Trust-Serv: Model-Driven Lifecycle Management of Trust Negotiation Policies for Web Services

Halvard Skogsrud
University of New South Wales
Sydney NSW 2052, Australia
halvards@cse.unsw.edu.au

Boualem Benatallah
University of New South Wales
Sydney NSW 2052, Australia
boualem@cse.unsw.edu.au

Fabio Casati
Hewlett-Packard Laboratories
Palo Alto, CA, 94304 USA
fabio.casati@hp.com

ABSTRACT

A scalable approach to trust negotiation is required in Web service environments that have large and dynamic requester populations. We introduce Trust-Serv, a model-driven trust negotiation framework for Web services. The framework employs a model for trust negotiation that is based on state machines, extended with security abstractions. Our policy model supports lifecycle management, an important trait in the dynamic environments that characterize Web services. In particular, we provide a set of change operations to modify policies, and migration strategies that permit ongoing negotiations to be migrated to new policies without being disrupted. Experimental results show the performance benefit of these strategies. The proposed approach has been implemented as a container-centric mechanism that is transparent to the Web services and to the developers of Web services, simplifying Web service development and management as well as enabling scalable deployments.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; H.3.5 [**Information Storage and Retrieval**]: Online Information Services

General Terms

Security, Management

Keywords

Conceptual modeling, Lifecycle management, Trust negotiation, Web services

1. INTRODUCTION

Traditional access control models rely on knowing requester identities in advance [8]. Web services typically have large and dynamic requester populations. This means that requesters' identities are seldom known in advance. Most existing Web applications deal with strangers by requiring them to first register an identity at the Web site. Such approaches do not fit into the Web service philosophy of dynamically choosing services at run-time.

Trust negotiation is an access control model that addresses this issue by avoiding the use of requester identities in access control policies [17]. Instead, access is granted based on trust established in a negotiation between the service requester and the provider. In this negotiation — called a trust negotiation[1] — the requester and the provider exchange *credentials*. Credentials are signed assertions describing attributes of the owner. Examples of credentials include membership documents, credit cards, and passports. The attributes of these credentials are then used to determine access. For instance, a requester may be given access to resources of a company by disclosing a credential proving she is an employee of that company. This example shows that the requester identity is not always needed to determine access. Credentials are typically implemented as certificates [11].

Although trust negotiation systems exist [3, 5, 12, 17], several issues still need to be addressed:

- *Trust negotiation policy specification.* Trust negotiation policies specify which credentials — and other resources — to disclose at a given state of the trust negotiation, and the conditions to disclose them. Specifying these trust negotiation policies using most existing policy languages is a complex task that generally requires time-consuming and error-prone low-level programming [9].

- *Trust negotiation policy lifecycle management.* Lifecycle management of policies — that is, the creation, evolution, and management of policies — is an often overlooked part of policy model design. Policies are rarely set in stone when first defined. Instead, they are modified and refined to reflect changing business strategies [14]. Lifecycle management of policies is especially valuable in the dynamic environments that characterize Web services. Enterprise security policies change because of mergers and acquisitions, internal reorganization, emerging competitors, new products, updated processes, changes to laws and regulations, etc.

Issues that must be considered in lifecycle management frameworks include how to update trust negotiation policies in a consistent manner and how to cope with *dynamic policy evolution*, that is, the change to a policy while there are active negotiations based on the policy being modified. The latter issue is particularly challenging, due to the need of minimizing the disruption to current requesters while making sure that the new policy is applied.

In this paper we propose a model-driven approach to trust negotiation in Web services. The framework, called Trust-Serv, features a trust negotiation policy model based on state machines. More importantly, this model is supported by both abstractions and tools that permit lifecycle management of trust negotiation policies. These are the salient features of the work:

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

[1] In the remainder, we will use the terms *trust negotiation* and *negotiation* interchangeably.

- To specify trust negotiation policies, we provide modeling abstractions that are used to extend the familiar state machine model. These abstractions provide the expressiveness required for a trust negotiation policy model, such as representations for the level of trust established, credential disclosures, provisions, and obligations.
- Trust-Serv supports lifecycle management of trust negotiation policies and instances. We introduce a set of change operations that are used to modify policies. Strategies are presented to allow not only evolution of policies, but also migration of ongoing negotiations to a new policy. To automatically determine the appropriate strategy for each negotiation, we use meta-policies. These meta-policies are specified independently of the negotiation policies. Additionally, since negotiation migration may cause negotiated access rights of requesters to be revoked, we present a scheme that is used to compensate such requesters.
- Trust negotiation and access control are managed and automated by software components called *negotiation controllers*. Negotiation controllers intercept messages directed to the Web service they control. They may accept or reject operation invocation requests, or they may initiate an interaction with the negotiation controller of the other party to negotiate trust before accepting the invocation. All this is transparent to the Web services. At the service level, the interaction only involves the business logic of the services, and it appears to take place directly between the Web services.

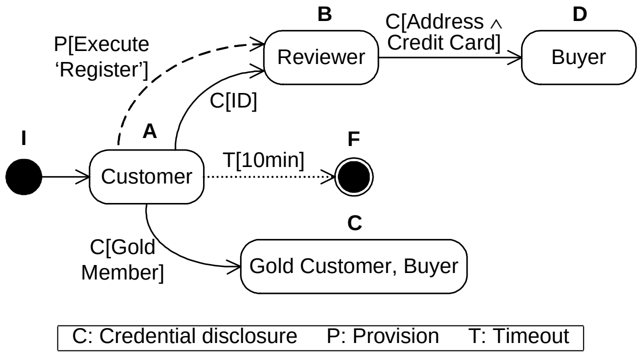
The remainder of the paper is structured as follows. We start by describing the trust negotiation policy model in Section 2. In Section 3, we present a proposed lifecycle management model. Section 4 describes the architectural support for policy evolution and negotiation migration, as well as the implementation of Trust-Serv and the results of experiments. In Section 5, we describe how our model-driven approach to trust negotiation is beneficial to developers of composite Web services. We discuss related work in Section 6 and conclude with a summary and directions for future work in Section 7.

2. MODELING TRUST NEGOTIATION POLICIES USING STATE MACHINES

Trust-Serv expresses trust negotiation policies as state machines, because of their formal semantics, and because they are well suited to describing the reactive behavior that characterizes trust negotiations [15]. Figure 1 shows an example of a trust negotiation policy for a bookshop service. We will refer to this policy in our examples.

2.1 States

States in the model represents the level of trust achieved by the requester so far in the negotiation. By entering a new state, Trust-Serv gives the requester access to more resources. Trust-Serv identifies two types of resources: operations of the Web service that is protected and credentials owned by the provider. Instead of assigning these resources directly to states, we use the abstraction of roles. Roles are semantic abstractions that describe some function performed by people or processes (e.g., *author* and *editor*). In role-based access control, permissions are assigned to roles rather than individual requesters [8]. In the Trust-Serv policy model, roles are mapped to states, which means that the roles of a state may be activated (i.e., acquired) by the requester once it reaches that state. Roles are cumulative, so previously activated roles are not deactivated when entering a new state.



Role	Operation	Credential
Customer	Register, Search	Verified by Visa
Reviewer	Write review	Elite Bookshops Member
Gold Customer	Special offers	
Buyer	Purchase	

Figure 1: A trust negotiation policy *P.I* for a bookshop service.

Example 2.1 (Roles). *Consider the policy in Figure 1. If a requester enters state A, that requester is allowed to activate the* `Customer` *role. Upon activating this role, the requester is given access to the service operations* `Register` *and* `Search`*, as well as the provider credential* `Verified by Visa`*.*

2.2 Transitions

In Trust-Serv, transitions are labeled with conditions that restrict when they may be fired. Briefly, the semantics are as follows: When the negotiation is in a state S, and an event occurs which satisfies the condition of a transition T where S is the input state, then the negotiation moves to the output state of T. Requesters explicitly trigger events by invoking operations such as credential disclosure.

Transitions are extended beyond traditional state machines to capture security abstractions necessary for trust negotiation. We have identified three types of transition conditions: *credential disclosures*, *provisions* or *obligations*, and *timeouts*.

- *Credential disclosure* conditions require the requester to disclose one or more credentials. Additionally, they may constrain the permitted values of attributes of the credentials. The transition labels that start with `C` are credential disclosure conditions. These conditions may be specified in a number of existing languages [3, 5, 9, 12]. Trust-Serv uses TPL (Trust Policy Language) [10], because it is expressive enough to describe credential disclosure conditions.
- *Provisions and obligations.* We represent provisions as service operations that must be invoked before the negotiation can proceed, while obligations are promises by the requester to invoke an operation some time in the future [4]. The transition between state A and B in Figure 1 marked with a dashed line, is a provision that requires the requester to invoke the `Register` operation to satisfy the condition. The intention is that requesters that do not possess an `ID` credential may instead register at the service and provide their identity through the `Register` operation.
- *Timeouts* are used to specify timed transitions. If no action is taken by the requester within a given time, a transition to another state may be forced by using timed transitions. This type of transition is used to abort abandoned negotiations by forcing them to a final state. The transition between state

A and F in Figure 1 marked with a dotted line, is a timed transition that requires the requester to take action within ten minutes of entering state A.

2.3 Trust Negotiation Controllers

Trust negotiation policies are interpreted by *negotiation controllers*. Each service is associated with a negotiation controller that interprets the service's trust negotiation policy. When a requester invokes an operation of the service, the invocation is intercepted by the controller. The controller may then forward the invocation to the service for processing, reject it, or it may initiate a negotiation with the requester to allow it to establish sufficient trust to allow the invocation.

Requesters may also deploy trust negotiation policies to protect their credentials, instead of interacting directly with the negotiation controller of the service. The task of the negotiation controller would then be to try to achieve sufficient trust to allow the requester to access the desired resource, within the limits of the requester's trust negotiation policy. Because the requester policy constructs are a subset of those used for a provider policy (requesters may not have service operations), we focus on provider policies.

Figure 2 illustrates how controllers are deployed to perform trust negotiation on behalf of both requesters and providers. The advantage of this architecture is that all the trust negotiation occurs at the controller level. At the service level, the interaction appears to take place directly between the requester and the service provider. This eliminates the need to encode trust negotiation logic in the Web service itself, which simplifies development and deployment.

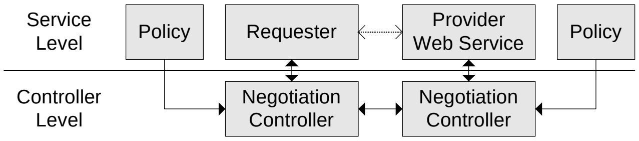


Figure 2: Interaction between service requesters and providers. Trust negotiation is managed and automated by the negotiation controllers and takes place at the controller level.

3. LIFECYCLE MANAGEMENT IN TRUST NEGOTIATION

In dynamic Web service environments, policies often need to be modified to accommodate changing business strategies. Changes to laws and regulations also force enterprises to update their policies. Security holes may be discovered, which need to be rectified. In general, lifecycle management has been recognized as an important problem and has been studied in several different domains (see e.g., [6, 13]). Our focus is to address the lifecycle management problem in the context of trust negotiation policies and of Web services, both conceptually and in terms of supporting tools and architectures.

Lifecycle management in trust negotiation policies is an important issue since, if it is not properly addressed, it could lead to policy breaches or to lower service quality, such as slower response time. Assume that a new policy *P* is defined for a service. All new negotiations will start according to *P*. However, simply aborting and restarting all current negotiations is not appropriate for several reasons. A considerable amount of work may be lost, and the number of ongoing negotiations may be so high that aborting and restarting all would cause severe disruption to other dependent services. For instance, if a popular Web site (e.g., Amazon.com) suddenly aborted all customer transactions, all current customers would have to repeat the steps of the purchasing process. This could cause customer frustration and ultimately lead to loss of income for the service as customers seek other services. The problem is even more critical in the case of long-running services, such as purchase order approval or employee relocation management, as both the likelihood of having policy changes during each service execution, as well as the amount of work lost by aborting the service increase in a very significant way. Instead, it should be possible for the service to modify its policy without disrupting ongoing negotiations.

Ad-hoc approaches to policy evolution and negotiation migration encounter scalability problems when used in Web service environments. Due to the potentially large requester populations, the number of concurrent negotiations could be large. It is thus infeasible to manually manage policy evolution. To address the problem, in the following we propose a framework that enables automated policy lifecycle management, built on top of the trust negotiation model presented earlier.

3.1 Basic Definitions

This section introduces definitions that will be used throughout the paper to describe our approach to lifecycle management in trust negotiation. We describe a trust negotiation policy P by the tuple

$\Sigma^P = \langle States^P, Transitions^P, Roles^P, \theta^P, \rho^P \rangle$

where $States^P$ is the set of states of P, $Transitions^P$ is the set of transitions of P, $Roles^P$ is the set of roles protected by P, θ^P is the *transition assignment* function, associating each transition to a source state and a target state, and ρ^P is the *role assignment* function, associating each role to a set of states. The domains and co-domains of θ and ρ are as follows ($\wp$ denotes powerset):

$\theta^P : Transitions^P \rightarrow States^P \times States^P$

$\rho^P : Roles^P \rightarrow \wp(States^P)$

Notice that a role may be mapped to several states, and several roles may be mapped to the same state. States C and D in the policy in Figure 1 show examples of these properties. Upon reaching state C, requesters may activate both the `Gold Customer` and the `Buyer` role. Requesters may acquire the `Buyer` role either by reaching state C or state D.

To be considered *legal*, i.e., syntactically correct, a negotiation policy has to meet certain criteria. We introduce the following definitions to formally define legality of a negotiation policy P. In the remainder, we will omit the policy identifier P where no ambiguity arises from the context.

- *Initial state:* The policy has a single initial state ι, where $\iota \in States$.
- *Successor function:* The successor function σ maps a state to the set of states that succeeds it. It is defined as:
  $\sigma(s) = \{t | t \in States \wedge (\exists u : u \in Transitions \wedge \theta(u) = \langle s, t \rangle)\}$
- *Successor transitive closure:* The transitive closure of the successor function, denoted $\sigma*$, maps a state to the set of its successors recursively. It is defined as:
  $\sigma*(s) = \{p | p \in \sigma(s) \vee (\exists t : t \in \sigma(s) \wedge p \in \sigma*(t))\}$
- *Reachability:* A state s is *reachable* iff it is in the successor transitive closure of the initial state, i.e., $s \in \sigma*(\iota)$.
- *Incoming and outgoing transitions:* The outgoing transitions function α maps a state s to the set of transitions for which s is the source. Similarly, the incoming transitions function ω maps a state s to the set of transitions for which s is the target. They are defined as:

$\alpha(s) = \{tr | tr \in Transitions \wedge (\exists t : t \in States \wedge \theta(tr) = \langle s, t\rangle)\}$
$\omega(s) = \{tr | tr \in Transitions \wedge (\exists t : t \in States \wedge \theta(tr) = \langle t, s\rangle)\}$

- *Source and target states:* The source state θ_s and target state θ_t of a transition tr are defined such that $\theta(tr) = \langle \theta_s(tr), \theta_t(tr)\rangle$.

With these definitions in place, we may define policy legality.

Definition 3.1 (Legality). *A trust negotiation policy P is* legal *iff every state is reachable from the initial state. Formally:*
$\forall s \in States^P : s = \iota \vee s \in \sigma * (\iota)$

3.2 Policy Change Operations

To allow policy updates, it is necessary to provide a set of change operations that can be applied to a policy. However, these operations need to be carefully constructed, to ensure they satisfy some desirable properties.

Firstly, the set of operations should ensure *structural consistency*. This means that the result of applying an operation to a legal policy should always result in a legal policy. Additionally, it means that changes may not cause any ongoing negotiation to end up in a situation such that it is not clear how to proceed, i.e., how to process incoming operation invocations. Secondly, the set of operations should be *complete*. This means that using only these operations, it should be possible to transform any legal policy *P.I* into any other legal policy *P.F*. Finally, the set of change operations should be *minimal*. The set is minimal if no proper subset of it is also complete.

An important result of the structural consistency requirement is that removing states becomes a delicate process. Firstly, if a policy developer wants to remove a state from a policy, she must first make sure that all the remaining states are still reachable. Secondly, any negotiations currently at that state would be left in an inconsistent state, since the behavior of such instances would be undefined. To handle the first problem, we specify the reachability requirement as a precondition of the operation to remove states. The second problem is handled by rolling back instances at that state to their previous states.

Having a set of operators that is complete, minimal, and structurally consistent guarantees that, through such operators, we can make any changes to any policy while avoiding the generation of illegal policies. It also avoids burdening the model with unnecessary change operations that would make the framework more complex without adding value.

Based on these properties, we define the set of change operations, hereafter called *primitives*. The definitions below detail the preconditions and effects of the primitives. We assume in the following that a policy $P.I$ is modified, resulting in an updated policy $P.F$.

- **AddTransition** (Transition tr, State s, State t): This primitive adds transition tr to the policy with source state s, and target state t.

  Precondition: $s, t \in States^{P.I} \wedge tr \notin Transitions^{P.I}$

  Effect:

  1. $Transitions^{P.F} = Transitions^{P.I} \cup \{tr\}$
  2. $\theta^{P.F} = \theta^{P.I} \cup \{tr \mapsto \langle s, t\rangle\}$

- **RemoveTransition** (Transition tr): This primitive removes transition tr from the policy. The precondition states that there must be at least one other transition to the target of tr. This maintains structural consistency by ensuring that state t is still reachable from the initial state ι.

  Precondition: $|\omega(\theta_t(tr)) - \alpha(\theta_s(tr))| \geq 1 \wedge tr \in Transitions^{P.I}$

  Effect:

  1. $\theta^{P.F} = \theta^{P.I} - \{tr \mapsto \theta^{P.I}(tr)\}$
  2. $Transitions^{P.F} = Transitions^{P.I} - \{tr\}$

- **MapRole** (Role r, State s): This primitive adds role r to the policy and maps it to the state s.

  Precondition: $s \in States^{P.I}$

  Effect:

  1. $Roles^{P.F} = Roles^{P.I} \cup \{r\}$
  2. $\rho^{P.F} = \rho^{P.I} \cup \{r \mapsto s\}$

- **UnmapRole** (Role r, State s): This primitive removes the mapping of role r to state s. Because r may be mapped to other states, it is only removed from $Roles^{P.F}$ if this is the last mapping for that role.

  Precondition: $true$

  Effect:

  1. $\rho^{P.F} = \rho^{P.I} - \{r \mapsto s\}$
  2. if $\rho^{P.F}(r) = \varnothing$
     $Roles^{P.F} = Roles^{P.I} - \{r\}$

- **AppendState** (State s, State r, Transition tr): This primitive adds the state s to the policy as a successor of the state r. The transition tr is added from state r to state s. The formal semantics are:

  Precondition: $r \in States^{P.I} \wedge s \notin States^{P.I} \wedge tr \notin Transitions^{P.I}$

  Effect:

  1. $States^{P.F} = States^{P.I} \cup \{s\}$
  2. $AddTransition(tr, r, s)$

- **RemoveState** (State s): This primitive removes the state s from the policy. Before s is removed, all roles mapped to s are unmapped, and all transitions in and out of s are removed. The precondition states that all the target states of outgoing transitions from s must be reachable even if the outgoing transitions of s are removed. The formal semantics are:

  Precondition: $\forall tr \in \alpha(s) : |\omega(\theta_t(tr)) - \alpha(s)| \geq 1$

  Effect:

  1. for each Role $r \in \rho^{P.I}(s)$:
     UnmapRole (r, s)
  2. for each Transition $t \in \alpha^{P.I}(s) \cup \omega^{P.I}(s)$:
     $\theta^{P.F} = \theta^{P.I} - \{t \mapsto \theta^{P.I}(t)\}$
     $Transitions^{P.F} = Transitions^{P.I} - \{t\}$
  3. $States^{P.F} = States^{P.I} - \{s\}$

Firstly, these primitives are complete because every trust negotiation policy may be developed by initially transforming it into a state machine with only an initial state. Secondly, the primitives are minimal because they deal with insertion or removal of distinct concepts (transitions, roles, and states). Finally, they are consistent, since after applying any primitive to a legal policy, the policy remains legal. As the proofs of these properties are lengthy, although intuitive, they are omitted.

3.3 Lifecycle Management of Negotiation Instances

Once a policy is changed, decisions must be taken on how to handle ongoing negotiation instances (i.e., negotiations) that began under a different policy. We identify *strategies* that can be used to manage the negotiation instances when policies are changed. These strategies are designed such that different strategies may be applied to each instance. We assume that an initial policy *P.I* is modified, resulting in a final policy *P.F*. The two most obvious strategies either allow the instances to complete according to the policy under which they begun, or abort the instances and restart them under the new policy. We detail these below:

- *Concurrent to completion.* The negotiation in progress according to *P.I* is allowed to complete according to *P.I*. This means the enforcement system might need to enforce more than one policy at a time (e.g., both *P.I* and *P.F*). This is done by creating one negotiation controller instance for each policy. The controller instance enforcing *P.I* is destroyed once all its negotiations have completed. This strategy is applicable when the provider can tolerate existing negotiations completing according to *P.I*. However, in many cases the provider may not allow this to happen. For instance, a change to applicable laws means that the provider must modify its operation to satisfy the new requirements. This strategy would be unacceptable in such situations.
- *Abort.* The negotiation instance is aborted, and all roles attained by the requester are deactivated. Depending on the implementation, a new negotiation instance following *P.F* may be created in place of the aborted instance. The main drawback of this strategy is that it may waste a lot of work that has already been done. The requester and provider may have already negotiated a high level of trust, but if this strategy is applied, the negotiation would have to start all over again.

The inadequacy of the two previous strategies emphasizes the need for better and more efficient solutions to this problem. This involves allowing ongoing negotiation instances to be *migrated* to the new policy. However, care needs to be taken when migrating negotiations, to avoid undesired results.

Example 3.1 (Compliance). *Consider the policy in Figure 1. Suppose that the credential disclosure transition between states A and B was modified as follows: The requester must also submit a* `Credit Card` *credential. At the same time, the condition of a* `Credit Card` *credential is removed from the transition between states B and D. The resulting policy* P.F *is shown in Figure 3 (the provision transition and the timed transition are removed for clarity of presentation). A requester may then have reached state B according to* P.I *without disclosing a* `Credit Card` *credential. If the policy is modified while this requester is at state B, and the negotiation is migrated to* P.F, *this requester will be able to proceed to state D and activate the* `Buyer` *role without disclosing its* `Credit Card`*. This may not be acceptable for the provider.*

Example 3.1 suggests that negotiations have to satisfy some condition in order to be effectively migrated to *P.F*. The condition is that the negotiation instance so far according to *P.I* is *compliant* to *P.F*. A negotiation instance is compliant to a negotiation policy if it is a valid instance of the policy. To be able to define compliance, we introduce the following variables relating to a negotiation N according to a policy P:

- $State_N^P$ denotes the state of P that N is currently at. The initial value of this variable is the initial state of P.

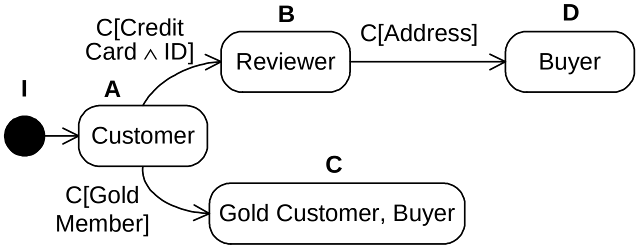


Figure 3: A modified trust negotiation policy *P.F* for a bookshop service.

- $VisitedStates_N^P$ denotes the ordered set of states of P visited so far by N. This set is initialized to contain only the initial state of P. States are appended to this set as soon as the requester enters them, thus $State_N^P \in VisitedStates_N^P$.
- $RolesActivated_N^P$ denotes the set of roles that are currently active for N.
- $TransitionsFired_N^P$ denotes the transitions of P taken by N to reach the $VisitedStates_N^P$.

Definition 3.2 (Compliance). *A trust negotiation instance N is* compliant *to a trust negotiation policy P if all of the following conditions hold:*

1. $\forall s \in VisitedStates_N : s \in States^P$
2. $\forall r \in RolesActivated_N : r \in Roles^P \wedge \exists v : v \in VisitedStates_N \wedge v \in \rho^P(r)$
3. $\forall t \in TransitionsFired_N : t \in Transitions^P \wedge \exists x, y : \{x, y\} \subseteq VisitedStates_N \wedge \theta^P(t) = \langle x, y \rangle)$

This definition states that the policy P must contain (i) all states visited by N, (ii) all active roles of N, which must be mapped to states visited by N, and (iii) all transitions fired by N, of which the source and target states must be visited by N.

Based on our definition of compliance, we present here two strategies for migrating running negotiation instances to new policies.

- *Migration to new policy.* The negotiation is migrated to *P.F*. If the negotiation instance is compliant to *P.F*, the migration is said to be *unconditional*. However, if the instance is not compliant to *P.F*, the migration is *conditional*. Conditional migrations require the negotiation instance to trace back steps in the policy until it reaches a state where it is compliant to *P.F*. If such a rollback is necessary, the appropriate roles are deactivated after the migration.
- *Migration to hybrid policy.* Instead of rolling back non-compliant negotiation instances, temporary policies may be defined for these negotiations. This ad-hoc policy will be a hybrid of *P.I* and *P.F*. Its function is to modify existing negotiation instances to comply with the requirements causing the policy change from *P.I* to *P.F*. Such a strategy is useful if the policy modification is critically important but a rollback is considered too disruptive.

Example 3.2 (Negotiation migration). *Consider the policy of the bookshop service, shown in Figure 1. Suppose the provider* *changed this policy as explained in Example 3.1, resulting in the policy* P.F *shown in Figure 3. Suppose further that a current negotiation had reached state B and activated the* `Reviewer` *role when the policy change occurred.*

If the provider uses the abort strategy, the negotiation is canceled and must be restarted. This is simple, but it might be considered a suboptimal solution because of the work lost.

The concurrent to completion strategy lets the negotiation continue according to the old policy. The provider must decide whether to allow this. If this is unacceptable, the provider must choose another strategy.

Using the migration to new policy strategy, the system first determines whether the negotiation is compliant to the new policy. The transition between states A and B has been fired by this negotiation. Since this transition was changed in the new policy, the negotiation is not compliant to the new policy. Trust-Serv then rolls back the negotiation to state A, and deactivates the role `Reviewer` *before the negotiation resumes following the new policy.*

3.4 Strategy Selection

Because the number of concurrent negotiation instances could be large, it is infeasible for the provider to manually examine each instance to determine which strategy to apply. Instead, we use meta-policies; sets of rules that describe the management of policies. The result of the evaluation of a negotiation instance with this meta-policy will determine the appropriate strategy for this instance. We call these meta-policies *strategy selection policies*.

A strategy selection policy consists of a sequence of rules. Each rule has two parts; a condition on variables of negotiation instances, and a migration strategy with an associated policy. The condition is a set of logic statements that state restrictions on the permissible values of negotiation instance variables. The last rule of every policy has a *true* condition. This rule is called the default rule, since it is used if no other rules match. The strategy part of each rule contains one of the evolution strategies presented above. It also contains a reference to the policy to which the instance is migrated if the condition evaluates to *true*.

Negotiation instances are evaluated against each of the rules in turn, until a match is found. Once a negotiation instance satisfies the condition of a rule, matching ceases and the instance is migrated to the specified policy using the specified strategy. Notice that the rules define a partition of the set of ongoing negotiations, meaning that for each ongoing negotiation there is exactly one strategy.

Example 3.3 (Strategy selection policy). *Figure 4 shows an example of a strategy selection policy. The policy* P.I *(Figure 1) is modified to achieve policy* P.F *(Figure 3). Rule 1 states that instances that have only visited state I and A are aborted. Rule 2 states that negotiation instances that have not visited state D are migrated to the new policy* P.F. *Finally, rule 3 is the default rule that specifies that all other instances are allowed to complete according to their current policy* P.I.

3.5 Honoring Obligations to Requesters

Issues regarding implicit agreements with requesters may arise when policies are changed while a negotiation is underway. When the provider informs the requester that it can acquire its desired role R at state S by providing a credential C, this is an implicit obligation from the provider to the requester. Essentially, the provider has just promised the requester that it only has to submit a single credential C to access role R.

Now suppose that the provider wishes to change the policy by removing the mapping of role R to state S (i.e., applying the primitive $UnmapRole(R, S)$). If this negotiation instance is migrated to the new policy after the requester has submitted credential C and activated role R, R will be deactivated. Now the requester has disclosed C to no avail, and it has not been able to acquire R, as promised by the provider.

```
<RULE ID="1">
  <CONDITION>
    <SUBSETEQ>
      <VARIABLE>VisitedStates</VARIABLE>
      <SET>I, A</SET>
    </SUBSETEQ>
  </CONDITION>
  <STRATEGY NAME="Abort" POLICY="null"/>
</RULE>
<RULE ID="2">
  <CONDITION>
    <NOTIN>
      <CONST>D</CONST>
      <VARIABLE>VisitedStates</VARIABLE>
    </NOTIN>
  </CONDITION>
  <STRATEGY NAME="Migrate to new policy"
            POLICY="P.F"/>
</RULE>
<RULE ID="3" TYPE="default">
  <STRATEGY NAME="Concurrent to completion"
            POLICY="P.I"/>
</RULE>
```

Figure 4: A strategy selection policy.

It is of vital importance that any lifecycle management model provides ways to handle these issues. Using the Trust-Serv trust negotiation policy model, the policy change that might cause such a situation is the removal of role-to-state mappings. To avoid situations where promises to the requester are broken, it is necessary to not deactivate roles activated by requesters, even if these role mappings are moved or removed. By letting requesters keep their roles, the promised resources are still available to requesters, and the implicit agreement is not broken.

However, there might be situations where the policy upgrade is considered vital by the service provider. For instance, it might be discovered that the previous trust negotiation policy was too weak and allowed some requesters to obtain resources that they should not be able to obtain. Such violations could even be in breach of laws in cases where it would allow access to privileged information. It is thus not possible to always allow requesters to keep their roles in the case of policy updates and instance migration.

To address this issue, we introduce a set of options that may be taken whenever negotiation instances are migrated to a new policy where role mapping have been removed. These options are executed by the provider's negotiation controllers. They permit role deactivations to be delayed, or provides compensation to the requester for obligations that are not upheld.

- *Delay the role deactivation.* The provider instructs the negotiation controller to delay all role deactivations by some specified time. A notice is sent out to all affected instances, informing them that some of their privileges will soon be revoked because of a forced role deactivation. This gives requesters a "grace period", during which they can adapt to the change by exercising the privileges they have achieved so far in the negotiation.

- *Compensate the requester.* The provider instructs the controller to deactivate the affected roles immediately, and notifications are sent to the affected requesters. To appease these requesters, offers of compensations are issued by the provider. This compensation can take many forms, including financial compensation. In Trust-Serv, we associate a *compensation role* with each role in the trust negotiation policy. This means that if a role is deactivated due to migration to another policy, the provider may offer membership in a compensation role to the requester. This role may then give the requester access to various forms of compensation offered by the service.

Example 3.4 (Compensation). *Figure 5 shows a fragment of the definition of a role named* `Gold Customer`*. It specifies that if a requester is a member of this role, and this membership is deactivated by a negotiation migration, then that requester instead becomes a member of the compensation role* `Discount`*.*

- *Let the requester decide.* This final option lets the requester decide between the two previous options (i.e., *delay the role deactivation* or *compensate the requester*). The affected negotiations are suspended while the negotiation controller issues notifications to the requesters asking for their preferred way of dealing with the issue. Once the requester replies with its choice, the instance is resumed and the controller takes the action indicated by the requester's reply.

```
<ROLE NAME="Gold Customer" ...>
  <COMPENSATION ROLE="Discount"/>
  ...
</ROLE>
```

Figure 5: An example of the specification of a compensation role.

Note that if the policy update does not involve removal of role-to-state mappings (i.e., the *UnmapRole* primitive), this issue does not arise. Also, even if role mappings are removed, if the chosen strategy for an instance is *concurrent to completion*, the update does not affect that instance, and no further action is necessary.

4. ARCHITECTURE AND IMPLEMENTATION

In order to support the trust negotiation model described in this paper, we propose an architecture for Trust-Serv that is specifically targeted at Web service environments.

4.1 Trust-Serv Architecture

The goal of the architecture is to substantially increase the level of automation in Web service development and deployment with respect to what is available today. We achieve this by factorizing into the middleware those chores common to the development of many Web services. The Trust-Serv architecture introduces the notion of Web service *containers* to manage the internal behavior of the underlying service and its interactions with service requesters and partners. Containers provide functionality necessary for Web services to support trust negotiation, as well as other functionalities, such as conversation management and exception handling. The run-time operation of the service container is directed by policies, such as trust negotiation policies, that may be defined for individual or groups of Web services. The advantage of this architecture is that developers who want to create a new service simply need to implement the business logic of the service and specify the trust negotiation policy. Tasks such as controlling negotiation instances and verifying credentials are delegated to the container, thereby considerably simplifying development. An overview of the extended architecture is shown in Figure 6.

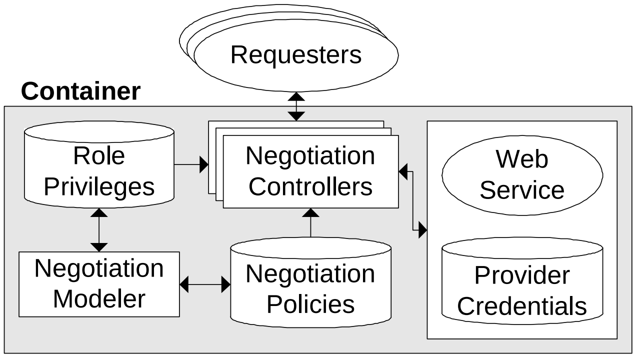


Figure 6: The service container architecture.

The negotiation controllers are implemented as Web services that provide the capabilities to participate in trust negotiations. At run-time, the negotiation controllers are responsible for receiving negotiation messages such as credential disclosures and service requests, determining if new negotiation instances should be created, and triggering transitions if their conditions are met. Messages are sent between negotiation instances and service instances as SOAP requests and responses [7]. Information needed by controllers to control trust negotiations is provided by translating the state machine representation of trust negotiation policies into rules. Due to space reasons, we do not discuss this here. However, details on this translation can be found in [15].

The negotiation controllers are able to intercept invocations to the protected service by implementing all the operations defined in the interface of the service (i.e., in the WSDL document [7]). However, for the operations of this interface, the negotiation controller simply acts as a mediator. That is, after the invocation is permitted, the implementations of these operations in the negotiation controllers only consist of a call to the corresponding operation in the protected service. Thus, requesters only interact with the protected service indirectly through the negotiation controllers.

To support lifecycle management, Trust-Serv offers a negotiation modeler, which is a CASE-like tool for Web service trust negotiation policies. It assists developers in specifying and modifying negotiation policies. Additionally, it allows policy developers to define strategy selection policies as XML documents. A negotiation policy is edited through a visual interface. This interface offers an editor for describing a state machine diagram of a negotiation policy. It also provides means to describe the conditions of transitions. The modeling functions available to the policy developer are the members of the set of primitives defined in the policy model. Additionally, the interface allows more complex functions to be built using the primitives. Once constructed, these functions are available in the editor. Figure 7 shows a screenshot of the negotiation modeler interface.

4.2 Implementation and Evaluation

The implementation of Trust-Serv is an extension to the Self-Serv platform [1]. Self-Serv is a middleware supporting Web service development based on standards such as SOAP, WSDL (Web Service Description Language), and UDDI (Universal Description,

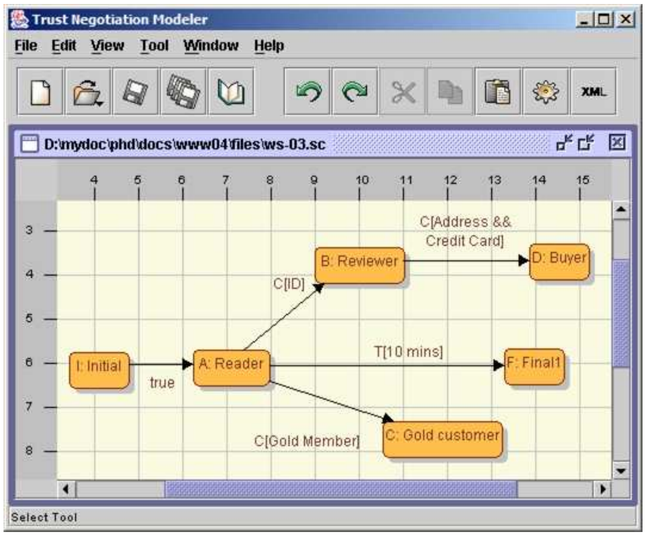


Figure 7: The Trust Negotiation Modeler interface.

Discovery, and Integration) [7]. To test the scalability of Trust-Serv, we focused on the performance advantages of the lifecycle management framework. The migration strategies migrates negotiation instances to an updated policy. Without such strategies, trust negotiation frameworks would be forced to abort all negotiation instances and restart then following the new policy. To measure the performance advantage, the experiments were implemented in Java and run using Sun's JDK 1.4 on an AMD Athlon 1 GHz with 256 MB memory.

Scaling well for a high number of negotiations is important for Web services because they frequently have many requesters. Similarly, scaling well for complex policies with many states is an important property, since Web service policies are often very complex. This is because complicated laws and regulations often apply to services offered, and the trust negotiation policy has to enforce these laws.

To measure the benefits of migration strategies, we generated a number of ongoing negotiations at various states of a policy P.I. We then create another policy P.F by changing P.I using the primitives. If no migration strategies are used, we assume that all credentials submitted so far by the requester are kept in a cache by the provider. This information can then be used by the controller of the new policy to advance the restarted negotiation through the new policy, without any further credential disclosures by the requester. We measured the time it took for the controller to evaluate this information and move the negotiation as close as possible to the target state using only the information already submitted by the requester before the policy change.

To compare, we measured the time it took to evaluate the negotiations against a strategy selection policy, migrating the instances accordingly, and advancing them as close as possible to their target state using information already submitted, in a manner similar to that used without migration strategies. Any performance advantage would come from the fact that the policy evaluation would not have to start from the initial state of the policy, but rather from the state of the new policy to which the negotiation was migrated.

Of course, these measurements would be highly dependent on what changes are made, and on the strategy selection policy. If the strategy selection policy was set to abort nearly all the negotiation instances, the performance with negotiation strategies would likely be very similar to the performance without negotiation strategies. Conversely, if most of the negotiation instances were allowed to complete according to the old policy, the migration strategies would seem to clearly give better performance.

To avoid favoring either of the two options (with or without migration strategies), we focused on the *migration to new policy* strategy. This strategy must determine if a negotiation instance complies with the new policy. Compliant negotiations would be easily migrated to the new policy. Non-compliant negotiations, on the other hand, would have to be rolled back, then migrated to the new policy, and finally advanced through the new policy as far as possible using credentials already disclosed by the requester. We designed the policy update to cause half of the active negotiations to be migrated unconditionally, while the other half was non-compliant and would have to be rolled back.

The graphs in Figure 8 show the performance with and without migration strategies. The graph on the left shows the time taken when negotiations are aborted and restarted, while the graph on the right shows the time taken when negotiations are migrated. Each approach was tested with policies with different number of states and different number of negotiation instances. Each of the tests was run 11 times, with the result of the first run thrown away, to ensure that issues such as initialization would not affect the final result. The times taken by the remaining runs were averaged to achieve the times seen in the graphs.

The times on the vertical axis show the performance benefit of using migration strategies. As we can see from the graphs, increasing the number of states causes only a linear increase in the time taken to migrate the negotiation instances. Similarly, increasing the number of negotiation instances causes a linear increase in the migration time. This shows that the migration strategies of Trust-Serv scales well both for a high number of negotiations, and for complex trust negotiation policies with many states.

5. JOINT ANALYSIS OF COMPOSITION AND TRUST NEGOTIATION LOGIC

A model-driven approach to trust negotiation provides benefits for developers of composite Web services. A composite Web service is an umbrella structure that aggregates multiple other elementary and composite Web services. In the composition, these services interact according to a given process model. For example, a composite Web service "Travel Planner" may aggregate multiple Web services for flight booking, travel insurance, accommodation booking, car rental, itinerary planning, etc., which are executed sequentially or concurrently. The process model underlying a composite service is specified using formalisms like state charts [1], or emerging standard composition languages such as BPEL4WS (Business Process Execution Language for Web Services) [16]. We have identified three composition scenarios in which our trust negotiation model is helpful.

5.1 Generation of composite service trust negotiation policy

In this scenario, we assume that a composition model is already defined by a service developer. The problem lies in inferring the trust negotiation policy of the composite service from the composition model and the trust negotiation policies of the component services. This is useful in cases where the developer starts by defining the composition model, and then needs to derive the trust negotiation policy that the composite service supports.

For example, assume that operation OP_C of the composite service S_C is implemented by invoking operation OP_1 of service S_1.

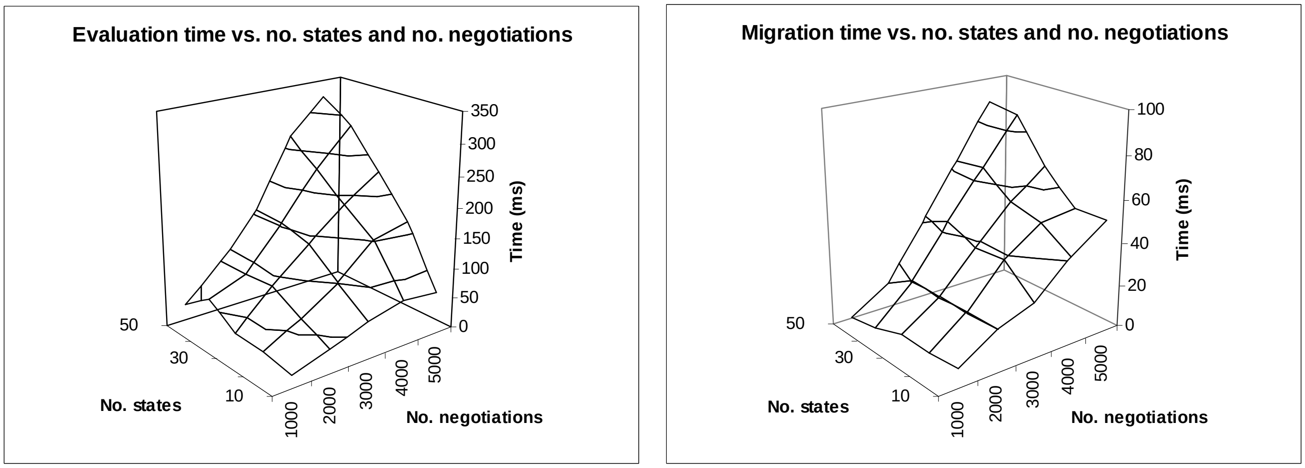


Figure 8: An evaluation of the performance advantages of migration strategies. The graph on the left shows the time taken to abort, restart, and reevaluate negotiations with respect to the new policy. The graph on the right shows the time taken to evaluate the negotiation instances against the strategy selection policy, determining which of the instances comply with the new policy, migrating the negotiation instances, and reevaluating those instances that could not be migrated unconditionally from the state of the new policy to which they were migrated.

If the condition of invoking OP_1 from the trust negotiation policy of S_1 requires credential C_C to be disclosed, this condition is added to the condition of invoking OP_C in the policy of S_C.

5.2 Generation of composition model

An even more useful approach consists of guiding the service developer in designing the composition model of the composite service, based on the trust negotiation policy that the composite service has to support. This is useful for instance in cases where some standardization body has defined the characteristics (including trust negotiation policies) that a certain service must support. The developer is then faced with the problem of designing a composition model and choosing component services that can implement the standardized trust negotiation policy. To assist the developer in this effort, it is possible to automatically generate a skeleton of a composition model starting from the trust negotiation policy that the composite service has to support. The developer may then extend the skeleton with the business logic required to implement the service functionality.

For instance, consider an operation OP_C of a composite service S_C that is implemented by invoking operation OP_1 of service S_1 and operation OP_2 of service S_2 in parallel. If the standardized trust negotiation policy of S_C states that the condition for invoking OP_C is C_C, then S_1 and S_2 must be chosen so that $C_C = C_1 \wedge C_2$, where C_1 and C_2 are the conditions for invoking OP_1 and OP_2 in the policies of S_1 and S_2, respectively.

5.3 Composite service trust negotiation policy validation

Given trust negotiation policies of a composite service and its component services, as well as a composition model of the composite service, abstractions of a trust negotiation model prove useful for checking the correctness of the trust negotiation policy of a composite service with respect to the composition model and the trust negotiation policies of the component services. Essentially, we want to avoid invocations of operations of the component services by the composite service if the requester to the composite service does not have the right to invoke those operations.

As an example, consider an operation OP_C of a composite service S_C that is implemented by invoking operation OP_1 of service S_1. If the trust negotiation policy of S_1 states that the condition for invoking OP_1 is C_1, then the condition C_C for invoking OP_C in the trust negotiation policy of S_C must be either C_1 or $C_1 \wedge C_2$, where C_2 represents additional conditions for invoking OP_C specified by the trust negotiation policy of S_C. If this is not the case, the validation fails. If, for instance, $C_C = C_1 \vee C_2$, the validation fails because this would allow OP_1 to be invoked without satisfying C_1.

6. RELATED WORK

Our work is related to efforts in providing policy languages for trust negotiation. Existing languages include early works such as PolicyMaker and KeyNote [9], as well as more recent efforts, such as IBM's TPL (Trust Policy Language) [10], the *RT* family of role-based trust management languages [12], the portfolio and service protection language presented in [5], as well as χ-Sec and χ-TNL [3]. Trust-Serv is complementary to all these efforts, as we use TPL to describe credential disclosure conditions. On the other hand, Trust-Serv provides a lifecycle management framework to support evolution of trust negotiation policies. To the best of our knowledge, none of these existing policy languages includes support for dynamic policy evolution.

Our visual model for representing trust negotiation policies as state machines is related to visualization efforts in IBM's Trust Establishment (TE) framework [10]. TE allows policies to be specified as graphs, where the nodes represent roles and the edges show which roles are accepted as issuers of credentials for membership in other roles. The framework also includes an editor that allows policies to be edited as graphs. However, since TE does not support trust *negotiation*, it does not support dynamic policy evolution.

The use of graphs to specify policies has also been studied by Yu et al. [17]. The aim of this work is to use policy graphs to prevent unnecessary policy disclosure during trust negotiation. In this model, each resource is assigned a policy in the form of a graph. A policy is attached to each node, and this policy is revealed upon the client reaching the predecessor node. The policy specifies the credential disclosures that are required before the client may reach the node and reveal more of the policy. There are three important differences between the policy graph model and our policy state machine model. Firstly, each policy graph can only be used to protect one resource, while state machine policies in Trust-Serv can protect any number of resources. Secondly, our model extends traditional state machines with security abstractions such as provisions and obligations. Finally, our model supports lifecycle management of both trust negotiation policies and negotiation instances.

Our concept of trust negotiation borrows from TrustBuilder [17]. This framework focuses on trust negotiation strategies and protocols. Negotiation strategies control which credentials to disclose, when to disclose them, and when to terminate a negotiation. The strategies are designed to work together with policies. If a policy determines that a credential may be disclosed, the strategy determines whether the disclosure is necessary, and when it should take place. Trust-Serv is complimentary to this work, as it adds support for dynamic policy evolution. In addition, our work features a container-centric architecture and implementation designed for Web services which permits scalable deployment of trust negotiation infrastructure.

Bertino et al. [2] introduced a model for specifying and enforcing authorization constraints in workflow management systems. While our policy model uses state machines, a common representation for process systems such as workflows, our model works on a different level than this workflow authorization model. The model proposed in [2] specifies authorizations on the individual tasks of the business process. These processes are executed internally, possibly with publicly exposed interfaces. Our model on the other hand, considers these public interfaces exposed as Web services. Therefore, state machines in our model do not represent business processes, but rather policies to restrict access to exposed interfaces of business processes. Another important difference is that the model in [2] does not support lifecycle management of policies. Finally, Trust-Serv provides a scalable implementation of trust negotiation for Web service environments.

Dynamic evolution of trust negotiation policies presents some unique and challenging issues. Firstly, because the number of ongoing negotiations may be large, we introduced strategy selection policies to automatically determine the appropriate strategy for each negotiation instance. Secondly, because the negotiation involves two parties (the requester and the provider), we had to address issues such as implicit obligations, and compensation if privileges were removed due to negotiation migration. In this regard, Trust-Serv provides a novel architecture for scalable deployment of trust negotiation with lifecycle management support in Web service environments.

7. CONCLUSIONS AND FUTURE WORK

We have presented Trust-Serv, a trust negotiation framework for access control in Web services. In particular, we have emphasized lifecycle management, which is an important issue that to date has not been addressed. We have shown how security abstractions can be modeled as extensions to traditional state machines. Based on a formalization of the model, we have presented policy evolution primitives, negotiation instance migration strategies, and a strategy selection policy language that allow running negotiations to be efficiently migrated to a new policy. Finally, we have shown how this framework implements service containers to enable scalable deployment. Although we have chosen Web services as our target applications, many of the results presented in this paper are applicable to other forms of services provided over networks.

A promising area for future includes expanding the model to handle composition of Web services. The approach to composite Web services will be based on the principles discussed in the joint analysis of trust negotiation and service composition. To enable composition, it is also necessary to study compatibility between requester and provider policies. That is, given the policies of a requester and a provider, is it possible to establish sufficient trust between the two parties to facilitate the desired interaction. This has been the subject of previous research, and we are investigating the application of this work to our model.

8. REFERENCES

[1] B. Benatallah, Q. Z. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1):40–48, Jan./Feb. 2003.

[2] E. Bertino, E. Ferrari, and V. Atluri. The Specification and Enforcement of Authorization Constraints in Workflow Management Systems. *ACM Trans. Information and System Security (TISSEC)*, 2(1):65–104, Feb. 2002.

[3] E. Bertino, E. Ferrari, and A. C. Squicciarini. χ-TNL: An XML-based Language for Trust Negotiations. In *Proc. 4th Int'l. Workshop Policies for Distributed Systems and Networks (POLICY'03)*, June 2003.

[4] C. Bettini et al. Provisions and Obligations in Policy Management and Security Applications. In *Proc. 28th Conf. Very Large Data Bases (VLDB'02)*, Aug. 2002.

[5] P. Bonatti and P. Samarati. A Unified Framework for Regulating Access and Information Release on the Web. *J. Computer Security*, 10(3):241–272, 2002.

[6] F. Casati et al. Workflow Evolution. *Data and Knowledge Eng.*, 24(3):211–238, Jan. 1998.

[7] F. Curbera et al. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, Mar./Apr. 2002.

[8] D. Ferraiolo et al. Proposed NIST Standard for Role-Based Access Control. *ACM Trans. Information and System Security (TISSEC)*, 4(3):224–274, Aug. 2001.

[9] T. Grandison and M. Sloman. A Survey of Trust in Internet Applications. *IEEE Comm. Surveys & Tutorials*, 3(4), 2000.

[10] A. Herzberg et al. Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers. In *Proc. IEEE Symp. Security and Privacy*, May 2000.

[11] R. Housley et al. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*. IETF RFC 2459, Jan. 1999.

[12] N. Li and J. Mitchell. RT: A Role-based Trust-management Framework. In *Proc. 3rd DARPA Information Survivability Conf. and Exposition (DISCEX'03)*, Apr. 2003.

[13] C.-T. Liu, S.-K. Chang, and P. Chrysanthis. Database schema evolution using EVER diagrams. In *Proc. ACM Workshop Advanced Visual Interfaces (AVI'94)*, June 1994.

[14] J. Rees, S. Bandyopadhyay, and E. H. Spafford. PFIRES: A Policy Framework for Information Security. *Comm. ACM*, 46(7):101–106, July 2003.

[15] H. Skogsrud, B. Benatallah, and F. Casati. Model-Driven Trust Negotiation for Web Services. *IEEE Internet Computing*, 7(6):45–52, Nov./Dec. 2003.

[16] S. Thatte, ed. *Business Process Execution Language for Web Services (BPEL4WS), Version 1.1*. www-106.ibm.com/developerworks/library/ws-bpel, May 2003.

[17] T. Yu, M. Winslett, and K. Seamons. Supporting Structured Credentials and Sensitive Policies through Interoperable Strategies for Automated Trust Negotiation. *ACM Trans. Information and System Security (TISSEC)*, 6(1), Feb. 2003.