Published in final edited form as:

Proc IEEE Int Symp Policies Distrib Syst Netw. 2012 July ; 2012: 17–24. doi:10.1109/POLICY.2012.13.

Policy Driven Development: Flexible Policy Insertion for Large Scale Systems

Barry Demchak and Ingolf Krüger

Computer Science and Engineering Department, University of California, San Diego, La Jolla, USA

Barry Demchak: bdemchak@ucsd.edu; Ingolf Krüger: ikruenger@ucsd.edu

Abstract

The success of a software system depends critically on how well it reflects and adapts to stakeholder requirements. Traditional development methods often frustrate stakeholders by creating long latencies between requirement articulation and system deployment, especially in large scale systems. One source of latency is the maintenance of policy decisions encoded directly into system workflows at development time, including those involving access control and feature set selection.

We created the Policy Driven Development (PDD) methodology to address these development latencies by enabling the flexible injection of decision points into existing workflows *at runtime*, thus enabling policy composition that integrates requirements furnished by multiple, oblivious stakeholder groups.

Using PDD, we designed and implemented a production cyberinfrastructure that demonstrates policy and workflow injection that quickly implements stakeholder requirements, including features not contemplated in the original system design. PDD provides a path to quickly and cost effectively evolve such applications over a long lifetime.

I. Introduction

A significant threat to the real world success of large scale Systems of Systems (SoS) lies in their inability to realize stakeholder requirements on time scales relevant to their large, diverse, and growing stakeholder populations. Changes in stakeholder requirements often require application modifications and re-deployment, which are often lengthy and error-prone processes. A common technique for avoiding these latencies is embedding policy decisions into application workflows (at *development* time), thereby enabling the specification of decision criteria at runtime via externally stored policy statements. This technique addresses known requirements, but it has limited value in addressing changing or emerging requirements. In this paper, we describe our Policy Driven Development (PDD) methodology, which empowers developers to address emergent requirements via *runtime* placement of policy decisions without incurring large deployment latencies, thereby enabling greater stakeholder enfranchisement.

Policy decisions represent the runtime configuration of an application based on a combination of policy statements and runtime parameters such as user identity, privileges

granted, and application and environment state. Given that an application can be modeled as a collection of workflows, existing practice calls for policy decisions to be embedded directly into those workflows, thereby constraining or modifying the workflow by choosing one execution path instead of (or in addition to) another. In this regard, policy decisions enable access control (through Grant/Deny decisions) and the incorporation or exclusion of application feature sets, thus avoiding development, maintenance, deployment, and administration costs otherwise incurred to support each user in each context.

While the value of embedding policy decisions is well established, there are also well established costs: workflows having excessive decision points present maintenance and evolution challenges (because they are often entangled, error-prone, and hard to test), and inserting, changing, or removing decision points (responsive to stakeholder requirement changes) incurs application testing and deployment overhead.

These costs are particularly significant in large scale SoS applications, which serve many stakeholder communities that are distributed across multiple domains and may be oblivious to each other. Over time, large stakeholder groups independently generate many requirement changes, some of which can be fulfilled by altering policy statements, while others require addition, modification, or removal of hard coded decision points and embedded workflows. The result is often increasingly complex (and unwieldy) policies and workflows, leading to long maintenance intervals.

SoS application viability depends on stakeholder requirements being fulfilled quickly and faithfully, but large scale systems have notoriously long maintenance and deployment cycles. Because such a system's user base tends to be large and global, system downtime for staging a deployment itself tends to be costly, which naturally justifies long intervals between deployments. Perversely, requirements realized during such intervals may be superseded even before the system can be re-deployed, and incorrectly implemented requirements can result in disaffected stakeholders over multiple releases.

Finally, in SoS development, not all source code may be available, thereby making source-level adjustment of decision points and workflows very difficult if not impossible.

To address these challenges, we developed Policy Driven Design (PDD), a methodology that applies principles of Service Oriented Architecture (SOA) and Aspect Oriented System Design (AOSD) to enable the addition, modification, or removal of policy decision points and workflows *at runtime*. PDD encourages the decomposition of requirements into simple workflows each of which addresses a separate concern, and which can be composed together and with other workflows at runtime based on policy decisions.

By shifting the focus of application development to runtime-composable workflows, PDD presents an opportunity to also shift the developer-stakeholder relationship. Rapid translation of stakeholder requirements into policy-mediated workflow reorganization encourages an emerging policy programming discipline that cooperates and coordinates with stakeholders more frequently and effectively than with traditional methodologies, thereby empowering otherwise disenfranchised SoS stakeholder communities and improving SoS viability.

PDD's workflow composition-based approach contrasts with recent approaches implementing usage and access control via predicates on dataflow independent of workflows ([1]), events defined on workflows at development time ([2]), and events emitted by objects defined at development time ([3]).

A. Workflows and Features

Under PDD, stakeholder requirements are expressed as a collection of workflows that coordinate to execute tasks that address requirement-related concerns. Such requirements can be either functional or non-functional (e.g., information availability, integrity, confidentiality, non-repudiation, and accessibility), and are often compound, complex, conflicting, and intertwined with other requirements. While there are a number of ways to factor requirements into workflows, PDD encourages a separation of concerns approach that leads to a collection of basic workflows that address a dominant decomposition. Aspects of requirements not addressed as basic workflows are addressed as ancillary workflows, which can be composed onto basic workflows or onto each other in order to either constrain or augment workflow behaviors. Such compositions may affect single workflow actions, or may crosscut a number of actions in one or more workflows.

Note that there is wide disagreement regarding the definition of a workflow, as the term serves a number of interests and purposes ([4]). No comprehensive formal workflow model exists, though workflows have been modeled formally using Petri nets, and expressed with various degrees of precision using π -calculus, UML activity diagrams, and others.

For the purpose of this paper, we adopt a generic working definition from ORC [5]: “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.” Such activities, themselves, can be further decomposed as either workflows or atomic operations. Equivalently, we observe that a workflow is a decomposable relation that transforms one set of inputs into a set of outputs; in Section III.A, we cast the SOA definition of service in these terms. In this paper, we use *workflow*, *relation*, and *service* interchangeably as duals, with *workflow* emphasizing a description of data and control flow (as in the business domain), *relation* emphasizing formal properties of workflows, and *service* applying to the implementation domain.

Ref. [6] defines an application *feature* as an optional or incremental unit of functionality, and addresses the consequences and issues surrounding feature design and interaction. We adopt that definition and assert that a feature can be implemented by a workflow or coordinated workflows.

B. Policy Driven Design

Under PDD, activities within a workflow are modeled as services with well-defined interfaces, and data flowing between activities is modeled as well-defined messages. PDD models decision points as policy evaluations carried out on messages exchanged by services, where a *policy* is a decision that chooses to either continue the workflow, execute a different workflow, or both. The policy and the service interaction to which it applies are specified by

a policy programmer (PP, which could be a stakeholder) and resides in external store, separately from the workflow to which it applies. Consequently, so long as service interfaces are observed, PDD improves the viability of applications under maintenance by enabling the composition of policy decisions and workflows into a running application, thereby avoiding many of the costs of deployment-based traditional methods.

PDD results in a clear positioning of policies within an otherwise highly complex, distributed software system. Policy checking can occur at that position, without having to trace many distributed software components into their internals or requiring access to source code.

Finally, policy injection via message interception presents opportunities for policy-based message transformation, thereby enabling policies that constrain or augment data flows per stakeholder requirements responsive to regulations such as HIPAA, SOX, PCI, GLBA, and NIH/NSF covenants – a capability difficult to realize using existing policy techniques.

C. Paper Structure

The contributions of this paper are: 1) a formal description of PDD's SOA-based composition of policy evaluation and policy-derived workflows, 2) a means for specifying (*at runtime*) the service interaction to which a policy applies, 3) leveraging composition to inject Grant/Deny policies, filter policies, and policy-selected application features in a SOA context, 4) a policy-based means for composing multiple policies defined on a single service interaction.

We start by describing a functioning and successful SoS case study that illustrates our use of PDD (Section II), followed by the theory behind PDD and how it enables policy evaluation (Section III). We then describe the application of PDD in our case study (Section IV), and evaluate its success (Section V). We discuss issues raised by PDD and describe its relationship to existing and future work (Section VI). Finally, we summarize our progress to date (Sections VII).

II. PALMS: A Running Example

The PALMS Cyberinfrastructure (or simply “PALMS”) [7] represents a case study that demonstrates the use of PDD and several issues surrounding it. PALMS was created to support the PALMS Project, which 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. It employs a number of principle investigators (PIs) each defining distinct studies involving study participants, data collection, and analysis, all subject to HIPAA, NIH, and other constraints and strictures. PALMS was created to serve the EB community at large, which consists of an emerging federation of independent PIs.

PALMS requirements focus on the collection and maintenance of participant, observation, and post-analysis data, and on access to such data and the analysis capabilities related to them. Given that each study is unique as to content, funding agency requirements, personnel

organization, and the personal preferences of its PI, a critical requirement is that PIs be able to control who can affect and access their data, analyses, and results, and under what conditions. Additionally, given that the PALMS community was and remains emergent, another critical requirement is that workflows be rapidly and economically adaptable to the community's emerging requirements. Today, PALMS realizes many of these requirements, and serves a growing community of over 20 research groups worldwide.

To achieve a good fit between budget constraints and the expectation of emerging requirements from an unknown but growing stakeholder base, we conceived of PALMS as a highly evolvable but complex SoS, where different concerns are modeled and implemented separately, and are composed into a functional system incrementally. We modeled PALMS using the Rich Service (RS) [8] architectural blueprint. A Rich Service is a type of Service Oriented Architecture (SOA) that models a workflow as an orchestration of loosely coupled services, where services themselves can be decomposed into workflows or implemented as atomic functions. They form a service hierarchy conforming to a composite pattern [9].

A Rich Service is a service that transforms one or more streams of input messages into one or more streams of output messages, and can be used to implement a workflow activity. Figure 1 shows the structure of a Rich Service as a simplification of the PALMS design. The Service/Data Connector (SDC) ❶ defines the Rich Service's input streams and output streams. The message router transforms ❺ the input streams to output streams by orchestrating interactions between Rich Application Services (RASs) ❷ – it passes messages (via a message transport ❹) to RASs, receives RAS output messages, and determines which RAS outputs feed into RAS inputs. Rich Infrastructure Services (RISs) ❸ perform functions that crosscut RAS interactions.

Given that the message router mediates RAS-to-RAS interactions, it can also enable crosscutting processing by interposing a RIS into the interaction, replacing the RAS-to-RAS interaction with a RAS-to-RIS-to-RAS interaction. Critically, we use this interposition to enable injection of policy-based constraints and features onto simple, basic workflows, responsive to emerging stakeholder requirements. Modifications to workflows involved modifications to (or compositions of) demonstrably independent service orchestrations, thereby reducing testing overhead and promoting workflow reusability.

Both RASs and RISs can, themselves, be Rich Services, thus enabling the decomposition of the abstractions each represents and completing the composite pattern. RASs or RISs that are decomposed as Rich Services (e.g., Device Repository) have SDCs ❹, RASs, RISs, and routing of their own.

As a demonstration of PDD, Figure 1 shows a simplified Rich Service that demonstrates the orchestration of Device Repository and Calculation Repository RASs to perform some function (e.g., defining resources available to a study).

Logically, PALMS is modeled as an RS represented by the PALMS SDC ❶. Each workflow is modeled as an orchestration of RASs ❷ implementing PALMS repository functions (e.g., Device Repository and Data Repository abstractions enable storage and retrieval of sensor device metadata and data, respectively). Each RAS can be decomposed

into workflows that validate and store repository data (not shown). Interactions between RASs can be intercepted by RISs ③ implementing crosscutting concerns (i.e., Policy Evaluation and Event Logging RISs), or context management services (i.e., a Context Management RIS) that crosscut the crosscutting concerns. Not shown are the particular routing functions that implement each RAS-based workflow, including interceptors that incorporate RISs into a workflow.

At a deployment level, the PALMS logical architecture is mapped onto a Mule Enterprise Service Bus (ESB) [10] augmented by PALMS' context management RIS (shown in Figure 1) to support the lifecycle and availability properties needed by a scalable, distributable, policy-centric SOA.

III. Theory of Solution

Under PDD, a policy decision is injected into a target workflow at a runtime-defined decision point, and may lead to the composition of a new workflow onto the target. The decision can be based on a number of factors, including data flows (or histories of data flows) managed by the target or other workflows, or the state of the application or external system. As such, PDD policies encompass traditional Grant/Deny decisions, and support obligation enforcement, filtering of data flows (including augmentation and decimation), and the injection of value-added concerns (including entire feature sets).

In this section, we describe the PDD mechanisms by which a policy decision is injected into a workflow. PDD expresses a workflow as an orchestration of services, so we first present basic service orchestration concepts, including a definition of service, service interactions, service contracts, and service composition. We then define orchestration in terms of service decomposition, and explain how a crosscutting concern (such as policy evaluation) is injected into a service orchestration, thereby creating a decision point. Finally, we explain how a single policy is evaluated and how policy composition occurs when multiple policies are defined at a single decision point.

As an illustration, we suppose that the PALMS RS (referred to as S) implements a function that stores new device metadata and then uploads data from those new devices (per Section Figure 1). The workflow involves invoking the Device Repository RAS (referred to as P) followed by the Data Repository RAS (referred to as Q); Q depends on information from P . From an external perspective, S implements the function; internally, S decomposes into an orchestration of P and Q .

A. Service Definition and Composition

In this section, we describe how Rich Services (including RASs and RISs) logically interact with each other, notwithstanding mediation by the message router. Specifically, we define how messages are exchanged, which forms the basis for interception that enables policy decisions.

A single Rich Service S implements a relation $f: I \rightarrow \wp(O)$ where I and O represent disjoint sets of directed *channels*, each containing an ordered set of *messages* M . Relative to S , I

represents *input* channels, O represents *output* channels, where I and O are defined by the Service Data Connector as shown in Figure 2a. We define a *channel valuation* at time t as $C_t = I_t \cup O_t$ where I_t is the sequence of messages on each input channel at time t , and O_t is the sequence of messages on each output channel at time t . A *channel history* C is defined as the union of all channel valuations over time, and can be partitioned into an input component I and an output component O . We define a *service contract* as the set of channel valuations for which $f: I \rightarrow \wp(O)$ holds, and S is said to *fulfill* a service contract if it maps I to O as defined in the service contract. Note that relation f can execute non-deterministically, and controls which channels are read, written, and when.

We define a service composition between two services P and Q as $P \otimes Q$ as shown in Figure 2b. Some of P 's input channels are connected to some of Q 's output channels, and vice versa. Such channels are called *hidden channels*, and their service contract is called the *hidden channel service contract*. Channels not connecting the two services define the *composition's* channel valuations, channel history, and service contract. Formal semantics for service composition are given in [11], and includes a demonstration that service composition is associative – $(P \otimes Q) \otimes R = P \otimes (Q \otimes R)$.

In the case of our sample workflow, the PALMS function is implemented by composing the RASSs, with a message containing device information flowing from P to Q via the hidden channel, and the composition returning a result via the SDC.

B. Service Decomposition and Routing

We define the *decomposition* of a service S as an orchestration of sub-services to achieve the functionality of S . 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 $P \otimes R_{PQ} \otimes Q$, and R_{PQ} mediates between P and Q (as shown in Figure 2c) and fulfills the hidden service contract for $P \otimes Q$.

Critically, R_{PQ} is free to interact with a different service W instead of Q , provided that W fulfills Q 's service contract. (Similarly, W can replace P – for the sake of convenience, we focus on replacement of Q and assert that arguments that apply to Q also apply to P .) In addition to facilitating the composition, R_{PQ} can interact with a RIS to implement functionality that crosscuts the $P \otimes Q$ interaction.

In our sample workflow, the router function would synchronously interact with the Policy Evaluation RIS to determine a possible replacement service, which might (instead) upload data to a different database than the default service. Alternatively, a routing function may communicate with other RISs (e.g., asynchronously with the Event Manager RIS to accomplish message logging) representing a composable feature.

C. Control and Filter Policies

Under PDD, policies are decisions that affect either the data flowing between services or the operations performed on the data. Conceptually, they leverage the duality between service orientation and data flow orientation. Under a service-centric view, a service represents an

action consuming input channel valuations and producing output channel valuations; under a channel-centric view, input valuations beget output valuations via a transform implemented as a service.

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). It represents a means by which new workflows can be composed into an existing workflow. W_{cp} is calculated by $cp_{PQ}(I_{Q_h}, \vec{C_{PQ}}, P, Q)$, where I_{Q_h} is Q 's hidden input channel, $\vec{C_{PQ}}$ is the $P \otimes Q$ channel history, P is the interaction service, and Q is the default target service. In the corresponding router interaction, $P \otimes R_{PQ} \otimes W_{cpPQ}$, R_{PQ} is composed with the calculated service W_{cpPQ} instead of the default service Q . A control policy chooses between various services (which may be either pre-existing or constructed at runtime, as allowed by the runtime orchestration system); the choice can also leverage the channel history for other interactions in the system.

Based on the channel-centric view, a *filter policy* fp calculates a service W_{fp} to intercept Q 's inputs I_{Q_h} and replaces them with other values I'_{Q_h} that are valid inputs to Q . It represents a means by which data flows can be constrained, augmented, inspected, or replaced. fp is a function $fp_{PQ}(I_{Q_h}, \vec{C_{PQ}}, P, Q)$ whose arguments are the same as for cp . The corresponding router interaction is $P \otimes R_{PQ} \otimes W_{fpPQ} \otimes Q$. Similar to a control policy, a filter policy returns W_{fp} , which effects the channel value transformation. In our sample workflow, a filter policy might configure upload parameters, log data uploads, or even delay uploads, per the user's credentials.

Both control and filter policies are expressions authored by a stakeholder or PP per emerging stakeholder requirements. They are inserted into a policy repository, indexed by the service interaction (P, Q) to which it applies.

The router function R_{PQ} , executes both filter and control policies for an interaction, and injects the resulting workflows. When both a filter and control policy exist for an interaction, the filter policy is executed before the control policy, meaning that the input channel is transformed before the target service is executed. In Figure 2c, a filter policy for $P \otimes Q$ in service S replaces I_Q , while a control policy replaces the service Q .

Note that Figure 2c also shows that the I_P channel allows communications between services Q and P . A filter policy can also exist for the I_P channel. When the service contract for P requires a reply from Q , a request/reply pattern exists – a filter policy applied to I_Q is called a pre-filter (because it precedes the execution of Q), and a filter applied to I_P is called a post-filter. If the I_P channel can support error message content, the P - Q interaction can fail gracefully, enabling a control policy to implement access control by returning an error when appropriate. (A stakeholder requirement may call for the graceful failure of Q – if I_P does not support error message content, the requirement implies follow-on requirements for P .)

A service returned by a control or filter policy is constrained to fulfill its service contract, but can have any activity or effect. It is free to execute a workflow involving services either serially, in parallel, or asynchronously. Exploiting this allows such policies to inject services

that execute workflows implementing application features as separate concerns maintaining their own state. 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 conveniently accessed by capturing it in repository-oriented services.

D. Composition Policies

While control and filter policies affect control and data flows, *composition policies* determine how such policies combine when multiple control or filter policies apply to the same service interaction, as could be the case when single or multiple stakeholders specify policies on the same workflow. A *control composition policy* is a decision that accepts multiple control policies and returns a single service W_{cp} that represents them. Likewise, a *filter composition policy* returns a single service W_{fp} that represents multiple filter policies. In our simple workflow, a composed filter policy might transform upload parameters, log the data upload, and delay it.

Generally, a composition policy has the form $\chi_{PQ}(E_{PQ}, I_{Qh}, \vec{C_{PQ}}, P, Q)$ where E_{PQ} is a set of control or filter policies applicable to $P \otimes Q$. Composition policies authored by a PP are inserted into a composition policy repository, indexed by the service interaction $P \otimes Q$ to which it applies.

For a given service interaction $P \otimes Q$, there can be at most one control composition policy and at most one filter composition policy. A composition policy is implemented by a Policy Evaluation RIS executing a four stage workflow: 1) retrieve set E_{PQ} from one or more policy repositories, 2) retrieve χ_{PQ} from a composition policy repository, 3) evaluate $W_{PQ} = \chi_{PQ}(E_{PQ}, I_{Qh}, P, Q)$, and 4) return W_{PQ} . If $|E_{PQ}| = 0$, no W_{PQ} is returned. If χ_{PQ} is undefined and $|E_{PQ}| = 1$, W_{PQ} is returned undefined.

While the router function R_{PQ} executes services based on filter and control policies, it calls the Policy Evaluation RIS to calculate filter and control compositions. Composition policies can be used to define policy hierarchy (e.g., one policy overriding all others), implement unanimous decision (e.g., returning a Grant decision only if all composed policies return Grant decisions), or create other evaluation strategies, though without any guarantees of conflict detection or mitigation.

IV. Case Study

The PALMS policy system is a component in the PALMS SoS architecture – it is both a standalone application and a Rich Service that provides policy evaluation functionality. It consists of policy authorship, policy repository, and policy evaluation RSs, themselves architected as RSs and modeled in Figure 3 (with SDCs and the router/messenger unshown for brevity). The Authoring service interacts with the Policy Repository service to retrieve and deploy policy expressions, and the Policy Evaluation service interacts with the Policy Repository to retrieve policy expressions for execution.

The Authoring service enables PPs to define and maintain policies on PALMS workflows. The Policy Repository service maintains policy repositories for each study, all studies run by a given PI (aka *study group*), the PALMS system administration as a whole (aka *admin*), and for policy compositions.

The Policy Evaluation service is composed onto all PALMS service interactions as a RIS via the interactions' routing function. It is decomposed into services that implement the workflow stages defined in Section III.D. Note that Figure 3 shows a Policy Evaluation RIS composed into Policy RS workflows, enabling the injection of policy on policy authoring itself.

A PALMS policy is written as an XQuery [12] expression, and is executed by Saxonica's Saxon XQuery processor. XQuery was chosen as the base policy language for a number of reasons. PALMS messages are transmitted as Java objects, which are fungible with XML documents, and are therefore easily and efficiently filtered in flight using XQuery and its close cousins XPath and XSLT. XQuery expressions can draw on library functions written either as XQuery functions or Java code, including domain-specific libraries and context access functions. They can be written and executed as plain text (or just-in-time compiled object), thereby avoiding compiler and runtime system dependencies in the authoring and repository services. They can be succinct enough to encode domain-level abstractions (as a DSL [13]), which helps maintain alignment between stakeholder requirements and policy expression, and invites stakeholders into the policy formulation exercise.

Note that the PALMS XQuery policy support library defines a number of policies as simple XQuery function calls (as in Sections IV.A and IV.B) that invoke more complex XQuery-based functions. This supports reuse of common policies; however, policy authors are free to author policies that, themselves, are complex XQuery statements, which may or may not call functions in the XQuery support libraries.

To support policy execution, the Context Manager RIS maintains state that has workflow, session, application, or customizable scope. The Context Manager is itself a separate concern that is composed into either the PALMS RS or the PALMS Policy RS, and exposes services callable by policies (via an XQuery library) or PALMS workflows. For example, the Context Manager RIS captures the user's X.509 credential at the outset of a PALMS API invocation and makes it available to policies for the duration of the API workflow. Note that because Mule enables access to services (including the Context Manager RIS) distributed across a network, maintaining context in a standalone service enables policies to access coherent state regardless of where they execute.

Initial and follow-on requirements specified by PALMS stakeholders include access control decisions, which a PP can factor into predicates applicable to workflows as either prefilters (used to qualify service requests), post-filters (used to redact excessive or inappropriate replies), and control policies (implementing Grant/Deny decisions). Access control decisions are supported by XQuery library functions that relate user identity to a hierarchically defined role-based (RBAC) and access control list-based (ACL) taxonomy maintainable in Grouper [14] by stakeholders themselves.

A. Filter Policies

An example of a post-filter can be applied to a message containing a list of data elements returned by a query to the Data Repository service. Suppose that under HIPAA requirements, a study's data stream should not carry personally identifying information (PII) unless the user is the study's principle investigator (PI) or research assistant (RA). A post filter residing in the study-level policy repository and applying to Data Repository interactions would be:

```
palms:filter-pii-by-role('PI,RA')
```

where **palms:** is an XQuery namespace, and the **filter-pii-by-role('PI,RA')** function calls an XQuery function that determines whether the current user fills the PI or RA roles, and conditionally eliminates all PII in the repository query result message. (In general, the definition of PII can be encoded into the policy, or can be accessed elsewhere by the policy.) The XQuery function leverages XQuery policy support libraries that assist in message transformation or interface to external services (e.g., determining the member list for a role, setting state, or executing a feature workflow such as audit capture). Note that the decisions made by a filter policy nominally depend on the user's credentials and the content of the message; they can also depend on state set by the policy or other policies, and maintained in services such as Context Manager.

Additionally, a filter policy is free to execute a workflow implementing an application feature (via an XQuery function call) using policy parameters, the current message, or system state to select the workflow or set the workflow parameters (e.g., implementing audit or provenance tracking features).

PALMS currently supports three filter composition policies, each written in XQuery:

```
palms:compose-all()
```

```
palms:compose-hierarchical()
```

```
palms:compose-preemptive()
```

where the **palms:compose-all()** function composes pre-filters – it chains together all filter policies defined for the service interaction in an undefined order. **palms:compose-hierarchical()** executes admin filters first, then study group filters, and finally study filters. **palms:compose-preemptive()** implements a filter hierarchy – it executes only admin filters, or study group filters (if there are no admin filters), or study filters (if there are no admin or study group filters).

The PALMS composition policies address the admin/group/study hierarchy concerns of the PALMS application. Other composition functions are possible, given different application requirements.

B. Control Policies

A PALMS control policy assumes that the interaction to which it applies follows a request/reply pattern. An example of a control policy can be used to prevent access to a repository service by insufficiently credentialed users:

```
if (palms:user-in-study-role('PI'))

then ( )

else 'Disallowed'
```

where the `user-in-study-role('PI')` function determines whether the current user has the PI role for the study referenced in the service message. If so, the workflow continues; otherwise, a different workflow is chosen – specifically, a trivial workflow that returns an error message instead of continuing the workflow. Other forms of the `else` clause specify a service that accepts the current message and might perform substitute processing, while still fulfilling the service contract (e.g., choosing a weaker analysis function per the user's credentials).

As with a filter policy, a control policy can set or base decisions on its own state or state maintained in other repositories.

PALMS supports two control composition policies:

```
palms:compose-unanimous( )

palms:compose-override( )
```

where the `palms:compose-unanimous( )` function executes all control policies and concatenates their results – if the result is `( )`, the workflow continues; otherwise, the concatenated error message is returned. The `palms:compose-override( )` function is similar to the `palms:compose-filter-preemptive( )` function – it executes control policies preferentially based on the repository from which they come.

V. Execution and Results

The success of the PALMS policy system is determined by whether it correctly implements access control and whether it can compose multiple policies reflecting independent requirements contributed by stakeholders oblivious to each other. (Note that policy execution speed is not a major PALMS success criteria – user-perceived execution latencies depend primarily on the rendering speed of the PALMS Browser component, its Internet-based communication with the PALMS service, and data repository access within the PALMS service. Consequently, policy execution speed did not emerge as a bottleneck, and so was not measured.)

PALMS' post-filters have successfully decimated query results for over a year, resulting in distinct users receiving only data intended for them. Also, control policies are successful in

reserving critical functionality (e.g., creating a study) according to role membership per PALMS' RBAC repository.

VI. Discussion, Related, and Future Work

Service Definition and Composition

At a logical level, PDD's service composition relies critically on concepts from interface-oriented design (IOD [15], the separation of an object's interface from its implementation for the sake of reusability, maintainability, and extensibility) and Design by Contract (DbC [16], using method pre/post-conditions and invariants to enable proof of system attributes). PDD augments IOD and DbC by defining interfaces more broadly as message streams that form input and output channels, thereby defining correct execution in terms of legal sequences of valid input and output values, and leading to constraints on protocols and service level agreements (SLAs). PDD's reliance on Rich Services enables a hierarchical view of system correctness.

Orchestration and Routing

Existing workflow modeling and execution languages (e.g., BPEL, WS-CDL, WSCL, MSCs, and UML) statically define service orchestrations. AO4BPEL [17] extends BPEL by enabling feature composition via interception. BPEL Business Rules Integration [18] extends BPEL by composing business rules onto service interactions. However, both extensions define interception points at development time. PDD's use of SOA-style interception enables selection of policy evaluation points at runtime, and uses router functions to create runtime-defined orchestrations.

Note that PDD assumes the ability to specify message interception at runtime, and that target workflows are broken into interceptable interactions. Under orchestration systems where either of these conditions is unmet, PDD is of limited value.

Policy Engines and Composition

Robust policy engines (e.g., PERMIS [19]) define secure identity, attribute, and policy management services that feed into a policy decision point (PDP) separate from an enactment point (PEP), per ISO 10181-3, with policies expressed in various languages (e.g., XACML, X-Sec, PERMIS, and Akenti). Using common methodologies, PDPs and PEPs are coded statically into a workflow. PDD's Policy Evaluation (PE) service and routing functions operate as PDPs and PEPs, respectively. Traditional policy engines could be wrapped into the PE, but this alone would not address the SoS requirement for runtime policy composition or message stream filtering. Integrating such engines into a workflow or composition framework (e.g., WS-CDL, et al from above) within a PDP would entail high overhead and complexity as compared to PDD's XQuery approach, and would not address message stream filtering.

Policy and Context

PDD enables the coordination of multiple policies via retained state (as with PALMS' Context Manager). Coordinated policies amount to stateful applications crosscutting one or

more workflows. For example, in a Multi Session Separation of Duties (MSoD) policy [19] a workflow action is recorded (by one policy) and a subsequent workflow action is constrained (by another policy). PERMIS packages and maintains coordinated policy applications as atomic entities (XML documents). Under SoS, such applications may be plentiful, and their packaging and management is a critical feature unsolved in PDD.

Similarly, PALMS' policies are currently developed *in vitro* using IDEs such as Syncro Soft's <oXygen/>, and then are deployed to policy repositories; PALMS provides policy execution logging, but leaves *in vivo* profiling, debugging, conflict detection, and correctness proofs as future development.

SOA-based Policy

xESB [1] integrates policy injection and evaluation as core ESB functionality, focusing on adding policy execution to existing SOAs. PDD's thrust is to drive the design of SOAs as collections of workflows constrained by injectable policy, and as SoS composed according to policy.

On each service interaction, xESB examines all policies to determine applicability; selection according to complex criteria or administrative domain may be computationally costly. PDD policy is tied to a particular interaction, thus avoiding such selection costs. As the set of service interactions changes, xESB policies automatically adapt, and PDD policies must be manually adapted. Either approach risks inappropriate policy execution depending on the semantics of the service interactions involved. xESB policies follow narrowly defined syntax and semantics, which cover a wide range of important obligation and rule use cases, and which can be easily model checked. PDD policies are expressed as XQuery library calls, enabling development of DSLs that encourage close alignment of policies with stakeholder interests. Expressing xESB decision and action primitives, and model checking policies are left for future PDD development.

The injected events of [2] enable selection of policy into workflow but without flexible composition of multiple policies (contributed by/for independent stakeholders) or runtime placement of PEPs per emerging stakeholder requirements.

Ponder2 [3] is an event based system that uses an interceptor approach to associate policies with specific service interactions similarly to PDD. Ponder2's policy language [20] is mature and robust with respect to numerous policy types – it does not allow the runtime composition of one Ponder2 application onto another, which is critical for building and maintaining large, highly available SoS, and for which PDD was specifically designed. Demonstrating XQuery libraries equivalent to Ponder is left for future development.

Policy Deployment

While PDD's repository-based policy deployment currently admits policy synchronization across a distributed environment, it is insufficient to guarantee consistent application of coordinated policies, where a workflow may begin execution with one version of a coordinated policy, and may end with a different, later version. Guaranteeing policy version consistency is left for future development.

Leveraging the robust, end-to-end policy distribution facilities of existing policy engines would achieve safety, trust, and security properties required for reliable regulatory enforcement (e.g., HIPAA), and is left for future development.

Orchestration Languages

Currently, PDD policies return only pre-existing services W_{PQ} , consistent with the Mule deployment platform's message routing strategy. Integration of an orchestration language (e.g., Orc or Clojure) would enable W_{PQ} to express orchestrations dynamically constructed by PDD policies, and is left for future development.

Stakeholder Enfranchisement

PDD enfranchises stakeholders via programmer-mediated policy authorship. DSL-based [13] policies encourage more direct stakeholder involvement, either through direct authorship or GUI-driven interfaces. Enabling this requires a process that reliably refines stakeholder-level requirements and workflows into workflows executable as services. Such a process doesn't yet exist, and would be a valuable future development.

VII. Summary and Conclusion

PDD addresses a significant threat to the real world success of large scale SoS posed by SoS' inability to realize stakeholder requirements on time scales relevant to its large, diverse, and growing stakeholder populations. Existing methodologies address evolving requirements in long and error-prone development cycles, often failing to meet stakeholder needs and creating disaffection that weakens a SoS and community.

By integrating SOA and AOSD concepts, PDD proposes a realignment of the relationship between developer and stakeholder so as to better leverage both, resulting in expeditious realization of stakeholder requirements with high fidelity. As such, it can benefit from support or repurpose of existing and emerging techniques and disciplines, including incremental correctness proofs, secure policy management, comprehensive workflow languages, and workflow visualizations coupled with policy injection. Based on PDD's success with the PALMS case study, these goals seem attainable.

Acknowledgments

This work was funded under the PALMS project funding from the Genes, Environment and Health Initiative NIH/NCI Grant U01 CA130771, under the iDASH project funding from NIH Grant U54 HL108460, and by the California Institute for Telecommunications and Information Technology (Calit2). We are grateful to Kevin Patrick (PALMS PI) and the anonymous reviewers for their insights.

References

1. Gheorghe, G.; Neuhaus, S.; Crispo, B. xESB: An Enterprise Service Bus for Access and Usage Control. Proceedings of the 4th IFIP Intl Conf on Trust Management (IFIPTM2010); Morioka, Japan. 2010;
2. Neisse, R.; Pretschner, A.; Giacomo, V. A Trustworthy Usage Control Enforcement Framework. Proceedings of the 6th Intl Conf on Availability, Reliability, and Security (ARES '11); Vienna. 2011;
3. [3 April 2012] Ponder2 Wiki. Jan 8. 2012 [Online]. Available: www.ponder2.net

4. van der Aalst, W.; ter Hofstede, A. Workflow Patterns. 2010. [Online]. Available: <http://workflowpatterns.com/>. [On 29 Sept 2011]
5. Kitchin, D.; Cook, W.; Misra, J. Proceedings of Concur'06. Bonn, Germany: 2006. A Language for Task Orchestration and its Semantic Properties.
6. Zave, P. Software Requirements and Design: The Work of Michael Jackson. Chatham, New Jersey: Good Friends Publishing Company; 2010. Modularity in Distributed Feature Composition; p. 267
7. PALMS, [Online]. Available: <http://ucsd-palms-project.wikispaces.com/>. [3 June 2011].
8. Arrott, M.; Demchak, B.; Ermagan, V.; Farcas, C.; Farcas, E.; Krüger, I.; Menarini, M. Rich Services: The Integration Piece of the soa Puzzle. Proceedings of the IEEE International Conference on Web Services (ICWS); Washington, DC. 2007;
9. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley; 1995.
10. Mulesoft Inc. [14 June 2011] [Online]. Available: <http://www.mulesoft.org/>
11. Krüger, IH. Distributed System Design with Message Sequence Charts. München: Technische Universität München; 2000.
12. W3C. [29 Sept 2011] XQuery 1.0: An xml Query Language. Second Jan 3. 2011 [Online]. Available: <http://www.w3.org/TR/xquery/>
13. Fowler, M. Domain-Specific Languages. Addison-Wesley; 2010.
14. Internet2, [Online]. Available: <http://www.internet2.edu/grouper/>. [14 June 2011].
15. Pugh, K. Interface Oriented Design: With Patterns. Pragmatic Bookshelf; 2006.
16. Meyer B. Applying 'Design by Contract'. Computer. Oct.1992 :40–51.
17. Charfi A, Mezini M. AO4BPEL: An Aspect-oriented Extension to BPEL. World Wide Web. 2007:309–344.
18. Rosenberg, F.; Dustdar, S. Business Rules Integration in BPEL - A Service-Oriented Approach. Seventh IEEE International Conference on E-Commerce Technology; Munich. 2005;
19. Chadwick, D.; Xu, W.; Otenko, S.; Laborde, R.; Nassar, B. Multi-Session Separation of Duties (MSoD) for RBAC. First International Workshop on Security Technologies for Next Generation Collaborative Business Applications (SECOBAP'07); Istanbul, Turkey. 2007;
20. Damianou, N.; Dulay, N.; Lupu, E.; Sloman, M. The Ponder Policy Specification Language. Proceedings of the Intl Workshop on Policies for Distributed Systems and Networks (Policy '01); Bristol. 2001;

Figure 1. Simplified PALMS Rich Service

Figure 2. Basic Rich Services

Figure 3. PALMS Policy Rich Service