A Model-Driven Engineering Approach to Requirement Elicitation for Policy-Reactive Cyberinfrastructures

Barry Demchak and Ingolf Krüger
Computer Science and Engineering Department
University of California, San Diego
La Jolla, CA, USA
{bdemchak,ikrueger}@cs.ucsd.edu

***Abstract*—A cyberinfrastructure (CI) is an Internet-based collection of computing services dedicated to providing data storage, computations, and visualizations to a stakeholder ecosystem. A major CI function is to execute workflows on behalf of stakeholders. Each stakeholder will participate in the CI only if the workflows incorporate certain requirements, which may vary from stakeholder to stakeholder. Additionally, because successful CI use by one stakeholder depends on the results of successful use by other stakeholders, a failure of the CI to enforce stakeholder requirements risks the viability of the entire CI. A critical enabler for CIs is the efficient elicitation of stakeholder requirements, called policies, and their accurate and timely enactment. This paper presents a technique that combines UML Activity Diagrams and a Domain Specific Language (DSL) to enable stakeholders to formulate identity- and environment-based access control policies in the context of a workflow. To demonstrate the technique, we recruited exposure biologists as domain experts interested in inserting access control policies into a workflow in the PALMS CI, a health monitoring system currently used at UC San Diego. We found that not only could the experts successfully formulate their policies, but that translation of these policies to the implementation level was quick and accurate. This work extends work in design-level security engineering techniques (UMLsec[1] and SecureUML[2]), Activity Diagram formalisms[3], and DSLs[4]. In leveraging workflow visualization, efficient policy articulation, and timely enactment, this technique encourages exploration of the requirement space by domain experts.**

Keywords-cyberinfrastructure, Domain Specific Language, DSL, elicitation, UML, Activity Diagram

I. INTRODUCTION

As an emerging class of large scale computing systems, cyberinfrastructures (CIs) are poised to become important enablers of community-based computational and information processing in academia, government, and commerce. As an Internet-based distributed collection of data storage, computation, and visualization resources, CIs provide a substrate on which stakeholder communities can build and deliver value by organizing CI resource access through automated processes called *workflows*. They also provide an infrastructure through which communities can create significant additional value via cooperation and exchange.

A major threat to the promise of these systems is their complexity, which, in part, arises from the explicit and implicit need to satisfy the requirements of all communities simultaneously. While many technologies can be harnessed to tame different aspects of cyberinfrastructure construction, such solutions are at significant risk of failure without first proceeding from a solid base of actionable requirements, followed up by a requirement stream that reflects changing stakeholder needs.

Model Driven Engineering (MDE) is a methodology that encompasses many of the technologies needed to build a CI. At its core, MDE enables the management of complexity through the use of abstractions, evidenced by models expressed predominantly in the Unified Modeling Language (UML). These models can be used to describe both system structure and flow at various levels of abstraction, from the highly abstract to the highly concrete, and from the requirements stage to application deployment.

A complimentary approach is Service-Oriented Architecture (SOA), which, at its core, represents computing activities as patterns of interaction between computing components where information is exchanged via messages. By specifying interactions between components representing CI resources, a SOA can be used to model CI workflows. A critical feature of SOA systems is the ability to intercept messages traveling between components – thus enabling message transformations and additional message routing that can respond to stakeholder requirements by altering or augmenting workflows without compromising existing functionality.

While MDE and SOA can be leveraged to elicit and realize functional requirements (and to a lesser degree, non-functional requirements), systems built using these technologies often react to changes in requirements only after long latencies, even if agile development methodologies are used. The consequence of such latencies can be severe to stakeholders whose requirements develop and evolve rapidly – their ability to contribute to the CI community and derive value from it can be compromised to the point where other communities that rely on them are compromised, too.

A major source of this problem in CIs is the number and diversity of stakeholder groups, the diversity of their requirements, and the limited resources developers have for eliciting requirements and enacting them quickly.

Our vision for a solution to this dilemma is to enable a more proactive stakeholder posture in the requirements elicitation and enactment process, with multiple benefits:

- precise requirement formulation
- low latency between requirement formulation and enactment
- high fidelity of enactment relative to real stakeholder requirements
- improved understanding of the requirement space by both stakeholder and developer

To accomplish this, we propose the Alternate Workflow Specification (AWS) technique, which exposes stakeholders to CI workflows modeled as UML Activity Diagrams. Under AWS, stakeholders can propose modifications to such workflows by formulating alternate workflows using a specially constructed Domain Specific Language. By conceiving the workflow in SOA terms, AWS proposes workflow insertion semantics based on SOA message interception techniques. Informally speaking, we call alternate workflows defined in this way *policies*.

Furthermore, we observe that for CIs that implement workflows using SOA technologies (including interception), there is a direct and natural correspondence between a stakeholder-written policy and the workflow a developer would author into an executing system. We call CIs that can execute authored policies *policy-reactive*.

Policies can be used to realize a wide range of functional and non-function requirements, including security, performance, fault tolerance, and so on.

In this paper, we limit the use of AWS to enacting access control on CI resources, where access control is defined as restrictions placed on the use of a resource based on some criteria, such as user identity. We demonstrate the use and potential efficacy of the technique by presenting it to a group of stakeholders in the PALMS CI, a health monitoring system operating at the University of California, San Diego.

We show that:

- A blending of MDE and SOA techniques is effective in eliciting access control requirements from domain experts
- A DSL can be used to define policies on existing workflows
- Refinements of DSL approach may be useful in improving domain expert performance in access control requirement elicitation and policy authorship
- Focusing on access control requirement elicitation yields other requirements

We know of no other approach that attempts to elicit and enact requirements in this way. However, various aspects of AWS were inspired by model driven security work at the domain design and analysis level [1][2], requirements syntax languages [4], and directed elicitation techniques [5].

The remainder of the paper describes the specifics of the AWS technique. Section II presents the theory underpinning AWS. Section III presents the PALMS-CI case study in which AWS is evaluated. Section IV describes an evaluation of AWS by PALMS domain experts. Section VI presents an analysis and interpretation of the results. The remaining sections present discussion, related work, future work, and conclusions.

II. THEORY OF SOLUTION

Under Model Driven Engineering, there are a number of well-known techniques for eliciting functional requirements from stakeholder communities, then modeling the structure of entities (e.g., UML class and object diagrams) and data flow (e.g., UML activity, sequence, and state diagrams) , and component interactions (e.g., UML sequence and collaboration diagrams) needed to support them. Within a collection of models, different views often represent different concerns, and modelers can combine views to explore the interplay of multiple concerns at various levels of abstraction.

AWS demonstrates how UML Activity Diagrams can be leveraged to superimpose the concern of access control onto a workflow model, thereby enabling the elicitation of access control requirements from domain experts. It draws on

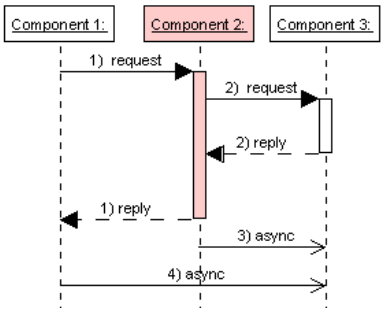

Figure 1. Service Interface Contract

formalisms basic to SOA, the semantics and graphic simplicity of Activity Diagrams, and the definition of applicable Domain Specific Languages (DSL). Using a DSL, AWS enables a user to specify access control policies on a workflow represented by an Activity Diagram, thereby enabling a developer to directly implement access control in an application.

A. Workflow and SOA

A *workflow* has been informally defined as the "computerised facilitation or automation of a business process" [6] where a business process involves the orchestration of one or more data flows between one or more activities, and one activity may depend on the execution of a previous activity. In its most basic form, a workflow begins with some data entering a system, where some activity transforms it and routes the result to one or more other activities. The workflow terminates after the last transformation is complete. Variability amongst workflows occurs as a result of operating on different data streams, executing different activities, executing activities in parallel or serially, and joining concurrently flowing data streams [7].

A workflow can be modeled in terms of services, where a *service* is logically defined as an interaction between two components via message exchange [8]. Using the service paradigm, complex component interactions can be modeled,

and formalisms exist for service composition, overlapping execution, and the analysis of properties such as liveness [9]. Additionally, models exist that demonstrate how a service can be hierarchically composed of other services, thereby enabling modeling of complex systems of systems [10]. Additionally, they describe how messages exchanged between components can be intercepted and then transformed, relayed, reconveyed, rerouted, or otherwise acted upon, thereby implementing dynamic routing and various crosscutting concerns such as security, reliability, and failure management.

While service definitions are often component-centric, a dual exists in a message-centric perspective: a service can be modeled as the transformation of one message into another by an intervening component. We define a *service interface contract* as the guarantees that can be asserted on incoming and outgoing messages. Components interacting with the intervening component can rely on its meeting its service interface guarantees. For example, Figure 1 is a sequence diagram showing a service interaction between three components. The service interface contract for Component 2 would be the content and semantics of the incoming messages `1) request` and `2) reply` and the outgoing messages `2) request`, `1) reply`, and `3) async`. Additionally, the contract would include any message timing or other interaction constraints specified.

In recasting a workflow activity as a service component, we bring service-oriented modeling and analysis to workflows. From a message-centric viewpoint, a workflow activity can be considered a message transformer (or filter) relative to an incoming and outgoing message or set of messages. Particularly, a component can be replaced with another component so long as it fulfills the service interface contract (explicitly or implicitly) defined for it. Additionally, incoming and outgoing messages can be intercepted and transformed. Finally, a component can be replaced by a collection of interacting components so long as the collection observes the original service interface contract.

B. Workflow and Activity Diagrams

Workflows can be described using a number of notations, each of which emphasize different workflow aspects or facilitate different analyses. In the MDE community, UML v2.1 Activity Diagrams are preferred as a high level graphical construct for relating activities, data flows, and synchronization – similar to a flowchart. It is commonly used to model workflow aspects of functional requirements, and in its detailed form, can represent exceptions, multiple kinds of conditionals on execution, the flow of typed data, and the state of data elements. It coordinates with other UML diagrams (e.g., class, object, sequence, and state), which present more detailed views of entity relationships and sequencing [11]

Specifically, at the core of an Activity Diagram (e.g., Figure 2) is the activity box, which represents a transformation of incoming data, a change in the system state, or both. Activity boxes are connected by directed flow edges, where the edge represents the flow of a typed data element from one activity into another. When an activity box accepts more than one flow edge, it is defined to suspend its processing until upstream activities have generated data elements for each of the flow edges. When an activity box emits more than one flow edge, it is defined to generate different data elements heading to different activity boxes. Activity Diagrams have special symbols representing forked and joined directed flows. A solid bar that accepts a single directed flow and emits multiple directed flows is defined as a fork; it copies the inbound data onto the outbound flows, and the connected activity boxes execute in parallel. A solid bar that accepts multiple directed flows and outputs a single flow is a join; the output flow is the data presented by the incoming flows, and it is emitted when all incoming flows present data. Additionally, a diamond symbol can indicate a split control flow, which directs a data flow to one activity or another. It can also indicate a merged data flow, which accepts data flows from multiple activities and immediately forwards whatever data it receives to a downstream activity box.

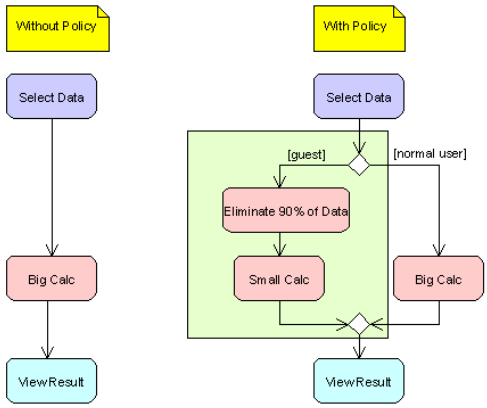

Figure 2. Activity Diagram with Policy Insertion

An Activity Diagram's directed data flow has semantics similar to a message flow in a service. However, unlike a service, activity boxes perform a transformation without being associated with a particular component. To conform activity boxes to service components, an Activity Diagram can be organized into partitions that can be labeled with a node name, which can be interpreted as a component in a service interaction.

AWS relies on Activity Diagrams because they present a familiar workflow metaphor accessible to non-developers such as domain experts representing stakeholder communities. At the same time, when partitioned, they can express many of the same interactions as a service can. Particularly, an activity box in a partition can be considered to present and adhere to a service interface contract based on the directed flows it accepts and generates. Therefore, it makes sense to discuss intercepting and transforming directed flows and replacing activity boxes with other activity boxes or networks of activity boxes, provided the result conforms to the original service interface contract, which may be relied upon by the connected activity boxes.

C. Access Control and Policy

Access control is a concern separate from, yet intimately connected, to workflow. Whereas a workflow relates activities and data flows, a system that exclusively executes workflows may not meet stakeholder requirements for limiting access to data or activities by inappropriate parties. Limited access can be implemented in many ways, including prohibiting access to data or an activity, or allowing access to only a subset of data or an activity's capabilities.

(Note that access control is a subset of security, which encompasses topics such as encryption, storage reliability, non-repudiation, and others.)

The criterion for access control involves three elements: the data flow or activity over which to exert control, the circumstance in which control should be asserted, and the particular control to be imposed.

The circumstance calculation involves a conditional often referencing a user's identity, attributes of the user, the state of the application, the state of the system (e.g., the current time), or any combination of these.

A common access control is to simply reject access and return an error message or exception. Alternatively, an incoming or outgoing data flow may be filtered, decimated, truncated, or de-resolved (e.g., a de-identification or a bounded randomization). In service terms, a generalization of these two approaches is to replace the target activity with a different activity. To reject access, the replacement activity would simply propagate an error message or exception. Alternatively, to affect a data flow, the replacement activity might consist of an ingress filter service connected to the original activity, which might be connected to an egress filter service – the filter services would alter the data flow according to the access control requirement.

Accordingly, we define a *subflow* as a portion of a workflow characterized by a service interface contract. We define a *policy* as a conditional replacement of a subflow with another subflow that meets the original subflow's service interface contract. While a policy can be used to implement many concerns in an SOA, we restrict it to access control in this paper. Figure 2 shows an example of a workflow before and after a policy (shaded) is inserted. The Small Calc subflow observes the same service interface contract as the Big Calc subflow, and is executed conditionally based on the user's role.

In a service-oriented analysis, access control can be implemented as a service separate from workflow activities. Loose coupling between these services allows an access control mechanism to be changed without impacting the service interface contract maintained between activities. For the purpose of requirements elicitation, it also promotes focus on access control instead of the implementation of the activity itself.

Note that within an executing application, policy is evaluated and executed by a *policy engine*, which is inserted as an interceptor along the data path controlled by the policy. The policy engine evaluates the conditional, and then replaces the target subflow as appropriate.

D. Domain Specific Languages and the Stakeholder

In order for a domain expert (representing a stakeholder community) to specify an access control policy, she must associate an existing data flow (evidenced in an Activity Diagram) with a condition and a replacement subflow specification. Whereas it is possible to specify all three components in "concise, plain English", such specifications often are fraught with ambiguity and misinterpretation [12], which can lead to protracted negotiations with developers and systems with unintended behaviors.

The criterion for a specification language are that it must enable an efficient and effective communication path between a domain expert and the developers, must not burden a domain expert with unfamiliar convention, and must be simple enough to use quickly and repeatedly in an agile development environment. Since the specification of conditions and actions is relative to a working system, it must relate concepts native to the system. That said, all AWS DSLs share a common structure, as follows.

The language for specifying the existing data flow is graphical – on a copy of an Activity Diagram, the domain expert puts a hatch mark on the associated directed flow edge.

The language for specifying the conditional is a standard boolean expression containing predicates that evaluate application entities, message content, application states, system states, and application-related functions. An example of such a DSL is presented in the case study in Section III.

The language for specifying the replacement subflow identifies a filter activity, which may be implemented as a composition of other filters.

The general form of a policy for ingress is:

```
If conditional, first action1, then action2,
                otherwise policy
```

where `conditional` is defined above, `action1` is an activity representing a filter having a name and filter-specific parameters, and `action2` is either `continue` (to execute the existing activity) or the name of a substitute activity. `policy` is defined as another ingress policy, and can also be simply `continue`. For example, the following policy can transform the "Without Policy" flow in Figure 2 to the "With Policy" flow.

```
If user is guest, first Eliminate 90,
          then Small Calc,
          otherwise continue
```

Similarly, the general form of a policy for egress is:

```
If conditional, finally action3
```

where `conditional` is defined above, and `action3` is the name of a filter activity and parameters.

Note that the exact form or legal values of a conditional, filter, or activity are not specified for two reasons. First, such policy statements will be evaluated and understood by human developers, not an automated language processor. Therefore, within limits, loose syntax can be tolerated.

Second, though it is important to apprise the domain expert of the existence and meaning of important domain entities and activities and their relationship so they can be used in policies, it is equally important to allow the domain expert to fabricate domain entities and activities as the need arises. Such fabrications are likely to contain the seeds of requirements for system modifications, which may come to light as developers negotiate with domain experts over the true meaning of these fabrications.

Finally, by keeping the DSL simple and ad-hoc, the domain expert can focus on defining policies instead of formatting them – leaving more time and energy for policy development.

III. PALMS CASE STUDY

The PALMS-CI is a cyberinfrastructure built at the University of California, San Diego to support the research of a worldwide community of exposure biologists. This community is represented by a number of principal investigators (PIs) that monitor and study human health as a function of geographical location and ambient conditions. Each PI may conduct one or more studies, which typically involves collecting data from sensors (e.g., heart rate, accelerometer, and GPS) worn by scores or hundreds of human subjects for periods of a week or more. Once the data is collected, either the PI or a research assistant (RA) uploads it into a PALMS repository, where it remains available for analysis and visualization. Additionally, the PI may agree to share raw or processed data with other investigators.

The PALMS-CI is a SOA based on a Rich Services [13] pattern executing on a Mule Enterprise Service Bus. PALMS-CI services implement the functionality of major domain entities (e.g., calculations, visualization, and repositories for studies, subject information, data, calculations, and calculation results) and services are connected via a message bus. The PALMS-CI presents itself as a single component that exposes its services via Web Services-based API calls using a request/reply pattern.

A PALMS user interacts with the PALMS-CI by using a web browser running JavaScript generated by the Google Web Toolkit (GWT). As such, the browser acts as an intermediary between the user and the PALMS-CI, and it presents the workflow experienced by the user.

To date, the PALMS-CI supports an end-to-end workflow consisting of creating a study, identifying a list of human subjects, uploading subject data, defining calculations on the data, and viewing data and calculated results.

Currently, there are no access control policies that would constrain the use or misuse of study data, either accidentally or by malicious outsiders. For this reason, only a small, close-knit group of researchers can make use of the system, and then only for processing anonymized information subject to Institutional Review Board (IRB) restrictions.

In order to service more PIs and their studies, PALMS-CI must support access control policies that meet the requirements of various stakeholder groups (e.g., PIs, funding agencies, IRBs, etc). Even within a stakeholder group different individuals may prefer certain policies over others, and the preference may vary from study to study.

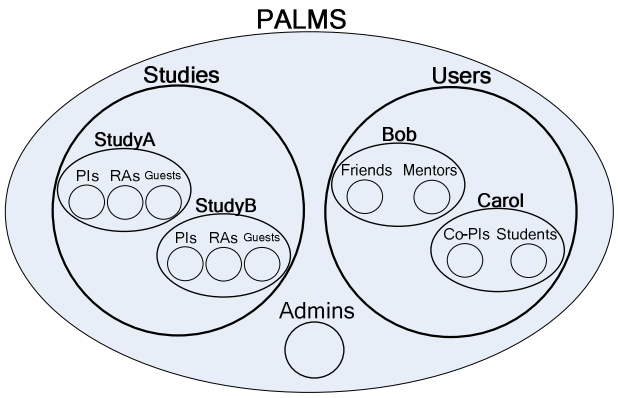

Figure 3. PALMS Group Ontology

The PALMS-CI development staff is capable of eliciting access control policy requirements from the known stakeholders and then enacting them within the CI. However, as the stakeholder population grows and becomes more diverse, the burden on the developers to service these requirements is borne at the expense of budget, long delivery times, and access control implementations that still fail to keep up with changing requirements.

A. User Identity and Grouping

The major PALMS-CI access control policy needs focus on restricting access to particular data or processes based on user identity. To achieve flexibility in targeting policies, we allow users to be organized into groups, consistent with an RBAC discipline[1] modified to accommodate a Facebook-style friends system. As shown in Figure 3, for a study-centric group structure, a PI can define groups of users that reflect the PI's organizational structure. For a given study, common groupings could be PIs, RAs, and Guests. For a friend-centric group structure, a PI can define groups of users that crosscut studies. Policies can be targeted toward study groups, friend groups, or individual users. User identity is established via logon credentials (i.e., userID and password) presented by the browser, verified by a caBIG Identity Provider[2], and evidenced by an x.509[3] certificate.

B. Activity Diagram and Workflow

To a PALMS user (and a domain expert), a PALMS workflow can be viewed as a linkage of activities accessible and experienced in the Browser. However, attempting to specify access control policies on flows and activities

[1] An RBAC system grants users access based on one or more groups the user belongs to (e.g., a person can play soccer if she is a member of the soccer team group)

[2] caBIG is a cyberinfrastructure for cancer research, and PALMS-CI uses its identity verification services instead duplicating them

[3] An x.509 certificate is a data structure produced by caBIG that services can use to verify an identity

described at this level result in ambiguous policy placement and policy definitions that act upon incomplete information – especially because activities exposed at the Browser level are actually implemented by calls to the PALMS-CI API. Accordingly, we elicit access control policy by first exposing subflows, which appear as request/reply interactions with PALMS-CI activities as shown in Figure 4.

C. *PALMS Domain Specific Language*

The PALMS DSL extends the base language defined in Section II.D, references PALMS-specific domain entities, and defines operations appropriate for them.

Access control policies can be specified on flows between the Browser and the PALMS-CI. Flows entering PALMS-CI activities are requests, and can be subject to ingress policies. Flows returning to Browser activities are replies, and can be subject to egress policies.

By convention, all PALMS-CI request messages contain the identity of the user represented by the Browser. Therefore, policy conditions can include conditions on the user, such as the user being a member of one or more groups. Additionally, when a request message identifies a study (e.g., the uploadMsg), membership in groups relative to that study can be tested.

An example of a system level attribute is the SysTime value, which is the current time of day. An example of a conditional that demonstrates the use of time in conjunction with a complex group membership test in an AcceptData interaction is:

```
((User in PIs or RAs)
    or (User in PALMS.Users.Carol.Students))
and (SysTime between 6AM and 10PM).
```

… where PIs and RAs are groups relative to the current study, and the Students group is fully qualified relative to the PALMS group ontology (see Figure 3).

An example of a filter appropriate for a reply policy is:

```
Keep entries where user in study
```

Note that a reply policy is considered to have access to values in the ingress message. Consequently, a reply conditional or filter can test a user identity.

An example of an activity useful as a substitute for an existing activity is Reply. A request policy that denies access to guests is:

```
if (user in Guests), then Reply "no access",
           otherwise continue
```

IV. Experiment and Design

The objectives of the experiment were to determine whether a PALMS domain expert using AWS could:

- specify the target, action, and location of an access control policy
- identify new access control-related criteria and actions
- identify new PALMS-CI requirements

Four PALMS domain experts (subjects 1-4) were chosen to receive instruction in AWS and then formulate access

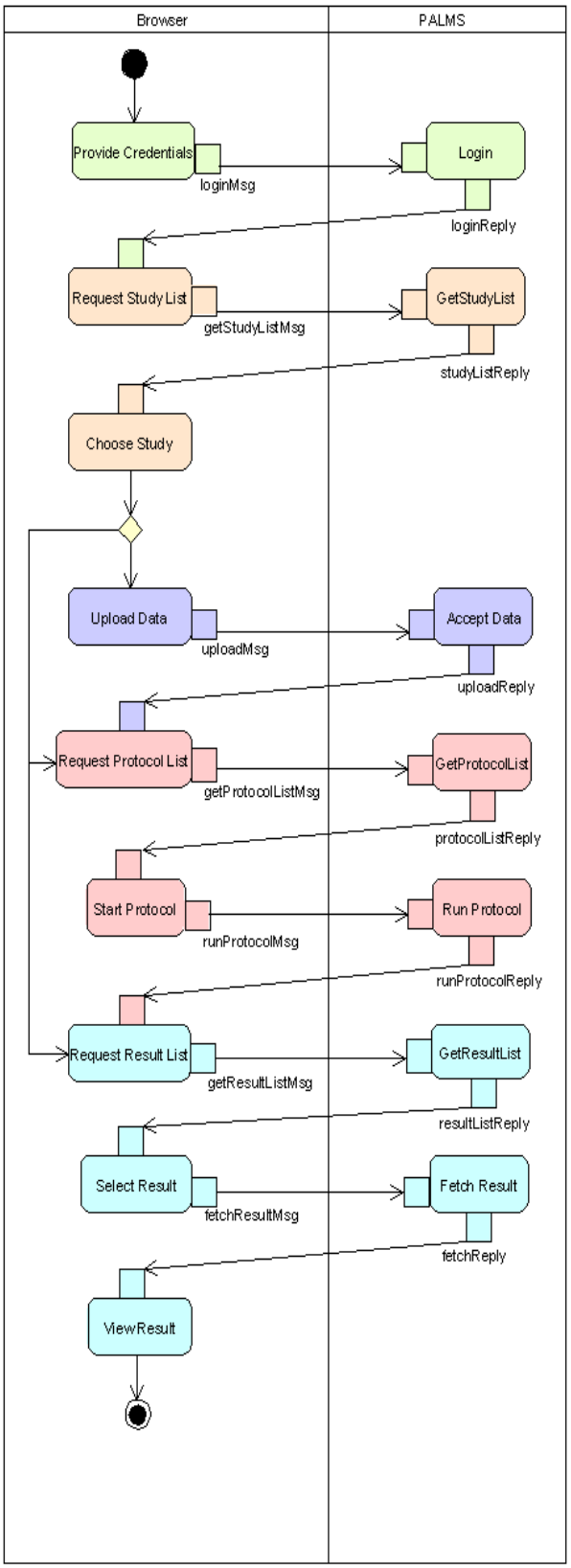

Figure 4. PALMS Workflow

control policies. All of the subjects were exposure biology investigators, and one (subject 4) also had substantial computer programming skills. Each subject has already requested that access control features be added to the PALMS-CI, and has participated in general discussions regarding the objectives of PALMS access control.

The AWS instruction consisted of a document containing:

- a description of the experiment objectives
- a brief orientation on a typical PALMS workflow expressed as an Activity Diagram
- a brief orientation on the PALMS user group scheme
- a tutorial on the elements of an access control policy (including specifying a conditional, a replacement workflow, and placement within the PALMS workflow)
- a description of the general form of a policy statement, both for a request-oriented policy and a reply-oriented policy

Each subject was given three training policy sets. The first set consisted of a one line textual description of a simple request-oriented policy, the workflow placement for the policy, and the actual policy expression. The second set was like the first set, except the policy was reply-oriented. In the third set, the policy placement and a slightly exotic expression were given, and the subject was asked to supply the textual description.

Each subject was given three exercises involving formulating and placing policies. The subject was instructed to try to use the AWS policy syntax for specifying a condition and a replacement workflow, and to try to use conditionals and filters they saw being used in the training set and other examples. The subject was also advised not to adhere slavishly to the DSL syntax, as deviations may give us clues to new PALMS requirements.

In the first exercise, the subject was given a text description for a simple request-oriented policy, and was asked to specify a matching policy placement and expression.

In the second and third exercises, the subject was asked to formulate policies they would like to see in PALMS, and then specify their placement and expression.

In all cases, the subject was asked to write his solutions down instead of describing them verbally. There was no time limit on the exercises. The experiment was administered in public places with modest ambient foot traffic.

A. *Threats to Validity*

The hypothesis of this experiment is an assertion that a PALMS domain expert could use AWS to achieve the three objectives listed in Section III.B. Naively speaking, the hypothesis is proved by a single instance of success. However, the experiment is designed to explore the degree to which the objectives are met, and the impediments to substantial and meaningful domain expert performance.

Given this, we do not assign tasks whose results can be evaluated as strict success or failure. Instead, our data is our own subjective observations of performance and the observations of the subjects themselves. Our conclusions are based on the implications of the observations regarding the efficacy of AWS in eliciting clear and actionable requirements.

While we intend that the experiment design would lead to sincere effort and thoughtful feedback from domain experts interested in effecting access control within PALMS, a number of factors can work against this:

- the description and tutorial materials could be poorly written or confusing
- the testing environment could have distractions
- the domain expert could be tired
- the domain expert could collude with others
- our assessment criteria could drift over time
- a small and uniform subject pool

While each of these factors are possibilities, we have acted to mitigate them by furnishing the experiment materials to the domain experts in advance of the experiment, by soliciting questions while going over the descriptions and tutorials immediately before running the experiment, by monitoring the test environment and the demeanor of the domain expert, and by interviewing four domain experts with different talent sets and backgrounds.

V. EXECUTION AND RESULTS

Each of the subjects required about 45 minutes to go through the discussion and tutorials, and about 15 minutes to work the exercises. Half of the subjects executed the experiment without assistance from an interviewer.

A. *Subject 1*

Subject 1 made a number of responses that challenged the AWS policy syntax and semantics. He consistently wrote request-oriented policy expressions containing a conditional and an action, but did not specify an alternative action. He explained that he viewed the conditional as a statement of an access exception that should lead to returning an error. In attempting to place policy in the workflow, he consistently placed request-oriented policy on reply flows. In conditionals, the subject checked user membership in a fully qualified group name (instead of using study-relative group names), and mis-specified that name. Additionally, he wrote declarations instead of filter expressions for actions in reply policies (i.e., instead of something like `filter out type not "GPS"`, he wrote return only GPS data). Finally, when authoring fresh policies, he preferred to operate on workflows not presented in the experiment. We drew these workflows so he could then articulate his policy.

Subject 1 successfully articulated non-access control requirements that occurred to him during the experiment. He suggested simplifying the PALMS workflow by taking advantage of information already available to PALMS, and creating a best-practices guide to optimize data collection and organization to take advantage of PALMS features. Regarding AWS, he expressed reservations that without a good way to verify the effect of a policy in advance and in the context of other policies, he would have difficulty trusting that the policy mechanism would achieve his policy

objectives as he intended. Finally, he wondered what policies might govern, who could specify policies, and under what conditions.

B. Subject 2

Subject 2 successfully formulated three request-oriented policies. Each policy was properly placed on an appropriate data flow. Two of the policies were incomplete because they did not specify alternative workflows.

One policy's conditional directly referenced a group for a particular study, which would have been an appropriate comparison only if the actual study involved in the workflow was the one named in the conditional. A more appropriate reference would have been to the group corresponding to the study identified in the message, which would have been correct for all studies.

Subject 2 did not offer any unsolicited requirements.

C. Subject 3

Subject 3 was interested in policies that addressed workflows not present in the experiment. Once we drew appropriate workflows, Subject 3 was able to place policies on the appropriate data flows. All policies were request-oriented and focused on denying access. One conditional checked for user membership in a group, but other conditionals checked for the study having a "sharable" attribute and the user having an "IRB certificate" attribute, neither of which are available in the current PALMS system, thereby signaling new requirements. In two of the three policies, the subject authored declarations instead of workflow substitutions – and the declarations did not map easily to a workflow substitution.

Once the policy elicitation was complete, the subject continued to write requirements and questions that had come to him while writing policies. Requirements were phrased as short predicates with general scope, and were not easily actionable (e.g., the addition of a provenance and curation capability, and then its use in making access control decisions based on inherited rights).

The subject's high level policy and requirement focus were evidence of engaging the PALMS access control policy topic, but in a way that would sometimes not lead to implementable policy without further negotiation.

D. Subject 4

Subject 4 consistently formed request-oriented policies correctly, and exceeded the exercise requirements by formulating a policy consisting of four separate policies each placed at different locations, and coordinating to form a larger policy.

He formed a reply-oriented policy that specified the return of an empty data set. While conceptually reasonable, the formulation failed to couch the action as a filter on the reply message – instead, it was formulated as an assertion.

Subject 4 used the policy elicitation exercise to raise a useful policy-mediated UI design question. He observed that in PALMS, many workflows are derived from a pattern of requesting a list of objects from the CI, allowing the user to choose from the list, and then requesting that the CI operate on the choice. He questioned whether a user should be presented with a list of all available objects, then allowing her to select one, only to be presented with an error message when attempting to operate on it if access to it is blocked. He suggested that a more user-friendly choice would be to filter unreachable entries. Right or wrong, the issue came up only as a result of having engaged in the policy elicitation in the first place.

Subject 4 attempted to formulate a policy based on data that was not kept by the PALMS-CI (i.e., allowing the view or deletion of calculation results by anyone other than the user that created them), thereby implying a requirement to change the application to support the policy.

VI. Analysis and Interpretation

The experiment demonstrated that the subjects understood and were favorably disposed to the general concept of access control applied to a workflow. It also demonstrated the potential of AWS to elicit novel requirements that could be realized by application modifications, pointing to what those modifications should be. Finally, it also demonstrated AWS's potential for eliciting requirements tangential or unrelated to access control, including requirements on the policy definition process itself. Clearly, AWS engaged stakeholders and focused them on discovering access control-oriented and other useful requirements.

While each of the subjects appeared to understand the general concept of applying access control policy placement in a workflow, there were various degrees of failure in formulating policy statements, in conceptualizing data flows, and in conceiving alternate workflows. The specific difficulties were:

- placing a request policy on a request data flow, and a reply policy on a reply data flow (1 subject)
- understanding when a workflow operates in the context of a study, thus enabling conditionals to reference study-local groups (2 subject)
- specifying and exercising reply-oriented filters that modify a reply message (4 subjects)
- specifying a request policy as a conditional workflow instead of as a simple criterion (2 subject)

In most cases, when a developer engaged a subject to negotiate the true meaning of an access control policy, a shared understanding was reached quickly because:

- the placement of the policy within the workflow was usually correct, and gave a strong hint as to the author's intent
- the policies themselves were simple, and running afoul of the intended DSL syntax did not obscure the author's intent – no policy proposed any complex workflow substitution

Because the policies were simple, there was no doubt that they could be easily and quickly implemented in the PALMS-CI, so this stage of the experiment was skipped.

Based on the policies articulated by the subjects, the classes of access control of major interest a) prohibit access to an activity based on user group membership or user identity, and b) cull a list of records returned by an activity based on a simple relationship between the user and a particular record component. In formulating policy, each of the subjects avoided the formalism of workflow substitution, and preferred to articulate statements that were criteria-based modifications to default flows (i.e., `deny if <some condition>, allow if <some condition>, cull based on <some condition>`).

Subjects may have been hesitant to engage attributes (other than user) found in request messages, reply messages, at the system level, or at the application level either because there was no need, or they didn't understand what attributes could be accessed. Since the subjects were not made aware of these attributes, subjects could have only guessed at their availability, and only in response to a requirement they had already thought of.

This issue could be solved by using a graphical user interface to present the Activity Diagram, and then to inform subjects of attribute choices in messages travelling along various data paths (as a result, perhaps of mousing over the data path) and available at the system and application level. Such a UI would encourage exploration of possibilities that could lead to the formulation of requirements otherwise not immediately apparent either by leveraging existing attributes or proposing capabilities that could lead to new attributes.

Likewise, a policy wizard would be useful in helping the subject create well-formed and properly targeted request and reply policies. Such assistance would alleviate the cognitive conflicts in attempting to discover and formulate policy at the same time as writing down a well-formed policy – the subject could focus solely on policy discovery and formulation instead of on form. Considering that policy articulation is likely to be a relatively rare activity for a domain expert, eliminating the load of constantly relearning writing mechanics seems particularly appropriate.

Finally, considering that the majority of policies identified by subjects fall into two basic forms, it seems appropriate to define a higher level DSL that maps to the general AWS DSL. The higher level DSL would provide facilities directly pertinent to user identification (for deny policies) and list culling (for reply policies), and forego general workflow replacement. Such a DSL could be implemented in either text-based or wizard-based forms.

An important aspect of the elicitation of access control policy is in motivating the domain expert to participate in such an exercise. While simplifying and focusing the DSL contributes to this, the larger issue of faith in the policy mechanism looms. As subject 1 indicated, a domain expert must have confidence that elicited policy requirements, if enacted, will have the desired effect on the system. This is particularly an issue when several stakeholder groups contribute policies that may interact with each other. Means must be devised to engender trust by simulating policy execution and demonstrating its effects – though this is outside of the scope of this paper, it is within reach of the Rich Service framework.

VII. Discussion and Related Work

Policy suitability depends on a clear understanding of the applicable service interface contracts, and the dimensions of these contracts are so far underdefined. Non-functional requirements (e.g., performance) amount to unstated interface contracts, and to the extent that a policy violates them, the policy will degrade the system. Additional work is needed in contract specification.

While AWS is capable of inserting user-supplied workflows, we have not explored the possibility of inserting workflows based on transformations (other than filtering) of existing workflows. We have yet to study the effect of multiparty protocols or delegation on the policy mechanisms.

In AWS, the contents of a request-oriented message is available to a conditional applied to a reply-oriented policy. This amounts to a policy session lasting for the duration of the target activity. It is unclear whether this is sufficient to implement separation of duties. This topic needs further attention.

While policy was discussed in terms of access control at a server level in a client/server system, no means is provided for the client (Browser) to ascertain the likely result of an operation before attempting it. If the client could ascertain this, it would make decisions at the user interface level in anticipation of a server interaction. It would be helpful for the policy mechanism to include a method whereby the client could determine whether a request would likely succeed or fail without actually executing it.

AWS does not attempt to perform any kind of policy validation or consistency checking. However, should the AWS DSL evolve towards a more rigorous form (e.g., through the use of GUI wizards), existing work [3] on Activity Diagram semantics may allow direct translation of stakeholder policies into executing CIs. Furthermore, conflict and completion checking may also be enabled.

Systems such as UMLSec [1] and SecureUML [2] have leveraged class diagrams annotated with OCL to improve application security and make guarantees about such systems. These approaches apply to system structure and object instances, which is a much harder concept for domain experts to grasp. These techniques apply at design time, and are accessible to knowledgeable security workers. AWS is a requirement elicitation tool operating on workflows. The objective of security specification is proof of completeness. The objective of AWS is configurability. These techniques are complimentary. We should explore the possibility of these systems tipping of AWS users of policy opportunity, and visa versa.

[4] discusses policy elicitation in declarative terms, including ubiquitous requirements. This causes a general rule to be applied according to general preconditions and triggers. AWS defines workflows at a particular flow. For large collections of workflows, specification of policy via AWS could be tedious and error prone. Adopting a declarative scheme may avoid this, and it need not replace the one-off AWS policy scheme.

[14] discusses a study probing the superiority of Activity Diagrams over EPC (statelike) diagrams for customers/users

and for requirements engineers. The study was inconclusive, but weighed in favor of Activity Diagrams for customers/users, which would describe the intended audience for AWS. It was interesting that requirements engineers preferred EPC diagrams, and it would be useful to consider them for polices insertable by technology-oriented stakeholders.

[15] describes building a secure system via a walkthrough of resources and reviewing their lifetimes. It focuses on allow/deny decisions, does not address filtering return results, and does not provide for access control configurable based on multiple criteria. As with UMLSec and SecureUML, this method is complimentary to AWS.

We are aware that AWS has a strong flavor of Aspect Oriented Programming, and we may exploit AOP features in future work. For now, our judgment is that the simple-minded conditional insertion paradigm of AWS is still under study and is sufficient for our target audience.

VIII. Future Work

While the textual form of the AWS DSL is successful in eliciting access control requirements, it does not yield policies that can be inserted directly into executing workflows. By using a GUI DSL instead, policies could be formatted in a controlled way that could result in automatic code generation leading to automatic policy enactment and a very good correspondence between stakeholder formulation and actual execution. This could result in high volumes of policies, which could conflict with each other and could incur composition errors when multiple policies target the same flow. Attempts to use AWS without addressing conflict checking, composition, completeness checking, would likely lead to unstable systems and a loss of user confidence.

IX. Conclusion

We have demonstrated that by using a blend of MDE and SOA techniques and analysis, it is possible to elicit meaningful access control requirements from domain experts not trained in access control techniques. We showed that although a DSL could serve as a specification language, the choice of DSL (perhaps a GUI DSL) would make a difference in the productivity and comfort level of stakeholders using AWS. Regardless, we found that using AWS to elicit policy produced novel, previously unarticulated requirements at both the access control level and a general application level. We believe that future work will improve AWS, and it may be applicable in policy domains beyond access control. There is still much work to do in the realm of policy-reactive CIs and in coordinating with existing and complimentary approaches. AWS is a good starting point.

Acknowledgment

We acknowledge Kevin Patrick, the principal investigator of the PALMS project, for his generous support in developing the PALMS-CI. We also acknowledge the four experiment subjects for their time and valuable insights regarding requirements and the policy proposition.

Request for approval of our experiment protocol has been made to UC San Diego's Institutional Review Board and is pending. We expect approval by mid-April 2010.

References

[1] J. Juerjens. *Security Systems Development with UML*. Springer-Verlag Berlin Heidelberg, 2003.

[2] T. Lodderstedt, D. Basin, and J. Doser. *SecureUML: A UML-Based Modeling Language for Model-Driven Security*. Proceedings of the 5th International Conference on The Unified Modeling Language. pp426-441, Springer Verlag, 2002.

[3] A. Bhattacharjee and R. Shyamasundar. *Activity Diagrams: A Formal Framework to Model Business Processes and Code Generation.* Journal of Object Technology. Vol 8, No 1, Jan 2009.

[4] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak. *EARS (Easy Approach to Requirements Syntax)*. 17th IEEE International Requirements Engineering Conference. Atlanta, GA, Sept 2009.

[5] G. Sindre. *Mal-Activity Diagrams for Capturing Attacks on Business Processes.* Lecture Notes in Compter Science. Springer Berlin/Heidelberg. Vol 4542/2007.

[6] The Workflow Reference Model. The Workflow Management Coalition, Jan. 1995.

[7] http://www.workflowpatterns.com/

[8] M. Broy, I. Krueger, and M. Meissinger. *A formal model of services.* ACM Transactions on Software Engineering and Methodology (TOSEM). Vol 16, Issue 1, 2007.

[9] http://www.springerlink.com/content/g0velc4fv7gp9qmw/

[10] https://sosa.ucsd.edu/ResearchCentral/download.jsp?id=149

[11] Process Aware Information Systems

[12] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5328644&isnumber=5328460&tag=1

[13] https://sosa.ucsd.edu/ResearchCentral/view.jsp?id=149

[14] G. Gross and J. Doerr. *EPC vs. UML Activity Diagram – Two Experiments Examining their Usefulness for Requirements Engineering.* 17th IEEE International Requirements Engineering Conference. Atlanta, GA, 2009.

[15] J. Viega. *Building security requirements with CLASP.* Proceedings of the 2005 workshop on Software engineering for secure systems – building trustworthy applications. St Louis, MO, 2005.