of this.

While my approach to the PDD policy languages (as exemplified in the PALMS languages described in Section 5.6) adopted the decision-action paradigm of many policy language, its objective was to strike a balance between:

- enabling stakeholders to understand and possibly write policies themselves
- promoting extensibility to meet unanticipated requirements of existing and future domains
- enabling the composition of external workflows in the context of base workflows (for System of Systems integrations)
- enabling injection of policies at runtime without requiring elaborate compilation or binding
- accessing diverse sources of data (for decision making)
- supporting rich interactions between services by accommodating (and not constraining) data exchanges

My choice of XQuery as the basis for a family of functional languages designed and presented as DSLs (as described in Section 5.6.5) demonstrates progress towards stakeholder enfranchisement (as described in Sections 6.2 and 1.1) and meets the challenges of enabling workflow composition and rich service interactions.

As described in Section 6.3.7, service interactions on which policies are evaluated are significantly slower compared to service interactions with no policies. (The policy system presents negligible overhead for service interactions where no policies are evaluated.) The evaluation time depends on:

- the complexity and size of the interaction message
- the time required by the XQuery engine to compile and execute the policy
- the complexity and size of the new interaction message (for filter policies)

From the perspective of the execution of an entire workflow, the policy contribution to execution time can be significant even when service execution times are included. To improve interaction times, a number of strategies can be adopted, following the experiences of xESB, Ponder, Ponder2, and others.

Particularly, the XQuery compilation and execution times can be reduced by pre-compiling policy expressions or caching compiled expressions. In the PALMS implementation, policies are read and staged by the Policy Repository service (described in Section 5.6.2.2). The effective policy is produced in the Policy Evaluator by combining the interaction's composite policy with corresponding control and filter policies based on the study or study group contained in the interaction message (as described in Section 5.6.3). The effective policy calculation can be relocated to the Policy Repository. Regardless of where the policy is calculated, caching the compiled policy would amortize the cost of a compilation over the life of the policy.

Additionally, message format conversions can be reduced or eliminated by harmonizing the message format with the policy execution language processor. PALMS messages are transported by the ESB as Java objects defined according to the needs of the service interaction they serve. However, the same information can be encoded as Saxon XDM structures, or other structures native to the XQuery processor, thereby avoiding ingress and egress transcoding (as described in Section 5.6.3). This would complicate the encoding and decoding of messages within services themselves, thereby violating the principle of interceptors adding value to otherwise oblivious service interactions (though this is just what xESB did). Alternatively, I observe that the PALMS DSLs are functional programs that can be expressed in Java, which consumes Java objects naturally. Java-based policies would access message data quickly and efficiently, and can use Java introspection to access or build variable data structures.

The benefits of minimalist design of existing policy languages inure not only to validation and verification, but to economical execution based on constraining message formats and contents, coding policies directly to the expected messages, constraining the complexity of messages, and native compilation of policies.

In contrast, the XQuery language enables workflow coders and policy writers to ignore these considerations – as an abstraction, it works well on structured data expressed as text (i.e., XML) and transformations (i.e., policy expressions as XQuery code) expressed as text. Consequently, policy programmers can easily create and evolve DSLs. However, the flexibility of XQuery can be acquired through judicious use of other languages, including Java, combined with expression pre-processing (similar to what already occurs with XQuery policy fragments as described in Section 5.6.2.2) and just-in-time compilation. The choice of query language and associated pre-processing can be managed at the Policy Repository level, thereby preserving all PDD abstractions.

This calls the question of what support is minimally required to define and evolve a DSL, and what skill sets are required to do this. To meet the goals of PDD, DSLs must be capable of evolving quickly and responsively to evolving stakeholder requirements. The skills must be available in the general programming population or from stakeholders themselves. Debugging DSL support libraries must be simple and reliable, and versioning and deployment must be addressed.

These issues drive the evolution of PDD and are discussed briefly in Section 1.1, but are beyond the scope of this dissertation.

### 7.4.6 Policy Versioning

Under PDD, a policy represents the implementation of a crosscutting concern composed onto a workflow, where the implementation may be a simple decision (based on no state, and maintaining no state), or may maintain state that affects its future behavior or is relied upon by other concerns. Insofar as a policy maintains a set of guarantees, a policy is incorrect if the guarantees are not met.

When an existing version of a policy is replaced with a new version, the new version must maintain not only its set of guarantees, but the guarantees of the original policy. Otherwise, the concern represented by the policy may fail overall.

For example, the MSoD concern (described in Section 7.4.1) guarantees that if a role executes one function, it cannot execute a second, proscribed function. The concern is represented by a policy that saves state in an IV-based key-value pair where the key and value have policy-defined structure. The policy can provide the guarantee so long as the policy uses the key-value pair consistently.

If a new version of the MSoD policy maintains this guarantee, though by using a different key-value pair, the new version is internally consistent, yet fails to maintain the guarantee of its predecessor. Consequently, a role may execute a proscribed function undetected, which would be an error. The state skew between policy versions over time is one example of how both a new and old policy version can be correct, but the guarantees of the concern can go unmet, and there are numerous other examples.

The general problem of maintaining concern guarantees in the face of policy evolution is a variation of the timeless problem of guaranteeing upward compatibility of applications over time, and includes deployment synchronization in distributed systems, and is yet unsolved. The problem is addressed in theory by [208], which proposes that guarantees be only strengthened over time.

PDD makes no attempt to solve this problem, and the problem exists in policies provisioned under PERMIS (in Section 7.4.1), Ponder [210] (and in Section 7.4.3), xESB (in Section 7.4.2), and other policy evaluation systems, whether policy is provisioned statically or at runtime.

## 7.5 Verification and Validation

PDD's focus on maintaining and reusing separate concerns discourages entanglement, resulting in reduced complexity of individual workflows, which eases assessment of workflow completeness and correctness, and lowers evolution costs. Additionally, because PDD leverages hierarchical decomposition available in Rich Services, it encourages a hierarchical approach to verification and validation of workflows.

In part, complexity shifts to assessing the completeness, correctness, and consistency of policy statements and their relationship to each other and to alternate and base workflows. Such complexity may be tackled by evolving existing visualization [198] and verification tools.

In this discussion, I use the terms *verification* and *validation* as they are commonly understood:

> *Software verification provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements for that phase by checking for consistency, completeness, and correctness of the software and its supporting documentation. Validation, on the other hand, is the confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled. [214]*

While automated verification of PDD policies has not been attempted, automated verification involving related policy approaches has been accomplished in a number of ways, including proof, simulation, and comprehensive examination.

Structural modeling techniques (described in Section 1.1) enable proofs based on factoring OCL in UML models [215] as aspects, though the carryover to workflow-based models is incomplete.

Verification of aspects [216] focuses on the correctness of an aspect relative to its specification, which contains assumptions regarding the base application and guaranteed properties after the aspect is woven into it. It model checks a linear temporal logic (LTL) expression that includes the assumptions, join points, and aspect advice. The authors claim to create modular proofs on weakly invasive aspects. This aspect verification strategy aligns well with PDD, given that composed workflows can be characterized by their property guarantees. However, pursuing this is beyond the scope of this dissertation.

Insofar as workflows (including applications containing numerous workflows) can be modeled using Colored Petri Nets (CPNs) [90], policy composition can also be modeled [74] (as workflows composed upon workflows). Given this, modeling policy-injected applications as CPNs may be able to verify domain-specific properties and properties such as liveness, safety, and reachability – this has not been tried with PDD-based policies. The ability to leverage composed workflows as CPNs aligns well with the PDD concept of policy injection, and therefore holds promise as an avenue for PDD policy verification. Similarly, [217] and [218] propose process algebra-based verification methods that act upon hierarchically composed workflows. However, pursuing this is beyond the scope of this dissertation.

Considering that PDD policies are intentionally broadly defined so as to enable expression of domain concepts for diverse stakeholder groups (via DSLs), a single policy language cannot be identified. Furthermore, the effects of PDD policies that inject feature sets (leading to System of Systems integrations) can be opaque to verification techniques. Factoring an arbitrary PDD policy into predicates that can be expressed to a model checker has not been done. Consequently, much work would be required before PDD policies could be verified via proofs.

Similarly, graph techniques such as those applied to Ponder (as described in Section 2.4.6) require a simple and unambiguous set of policy operations, which contradicts the real diversity of PDD-oriented DSLs.

The RBAC model used by Ponder is hierarchical, similar to the PALMS model described in Section 5.6.1.3, but without distinguishing study-based and organization-based roles, and without support for permissions and ACLs. Additionally, Ponder has a concept of meta-policies, which are policies that determine which policies are allowed in a system, including disallowing conflicting policies. Meta-policies are similar to PDD's composite policies (as described in Section 4.2.7), though with different objectives – PDD composite policies determine how multiple filter and control policies compose onto an interaction, while meta-policies determine whether polices are allowed at all.

Note that Ponder and other policy languages enable proof and graph techniques by focusing on logical predicates minimalistically defined. Given the flexibility of the PDD's DSL approach, DSLs designed along the same minimalist logic lines may be eligible for verification via proof and graph techniques, too. However, pursuing this is beyond the scope of this dissertation.

To the extent that policy injection represents a System of Systems composition (as with the MSoD policies in Section 7.4.1, and with the Audit system in Section 5.6.4), unit testing strategies can provide a practical means for achieving confidence that an injected concern functions as intended, and provides a means for testing the composition in isolation. In the PDD context, a unit test would consist of a workflow into which subject policies are injected. For the PALMS implementation, recording results can be implemented with calls from the workflow directly to log4j [189] functions or Audit system services.

Finally, validation activities pertain to the relationship between stakeholders, their requirements, and how their requirements are reflected as stated requirements. This is beyond the scope of this dissertation.

## 7.6 Policy Programmers and Enfranchised Stakeholders

A long term goal of PDD is to enable stakeholders to inject policies into an application quickly, accurately (with respect to requirements), and directly, without depending on the availability of trained programmers to implement (or misinterpret [219]) requirements. Conversely, this goal implies that trained programmers would be relied upon to maintain workflows encoded using traditional programming systems, thereby creating skeletons of applications, which stakeholder policies would customize. Realizing this goal requires a close mapping between stakeholder requirements and high level workflows that can automatically be refined into executable workflows. In turn, this requires both requirement organization disciplines that are still evolving and an understanding of what information must be added or discarded at each step of workflow refinement.

### 7.6.1 Survey of Stakeholder Policy Authorship

To explore the idea of stakeholders authoring and contributing policy directly to workflows, I created and executed an experiment where four domain experts familiar with PALMS were asked to create policies applicable to a high level PALMS workflow [220]. The domain experts were exposure biologists and computer programmers aware of the objectives of PALMS and familiar with PALMS system operations. Preparation for the experiment included:

- a brief tutorial on workflows as expressed using UML Activity Diagrams (using a prominent PALMS workflow)
- examples of access control policies (formulated as a conditional coupled with an alternate workflow)
- examples of filter policies (expressed as filters on request and reply data)

The tutorials included explanations of requirements and the simple policy DSL that expressed them in a PALMS context.

The hypothesis was that a PALMS-aware domain expert could:

- understand PALMS in terms of a workflow diagram
- formulate a requirement that could be enacted in PALMS
- write and inject a policy expression given a textual requirement

To test this, subjects were asked to:

- correlate a text-based requirement with a DSL-based policy expression (and vice versa)
- phrase a text-based requirement given a DSL-based policy expression
- verbally describe how policies could affect a workflow
- author one or more policies
- conceive of a new requirement

As a group, the subjects had mixed success in accomplishing each objective, but they shed valuable light on my assumptions related to articulating policies and composing them onto workflows. In each case, subjects were able to understand the relationship between the UML Activity Diagram and the PALMS workflow they already understood, they were receptive to the relationship, and they could determine where in a workflow an access control policy should be injected. They also understood the translation between a textual requirement and a DSL-based policy expression, and vice versa. One biologist readily wrote a compound policy expression using Boolean logic, and two subjects asked for additional Activity Diagrams depicting workflows into which they were interested in injecting policy.

On the other hand, more than one subject questioned whether the injection of a single access control policy in a single workflow could reliably suffice to fulfill an access control requirement application-wide. Additionally, one subject expressed doubt that a workflow diagram could accurately represent application control- and data-flow, and so was unconvinced that injecting a correct and complete policy in an actual running application would meaningfully affect application behavior.

Several subjects conceived policy only in terms of access control, and not in terms of filtering data flow or feature injection. Given this, the specification of an alternate workflow for a control policy (for returning an access control error) seemed redundant and unnecessary – they believed that access control policies could be stated by predicates amounting to guard expressions. One subject did attempt to write a filter policy, but expressed it as a predicate on the data flow instead of as a function call (despite being familiar with the DSL beforehand).

More than one subject attempted to create requirements that evaluated data not available to the application (e.g., requiring that a user have a certificate from an Institutional Review Board). This created confusion because he didn't realize that not all attributes are available at all times, and that making a particular attribute available is itself a requirement. (This amounts to a requirement derived from a requirement, and is addressed further in Section 7.8.3.)

Ultimately, no subject could consistently express a requirement as a DSL-based policy expression. Reasons for the failures included unfamiliarity with the DSL expression syntax, an imprecise understanding of the data exchanged in an interaction between workflow activities, general disinterest in function calls and alternate workflows, and disinterest in rigorous requirement articulation and decomposition.

Some requirements identified by subjects could not be written as policies injectable into the high level workflow, as the requirement actually targeted an interaction present only in a workflow refinement, which could only be expressed as a decomposition of a workflow activity.

Figure 69 shows an example of policy injection into a decomposed workflow. The figure shows two simple horizontal UML Activity Diagrams stacked one on top of the other. The top Activity Diagram depicts a simple workflow for creating a new PALMS study (and is not related to the workflow described in Chapter 3), where the user creates a study, and then associates a data collection device with the study. The bottom Activity Diagram depicts a decomposition of the Add Device activity, where a user adds a device by listing the available devices, selecting one device, and adding it to the study's device list. The rectangles attached to activity ovals represent data passed to or by an activity, and which is also available for use in a policy expression. (Note that UML Activity Diagrams have a formal notation for activity decomposition, but it is not concise. In Figure 69, slanted lines represent activity decomposition.)

In this example, the stakeholder's requirement is that only a research assistant (RA) should be allowed to add a GPS device to a study, as indicated in the cloud marked "1:". However, the requirement cannot be effected on the high level workflow because the user has not selected any device, and the user could select a non-GPS device. Instead, the requirement is effected on the decomposed workflow, after the device to be added is known.

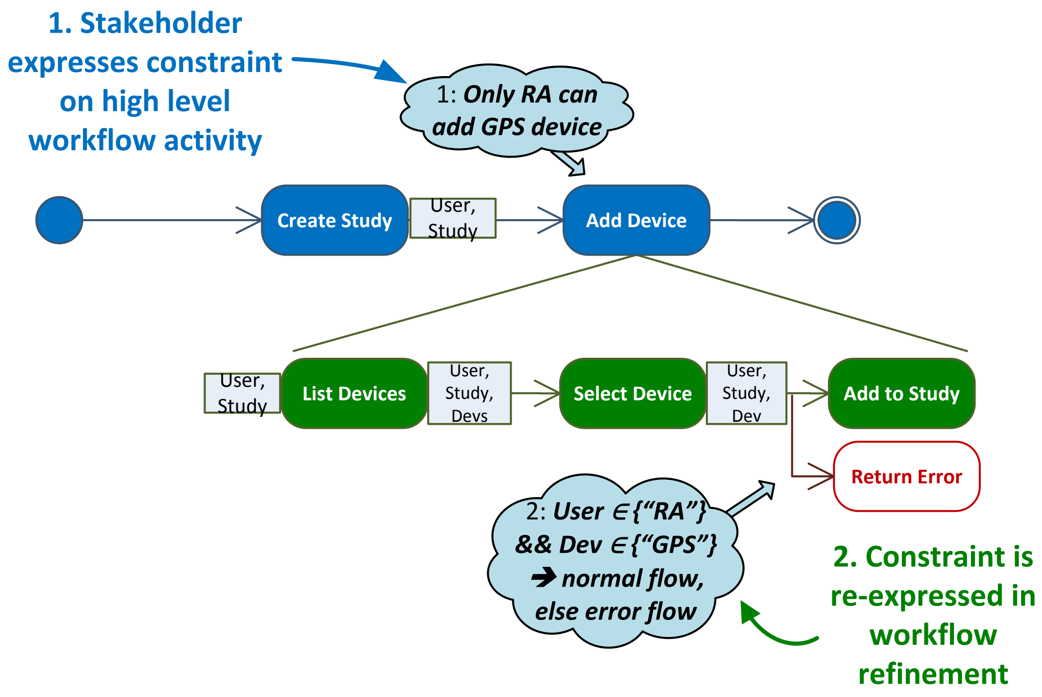

1. Stakeholder expresses constraint on high level workflow activity
1: Only RA can add GPS device
Create Study
User, Study
Add Device
User, Study
List Devices
User, Study, Devs
Select Device
User, Study, Dev
Add to Study
Return Error
2: User ∈{"RA"} && Dev ∈{"GPS"} ➔ normal flow, else error flow
2. Constraint is re-expressed in workflow refinement

Figure 69. Policy Applied to Decomposed Workflow

### 7.6.2 Policy Programming

Knowledge of where in a workflow to inject a policy depends on knowing service contracts for each interaction in the workflow (and are often under-specified), the workflow control flows and data dependencies, the data on which the policy depends, and the intent of the stakeholder. While it is plausible that a stakeholder might be able to determine the interaction on which to inject policy, and then phrase policies formed according to DSL rules, feedback from test subjects indicate this expectation is not realistic for domain experts not constantly and intimately involved with PALMS workflows.

Tool support could empower stakeholders to locate policy injection points and properly phrase policy expressions. For example, a tool could have the following features:

- graphically show workflows, and drill down on workflow decompositions
- show data available on each workflow interaction
- allow form-based policy specification, where a policy expression is automatically composed from form fields
- show workflows composed onto workflows
- maintain an inventory of alternate workflows that can be composed onto workflows

While such an authoring aid would benefit all policy programmers in writing control policies and filters, it does not address the general problem of authoring policy-based feature composition (as described in Sections 4.2.6 and 5.6.4) or creating novel, stateful coordinated policies (such as the MSoD solution in Section 7.4.1). As observed by study subjects, placement of a policy does not guarantee that a requirement is completely addressed.

For these reasons, PDD requires the participation of a *policy programmer* who can both precisely understand application workflows and dataflows, and precisely understand the requirements posed by stakeholders. A person operating in such a role would have software engineering skills that would enable the design, maintenance, and upgrade of DSLs that track stakeholder concerns. The policy programmer would interface intimately with stakeholders to discover and elaborate stakeholder requirements, and review exact policy expressions with stakeholders to verify that policies maintain fidelity to actual requirements. (A defensible analogy is the relationship between a business analyst and stakeholders in designing and implementing business processes.)

The policy programmer role exists in contrast to the role of *traditional programmer*, which I define as someone who uses traditional programming techniques (e.g., agile and waterfall processes, requirements elicitation, and integrated development environments) to create applications that are delivered in installments over time. They deliver application functionality slowly and often without fidelity to requirements from a stakeholder perspective, or deliver applications too slowly to keep up with emerging requirements.

As described in Section 1.3.3 (including Figure 1), the benefit of the policy programmer role is time to market while maintaining fidelity to stakeholder requirements. While traditional programming relies on well-developed tools that generate fast and efficient code and can coordinate with model checkers to deliver basic guarantees, policy programming does not yet have such support. Consequently, it offers a tradeoff between execution time (as described in Section 6.3), strong guarantees (as discussed in Sections 1.1 and 7.5), and time to market. Additionally, as policy programming tools evolve to provide stronger guarantees, more complex policies become more routinely feasible.

For either the policy programmer or the traditional programmer, functional requirements represent opportunities to inject policy (or code) into a limited number of interactions, and provide definitive tests that the application works as intended. In contrast, non-functional requirements define *how* a function must perform, and often result in composition of concerns onto many workflows. This often requires a thorough knowledge of all workflows, and is more likely under the purview of traditional programmers. Section 1.1 addresses complimentary approaches to implementing both functional and non-functional requirements completely.

In practice, on the PALMS project, policies are written and injected by trained programmers filling the policy programmer role as described above.

Additionally, while the traditional and policy programmer roles address separate development concerns, there exist potential conflicts that are not currently addressed within PDD. Workflows and interfaces in traditional applications (without policy injection) are free to change responsive to stakeholder requirements as a consequence of traditional development processes – control and data flows are certified as valid before an application update is released, and their consistency and coherency are the responsibility of the traditional developer. Considering that a valid policy relies on and adheres to the service contract for the interaction into which it is injected, a valid policy can be rendered invalid should the service contract change. This can happen accidentally (as described in Section 7.1.1) during normal development. However, to the extent that service interfaces change rarely, the likelihood of a policy becoming accidentally invalid is low. Furthermore, because the policy system correlates policies to service interactions, detection of an impending incompatibility between new code and existing policy can be detected and reported automatically to an otherwise oblivious traditional developer during the application development process.

### 7.6.3 Complimentary Approaches

During the policy authorship survey described in Section 1.1, I found that engaging subjects in detailed discussions focused on workflows (as Activity Diagrams) and possible policies (as both text and DSL expressions) resulted in the elicitation of then-undiscovered tangential and on-point requirements that could be addressed either at the base workflow or policy level. As a requirement elicitation exercise, a policy walkthrough is similar to the lifecycle walkthrough proposed in [221], which builds a secure system via a walkthrough of resources and a review of their lifetimes. Although [221] focuses on allow/deny decisions related to access control, it can be extended to address request and reply filtering addressed by PDD's pre- and post-filter policies.

Note that Activity Diagrams are not the only notation by which workflows can be evaluated. [219] conducts an experiment that evaluates Activity Diagrams compared to Event-driven Process Chains (EPCs) used by either engineers or end users (i.e., stakeholders) to define a workflow based on requirements, or to understand an existing workflow. The authors showed that Activity Diagrams performed better than EPCs when used by requirements engineers, but could not make a statement regarding their usefulness to end users. Sans an interactive exploration tool (as proposed above), this argues for the premise of a policy programmer having engineering skills.

More than one subject in my study (described in Section 1.1) preferred to state both access control and filter policies as predicates that either enabled access or constrained data flow. As a language syntax, [222] proposes a rule-based, template-oriented declarative language (EARS) to express requirements using natural language while avoiding ambiguity, complexity, and vagueness. In a workflow setting, these rules would appear as preconditions and triggers, which could be further translated to DSL expressions appropriate for the requirement domain. As such, this syntax could contribute to the policy authorship tool proposed above, and could be extended to include feature composition.

The PALMS system was architected based on the Rich Services Development Process (RSDP, as described in Section 5.3), which follows principles of Model Driven Engineering that relate requirements to domain models, service models, and service orchestrations that define workflows. In a robust process, elements of each model are traceable to elements in other models. For example, workflows are traceable to the services that comprise them, which are traceable to elements in domain models. Both workflows and domain models trace to original requirements. These traces can be leveraged to assist in determining whether a policy set provides guarantees that critical requirement properties are implemented. For example, UMLSec [55] and SecureUML [223] both aim to leverage class diagrams annotated with OCL to improve application security by demonstrating complete coverage of security policies. These techniques apply at application design time, but are available to policy programmers to assist in determining PDD policy placement and content. Model-driven security has been studied by a number of groups, and includes UMLSec, security aspects, intrusion detection aspects, AVISPA, SMV, and Alloy, all of which are compared in [224].

Because PDD relies on workflows as models of service orchestration, a rigorous expression of a workflow (in the guise of a rigorously drawn Activity Diagram) provides a combination of data flow specification and activity decomposition useful in the precise placement and definition of policy expressions. To the extent that an Activity Diagram rigorously represents application workflows (as is proposed in [225]) an interactive policy definition tool can be used to help generate a policy that maintains the service contract of the service interaction into which it is injected, though one has not yet been implemented.

Note that given an Activity Diagram that rigorously represents application requirements (as workflows), the automatic decomposition (or refinement) of the diagram into workflows that directly map to services (or other direct implementations) is not well understood, and is therefore typically produced by human programmers in stepwise fashion. However, once such refinements exist, [71] proposes a methodology for verifying that the refinement meets the pre- and post-conditions of the refined diagram.

Similarly, for BPMN diagrams, [226] provides a process semantic for BMPN diagrams based on CSP [113] (discussed in Section 2.2.3), and demonstrates refinement of BPMN-based models (using CSP refinement relations) and automatic behavioral proofs related to such models (using the automated FDR model checker). Separately, [227] and [228] adopt an OWL-based ontological approach to refinement checking as part of larger model-driven engineering products.

## 7.7 Building and Maintaining PDD Applications

As described in this dissertation, a PDD application is a SOA that employs policy injection techniques to compose requirements (as workflows) onto existing workflows, where the content of a workflow reflects a separation of concerns, and workflow composition reflects the creation of a System of Systems. Section 5.3 and 5.4 describe the basic process of framing an application as a Rich Service, which captures the concepts of hierarchical workflow decomposition in the MDA-compatible RSDP process.

A major challenge in creating an application addressable by PDD is in eliciting, organizing, and modeling requirements as separate concerns, which may stand alone or may exist as higher level concerns that combine and orchestrate lower level concerns. While RSDP's service elicitation phase assumes that requirements are gathered, factored into a (possibly UML-based) domain model, and eventually transformed into service-based workflows, RSDP does not prescribe any modeling process to achieve this – it can be chosen or designed situationally.

Modeling processes appropriate for separating requirements as base and composed workflows are provided by the Aspect community [229]. The AoURN [230] is an aspect-oriented variant of the User Requirements Notation [231] (URN) standard that creates a framework that models use cases, NFRs, and other concerns, and provides guidelines for concern identification – it captures concern dependencies, conflicts, and resolutions, resulting in a partitioning of the requirement space that informs subsequent aspect-oriented design. Similarly, [232] describes a repertory grid technique that identifies aspects at the requirements phase based on analysis of terminology and goal descriptions, and [233] describes the EA-Miner tool-based approach for automatically extracting aspect-oriented requirements models from natural language text in requirements documents.

Insofar as MDA envisions and supports the translation of requirements models into domain models, RSDP incorporates this flow as an intermediate step towards a service-oriented workflow model. Existing aspect-oriented techniques [234] [235] support modeling requirements separable from a base model, yet composable onto it, and [108] demonstrates interactive testing and verification of such models. Encapsulation-oriented design rules [104] can assist in formulating base workflows and composable aspects so as to maintain independence that allows both to evolve independently. As a refinement and extension of these techniques, [236] demonstrates a methodology for expressing software product lines as orthogonal concerns in the FOSD domain.

The COMPASS [237] approach combines aspect-oriented requirements elicitation and domain modeling (as application architecture) into a single process, where requirements are expressed in an aspect-oriented requirements specification language (RDL) and transformed into the COMPASS architectural definition language (AO-ADL). Under the RSDP process, the AO-ADL model is manually transformed into concrete services and workflows as Rich Services.

Note that in the RSDP process described in Section 5.3, crosscutting concerns that are discovered during the requirements elicitation phase are carried into subsequent phases, and result in the creation of RISs injected into crosscut service interactions. PDP presents and encourages an additional option: coding crosscutting concerns as workflows selected by injectable policies. Aspect-oriented requirements extraction techniques formalize and improve the detection of such crosscutting concerns as compared to ad-hoc methods. Because crosscutting concerns are coupled to base workflows via restrictive and well-defined interfaces, separate concerns can be developed in parallel RSDP instances, with the expectation of runtime linkage via injected policy (as in Section 5.6.4). During any RSDP execution, it is quite possible that developers would discover conceptualization errors and hidden dependencies, necessitating changes in the base workflow, injected workflow, or both. As a spiral development process, RSDP tolerates this well, providing a graceful path looping back to incremental refactoring and remodeling. Changes that affect workflows (as services) in development in interdependent RSDP efforts can cause similar looping within those processes.

## 7.8 Gaps and Insights

Section 2.7.6 identifies a number of gaps in existing contributions that must be filled in order to implement the PDD vision. Most (but not all) of these gaps are addressed by the methodology presented in Chapter 4 as realized by the case study in Chapter 5 and evaluated in Chapter 6. This section discusses PDD's successes and shortcomings relative to those gaps.

The identification of policy injection sites is addressed as an implementation issue via the Interaction Repository in Section 5.6.2.1, while the actual runtime injection of policy is addressed in Sections 4.2 and 5.6.3, and is evaluated in Section 6.3.

Sections 4.3 and 5.5.4 describe how state is composed onto a workflow and how it can be leveraged by injected policies. They also describe state maintenance according to other lifecycles, including session-oriented and application-oriented (i.e., global) scopes.

The composition of multiple policies onto a single injection site is performed via composition policies, as described in Section 4.2.7 and implemented in Section 5.6.2.2.4 as a reflection of the structure of PALMS community.

The ability of composed workflows to act as base workflows for further composition is inherent in the PDD policy injection theory described in Section

4.1.1.2 and demonstrated in Section 5.6.4. The combination of this composition capability with PDD's context system results in an understanding of composed workflows as Systems of Systems, where both the source and target workflows can be specified, developed, and maintained separately. This results in the scalability and reusability of requirement sets and the development efforts that implement them, consistent with the overall goals of SOAs.

The ability to guarantee a consistent relationship between policy state and policies themselves has been addressed in Section 7.4.6, and remains an unfilled gap in PDD (and other policy evaluation systems).

### 7.8.1 Gap in Service Contract Checking

As described in Section 4.1.2, PDD requires that injected policies adhere to the service contracts defined for a service interaction. Given that a service contract defines the validity of input channels consumed by a service and output channels produced by a service, failure of a policy to adhere to a service contract can have unpredictable (including erroneous) results. (Additionally, adherence to service contracts is no guarantee that a policy actually implements functionality faithful to a requirement.)

Under PDD, policy adherence to a service contract means that the policy decision and the workflow it selects must consume the same input channel (i.e., input message) as the target service, and the selected workflow must produce the output channel (i.e., output message) expected by the source service (per Section 4.1.3). Under PALMS, channels consist of Java objects, so a service contract specifies channels consisting of typed Java objects.

The description of a service contract in PALMS is simplistic (as described in Section 5.6.2.1) and functional, but cannot be used to guarantee that a policy fulfills a service contract. The PALMS service contract specification identifies the type of a valid response message acceptable to the source service in a simple request/reply interaction pattern. More complete specifications would represent both the source and target services, and might include message types (as a syntactic interface), channel definitions (as a primitive semantic interface), and interactions (e.g., MSCs, as a more context-laden semantic interface), each leading to different guarantees. (Interaction specifications would cover not only interactions involving a given service pair, but a history of other interactions that lead to it, and could also depend on the actual channel contents for a particular interaction.)

Sans any contract checking, it is up to a policy itself to verify that its input channel is valid for its decision and the service it returns, and that the selected service produces a valid output channel. This is a weak guarantee.

In strongly typed languages (e.g., Java), syntactic guarantees for injected functions (e.g., functors) would be enforced by a compiler for both input and output channels (based on function signatures), and model checking would demonstrate semantic assertions (based on channel properties, represented by actual parameter values).

Achieving syntactic guarantees for PALMS policies would require that service interface specifications (e.g., Java message types) be specified for policy decisions, service alternatives, and source and target services in base workflows. Static (compiler-style) contract checking can tentatively evaluate the compatibility of a policy with a service interaction at policy definition time, though a more reliable check would occur at runtime during policy injection. Semantic checking can be performed at runtime as dynamic model checking that compares service interactions (including policy injections) to an interaction specification [238].

A strongly typed language provides guarantees based on inspection of function source, which can reveal whether a function plausibly meets a syntactic interface through inferences that typed input channels can generate typed output channels. The XQuery compiler (that processes XQuery-based PALMS policies) provides typed function interface guarantees pertaining to *XML structures* (which differentiate between a string, a list of string, or a tree of nodes), but not *Java object types* (which describe PALMS messages). So, there is no simple way to verify that an XQuery-encoded decision or selected workflow adheres to service interface specifications.

Alternatively, extending strong type checking onto a policy (including the decision and alternate workflows) can be achieved by changing the policy language to one whose structure and semantics align with PALMS messages (i.e., Java objects) and includes strong type checking, yet fulfills the criteria that led to the selection of XQuery as PALMS' policy language (i.e., message processing, runtime injectability, and accessibility to stakeholders and policy programmers, per Section 5.6.1). One possibility is Groovy, which accepts and emits Java objects. Given the availability of embeddable Java compilers, a combination of Java and a higher level macro facility may also suffice for this. Using Java would admit additional pre-existing model checking systems that could guarantee semantic properties of decisions and alternate workflows. Alignment of the policy language with interaction messages would also address substantial performance penalties incurred through the use of XQuery, as discussed in Section 6.3.7.

PDD does not address how service contract checking might be performed, and PALMS does not implement it -- this represents future work.

### 7.8.2 Gap in Testing and Fidelity Assessment

As a methodology, PDD offers means for framing requirements as policies and injecting them upon unprepared workflows. However, PDD gives little guidance in the testing and verification of such policies either as to their fidelity to requirements or their relationship to base workflows.

As described in Section 7.5, a number of modeling and simulation techniques can be adapted to help demonstrate basic policy guarantees such as liveness and reachability, provided that policy decisions and workflows are modeled at the interface level.

However, the fidelity of a collection of policies to a requirement is left unaddressed. Model Driven Engineering techniques have proven effective in defining and refining requirements leading to application architectures [8] [239] [240], including RSDP as described in Section 5.3.

While such techniques can be used or adapted to produce and test policies, as to PDD, they represent future work.

### 7.8.3 Requirement Feedback Loops

A variant of the problem of defining and realizing requirements occurs when attempting to inject a policy onto a workflow, where the policy must rely on information not available to the workflow or must produce information not handled by the workflow. Examples of this are described in Sections 5.6.2.1 and 1.1, which provide signals that the requirement implemented by the new workflow interacts with base requirements, and implies that the base requirements (and the workflows that implement them) are themselves incomplete and must evolve. Such requirement feedback loops are within the purview of MDE techniques described above.

### 7.8.4 Implementation Platforms

As described in Section 1.3.2, taking advantage of late decision binding requires that workflow interactions be exposed for interruption and modification at runtime. PDD is framed in SOA terms because SOAs provide these capabilities at a conceptual level, and ESB frameworks (as platforms for SOA execution) enable their realization. However, to the extent that service interactions can be exposed and interrupted in other frameworks, PDD's policy injection can provide composition value in a workflow context. When Web Services are mediated by SOAP-based remote procedure calls (RPCs), and are implemented using proxies at both the source and target service, PDD can be implemented by replacing both the source and target proxies with new proxies that pass parameters as part of an interservice message, with the target proxy also calling a policy evaluator service (as in Sections 4.3 and 5.5.4). Unlike implementation at the ESB level, this requires that developers have control over the RPC proxy libraries linked to a Web Service, and this is often not the case. Implementing PDD under Web Services is beyond the scope of this dissertation.

## 7.9 Summary

In this chapter, I compared PDD to a number of composition, context, and policy evaluation approaches, and discussed a number of issues bearing on a robust deployment of policy within real world environments.

In Section 7.1, I compared PDD to Aspect Oriented Software Design and Policy-based design, and observed that both methodologies apply crosscutting concerns at the program design and compile stage. Furthermore, both methodologies focus on optimization of programming processes and concerns, and not on mapping of requirements onto workflows. Therefore, they do not directly support the runtime policy injection, workflow context support, and feature injection support provided by PDD for general requirement composition onto workflows.

In Section 7.2, I compared PDD's context management features to state management in important distributed processing paradigms, including Struts, REST, and AJAX. In each case, I demonstrated various degrees and types of support for workflow and distributed state, showing that none of these paradigms support these capabilities sufficiently to enable runtime-based workflow composition.

In surveying prominent workflow and orchestration languages and systems (in Section 7.3), I demonstrate they generally provide no functionality that enables runtime workflow injection, they are no scalable, they provide no workflow context support, or all three. Consequently, they are not well suited for runtime composition of stakeholder requirements onto existing workflows.

I compared PDD to important features in a number of policy evaluation systems in Section 7.4. Whereas PERMIS is a well-developed policy evaluation and management system, it supports a PDP/PEP policy model that integrates policy evaluation with applications at the source code level at the time of development and compilation. While xESB allows the composition of policy onto running systems, it provides no support for requirement-oriented workflow injection. Both PERMIS and xESB evaluate policies as first order logic expressions, and not as DSLs designed to enfranchise stakeholders.

In Section 7.5, I surveyed various program verification techniques and strategies as applied to policy and policy-injected workflows, and discussed additional work needs to be done to enable validation of policy-mediated workflow injections.

Finally, in Section 7.6, I presented a vision of a policy programmer whose activities focus on realizing stakeholder requirements at runtime, where such requirements can be expressed as the policy-based composition of workflows onto base workflows, given a knowledge of (but not the ability to change) interactions in base workflows. This role contrasts with the definition of a traditional programmer, who is free to realize requirements as new workflows with new service interactions, implemented at the source code level, with attendant deployment delays and risks.

# CHAPTER 8

# SUMMARY AND OUTLOOK

In the previous chapters, I presented a new development methodology called Policy Driven Development (PDD), which is my approach to designing complex systems (particularly cyberinfrastructures) so as to improve their evolvability over time. By nature, a cyberinfrastructure serves a community of multiple collaborating stakeholder groups, each with a stream of different requirements that must be met quickly in order for stakeholders to remain engaged and for the cyberinfrastructure to thrive. Paradoxically, as the cyberinfrastructure becomes larger and more complex, requirement realization and re-deployment require more time, potentially creating long lag times between the articulation of stakeholder requirements and their realization in delivered systems. These delays can be traced to the practice of binding requirements into an application early in the development process (so-called *early binding*), which is common to many current design and programming techniques.

PDD seeks to improve cyberinfrastructure evolvability by dramatically reducing the time needed to realize stakeholder requirements. PDD's overall strategy starts with viewing a cyberinfrastructure as a collection of base workflows that implement partial behaviors, and viewing a requirement as a collection of workflows that also implement partial behaviors and can be conditionally composed onto the cyberinfrastructure. Requirements can represent constraints on workflows, concerns that crosscut multiple requirements (and workflows), or standalone applications that implement substantial feature sets. As such, requirement composition represents a System of Systems integration, where both sets of workflows represent partial behaviors composed together.

A conditional composition involves a decision and a set of alternative workflows, and is called a *policy*. The two key insights of PDD are a) to express a policy in a Domain Specific Language (DSL) custom-designed to align with the stakeholders' view of a requirement, and b) to inject a policy into a base workflow at runtime (so-called *late binding*). PDD proposes a new programmer role (called *policy programmer*) that designs a DSL responsive to stakeholder requirements, collaborates with stakeholders to express requirements as policy, and injects policies into an executing system.

Critically, PDD provides mechanisms that a) enable the composition of multiple policies (representing requirements tendered by independent stakeholders) onto the same base workflow, b) enable injected workflows to function as base workflows onto which new workflows can be composed, and c) enable related policies to maintain and share state.

An application built using PDD principles can be expressed as a Service Oriented Architecture (SOA), where workflows are represented by service orchestrations, and all service interactions are subject to policy injections.

My contributions to PDD are described below, and include:

- An engineering approach to the realization of stakeholder requirements in SOA-based cyberinfrastructures (CIs) via runtime policy injection
- A demonstration of a SOA-based CI (PALMS) that enables runtime policy injection
- A demonstration of the creation and use of Domain Specific Languages (DSLs) to articulate injectable policy
- An evaluation of runtime policy injection (in the context of PALMS)
- An evaluation of the use of DSLs (in the context of PALMS)
- Insights for improving the performance of injected policies and widening the stakeholder audiences they address

In Chapter 2 of this my dissertation, I first surveyed the means by which choice (as policy) and workflow composition are represented in existing contributions, including computational models, software development methodologies, implementation mechanisms, and pertinent pattern sets. Chapter 2 presents an analysis of gaps in existing contributions that must be addressed in the course of realizing PDD.

Chapter 3 presented the GetStudyList workflow as a running example drawn from the PALMS case study in Chapter 5 and used to motivate and illustrate features of PDD beginning in Chapter 4.

In Chapter 4, I presented a Rich Service-based foundation that addresses the key PDD issues responsive to the gaps identified in Chapter 2. At a high level, I defined policies in the abstract, defined semantics for a service interaction, and defined the semantics for the injection of policy into a service interaction. I distinguished between three types of policies, where control policies influence control flow, filter policies customize data flow, and composition policies determine the effective control or filter policy when more than one of these policies is defined for a particular service interaction. Finally, I defined a context system that composes workflow-based data flows – the dual of control flows – to enable policies to maintain state valid across a workflow in a distributed (and scalable) system.

As a real world demonstration of PDD foundations, Chapter 5 presented the PALMS Cyberinfrastructure (PALMS) as a Rich Service implemented on an Enterprise Service Bus (ESB) and supporting a worldwide community of researchers. PALMS implements DSLs based on XQuery syntax and using XQuery-based libraries, with DSLs for access control, policy composition, auditing, and general feature injection. I also described the specific ESB-based message interception mechanisms by which PALMS policies are injected into PALMS workflows, including how interaction and policy repositories function to deliver actionable policy during workflow execution, how policies are evaluated and enacted in a service interaction, and how the context system is implemented to support workflow state for the duration of a workflow and for other lifecycles.

In Chapter 6, I evaluated the PALMS PDD implementation along key performance dimensions, including contributions to evolvability, ease of stakeholder use, and execution speed. PALMS' PDD implementation demonstrated substantial productivity gains (10x) in the maintenance of PALMS workflows (via OOP techniques) by making use of both composed workflows and data flows. I also demonstrated a process whereby a key DSL was evolved to enable greater synergies between stakeholder and policy programmer, thereby enabling easier and more fluid access control policy maintenance. Finally, I reported on a number of time tests that profiled policy execution in common scenarios. They demonstrated acceptable execution costs for typical combinations of control policies, filter policies, and message payloads, and they revealed high costs for processing complex message payloads (which I address further in Chapter 6 and later in this chapter).

In Chapter 7, I compared PDD (and its PALMS implementation) to the existing contributions identified in Chapter 2, particularly regarding how it addresses the technology gaps I identified. I found that PDD filled gaps relating to workflow injection and workflow state management, while leaving policy modeling, secure deployment, and verification issues unaddressed as further discussed in Section 8.1. I described how a user study informed my definition of the role of the policy programmer as distinct from a stakeholder who defines and injects policy directly. In addition, I surveyed a number of approaches that are complimentary to PDD and which can be used to refine requirements and workflows, and I demonstrated critical properties of particular workflow compositions, including correctness and completeness at either the model or code level.

Together, these results demonstrate that PDD techniques can enable the rapid composition of requirements (represented by policies) onto applications designed and implemented according to PDD principles, thereby dramatically improving the evolvability of the overall system, inuring to the benefit of existing and emerging stakeholder communities. However, insofar as policy programming opens a new (and parallel) front in the relationship between programmers and stakeholders, it presents challenges to process, design, and code management; to verification; and to secure and consistent deployment as discussed in Section 8.2.

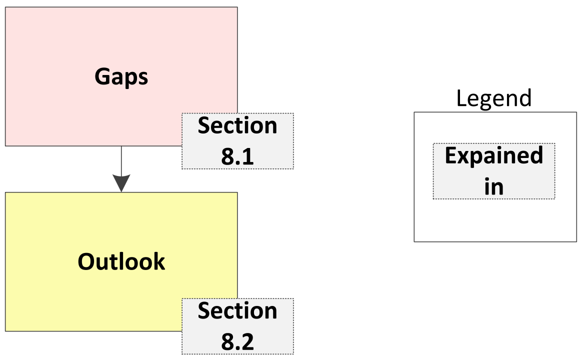


Figure 70. Chapter 8 Flow

As shown in Figure 70, the remainder of this chapter is devoted to describing the gaps addressed and unaddressed in this dissertation (Section 8.1) and the outlook for further PDD development (Section 8.2).

## 8.1 Gaps

In Chapter 2, I summarize my analysis of existing contributions that might bear on a comprehensive technical solution to cyberinfrastructure evolvability using a late binding strategy. In this section, I reprise the summary as a roadmap to such a solution, and as a way of measuring the contribution of this dissertation towards that solution.

Table 11. Gaps Addressed in this Dissertation

| Addressed | Gap |
|---|---|
| **Yes** | Identification of policy injection site at runtime |
| **Yes** | Injection of policy at runtime |
| **Yes** | Tracking workflow-based policy -centric state |
| **Yes** | Composition of multiple policies onto a single injection site |
| **Yes** | Enabling composition onto injected workflow |
| **No** | Verification of interface and semantic compatibility between policy and base workflow |
| **No** | Incremental testing and proofs that policies implement requirements |
| **No** | Enabling a consistent relationship between state and policy across policy deployments |

As shown in Table 11, of the eight gaps I identified, PDD solves the five that pertain to the foundations (as described in Chapter 4) and mechanics (as described in Chapter 5) of policy definition and injection. Additionally, PDD demonstrates the use of DSLs as a means for engaging and enfranchising stakeholders directly in the policy programming activity.

The remaining gaps (and other issues) are discussed in Section 7.8 and are further addressed in Section 8.2.

## 8.2 Outlook

I began this dissertation with a look back at the ancient SOARS system, which played the role in 1975 that a modern cyberinfrastructure would play today. I described how SOARS was functional but incapable of evolving quickly, and how programmers made many decisions that were more appropriately made by domain experts (as stakeholders). Since then, the field of software engineering has produced significant advances in disciplines, techniques, methodologies, and tools (as described in Chapter 2) responsive to stakeholders' increasingly complex requirements while reducing software development risk.

As large systems evolve from being productivity tools into being the substrate on which communities develop, they grow to reflect the relationships between stakeholder groups comprising these communities. Ultimately, I believe that large software systems (e.g., cyberinfrastructures) that support dynamic and interconnected communities will coevolve with the community to function as vehicles for the integration of complex systems as Systems of Systems (SoS). To the extent that such systems can evolve quickly and reliably responsive to new and changed stakeholder requirements (including new value drivers), these systems can support the vitality of such a community. Conversely, when they react slowly or non-responsively, they threaten community vitality.

In this dissertation, I have defined and demonstrated the Policy Driven Design (PDD) methodology, which supports the design and implementation of large scale systems able to respond to new and changing stakeholder requirements quickly and accurately. By enabling the composition of requirements (as workflows) at runtime via policies articulated using DSLs, PDD enables the formation and transformation of complex systems, and it encourages stakeholders to participate directly in requirement realization that produces rapid system evolution.

However, PDD is young and has yet to adopt processes and mechanisms that enable robust end-to-end guarantees required in a mission-critical industrial setting. The remainder of this section is devoted to discussing opportunities for extensions and improvements to PDD that could enable PDD's use in this environment.

PDD policy authorship and injection represent a separate and parallel activity relative to traditional system development processes. This presents significant challenges in maintaining coherence between the PDD-based and traditional development tracks. These challenges can be addressed at both development and deployment time. Chapter 7 identifies and discusses issues in maintaining *a priori* a semantic match between these tracks by focusing on a shared and consistent view of service contracts via Model Driven Development (MDD) and language definition techniques. It discusses similar issues at deployment time via a combination of MDD, modeling, model checking, and simulation techniques aimed both at producing policies that align with requirements and demonstrating the fidelity between requirements and policy execution. PDD must leverage these techniques in order to demonstrate end-to-end fidelity to requirements – there is much work to be done here.

Similarly, consistency issues arise across multiple generations of a single policy collection, particularly where policies are stateful, and the states' semantics are not consistently observed from one policy generation to another. This is discussed in Chapter 7, as are issues pertaining to secure and consistent policy deployment. The importance of these issues increases as a system scales into a highly distributed environment and policy programmers become adept at requirements discovery and injection, while ignoring the complexities of application distribution that systems and application programmers routinely manage. There is much work to be done here.

While Chapter 7 describes the process of building an application under PDD principles, this dissertation does not specifically address the process or prospects for using PDD principles on large scale systems not created using PDD principles. Additionally, real world Systems of Systems are often assembled from not only new (possibly PDD-observant) components, but from a mix of legacy services and other non-PDD services. As a practical matter, this common case must be addressed, perhaps using the implementation platform discussion in Chapter 7 as a point of departure.

In Chapter 6, I evaluate a group of performance-related hypotheses relative to tests performed on the PALMS implementation of PDD. While timings indicate that PALMS' PDD implementation performs acceptably well for typical combinations of control policies, filter policies, and message payloads, they reveal high costs for processing complex message payloads. The primary cost driver is the size and complexity of messages exchanged during a service interaction intercepted by a policy evaluation – such messages incur significant translations costs between Java object form (suitable for service interactions) and XML form (suitable for XQuery-based policy evaluation). By harmonizing interaction message formats with the underlying policy execution language, the costs of policy evaluation for these cases can be better reflect their benefits. There is much work to be done here.

As designed, PDD contemplates policy injection on service interactions where the participating services can be uniquely named. While this injection criteria is sufficient for many cases, other criteria can be leveraged to refine the injection decision for greater specificity, including a history of previous interactions and an evaluation of channel history, as described in Chapter 7. Similarly, while the result of a policy evaluation (as described in Chapter 4) is defined as a workflow, the PALMS implementation of PDD returns only a service, which acts as a workflow proxy that manifests an actual workflow through decomposition. Under PALMS, it is difficult for a policy to calculate and return a workflow dynamically – PALMS provides no way to express a workflow object or to execute one directly. Furthermore, if it did, identifying policy injection points within the dynamic workflow would require further study, as such workflow interactions might not be identifiable in advance, when policy authorship occurs. This limits deep SoS composition available for non-dynamic services, and suggests a need for pattern-based injection criteria based on behavior or workflow structure. Additionally, the possibility of executing dynamic workflows may result in revisiting the choice of using an ESB to orchestrate PALMS' workflows, as ESBs (so far) do not execute dynamic routings, and it may make little sense to have an execution engine for base workflows distinct from injected workflows. These considerations provide food for thought in choosing a next generation PDD execution platform – such a project may also encompass the choice of message format as presented above. Answers to both questions may arise out of combining emerging functional programming-based frameworks such as OpenRichServices [241] with yet-to-be-developed workflow pattern matching facilities. There is much work to be done here.

Insofar as policies represent separate concerns, maintaining policies as separate, composable entities removes the risk of entanglement and scattering represented in early binding approaches. However, PDD does not eliminate entanglement and scattering – it relocates them to the composition policy level, which seeks to resolve execution ordering and administrative priority issues when two policies are injected into the same service interaction. Significantly, composition policies serve to encapsulate this entanglement so it can be addressed definitively as a separate concern. Additionally, while scattering still occurs when elements of a policy collection are injected into different interactions, managing the consequences of scattering can still be done at the composition policy level. Ultimately, PDD encapsulates the issues of entanglement and scattering, but does not eliminate them -- understanding management of entanglement and scattering is an area for further study.

In summary, the PDD methodology and my experiences in using it to build a real-world cyberinfrastructure strongly suggest the value of building complex systems as flexible and highly evolvable Systems of Systems by leveraging a late binding paradigm to achieve composition at runtime. While stakeholders in a modern SOARS system would reap benefits from participating in policy programming via quick and accurate requirement realization (especially assuming the improvements to PDD described above), the approach has potential to pay future dividends should the university align itself with other institutions either virtually, ephemerally, or both, where PDD principles can be employed to realize computing systems that parallel and support inter-institutional relationships.

This scenario suggests that the true value of PDD's late binding approach is to preserve and unleash value that is currently lost in non-PDD systems due to choices made prematurely via early binding.

# APPENDICES

## APPENDIX A – Graphical Notations

In this dissertation, I use both standard notations and ad-hoc notations to convey important relationships. In this appendix, I explain these notations.

### A.1.1 Unified Modeling Language

The Unified Modeling Language consists of modeling standards defined and maintained by the Object Management Group [242] (OMG), which manages standards useful in the practice of Model Driven Architecture [171] (MDA)[5]. Over time, the OMG has published a number of UML versions, with each version defining a number of modeling diagrams, their form, and their semantics. Each diagram can be used to model some aspect of a software engineering project. According to [243], UML v2.2 contains 14 types of diagrams covering both application structure and behavior. In this dissertation, I use UML class diagrams to show structural relationships, and both UML sequence and activity diagrams to show behavior. The following sections give brief tutorials on my use of these diagrams, which uses only a subset of UML. The full UML language offers much richer features and semantics useful outside of this dissertation. A more extensive tutorial is presented in [40], and UML style tips are presented in [244].

[5] OMG defines MDA as "*an open, vendor-neutral approach to the challenge of business and technology change*" … which "*separates business and application logic from underlying platform technology*".

Note that common use of UML varies from the very informal (back-of-napkin) to the very formal (directly executable). In this dissertation, I use UML informally as a way of defining relationships for the sake of discussion, but not formally enough to use in the context of other formal models.

Note that UML allows the use of color in class diagrams, but allows each diagram to define its own color scheme. In my diagrams, I use color to group related elements.

### A.1.2 Class Diagrams

A class diagram depicts a collection of related entities and the ontological relationships between them, and enables reasoning about the entities and their relationships.

In my class diagrams, an entity is an abstraction representing an information container and a collection of pertinent methods. It may be possible to instantiate the entity, or the entity can be used as a component of another entity's definition or its content. Class diagrams are commonly understood in terms of Object Oriented Programming (OOP) principles, where an entity equates to a *class*, and relationship is represented by an *association* that may relate two or more classes. Examples of OOP associations include subclassing, referencing, and encapsulation, all of which are binary relationships. Domain-specific associations are often binary, though they can be n-ary.

As shown in the class examples in Figure 71, all classes have names, and may have data elements and/or methods. They are represented visually by rectangles, with separate regions for class name, data elements, and methods – empty regions can go undisplayed. Classes can be adorned with *stereotypes*, which endow class properties and behaviors that are defined elsewhere.

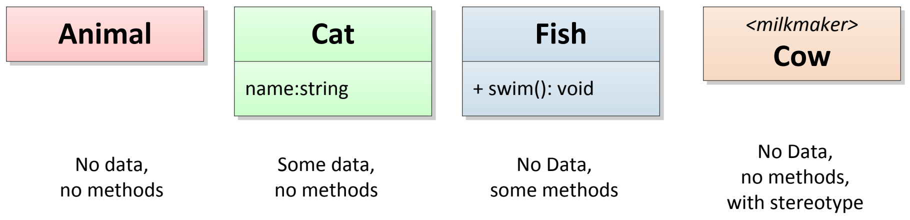


Figure 71. Styles of UML Class Entities

A number of association examples are shown in Figure 72. Binary associations are displayed as lines between classes, where classes play roles within an association. For relationships defined under OOP, lines have different appearances to indicate different kinds of associations. Figure 72 shows a number of OOP relationships: the `Cat` and `Fish` classes are subclasses (i.e., kinds) of `Animal`; a `Cat` is associated with (i.e., references) a `Bed` (though the `Bed` is not part of the `Cat`); and a `Cat` has (i.e., contains) `Legs`, each of which are part of the same `Cat`.

An association can have a descriptive name, and when a class fulfills a role in an association, the class can be annotated with the name of the role it plays. In the example, Eats is an association that applies to the Cat (as Predator) and Fish (as Prey).

If an association places a constraint on the number of instances that can fulfill a role, the association can be annotated with a multiplicity indicating the constraint (with "*" indicating any number, including zero). In the example, many Cats can eat a Fish, a Cat can eat many Fish, and a Cat can have between zero and four legs.

Finally, if a role is oblivious to other association roles, the line that connects to the class ends in an open arrow. In the example, the Bed is oblivious to the Cat.

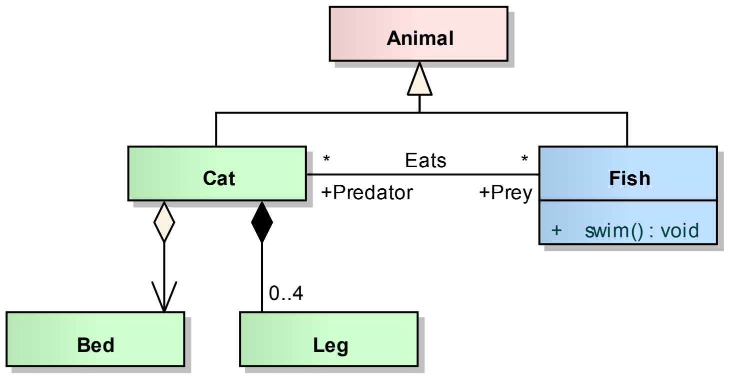


Figure 72. UML Associations

For n-ary associations, lines are connected through a diamond, and is shown in Figure 73 as an Ecosystem relationship between a River that flows, a Cat that sips water, and a Fish that swims in the water.

An association class is a combination of an association and a class. It not only defines relationships between roles, but also defines data and methods pertinent to the relationship. Figure 73 shows that the `Fish`'s `Swim` relationship is adorned with a `Vector` indicating `direction` and `speed`.

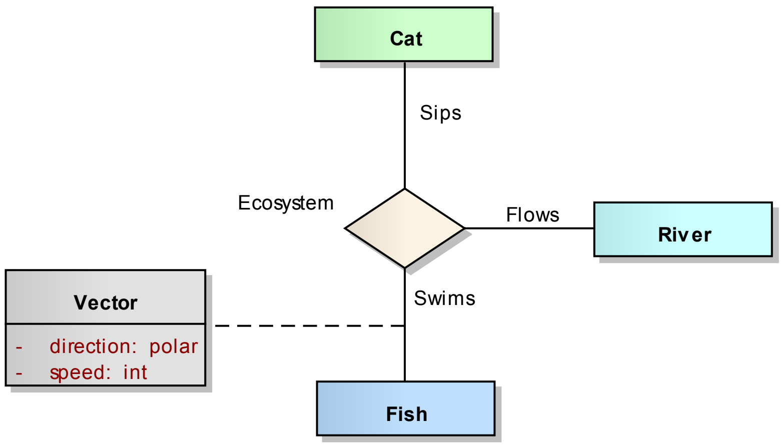


Figure 73. n-ary Association and Association Class

## A.1.3 Sequence Diagrams

A sequence diagram depicts a collection of roles and the interactions between them, and enables reasoning about the entities and their interactions.

In my sequence diagrams (exemplified in Figure 74), a role represents an instance of a class (shown as a rectangle) coupled with a *lifeline* (shown as a drop-down line terminate by a black mark). The lifeline represents time, which starts at the top of the line and proceeds toward the bottom. Two roles can interact if one class (i.e., the source) is capable of sending a message and the other class (i.e., the target) is capable of receiving it. In this context, a message represents a typed communication (including a structured packet, a parameter list, or other data-laden entity). Interactions are shown as events positioned on the lifeline, thereby establishing interaction ordering and proving a basis for discussions of causality.

As shown in Figure 74, the name of the role is the same as the class it represents, and can be qualified by an instance identifier for the sake of differentiating roles having the same class. An interaction is represented by a directed line originating at a source role and terminating at a target role, and intersects a lifeline below all previous interactions. Different line patterns and arrows represent assumptions regarding message transmission and relationship to other messages. In my sequence diagrams, a solid line with a solid arrow indicates a request, and a dashed line with an open arrow indicates its reply. A solid line with an open arrow indicates an asynchronous request, which may or may not have a reply. I annotate each interaction with the message contents it represents.

Figure 74 shows `Role1` sending a request `Msg1` to `Role2` (at the top), with `Role2` replying `Resp1` (at the bottom). After the request, and before the reply, `Role2` sends a request to `Role3`.

A sequence diagram that contains mutually exclusive sequences of messages groups alternative sequences in compartments in an Alt box, where one compartment is separated from another by a dashed line. This is shown in

Figure 74 as `Role3` choosing to return either `Resp2a` or `Resp2b`, depending on some criterion evaluated by `Role3`.

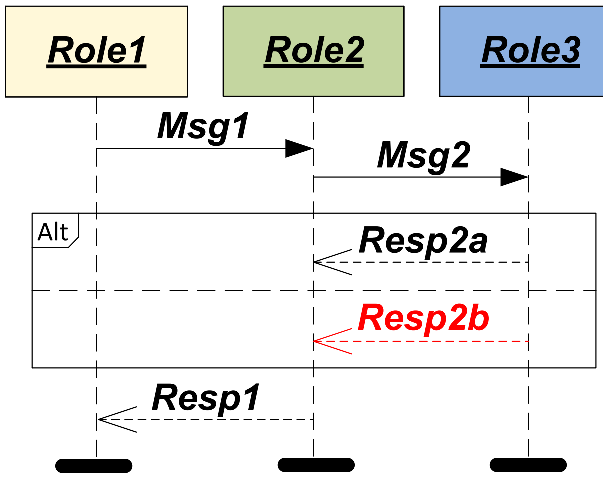


Figure 74. Sample UML Sequence Diagram

### A.1.4 Activity Diagrams

An activity diagram depicts a workflow as a collection of activities related by flowing data, and enables reasoning about the control flows and data flows. At a high level, an activity diagram begins, performs processing (via one or more activities), and ends. Activities themselves can be decomposed along the same lines.

Figure 75 shows a simple activity diagram that models a dog grooming service. The service begins when the dog arrives (marked by a black *initial node*) and ends when the dog leaves (marked by a hollowed *final node*). The service itself accepts an object (i.e., Dirty Dog) and produces an object (i.e., Done Dog), and exists as a decomposed activity that transforms input to output. The lines connecting nodes and activities are directed to indicate flow, and implicitly carry an object that is supplied by the source and consumed by the target. For clarity, the type of object produced or consumed can be articulated by joining a *parameter pin* rectangle, a line, and an activity, then labeling the pin with the type of object type (e.g., the Dirty Dog and Done Dog pins).

My activity diagrams model time in simple terms – there is no time quantum, and when an object is produced by a source activity, it is consumed by a target activity. An activity diagram defines a workflow that can be instantiated many times, where each instance executes independently of other instances, maintains its own state, and processes its own data.

The Dog Grooming activity consists of a number of activities that consume and produce objects (in this case, a dog). A directed path can be qualified by a guard expression (displayed within brackets, e.g., Healthy vs Unhealthy), and can be forked into multiple paths that execute independently (displayed as a black bar that consumes an object and duplicates it along independent flows, e.g., the Wash/Dry, and Clip Nails paths). Independent paths can also be joined (displayed as a black bar that consumes an object on any of several paths, and forwards it to a downstream activity).

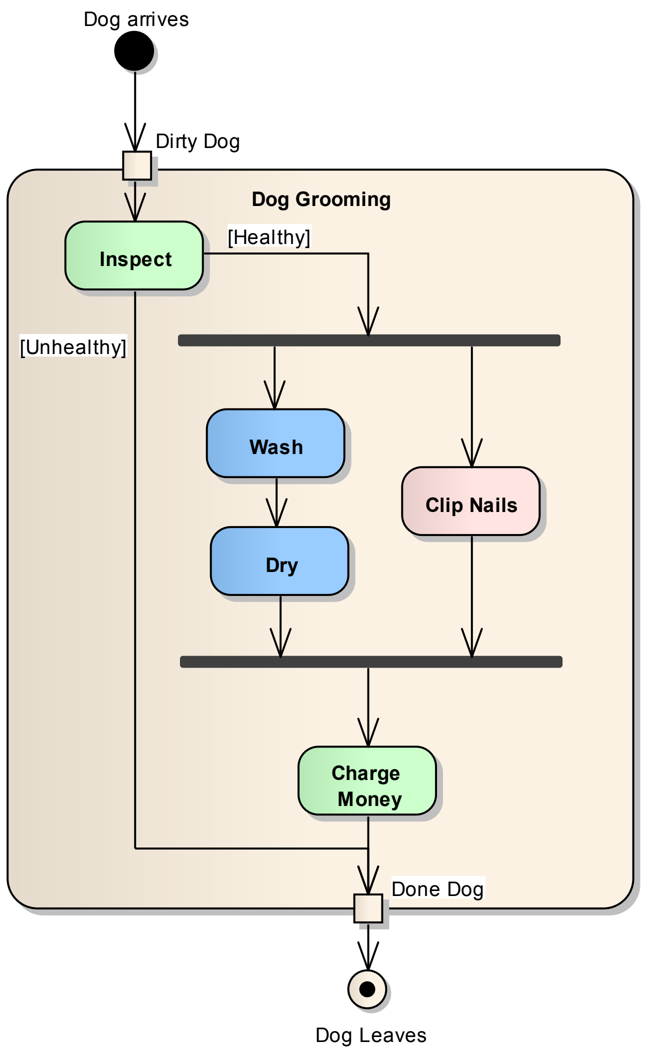


Figure 75. UML Activity Diagram

## A.2 Workflow Sketch

A workflow sketch is my own notation for compactly representing interactions between activities (in a workflow model) or, equivalently, services as representations of activities. Activities are represented by named rectangles, and messages are represented by directed lines, as shown in Figure 76. A request/reply pattern is indicated by two services sharing two directed lines (as shown for `Activity1` and `Activity2`). Each message line (called an *interaction*) is marked with a numeral, which is unique to the sketch.

When appropriate, a sketch can contain a legend that correlates a message line with message content, thereby indicating the data exchanged between two activities. For example, interaction ❷ passes a message called `Msg2`, which contains values `c` and `d`.

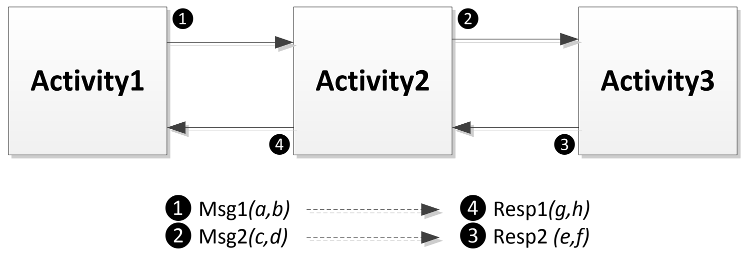


Figure 76. Workflow Sketch

Note that a workflow sketch shows only activity interactions, but does not specify the composition or decomposition that produced those interactions. For example, in Figure 76, each of the three activities could be elements of a single workflow. It is also possible that Activity 1 decomposes into Activity 2 interacting with Activity 3. In a workflow sketch, interactions are the focus of the discussion, and not the structure of the workflows that produced the interactions.

# APPENDIX B Existing Contributions

In this dissertation, I compare facets of PDD to existing contributions, which can be categorized as computational models, software development methodologies, mechanisms, and patterns. In this appendix, I introduce these contributions so as to position them in a discussion on choice, workflow specification and composition (as injection), and state management, verification and modeling, and deployment elsewhere in this dissertation.

## B.1 Models of Computation

Computational models are generally not intended as programming languages used to author and deliver applications. Instead, they abstract out some aspect of an application so as to answer particular questions about it (e.g., determining liveness, safety, or reachability) without enduring the complexity introduced by issues unrelated to the question. However, as logical frameworks, they underlie and inspire important features of programming methodologies, mechanisms, and techniques that can either support PDD or can be recognized as insufficient to support PDD, and thereby provide inspiration and theoretical support for components of PDD. In this section, I describe the computational models and process algebras that inspire PDD and provide counterpoints to it.

### B.1.1 Turing Machines

A Turing machine is a very basic process description first proposed by Alan Turing in 1936 [84]. Turing machines (and related finite state automata) express computing processes as transitions between states, where a start state represents the beginning of a computation, and accept and reject states represent the end of a computation. Based on input data, a Turing machine sequences from state to state (including intermediate states) according a transition function, and it generates output data (which may be subsequently used as input data, depending on the type of Turning machine in use).

The fundamental purpose of a Turing machine is as a framework for organizing and evaluating the inputs, state spaces, state transitions, and output so as to determine whether particular output can be generated from particular input, and to determine the time and space needed to achieve this.

### B.1.2 Petri Nets

A Petri Net [78] [87] is a graphical notation invented by Carl Adam Petri in August, 1939, to describe states and transitions in chemical processes; it has been extended in a number of forms to model computing processes. At its heart, a Petri Net is a relationship between *places*, *transitions*, and *arcs*, all of which cooperate to manage the flow of data elements called *tokens*. A place represents a state, and is a repository for tokens – one or more places are connected via directed arcs to a transition. Relative to a transition, places at an arc source are called *input places*, and places at an arc destination are called *output places*. A transition is *enabled* (and can *fire*) if all input places connected to it contain at least one token. When a transition fires, it removes a token from each input place, and places tokens on each output place. A token present in a place represents a *condition*, and a transition represents an *event* – an event can fire if all conditions are met. A particular configuration of tokens on each place in an entire system is called a *marking*; a marking represents a system state.

In a Petri Net, time is modeled as a forward-only sequence of quanta, where in each quantum all transitions are evaluated, and transitions that can fire do fire.

### B.1.3 π-calculus

π-calculus [91] is an algebra developed by Robin Milner in 1992 based on the CCS process algebra [245]; it models concurrent systems in terms of messages exchanged by processes. Under π-calculus, a *process* represents a computational unit that accepts input and produces output over *channels*, where a channel carries a *message* and acts as a connection between processes. A message represents a collection of data elements, which may include references to one or more channels[6], and a process can propagate those channel references in subsequent messages or can send messages itself on those channels. A π-calculus expression can model the creation of channel and process instances dynamically at runtime.

To receive input, a process waits for a message on a channel, and its execution is blocked until another process sends a message on that channel. Multiple processes can write to a channel, and there is no guarantee of the

[6] CCS cannot do this.

ordering of the messages accepted by the channel, or of the order in which processes that become unblocked will execute.

Because π-calculus can model communications between dynamically created and connected process instances, it forms a workable basis for modeling dynamically configurable systems (aka *mobile*, including agent-based systems). Numerous extensions to π-calculus have been proposed [246], including providing better definition of message content [247] and conveying processes themselves as payloads in channel messages [248]. While π-calculus and its derivative calculi are generally not used for general programming (due to their focus on low-level message passing and synchronization at the expense of code organization abstractions), they can be used for modeling protocols in various contexts, are supported by model checkers (to determine deadlock, livelock, and other properties), and have influenced the concurrency and message passing models of existing programming languages, including BPEL [249] and Orc [250], and Service Oriented Architectures (SOAs).

### B.1.4 λ-calculus

λ-calculus is an algebra created by Alonzo Church in 1936 as a mathematical logic for describing and analyzing computations [251]. At its heart, it defines a computation as a function that may perform a calculation or decompose into other functions. A function accepts arguments and returns a value, which in turn may be used as an argument to another function, may be bound to a variable, or may be used as a function itself. Executing a λ-calculus expression involves reducing it to a single result according to the λ-calculus reduction rules and their implementation strategies.

A strict definition of λ-calculus involves only a minimum set of operators, and is therefore inconvenient as a programming language. A number of so-called *functional* languages have been defined based on λ-calculus principles, including Scheme, ML, Lisp, and Clojure [116]; they incorporate various convenience functions, syntactic sugar, special forms, and typing systems that improve the economics of programming and maintaining λ-calculus expressions.

## B.2 Software Development Methodologies

Programming methodologies exist to enable application developers to organize their approach to requirement gathering and factoring, application architecture and design, and coding and maintenance. Historically, a new programming methodology emerges when the economics of existing methodologies are strained due to any of a number of factors, including the complexity of requirement sets, diversity of stakeholder groups, the number and complexity of concerns addressed in an architecture or design, and the size and complexity of a code base. In this section, I describe the software development methodologies that inspire PDD and provide counterpoints to it.

### B.2.1 Modular Programming

According to [100], modular programming is a discipline that enables the creation and maintenance of large systems by decomposing the system into concerns packaged as modules with clear interfaces and boundaries.

Accordingly, multiple modules can be programmed in parallel, thereby reducing time to market. Additionally, product changes that can be isolated to one module demonstrably don't affect other modules, thereby reducing maintenance and testing costs and improving reusability. Finally, whereas an entire application can be large and complex, modules represent subsets that can be more easily understood than the whole, thereby improving prospects for good design decisions. Module definition focuses on the implementation of arguably independent abstractions, which then can be orchestrated to realize a program's requirements. Modular programming was first supported in languages such as assembly, C, Cobol, and Fortran.

### B.2.2 Structured Programming

Structured programming is a design and programming technique that progressively refines a high level abstraction into an orchestration of low level abstractions via decomposition, and was first supported in languages such as Pascal. It leverages the Böhm-Jacopini theorem [252] which claims that workflows can be decomposed into combinations of three normalized patterns: a sequence, a choice, and a loop. As a corollary, each pattern has a single definable entry and exit point [253], with definable entry and exit properties useful in defining and verifying properties of the higher level abstraction. Such reasoning forms the basis of replacing one workflow with another, yet preserving the semantics of the higher level abstraction.

### B.2.3 Object Oriented Programming

Object-oriented programming (OOP) is a programming style that supports the Object-oriented design methodology, and was first supported in languages such as Java, C++, and Smalltalk. As defined in [101], it features object definition explicitly via an interface, data encapsulation, polymorphism, inheritance, and open recursion. Generally, object functionality is accessed via a call to a method whose actual code is resolved at runtime. The call is modeled as a message exchange, though it can be implemented in numerous ways, including as an actual message exchange or a stack-oriented function call.

### B.2.4 Aspect Oriented Programming

Aspect-oriented programming (AOP) [32] is a programming mechanism that seeks to eliminate the code scattering and entanglement created when crosscutting concerns are implemented in base workflows. It organizes such concerns centrally (as *advice*) and defines composition rules (called *pointcuts*) that together form an *aspect*. A pointcut determines where in a base workflow to inject a new workflow, and possible injection locations are called *join points*. A join point identifies an activity in a workflow, and a pointcut is a predicate that identifies one or more join points. The aspect encodes both an injected workflow and its relationship to join points, including executing before the join point, after the join point, or around (possibly replacing) the join point.

AOP is implemented as extensions to many common languages and frameworks, including Java (as AspectJ [11]), BPEL (as AO4BPEL [146]), and others. Each implementation defines its own join point model, pointcut language, and advice language. Consequently, the particular capabilities of AOP vary with the implementation.

### B.2.5 Execution Frameworks

#### Enterprise Service Bus

An Enterprise Service Bus (ESB) [120] executes workflows by routing messages between service components. Abstractly, a workflow is specified as a graph where service components are nodes that are connected by edges representing unidirectional message routings (connecting a source to a target). A routing can be defined statically (and encoded in a declarative language such as XML) when the workflow is authored, and nodes or routings can be added or removed at runtime. Interceptors [254] are functions that can intercept and process a message in flight, and can be assigned either statically or dynamically to an interaction between a source node and a target node.

### B.2.6 Policy Engines

The topic of policy-based access control has been addressed by [204] and others, and assumes the execution of policy statements by a policy engine built specifically to implement secure identity, attribute, and policy management services that implement choice in workflows. The relationship of a policy engine to an application is defined in ISO 10181-3 [255], and includes a policy decision point (*PDP*, where a decision is made) and a policy enactment point (*PEP*, where a workflow is selected based on the decision). A policy engine manages predicates (called *policies*) written in a policy language (e.g., Ponder, XACML, X-Sec, PERMIS, and Akenti) and evaluates a policy on behalf of a PDP. Policies are typically written by programmers and are deployed while an application is running. Commonly, PDPs and PEPs are coded explicitly into an application, as are the alternate workflows selected by the PEP. Policies executed by policy engines are oriented toward access control decisions, and can maintain their own state, but usually cannot access workflow, application, and environment state.

**PERMIS (Privilege and Role Management Infrastructure Standards)**

The PERMIS policy infrastructure [121] is a mature policy evaluation system that incorporates a policy engine as a component of an overall strategy to implement and manage injection of access control decisions into enterprise applications. It encompasses secure identity, attribute, and policy management services that feed into a policy decision point (PDP) separate from an enactment point (PEP) as described above. Typically, a programmer encodes a call to a PDP, which evaluates a policy identified in the call and returns a decision (often allow/deny) which is then used in a PEP to parameterize a workflow or distinguish between alternate workflows. PERMIS functions as a PDP, whereas the PEP is application-dependent, and is often statically coded to interpret and act upon the PDP's decision.

**BPEL Process Integration with Business Rules**

BPEL [122] is a block structured scripting language for sequencing services in workflows. Recent versions have variables, XML support, looping control constructs, transaction management, fork/join, and exception handling. Similar to PERMIS, Oracle's BPEL Process Integration with Business Rules [123] defines a decision service that evaluates business rules to render a decision.

**xESB: Integration of Policy with ESBs**

xESB [124] represents a different approach to runtime policy injection – it intercepts messages exchanged by interacting services in a SOA executing on an ESB. It executes policies that affect either the base control flow or data flow, and represents crosscutting concerns as business rules (i.e., policies) maintained separately from the base application.

Under xESB, rules exist in a collection that is examined in toto on each and every service interaction – the collection also contains statically defined counters, timers, and hashes that track rule state.

## B.3 Workflow Context in Distributed Systems

By nature, programming languages define workflows, and workflow activities depend on either control-related or data-related context. Each programming language defines workflows and context in terms (and with limitations) that suit the language's purpose. For example, in Java, workflow activity orchestration is organized as sequential statements, with decomposition implemented as method calls. Similarly, variables defined in method blocks and as method parameters function as workflow variables and messages, while variables defined in classes and packages have more global lifecycles.

Analogously, modern frameworks and architectures that support distributed systems frame the maintenance of state based on the workflow assumptions underlying their target applications. In this section, I describe context maintenance in Struts, REST, and AJAX, as examples of different tradeoffs.

### B.3.1 Struts

The Struts [126] system is a server-based Java environment that executes workflows on behalf of clients. Under Struts, state is maintained as Java beans, with request, session, and application lifecycles.

A request bean is associated with a single workflow executing in a single thread on a single computer.

A session bean is associated with a particular client (via a browser cookie or URL rewriting). Conceptually, a session exists for each client (as represented by a browser), but session memory and attributes are not actually allocated until a workflow accesses the session. The session is deleted when the browser leaves the application, which is approximated by a combination of the cookie being a browser "session cookie" and the session timing out according to a Struts-based session policy. Multiple workflows may access the session serially or in parallel, and so Struts provides a measure of thread-safe access. The client can assume that session beans are secure from client to client, but that they are visible to all requests on behalf of a given user. The maintenance of a session bean depends on a session reference passed using an HTTP protocol between a client (browser) and the Struts system – via a cookie or URL rewriting.

An application bean is equivalent to a global variable, and represents state available to all workflows executing on all computers in a distributed system.

### B.3.2 REST

A RESTful application follows the principles of the REST (Representational State Transfer) [127] architectural style, which distinguishes application state from data (so-called hypermedia). Application state resides on clients, and hypermedia (and other resources) resides on servers. A client accesses a server via a self-contained request that makes no assumptions about state stored on the server, and specifies data (and computing resources) as a resource identifier that the server can map to a real resource instance. In contrast to Struts, REST client state is local (and not distributed), which gives clients flexibility in responding to server failures, and promotes server scalability.

### B.3.3 AJAX

AJAX [128] [129] is a collection of technologies aimed at providing a fluid experience for users executing client-server application in a web browser. Under AJAX, a web application communicates with a single server by sending a request and then continues executing without waiting for a response. When the server's response arrives, the client invokes a function registered for the exchange. AJAX applications generally exchange XML or JSON data, and manage workflows and user interfaces using JavaScript, the Document Object Model (DOM), and XHTML/CSS.

From a high level perspective, AJAX applications involve a collection of clients interfacing with a single server, where the server maintains common data, and clients maintain the state of the user interface. A workflow is defined as a series of one or more exchanges between a client and server, and can involve keeping state on both the client and server (as well as the sequential execution of callback functions on the client and API calls on the server). For example, a server that implements servlet [256] capabilities stores session-oriented state that persists until a (configurable) inactivity timeout occurs, and the state can be used in combination with client requests to drive server workflows.

Alternatively, a developer can define a server using a REST model, where the server maintains no state, and server resources are identified by reference. In this case, the client-server protocol involves simple request-reply exchanges, and the client maintains state pertaining to both the user interface and the server.

For example, a typical AJAX workflow may fetch a long list in chunks, issuing one request for each chunk, and ordering the requests starting with the beginning of the list and proceeding in order to the end. In a REST model, each request would contain the list's resource identifier and an index. In a servlet model, a series of requests could set the resource identifier, and could then fetch successive chunks, with the resource identifier and the index retained at the server. Other designs are possible, and depend on tradeoffs between the ability of a client to maintain state and consistency guarantees made by the server (which often come at the cost of server scalability).

In any case, JavaScript executes client-side workflows based on client-resident state, which may be held as global variables and in closures, particularly closures associated with server requests. For example, in a REST model, a closure would maintain the index passed to the server, thereby allowing the client to correlate a server reply with the index that generated it.

## B.4 Ponder Policy Verification

Ponder is a highly successful environment that implements the policy Ponder language [130]. Ponder policies are written in a unique declarative language expressed in terms of subjects, targets, actions, and conditionals. A subject identifies the principals to which the policy applies. A target identifies the objects (e.g., resources and service providers) to which the policy applies, and which may expose custom methods accessible within the policy's actions and conditionals. An action is a program fragment that executes some series of operations that, themselves, may have constraints and may execute methods. A conditional determines when the policy becomes active, and can incorporate external factors as well as set-based calculations involving

Ponder's RBAC [115] database (which follows a hierarchical role model, similar to Grouper [178]). Ponder supports numerous policy types, including authorization, obligation, delegation, information filtering, and refrain policies.

Additionally, Ponder has a concept of meta-policies, which are policies that determine which policies are allowed in a system, including disallowing conflicting policies.

# APPENDIX C Patterns for Object Oriented Programming

As described in [33], design patterns capture solutions that have developed and evolved over time. Twenty three patterns pertaining to Object Oriented Programming (OOP) are presented in [33], where many of these patterns apply to design and architecture beyond OOP.

In this dissertation, I focus on the Strategy and Composite patterns, which are well described abstractly and concretely in [33], and are summarized in this Appendix for convenience.

## C.1 Strategy Pattern

Generally, a Strategy pattern represents a decision that chooses between a number of algorithms, and is abstractly similar to a *policy* as defined in this dissertation. At its core, a Strategy pattern contemplates the use of an algorithm, where the particular algorithm chosen is deferred.

In concrete terms, an OOP programmer can implement a Strategy pattern in a number of ways. For example, supposing a program feature is implemented by an algorithm that is explicitly called, and a stakeholder requires a choice of algorithms instead, and the choice is made at runtime. The programmer can virtualize the algorithm at compile time and choose the particular algorithm at runtime as follows:

1. Create an interface (as an interface class, abstract class, or base class) that functionally defines the algorithm
2. Create the existing algorithm as an instance of the interface
3. Create alternative algorithms as other instances of the interface
4. Replace the use of the algorithm with a reference (e.g., pointer) to a class that implements the interface
5. Based on some criteria evaluated before use of the reference, choose an algorithm and assign an instance of it to the reference
6. Use the reference instead of the original algorithm

The result can be graphically depicted as shown as a UML class diagram in Figure 77, where an application exercises three different features during its execution. Each of the features have alternative algorithms, each of which implements the interface for the feature it implements. When the application executes a feature, it calls an instance of one of the algorithms that implements it.

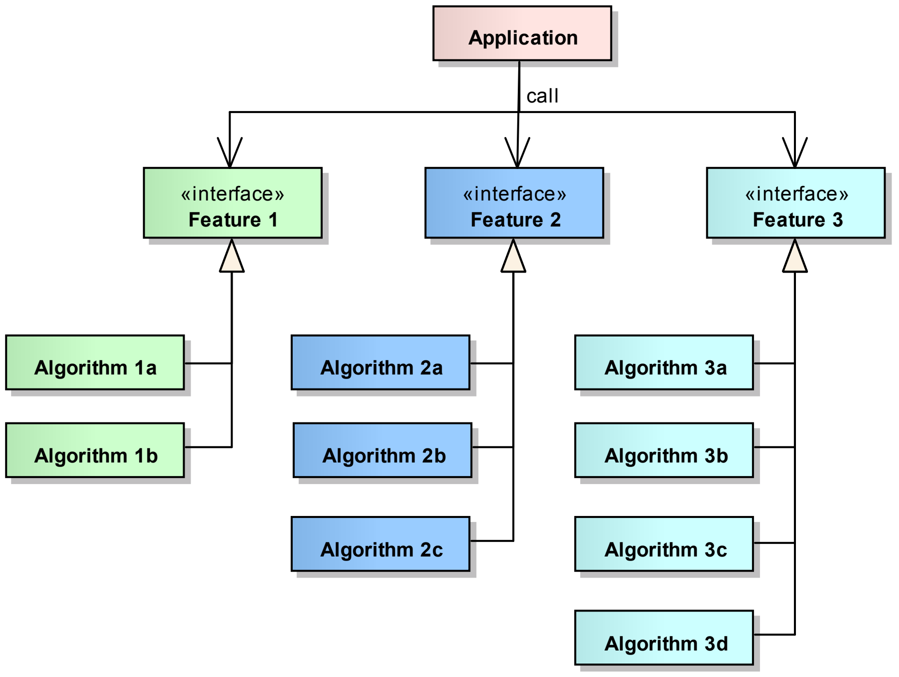


Figure 77. OOP Strategy Pattern

The deferred algorithm definition envisioned in OOP requires that the algorithms and the choice that distinguishes amongst them be explicitly coded into the application, whereas PDD's vision is to inject the choice and call at runtime.

## C.2 Composite Pattern

A Composite pattern represents a data structure defined recursively and can be thought of as a tree of nodes where all nodes derive from a common base class (and therefore expose common attributes and operations). As shown in the UML class diagram in Figure 78, a node can be either a standalone data structure or can itself be a Composite pattern that may have children (thereby creating the recursion).

A common example of a Composite pattern is a hierarchical file system. Following the model in Figure 78, a file would be a Leaf – it has no children. A directory would be a Composite – it may have children that are either files or directories. The common base class (called Component) – could have a timestamp element, which both files and directories would inherit.

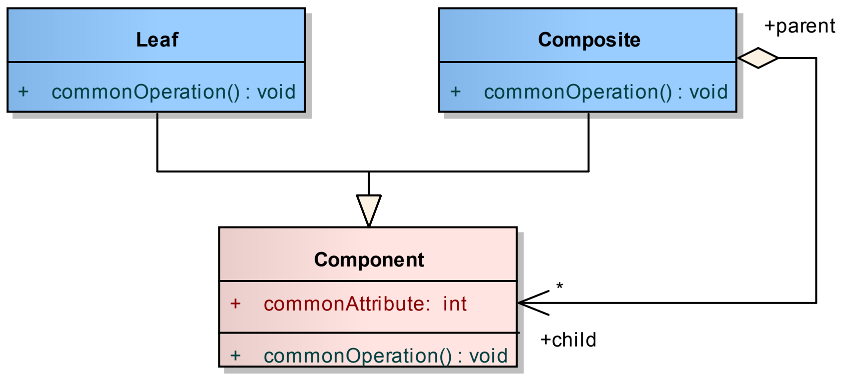

Leaf
+ commonOperation() : void
Composite
+ commonOperation() : void
+parent
Component
+ commonAttribute: int
+ commonOperation() : void
*
+child

Figure 78. OOP Composite Pattern

From a PDD viewpoint, a service is an example of a Composite pattern because it can implement its function by orchestrating interactions between a collection of services – and those services can be either decomposed further or can implement a self-contained calculation.

## REFERENCES

[1] Canfora, G., and Cimitile, A. (2000). Software Maintenance. In S. K. Chang, *Handbook of Software Engineering and Knowledge Engineering* (pp. 91-120). Singapore: World Scientific Publishing Co. Ptc. Ltd.

[2] Luer, C., Rosenbaum, D., and van der Hoek, A. (2001). The evolution of software evolvability. *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE)* (pp. 134-137). Vienna, Austria: ACM Press.

[3] Meyer, B. (1997). *Object-Oriented Software Construction (2nd edition).* Upper Saddle River: Prentice Hall.

[4] Lientz, B. P., and Swanson, E. B. (1979, April). Software Maintenance: A User/Management Tug of War. *Data Management*, pp. 26-30.

[5] Ciraci, S., and ven den Broek, P. (2006). Evolvability as a Quality Attribute of Software Architectures. *The International ERCIM Workshop on Software Evolution 2006 (EVOL 2006)*, (pp. 29-31). Lille.

[6] Martufi, G. (2007). Software Evolvability: An industry's view. *Proceedings of the 2nd Open Workshop on Resilience in Computing Systems and Information Infrastructures (ReSIST).* Rome, Italy.

[7] Cook, S., Ji, H., and Harrison, R. (2000). *Software Evolution and Software Evolvability.* University of Reading.

[8] Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002). *Fundamentals of Software Engineering, 2nd Edition.* Upper Saddle River: Pearson Education, Inc.

[9] Chaumun, M. A., Keller, R. K., and Lustman, F. (2002). Design Properties and Evolvability of Object-Oriented Systems. In H. Erdogmus, and O. Tanir, *Advances in Software Engineering* (pp. 197-224). New York: Springer-Verlag.

[10] Trieber, M., Juszczyk, L., Schall, D., and Dustdar, S. (2010). Programming Evolvable Web Services. *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS '10)* (pp. 43-49). Cape Town: Association for Computing Machinery.

[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An Overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming* (pp. 327-353). London: Springer-Verlag.

[12] Bosch, J. (2000). *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* New York: Addison-Wesley Professional.

[13] Mannaert, H., Verelst, J., and Ven, K. (2012). Towards evolvable software architectures based on systems theoretic stability. *Software -- Practice & Experience*, 89-116.

[14] Medvidovic, N. (1996). *A Classification and Comparison Framework for Software Architectural Description Languages.* Irvine: University of California, Irvine.

[15] Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices.* Upper Saddle River: Prentice Hall.

[16] Beck, K., and Andres, C. (2004). *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional.

[17] Verelst, J. (2004). The influence of the level of abstraction on the evolvability of conceptual models of information systems. *Proceedings of the 2004 International Symposium on Emprical Software Engineering (ISESE '04)* (pp. 17-26). Redondo Beach: IEEE Computer Society.

[18] Cook, S., Harrison, R., and Wernick, P. (2006). *Information System Evolvability, Feedback and Pattern Languages.* University of Reading.

[19] Khurana , H., Bobba, R., Yardley, T., Agarwal, P., and Heine, E. (2010). Design Principles for Power Grid Cyber-Infrastructure Authentication Protocols. *Proceedings of the 2010 43rd Hawaii International Conference on System Sciences (HICSS '10)* (pp. 1-10). Kauai: IEEE Computer Society.

[20] National Science Foundation. (2007, March). *Cyberinfrastructure Vision for 21st Century Discovery.* Retrieved October 28, 2012, from http://www.nsf.gov/pubs/2007/nsf0728/nsf0728_1.pdf

[21] Seidel, E. (2008, November 16). *The Importance of Cyberinfrastructure for Research and Education.* Retrieved October 28, 2012, from http://www.nsf.gov/sbe/secure/advcom1108/Presentations/04.Seidel_SBE_Advisory.pdf

[22] Atkins, D., Droegemier, K., Feldman, S., Garcia-Molina, H., Klein, M., Messerschmitt, D., et al. (2003). *Revolutionizing Science and Engineering Through Cyberinfrastructure.* Washington, DC: National Science Foundation.

[23] National Science Foundation. (2007, March). *Cyberinfrastructure Vision for 21st Century Discovery.* Retrieved June 3, 2011, from http://www.nsf.gov/pubs/2007/nsf0728/nsf0728.pdf

[24] Demchak, B., Kerr, J., Raab, F., Patrick, K., and Krüger, I. H. (2012). PALMS: A Modern Coevolution of Community and Computing Using Policy Driven Development. *Proceedings of the 2012 45th Hawaii International Conference on System Sciences (HICSS '12)* (pp. 2735-2744). Maui: IEEE Computer Society.

[25] Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 1053-1058.

[26] Dijkstra, E. W. (1968). Go To Statement Considered Harmful. *Communications of the ACM*, 147-148.

[27] Kernighan, B. W., and Plauger, P. J. (1974). *The Elements of Programming Style*. New York: McGraw Hill.

[28] McKnight, W. L. (2002, July). What is Information Assurance? *Crosstalk: The Journal of Defense Software Engineering*, pp. 4-6.

[29] Nuseibeh, B., and Easterbrook, S. (2000). Requirements Engineering: A Roadmap. *Proceedings of the Conference on The Future of Software Engineering* (pp. 35-46). Limerick, Ireland: Association for Computing Machinery.

[30] Lee, E. A., and Seshia, S. A. (2011). *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Berkeley: LeeSeshia.org.

[31] Wiegers, K. (2003). *Software Requirements (2nd Edition)*. Redmond: Microsoft Press.

[32] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., et al. (1997). Aspect-Oriented Programming. *Proceedings of the Eur Conf on OOP*. Helsinki, Finland: Springer-Verlag LNCS 1241.

[33] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[34] Fowler, M. (2005, June 26). *InversionOfControl*. Retrieved October 5, 2012, from http://martinfowler.com/bliki/InversionOfControl.html

[35] Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley.

[36] Bashar, N., and Easterbrook, S. (2000). Requirements engineering: a roadmap. *Proceedings of the Conference on the Future of Software Engineering* (pp. 35-46). Limerick: ACM Press.

[37] QFD Institute. (n.d.). Retrieved March 31, 2012, from www.qfdi.org

[38] Cohn, M. (2004). *User Stories Applied: For Agile Software Development.* Redwood City: Addison Wesley Longman Publishing Co., Inc.

[39] Cockburn, A. (2000). *Writing Effective Use Cases.* Boston: Addison-Wesley Longman Publishing Co., Inc.

[40] Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition).* Boston: Addison-Wesley Longman Publishing Co., Inc.

[41] Withall, S. (2007). *Software Requirement Patterns.* Redmond: Microsoft Press.

[42] Arkin, A. (2002). *Business Process Modeling Language*. Retrieved Sept 29, 2011, from http://www.bpmi.org/downloads/BPML1.0.zip

[43] Chung, L., and Leite, J. (2009). On Non-Functional Requirements in Software Engineering. In A. T. Borgida, V. K. Chaudhri, P. Giorgini, and E. S. Yu, *Conceptual Modeling: Foundations and Applications* (pp. 363-379). Berlin Heidelberg: Springer-Verlag.

[44] Bennington, H. D. (1987). Production of large computer programs. *Proceedings of the 9th international conference on Software Engineering* (pp. 299-310). Monterey: IEEE Computer Society Press.

[45] Martin, R. C. (2002). *Agile Software Development.* Upper Saddle River: Prentice Hall PTR.

[46] Boehm, B. (1986). A Spiral Model of Software Development and Enhancement. *SIGSOFT Software Engineering Notes*, 14-24.

[47] Demchak, B., Farcas, C., Farcas, E., and Krueger, I. (2007). The Treasure Map for Rich Services. *Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration (IRI)* (pp. 400-405). Las Vegas, USA: IEEE.

[48] Young, R. R. (2004). *The Requirements Engineering Handbook.* Norwood, MA: Artech House, Inc.

[49] Cysneiros, L. M., and Leite, J. (2001). Using UML to reflect non-functional requirements. *In Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research (CASCON '01).* Toronto: IBM Press.

[50] Sindre, G., and Opdahl, A. (2005). Eliciting security requirements with misuse cases. *Requirements Engineering*, 34-44.

[51] Mouratidis, H., Manson, G. A., and Giorgini, P. (2003). Analysing Security Requirements of Information Systems Using Tropos. *Procedings of the 5th*

*International Conference on Enterprise Information Systems*, (pp. 623-626). Angers, France.

[52] Wang, L., Wong, E., and Xu, D. (2007). A Threat Model Driven Approach for Security Testing. *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems (SESS '07).* Minneapolis: IEEE Computer Society.

[53] Krüger, I. H., Meissinger, M., and Menarini, M. (2007). Runtime verification of interactions: from MSCs to aspects. *Proceedings of the 7th international conference on Runtime verification* (pp. 63-74). Vancouver: Springer-Verlag.

[54] Alam, M., Breu, R., and Hafner, M. (2007). Model-Driven Security Engineering for Trust Management in SECTET. *Journal of Software*, 47-59.

[55] Juerjens, J. (2003). *Secure Systems Development with UML*. Berlin Heidelberg: Springer-Verlag.

[56] Burt, C. C., Bryant, B. R., Raje, R. R., Olsen, A. M., and Auguston, M. (2003). Model Driven Security: Unification of Authorization Models for Fine-Grain Access Control. *Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)* (pp. 159-173). Brisbane: IEEE Computer Society.

[57] Cysneiros, L. M., and Leite, J. (2004). Nonfunctional Requirements: From Elicitation to Conceptual Models. *IEEE Transactions on Software Engineering*, 328-350.

[58] Rashid, A., and Chitchyan, R. (2008). Aspect-oriented requirements engineering: a roadmap. *Proceedings of the 13th international workshop on Early Aspects* (pp. 35-41). Leipzig: Association for Computing Machinery.

[59] Oldevik, J., and Haugen, Ø. (2007). Architectural Aspects in UML. In *Model Driven Engineering Languages and Systems (LNCS)* (Vol. 4735/2007, pp. 301-315). Berlin Heidelberg: Springer-Verlag.

[60] Zave, P. (2010). Modularity in Distributed Feature Composition. In B. Nuseibeh, and P. Zave, *Software Requirements and Design: The Work of Michael Jackson* (p. 267). Chatham, New Jersey: Good Friends Publishing Company.

[61] Kim, C., Kästner, C., and Batory, D. (2008). On the Modularity of Feature Interactions. *Proceedings of the 7th international conference on Generative programming and component engineering* (pp. 23-34). Nashville: Association for Computing Machinery.

[62] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-Oriented Domain Analysis (FODA). *CMU/SEI-90-TR-21*. Pittsburgh: Carnegie-Mellon University Software Engineering Institute.

[63] Apel, S., Lengauer, C., Möller, B., and Kästner, C. (2008). An Algebra for Features and Feature Composition. *Proceedings of the 12th international conference on Algebraic Methodology and Software Technology* (pp. 36-50). Urbana: Springer-Verlag.

[64] Stoiber, R., Fricker, S., Jehle, M., and Glinz, M. (2010). Feature Unweaving: Refactoring Software Requirements Specifications into Software Product Lines. *Proceedings of the 2010 18th IEEE International Requirements Engineering Conference (RE '10)* (pp. 403-404). Sydney: IEEE Computer Society.

[65] IBM Corporation. (n.d.). *Morphogenic Software*. Retrieved May 19, 2012, from http://www.research.ibm.com/morphogenic/

[66] Robinson, W. N., Pawlowski, S. D., and Volkov, V. (2003). Requirements Interaction Management. *ACM Computing Survey*, 132-190.

[67] Van Lamsweerde, A. (2001). Goal-Oriented Requirements Engineering: A Guided Tour. *Proceedings of the 5th IEEE International Symposium on Requirements Engineering* (pp. 249-263). Toronto: IEEE Computer Society.

[68] Apel, S., and Hutchins, D. (2010). A calculus for uniform feature composition. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19:1-19:33.

[69] Cook, W. R., Patwardhan, S., and Misra, J. (2006). Workflow Patterns in Orc. *Coordination'06, volume 4038 of LNCS* (pp. 82-96). Berlin Heidelberg: Springer-Verlag.

[70] van der Aalst, W., and ter Hofstede, A. (2010). *Workflow Patterns*. Retrieved Sept 29, 2011, from http://workflowpatterns.com/

[71] Jurack, S., Lambers, L., Mehner, K., and Taentzer, G. (2008). Sufficient Criteria for Consistent Behavior Modeling with Refined Activity Diagrams. *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems (MODELS '08)* (pp. 341-355). Toulouse: Springer-Verlag.

[72] Leymann, F., and Roller, D. (1999). *Production Workflow: Concepts and Techniques*. Prentice Hall.

[73] Eder, J., Gruber, W., and Pichler, H. (2005). Transforming Workflow Graphs. *First International Conference on Interoperability of Enterprise Software and Applications* (pp. 23-25). Genf, Switzerland: Springer-Verlag.

[74] Pankratius, V., and Stucky, W. (2005). A formal foundation for workflow composition, workflow view definition, and workflow normalization based on petri nets. *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling - Volume 43* (pp. 79-88). Newcastle, New South Wales, Australia: Australian Computer Society, Inc.

[75] Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., and Mock, S. (2004). Kepler: an extensible system for design and execution of scientific workflows. *16th International Conference on Scientific and Statistical Database Management*, (pp. 423-424). Santorini.

[76] Object Management Group. (2012). *BPMN Information Home*. Retrieved September 21, 2012, from http://www.bpmn.org/

[77] Carlsen, S. (1997). *Conceptual Modeling and Composition of Flexible Workflow Models.* Information Systems Group, Department of Computer and Information Science. Trondheim: Norwegian University of Science and Technology.

[78] Petri, C. A., and Reisig, W. (2008). Petri net. *Scholarpedia*, 6477.

[79] Sadiq, W., and Orlowska, M. E. (1999). Applying Graph Reduction Techniques for Identifying Structural Conflicts in Process Models. *Proceedings of the 11th International Conference on Advanced Information Systems Engineering* (pp. 195-209). Heidelberg: Springer-Verlag.

[80] Fernández, M. (2009). *Models of Computation.* London: Springer-Verlag.

[81] Baeten, J., Basten, T., and Reniers, M. (2010). *Process Algebra: Equational Theories of Communicating Processes.* Cambridge, England: Cambridge University Press.

[82] Baeten, J. (2005). A brief history of process algebra. *Theoretical Compter Science - Process algebra*, 131-146.

[83] British Computer Society Formal Aspects of Computing Science Specialist Group. (2012, January 12). *Formal Methods Wiki*. Retrieved March 8, 2012, from http://formalmethods.wikia.com/wiki/Formal_methods

[84] Michael, S. (2006). *Introduction to the Theory of Computation.* Boston: Thomson Course Technology.

[85] Holzmann, G. J. (1991). *Design And Validation Of Computer Protocols.* Englewood Cliffs, New Jersey: Prentice-Hall.

[86] *Composition of State Machines.* (n.d.). Retrieved May 25, 2012, from http://cs.ioc.ee/~margo/aat/03_Composition_of_state_machines.pdf

[87] Johnsonbaugh, R. (n.d.). *8.5 Petri Nets*. Retrieved March 5, 2012, from condor.depaul.edu/rjohnson/dm7th/petri.pdf

[88] Fehling, R. (1993). A Concept of Hierarchical Petri Nets with Building Blocks. *12th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1993* (pp. 148-168). London: Springer-Verlag.

[89] Hamadi, R., and Benatallah, B. (2003). A Petri net-based model for web service composition. *Proceedings of the 14th Australasian database conference - Volume 17* (pp. 191-200). Adelaide, Australia: Australian Computer Society, Inc.

[90] Jensen, K. (2009). *Coloured Petri Nets*. Berlin Heidelberg: Springer-Verlag.

[91] Milner, R. (1999). *Communicating and Mobile Systems: The π-calculus*. Cambridge, England: Cambridge University Press.

[92] Nestmann, U., and Pierce, B. C. (1996). Decoding Choice Encodings. *Proceedings of the 7th International Conference on Concurrency Theory* (pp. 179-194). London: Springer-Verlag.

[93] Pierce, B. C., and Turner, D. N. (1997). Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner* (pp. 455-494). MIT Press.

[94] Lumpe, M. (1999, January). A π-Calculus Based Approach for Software Composition. *PhD Thesis*. Bern, Switzerland: Institute of Computer Science and Applied Mathmatics, The University of Bern.

[95] Abelson, H., Sussman, G. J., and Sussman, J. (1996). *Structure and Interpretation of Computer Programs*. Cambridge, MA: The MIT Press.

[96] Niehren, J., Schwinghammer, J., and Smolka, G. (2006). A Concurrent Lambda Calculus with Futures. *Theoretical Computer Science*, 338-356.

[97] Wadler, P. (1992). The Essence of Functional Programming. *Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 1-14). Albuquerque: Prentice Hall.

[98] Jones, S. L., and Wadler, P. (1993). Imperative functional programming. *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '93)* (pp. 71-84). Charleston: ACM.

[99] Marick, B. (2011, March 21). *Monad Tutorial, Part 4 (State Monad, Parts of a Monad)*. Retrieved March 24, 2012, from http://vimeo.com/21307543

[100] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 1053-1058.

[101] Pierce, B. C. (2002). *Types and Programming Languages.* Cambridge: MIT Press.

[102] Booch, G. (1994). *Object-oriented analysis and design with applications (2nd ed).* Redwood City: Benjamin-Cummings Publishing Co., Inc.

[103] Evans, E. (2004). *Domain-Driven Design.* Addison-Wesley Professional.

[104] Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., et al. (2005). Information Hiding Interfaces for Aspect-oriented Design. *Proceedings of the 10th European Software Engineering Conference (ESEC/FSE '13)* (pp. 166-175). Lisbon: Association for Computing Machinery.

[105] Clemente, P. J., Hernandez, J., and Sanchez, F. (2007). Driving Component Composition from Early Stages Using Aspect-Oriented Techniques. *40th Annual Hawaii International Conference on System Sciences (HICSS'07)* (p. 257a). Hawaii: IEEE Computer Society.

[106] Xerox Corporation. (n.d.). *Load-Time Weaving*. Retrieved May 31, 2012, from http://www.eclipse.org/aspectj/doc/released/devguide/ltw.html

[107] Lakhani, J., Akkawi, F., Bader, A., and Elrad, T. (2001). Dynamic Weaving for Building Reconfigurable Software Systems. *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems* (pp. 152-184). Tampa Bay: Springer-Verlag.

[108] Fuentes, L., and Sánchez, P. (2009). Dynamic Weaving of Aspect-Oriented Executable UML Models. In S. Katz, H. Ossher, R. France, and J.-M. Jézéquel, *Transactions on Aspect-Oriented Software Development VI* (pp. 1-38). Berlin/Heidelberg: Springer-Verlag.

[109] Greenwood, P., and Blair, L. (2006). A Framework for Policy Driven Auto-adaptive Systems Using Dynamic Framed Aspects. In A. Rashid, and M. Aksit, *Transactions on Aspect-Oriented Software Development II* (pp. 30-65). Berlin/Heidelberg: Springer-Verlag.

[110] Ossher, H., and Tarr, P. (2000). *Multi-Dimensional Separation of Concerns and The Hyperspace Approach*. Retrieved May 31, 2012, from http://researchweb.watson.ibm.com/hyperspace/Papers/sac2000.pdf

[111] Rashid, A. (2008). Aspect-Oriented Requirements Engineering: An Introduction. *Proceedings of the 2008 16th IEEE International Requirements Engineering Conference (RE '08)* (pp. 306-309). Barcelona: IEEE Computer Society.

[112] Navasa, A., Pérez, M., Murillo, J., and Hernández, J. (2002). Aspect Oriented Software Architecture: a Structural Perspective. *Proceedings of the 1st International Conference on Aspect-Oriented Software*

*Development (Workshop on Early Aspects)*. Enschede: Association for Computing Machinery.

[113] Hoare, C. A. (1978). Communicating sequential processes. *Communications of the ACM* (pp. 666-677). New York: Association for Computing Machinery.

[114] Oracle Corporation. (2001, August 8). *Java Authentication and Authorization Service (JAAS) Reference Guid*. Retrieved June 6, 2012, from http://docs.oracle.com/javase/1.4.2/docs/guide/security/jaas/JAASRefGuide.html

[115] Sandhu, R. S., and Samarati, P. (1994). Access Control: Principles and Practice.

[116] Halloway, S. (2009). In S. Holloway, *Programming Clojure*. Pragmatic Bookshelf.

[117] Krüger, I. H. (2000). Distributed System Design with Message Sequence Charts. *Dissertation*. München: Technische Universität München.

[118] Araújo, J., and Moreira, A. (2005). Integrating UML Activity Diagrams with Temporal Logic Expressions. *Proceedings of the 10th International Workshop on Exploring Modeling Methods for Systems Analysis and Design (EMMSAD'05)* (pp. 91-98). Porto: CEUR-WS.org.

[119] Whittle, J., Moreira, A., Araújo, J., Jayaraman, P., Elkhodary, A., and Rabbi, R. (2010). An Expressive Aspect Composition Language for UML State Diagrams. *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS '10)* (pp. 514-528). Nashville: Springer-Verlag Berlin Heidelberg.

[120] Chappell, D. A. (2004). *Enterprise Service Bus.* Beijing: O'Reilly.

[121] Chadwick, D., Zhao, G., Otenko, S., Laborde, R., and Nguyen, T. (2008). PERMIS: a modular authorization infrastructure. *Concurrency and Computation: Practice & Experience*, 1341-1357.

[122] Juric, M. B. (2006, July 10). *BPEL: Service composition for SOA*. Retrieved September 21, 2012, from http://www.javaworld.com/javaworld/jw-07-2006/jw-0710-bpel.html

[123] Oracle Corporation. (n.d.). *Oracle BPEL Process Manager Developer's Guide*. Retrieved June 8, 2012, from http://docs.oracle.com/cd/E11036_01/integrate.1013/b28981/decision.htm#CHDHDDCF

[124] Gheorghe, G., Neuhaus, S., and Crispo, B. (2010). xESB: An Enterprise Service Bus for Access and Usage Control. *Proceedings of the 4th IFIP International Conference on Trust Management (IFIPTM2010).* Morioka, Japan.

[125] *Ponder2 Wiki*. (2012, January 8). Retrieved April 3, 2012, from www.ponder2.net

[126] The Apache Software Foundation. (n.d.). Retrieved Sept 29, 2011, from http://struts.apache.org/

[127] Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Irvine, California: University of California.

[128] Garrett, J. J. (2005, February 18). *Ajax: A New Approach to Web Applications*. Retrieved September 13, 2012, from http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

[129] World Wide Web Consortium (W3C). (2010, Aug 3). *XMLHttpRequest*. Retrieved Sept 29, 2011, from http://www.w3.org/TR/XMLHttpRequest/

[130] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. (2001). The Ponder Policy Specification Language. *Proceedings of the Intl Workshop on Policies for Distributed Systems and Networks (Policy '01).* Bristol.

[131] Bandara, A. K., Lupu, E. C., and Russo, A. (2003). Using Event Calculus to Formalise Policy Specification and Analysis. *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY '03)* (pp. 26-39). Lake Como: IEEE Computer Society.

[132] Bertino, E., Bonatti, P. A., and Ferrari, E. (2001). TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)*, 191-233.

[133] Zhao, Y., and Parisi-Presicce, F. (2005). Policy Analysis and Verification by Graph Transformation Tools. *Electronic Notes in Theoretical Computer Science (ENTCS)* (pp. 101-112). Amsterdam: Elsevier Science Publishers.

[134] Fowler, M. (2003). *Patterns of Enterprise Application Architecture.* Boston: Addison-Wesley.

[135] Hohpe, G., and Woolf, B. (2004). *Enterprise Integration Patterns.* Boston: Addison-Wesley.

[136] Manolescu, D. A. (2004). *Patterns for Orchestration Environments.*

[137] Rotem-Gal-Oz, A. (2012 (est)). *SOA Patterns.* Shelter Island, NY: Manning Publications Co.

[138] Barros, A., Dumas, M., and ter Hofstede, A. (2005). *Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection* . Queensland University of Technology: Technical Report FIT-TR-2005-02.

[139] Martin, R. C. (1997, March 7). *The Dependency Inversion Principle*. Retrieved October 6, 2012, from http://www.objectmentor.com/resources/articles/dip.pdf

[140] Chappell, D. (2007, July). *Introducing SCA*. Retrieved September 12, 2012, from http://www.davidchappell.com/articles/Introducing_SCA.pdf

[141] SpringSource. (n.d.). *springsource community*. Retrieved September 12, 2012, from http://www.springsource.org/

[142] World Wide Web Consortium (W3C). (2006, April 25). *Web Services Policy 1.2 - Framework (WS-Policy)*. Retrieved September 12, 2012, from http://www.w3.org/Submission/WS-Policy/

[143] Object Management Group. (2011, January). *Business Process Model and Notation.* Retrieved March 1, 2012, from http://www.omg.org/spec/BPMN/2.0/PDF/

[144] White, S. A. (2004, March). *Process Modeling Notations and Workflow Patterns*. Retrieved September 30, 2012, from BPTrends: http://www.omg.org/bp-corner/bp-files/Process_Modeling_Notations.pdf

[145] Charfi, A., Müller, H., and Mezini, M. (2010). Aspect-Oriented Business Process Modeling with AO4BPMN. *6th European Conference on Modelling Foundations and Applications (ECMFA 2010)* (pp. 48-61). Paris: Springer.

[146] Charfi, A., and Mezini, M. (2007). AO4BPEL: An Aspect-oriented Extension to BPEL. *World Wide Web*, 309-344.

[147] Barros, J. P., and Gomes, L. (2003). Towards the Support for Crosscutting Concerns in Activity Diagrams: a Graphical Approach. *Proceedings of the 4th AOSD Modeling with UML Workshop*. San Francisco.

[148] Singh, M. P., Chopra, A. K., and Desai, N. (2009, November). Commitment-Based Service-Oriented Architecture. *Computer*, pp. 72-79.

[149] Rebêlo, H., Lima, R., and Cornélio, M. L. (2012). *Implementing JML Contracts with AspectJ: Improving instrumentation and checking of JML contracts*. Saarbruecken: LAP LAMBERT Academic Publishing.

[150] Ecma International. (2011, June). *ECMAScript Langage Specification*. Retrieved October 8, 2012, from http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[151] Organization for the Advancement of Structured Information Standards (OASIS). (2004, October 19). *UDDI Version 3.0.2*. Retrieved October 8, 2012, from http://uddi.org/pubs/uddi-v3.0.2-20041019.pdf

[152] The Eclipse Foundation. (n.d.). *AJDT: AspectJ Development Tools (The Visualiser)*. Retrieved 16 2012, June, from http://www.eclipse.org/ajdt/visualiser/

[153] Zhang, F., Qin, Z., and Zhou, S. (2004). Policy-Tree Based Proactive Defense Model for Network Security. In *Grid and Cooperative Computing - GCC 2004 Workshops* (pp. 437-449). Berlin/Heidelberg: Springer.

[154] Arrott, M., Demchak, B., Ermagan, V., Farcas, C., Farcas, E., Krüger, I., et al. (2007). Rich Services: The Integration Piece of the SOA Puzzle. *Proceedings of the IEEE International Conference on Web Services (ICWS)* (pp. 176-183). Washington, DC: IEEE Computer Society.

[155] Ermagan, V., Krueger, I., and Menarini, M. (2007). Model-based failure management for distributed reactive systems. *Proceedings of the 13th Monterey conference on Composition of embedded systems: scientific and industrial issues* (pp. 53-74). Paris, France: Springer-Verlag.

[156] Demchak, B., Ermagan, V., Farcas, E., Huang, T.-J., Krüger, I. H., and Menarini, M. (2008). A Rich Services Approach to CoCoME. In A. Rausch, R. Reussner, R. Mirandola, and F. Plášil, *The Common Component Modeling Example* (pp. 85-115). Berlin, Heidelberg: Springer-Verlag.

[157] Broy, M., and Stølen, K. (2001). *Specification and Development of Interactive Systems.* New York: Springer-Verlag.

[158] Krüger, I. H. (2012). *Services, SOAs and Integration at Scale*. La Jolla: University of California, San Diego.

[159] Meyer, B. (October 1992). Applying 'Design by Contract'. *Computer*, 40-51.

[160] Pugh, K. (2006). *Interface Oriented Design: With Patterns.* Pragmatic Bookshelf.

[161] Gama, P., and Ferreira, P. (2005). Obligation Policies: An Enforcement Platform. *Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY '05)* (pp. 203-212). Stockholm: IEEE Computer Society.

[162] Alfaro, L., and Henzinger, T. (2001). Interface automata. *8th Euro Software Eng Conf*, (pp. 109-120). Vienna, Austria.

[163] Northrup, L. (2006). *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Pittsburgh: Carnegie Mellon University.

[164] Software Engineering Institute (SEI). (2012). *Ultra-Large-Scale Systems*. Retrieved October 27, 2012, from http://www.sei.cmu.edu/uls/

[165] Sullivan, K. (2011, August 22). A Cyber-Social Systems Approach to the Engineering of Ultra-Large-Scale National Health Information Systems. Washington, DC, USA: Institute of Medicine of the National Academies.

[166] Foster, I., Kesselman, C., and Tuecke, S. (2001). The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 200-222.

[167] PALMS. (n.d.). Retrieved June 3, 2011, from http://ucsd-palms-project.wikispaces.com/

[168] National Institutes of Health. (n.d.). *Genes, Environment and Health Initiative (GEI)*. Retrieved June 3, 2011, from http://www.gei.nih.gov

[169] (n.d.). Retrieved September 23, 2012, from GPS-HRN: http://www.gps-hrn.org/

[170] National Cancer Institute (caBIG). (n.d.). Retrieved September 23, 2012, from caBIG: https://cabig.nci.nih.gov/

[171] Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture(TM): Practice and Promise*. Boston: Addison-Wesley Longman Publishing Co, Inc.

[172] Sparx Systems. (n.d.). Retrieved June 14, 2011, from http://www.sparxsystems.com/

[173] Google Inc. (n.d.). *Google Web Toolkit*. Retrieved June 14, 2011, from http://code.google.com/webtoolkit/

[174] Mulesoft Inc. (n.d.). Retrieved June 14, 2011, from http://www.mulesoft.org/

[175] The Apache Software Foundation. (n.d.). Retrieved June 14, 2011, from http://cxf.apache.org/

[176] National Cancer Institute (caBIG). (n.d.). *Dorian*. Retrieved June 14, 2011, from https://wiki.nci.nih.gov/display/caGridKC/Dorian

[177] The Internet Society. (1999, January). Retrieved June 14, 2011, from http://www.ietf.org/rfc/rfc2459.txt

[178] Internet2. (n.d.). Retrieved June 14, 2011, from http://www.internet2.edu/grouper/

[179] Richards, M., Monson-Haefel, R., and Chappell, D. A. (2009). *Java Message System, Second Edition.* Sebastopol: O'Reilly Media, Inc.

[180] Red Hat, Inc. (2004). *Chapter 15: Criteria Queries*. Retrieved July 23, 2012, from Hibernate Community Documentation: http://docs.jboss.org/hibernate/core/3.3/reference/en/html/querycriteria.html

[181] *org.hibernate Interface Criteria*. (n.d.). Retrieved July 23, 2012, from http://www.dil.univ-mrs.fr/~massat/docs/hibernate-3.1/api/org/hibernate/Criteria.html

[182] Zhang, X. (2011, February 8). *PALMS Criteria API Tutorial*. Retrieved July 23, 2012, from https://sosa.ucsd.edu/confluence/display/PALMSGD/PALMS+Criteria+API+Tutorial

[183] World Wide Web Consortium (W3C). (2011, Jan 3). *XQuery 1.0: An XML Query Language (Second Edition)*. Retrieved Sept 29, 2011, from http://www.w3.org/TR/xquery/

[184] Kay, M. (n.d.). *Saxonica: XSLT and XQuery Processing*. Retrieved Sept 29, 2011, from http://www.saxonica.com

[185] Codehaus. (n.d.). *XStream*. Retrieved Sept 29, 2011, from http://xstream.codehaus.org/

[186] 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 Transactions on Information Systems Security*, 34-64.

[187] SAP. (n.d.). Retrieved August 21, 2012, from http://www.crystalreports.com/

[188] Eclipse Foundation. (n.d.). Retrieved August 23, 2012, from www.eclipse.org

[189] The Apache Software Foundation. (n.d.). Retrieved August 23, 2012, from logging.apache.org

[190] SyncRO Soft SRL. (n.d.). *<oXygen/> xml editor*. Retrieved August 23, 2012, from www.oxygenxml.com

[191] Ocean Observatories Initiative. (2011). Retrieved Sept 29, 2011, from http://ci.oceanobservatories.org/

[192] University of California. (n.d.). Retrieved August 26, 2012, from https://sosa.ucsd.edu/confluence/display/CitiSensePublic/CitiSense

[193] Patrick, K., Wolszon, L., Basen-Engquist, K., Demark-Wahnefried, W., Prokhorov, A., Barrera, S., et al. (2011). CYberinfrastructure for COmparative effectiveness REsearch (CYCORE): improving data from cancer clinical trials. *Journal of Translational Behavioral Medicine: Practice, Policy, Research*, 83-88.

[194] VMware. (2010). *Timekeeping in VMware Virtual Machines*. Retrieved August 27, 2012, from http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf

[195] Ueno, K., and Tatsubori, M. (2006). Early Capacity Testing of an Enterprise Service Bus. *Proceedings of the IEEE International Conference on Web Services (ICWS '06)* (pp. 709-716). Chicago: IEEE Computer Society.

[196] World Wide Web Consortium. (2004, Feburary 11). *Web Services Architecture*. Retrieved September 27, 2012, from http://www.w3.org/TR/ws-arch/

[197] Siddiqi, S., and Atlee, J. (2000). A Hybrid Model for Specifying Features and Detecting Interactions. *Computer Networks*, 471-485.

[198] Robillard, M. P., and Murphy, G. C. (2007). Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM), 16*(1).

[199] *aopmetrics Project home*. (n.d.). Retrieved September 5, 2012, from http://aopmetrics.tigris.org/

[200] Zhang, J., Cottenier, T., van den Berg, A., and Gray, J. (August, 2007). Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. *Journal of Object Technology*, 89-108.

[201] Reddy, Y. R., Ghosh, S., France, R. B., Straw, G., Bieman, J. M., McEachen, N., et al. (2006). Directives for Composing Aspect-Oriented Design Class Models. In A. Rashid, and M. Aksit, *Transactions on Aspect-Oriented Software Development I* (pp. 75-105). Berlin/Heidelberg: Springer-Verlag.

[202] Alexandrescu, A. (2001). *Modern C++ Design : Generic Programming and Design Patterns Applied.* Boston: Addison-Wesley Provessional.

[203] Rosenberg, F., and Dustdar, S. (2005). Business Rules Integration in BPEL - A Service-Oriented Approach. *Seventh IEEE International Conference on E-Commerce Technology* (pp. 476-479). Munich: IEEE.

[204] Shebab, M., Bertino, E., and Ghafoor, A. (2006). Workflow authorisation in mediator-free environments. *International Journal of Security and Networks*, 2-12.

[205] World Wide Web Consortium (W3C). (2004, December 17). *Web Services Choreography Description Language Version 1.0*. Retrieved September 12, 2012, from http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/

[206] Chadwick, D., Xu, W., Otenko, S., Laborde, R., and Nassar, B. (2007). Multi-Session Separation of Duties (MSoD) for RBAC. *1st International Workshop on Security Technologies for Next Generation Collaborative Business Applications (SECOBAP'07)*. Istanbul, Turkey.

[207] Pretschner, A., Hilty, M., Basin, D., Schaefer, C., and Walter, T. (2008). Mechanisms for Usage Control. *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS '08)* (pp. 240-244). Tokyo: Association for Computing Machinery.

[208] Pretschner, A., Schültz, F., Schaefer, C., and Walter, T. (2009). Policy Evolution in Distributed Usage Control. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 109-123.

[209] Neisse, R., Pretschner, A., and Giacomo, V. (2011). A Trustworthy Usage Control Enforcement Framework. *Proceedings of the 6th Intl Conf on Availability, Reliability, and Security (ARES '11)*. Vienna.

[210] Dulay, N., Lupu, E., Sloman, M., and Damianou, N. (2001). A Policy Deployment Model for the Ponder Language. *Procedings of the 7th IEEE/IFIP International Symposium of Integrated Network Management (IM '01)* (pp. 14-18). Seattle: IEEE Press.

[211] Ribeiro, C., Zúquete, A., Ferreira, P., and Guedes, P. (1999). SPL: An access control language for security policies with complex constraints. *Proceedings of the Network and Distributed System Security Symposium*, (pp. 89-107).

[212] World Wide Web Consortium (W3C). (n.d.). *Enterprise Privacy Authorization Language (EPAL 1.2)*. Retrieved September 10, 2012, from http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/

[213] World Wide Web Consortium (W3C). (n.d.). *PolicyLangReview*. Retrieved September 10, 2012, from http://www.w3.org/Policy/pling/wiki/PolicyLangReview

[214] Nadler, R. (2009, March 26). *Software Verification vs Validation*. Retrieved September 11, 2012, from Bob on Medical Device Software: http://rdn-consulting.com/blog/2009/03/26/software-verification-vs-validation/

[215] Song, E., Franca, R., Kim, H., and Ray, I. (2007). Checking Policy Enforcement in an Access Control Aspect Model. *Proceedings of the International Conference on Convergence Technology and Information Convergence (CTIC) '07*. Anaheim: Association for Computing Machinery.

[216] Goldman, M., and Katz, S. (2007). MAVEN: Modular Aspect Verification. *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems (TACAS '07)* (pp. 308-322). Braga: Springer-Verlag.

[217] Salomie, I., Cioara, T., Anghel, I., Dinsoreanu, M., and Salomie, T. (2007). A Layered Workflow Model Enhanced with Process Algebra Verification for Industrial Processes. *IEEE International Conference on Intelligent Computer Communication and Processing* (pp. 185-191). Cluj-Napoca, Romania: IEEE.

[218] Terpstra, F., and Adriaans, P. (2007). New directions in Workflow formalisms. *UK e-Science All Hands Meeting*. Nottingham.

[219] Gross, A., and Doerr, J. (2009). EPC vs. UML Activity Diagram - Two Experiments Examining their Usefulness for Requirements Engineering. *17th IEEE International Requirements Engineering Conference (RE '09)* (pp. 47-56). Atlanta: IEEE Computer Society.

[220] Demchak, B., and Krüger, I. (2012). *A Model-Driven Engineering Approach to Requirement Elicitation for Policy-Reactive Cyberinfrastructures*. La Jolla: University of California, San Diego, Computer Science Department.

[221] Viega, J. (2005). Building security requirements with CLASP. *Proceedings of the 2005 Workshop on Software Engineering for Secure Systems Building Trustworthy Applications* (pp. 1-7). St Louis: Association for Computing Machinery.

[222] Mavin, A., Wilkinson, P., Harwood, A., and Novak, M. (2009). Easy Approach to Requirements Syntax (EARS). *Proceedings of the 2009 17th IEEE International Requirements Engineering Conference (RE '09)* (pp. 317-322). Atlanta: IEEE Computer Society.

[223] Lodderstedt, T., Basin, D. A., and Doser, J. (2002). SecureUML: A UML-Based Modeling Language for Model-Driven Security. *Proceedings of the 5th International Conference on The Unified Modeling Language* (pp. 426-441). Dresden: Springer-Verlag.

[224] Kasal, K., Heurix, J., and Neubauer, T. (2011). Model-Driven Development Meets Security: An Evaluation of Current Approaches. *44th Hawaii International Conference on System Sciences (CD-ROM)*. Poipu, Hawaii, USA: The IEEE Computer Society Press.

[225] Bhattacharjee, A., and Shyamasundar, R. K. (2009). Activity Diagrams: A Formal Framework to Model Business Processes and Code Generation. *Journal of Object Technology*, 189-220.

[226] Wong, P. Y., and Gibbons, J. (2008). A Process Semantics for BPMN. *Proceedings of the 10th International Conference on Formal Methods and Software Engineering* (pp. 355-374). Kitakyushu-City: Springer-Verlag.

[227] Prater, J., Mueller, R., and Beauregard, B. (2012). An Ontological Approach to Oracle BPM. *Proceedings of the 2011 joint international conference on The Semantic Web (JIST '11)* (pp. 402-410). Hangzhou: Springer-Verlag.

[228] Parreiras, F. S. (2012). *Semantic Web and Model-Driven Engineering.* Piscataway: Wiley-IEEE Press.

[229] Chiba, S., and Leavens, G. T. (n.d.). *LNCS Transactions on Aspect-Oriented Software Development*. Retrieved October 12, 2012, from http://www.springer.com/computer/lncs?SGWID=0-164-2-109318-0

[230] Mussbacher, G., Amyot, D., and Araújo, J. (2010). Requirements Modeling with the Aspect-oriented User Requirements Notation (AoURN): A Case Study. In S. Katz, and M. Mezini, *Transactions on aspect-oriented software development VII* (pp. 23-68). Berlin/Heidelberg: Springer-Verlag.

[231] International Telecommunication Union. (2008, November). Retrieved October 12, 2012, from User requirements notation (URN) - Language definition: http://www.itu.int/rec/T-REC-Z.151/en

[232] Niu, N., and Easterbrook, S. (2007). Analysis of Early Aspects in Requirements Goal Models: A Concept-Driven Approach. In A. Rashid, and M. Aksit, *Transactions on Aspect-Oriented Software Development IV* (pp. 40-72). Berlin/Heidelberg: Springer-Verlag.

[233] Sampaio, A., Rashid, A., Chitchyan, R., and Rayson, P. (2007). EA-Miner: Towards Automation in Aspect-Oriented Requirements Engineering. In A. Rashid, and M. Aksit, *Transactions on Aspect-Oriented Software III* (pp. 4-39). Berlin/Heidelberg: Springer-Verlag.

[234] Nouh, M., Ziarati, R., Mouheb, D., Alhadidi, D., Debbabi, M., Wang, L., et al. (2010). Aspect Weaver: a Model Transformation Approach for UML Models. *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research* (pp. 139-153). Toronto: IBM Corporation.

[235] Fuentes, L., and Sánchez, P. (August, 2007). Designing and Weaving Aspect-Oriented Executable UML Models. *Journal of Object Technology*, 109-136.

[236] Groher, I., and Voelter, M. (2009). Aspect-Oriented Model-Driven Software Product Line Engineering. In S. Katz, H. Ossher, R. France, and J.-M. Jézéquel, *Transactions on Aspect-Oriented Software Development VI* (pp. 111-152). Berlin/Heidelberg: Springer-Verlag.

[237] Chitchyan, R., Pinto, M., Rashid, A., and Fuentes, L. (2007). COMPASS: Composition-Centric Mapping of Aspectual Requirements to Architecture. In A. Rashid, and M. Aksit, *Transactions on Aspect-Oriented Software Development IV* (pp. 3-53). Berlin/Heidelberg: Springer-Verlag.

[238] Krüger, I., Meisinger, M., and Menarini, M. (2007). Runtime Verification of Interactions: From MSCs to Aspects. *Proceedings of the 7th International Converence on Runtime Verification (RV '07)* (pp. 63-74). Vancover: Springer-Verlag.

[239] Liu, W. W. (2009). *Refactoring-based Requirements Refinement Towards Design*. Toronto: University of Toronto.

[240] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2002). Modeling Early Requirements in Tropos: A Transformation Based Approach. *Second International Workshop on Agent-Oriented Software Engineering II (AOSE '01)* (pp. 151-168). Montreal: Springer-Verlag.

[241] Krüger, I., Demchak, B., and Menarini, M. (212). Dynamic Service Composition and Deployment with OpenRichServices. In M. Heisel, *Software Service and Application Engineering: Essays Dedicated to Bernd Krämer on the Occasion of His 65th Birthday* (pp. 120-146). Berlin Heidelberg: Springer.

[242] Object Management Group (OMG). (n.d.). Retrieved October 16, 2012, from http://www.omg.org/

[243] Object Management Group (OMG). (2009, February). *UML 2.2*. Retrieved October 16, 2012, from http://www.omg.org/spec/UML/2.2/

[244] Ambler, S. W. (2005). *The Elements of UML 2.0 Style*. Cambridge: Cambridge University Press.

[245] Milner, R. (1982). *A Calculus of Communicating Systems*. Secaucus, NJ: Springer-Verlag.

[246] Zilio, S. D. (2001). Mobile processes: a commented bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. D. Ryan, *Modeling and verification of parallel processes* (pp. 206-222). New York: Springer-Verlag.

[247] Milner, R. (1991, October). The Polyadic Pi-Calculus: a Tutorial. *Technical Report ECS-LFCS-91-180*. UK: Computer Science Department, University of Edinburgh.

[248] Sangiorgi, D. (1993, May). Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. *PhD Thesis*. UK: Computer Science Department, University of Edinburgh.

[249] OASIS. (2007, April 11). *Web Services Business Process Execution Language Version 2.0.* Retrieved March 1, 2012, from http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[250] Kitchin, D., Cook, W., and Misra, J. (2006). A Language for Task Orchestration and its Semantic Properties. *Proceedings of Concur'06* (pp. 477-491). Bonn, Germany: Springer.

[251] Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics.* Amsterdam: North Holland.

[252] Böhm, C., and Jacopini, G. (1979). Flow diagrams, Turing machines and languages with only two formation rules. *Classics in software engineering*, 11-25.

[253] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. (1972). *Structured Programming.* London: Academic Press.

[254] Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects.* Chichester, UK: Wiley.

[255] International Organization for Standardization. (2006, September 20). *ISO/IEC 10181-3:1996.* Retrieved January 21, 2012, from www.iso.org

[256] Oracle Corporation. (n.d.). *JSR-000154 JavaTM Servlet 2.5 Specification*. Retrieved September 13, 2012, from http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html

[257] Lee, L. (1981, November 7). Retrieved from A Service Oriented Approach to Wings, Photography, and Exascale Databases: http://fatlucas.com