On Context in Authorization Policy

Patrick McDaniel*
AT&T Labs – Research
pdmcdan@research.att.com

Abstract

Authorization policy infrastructures are evolving with the complex environments that they support. One key, but not yet well understood, aspect of policy is the need and support of context. Often implemented as condition functions or predefined attributes, context is used to more precisely control when and how policy is enforced. This paper considers context requirements and services in authorization policy. We classify the use, properties, and security requirements of context evaluation. A key observation gleaned from this classification is the degree to which context functions share common properties. The Antigone Condition Framework (ACF) exploits these commonalities to provide a general purpose service and associated API used to defined and implemented context. We present and illustrate the prototype ACF design, and conclude by considering directions for future work.

1 Introduction

Authorization policy describes how access to protected resources is governed. Historically, these policies have mapped identities to collections of rights over sets of objects according to some system model [23]. Policies give the supported systems a road-map to operation, and allow administrators to develop coherent strategy for protecting the environment.

Policy technologies have evolved in lock-step with the networks and environments they support. For example, novel access control models deal with the complexity of managing the large and fluid environments (e.g., RBAC [22]) or address the requirements of specific information models (e.g., lattice [1, 21]). Similarly, evolving policy infrastructures address widely distributed systems (e.g., KeyNote [4]) or arbitrate the policies of multiple endpoints [14, 12]. These and many other works have served to increase the flexibility and ease with which access to protected resources is managed.

Context is increasingly used by policy infrastructures to allow environmental factors to influence when and how policy is enforced. When specified in an authorization policy, context defines the conditions that must (or must not) hold for the policy to be applied. For example, the following context definition (from [20]),

Token Type:	printer_load
Defining Authority:	local_manager
Value: $< 20\%$	

defines context associated with a print queue. This context evaluates the status of a print queue by asking the `local_manager` if the print queue is less than 20% of its maximum capacity. Note that how this context is evaluated may be complex: an authority must be contacted (presumably using some secure means), and the result must be transformed (check whether queue size less than 20%). A policy using this context might regulate print job submission based on this context. Hence, authorization policy (and indirectly context) is used to regulate the print queue length.

Context is modeled throughout as parameterized functions called *conditions*. Traditional conditions used in authorization policy simply poll local state variables (e.g., test values recorded in the KeyNote action environment [6]). Recently, policy systems have begun to embrace more semantically rich conditions [16]. Such conditions can test the state of the environment, presence of qualified or authorized entities, or perform complex algorithms (e.g., query and interpret print queue length). However, existing implementations of these conditions are largely *ad hoc*: each new condition is typically hard coded in policy evaluating software.

This paper considers context requirements and services in authorization policy. We begin by classifying the use, properties, and security requirements of context evaluation. While context is represented and obtained by contemporary policy infrastructures in vastly different ways, these classi-

*This work is supported in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

fications show that they all are defined along common dimensions. It is the identification of these dimensions that led to the design of the Antigone Condition Framework (ACF). Built within the existing Antigone/Ismene policy framework [15, 13], ACF provides a general purpose service and associated API used to integrate context into arbitrary policy infrastructures. We present and illustrate the ACF design, and conclude by considering directions for future work.

This work extends the traditional notion of policy conditions. Throughout, conditions are not seen as singular expressions over a fixed set of attributes, but viewed as programs. One key aspect of this extended view is the need for additional security infrastructure. Because the policy is driven by external forces, it is necessary to ensure that the means by which these forces are measured is consistent with local requirements (i.e., data is faithfully obtained from trusted sources). Condition security largely been outside the scope of contemporary works in authorization policy, and is a central topic of this paper.

2 Related Work

Many policy infrastructures do not explicitly support conditions. For example, the KeyNote system is general purpose framework used to govern authorization policy. Signed policies, called credentials, define the conditions under which an authority grants access to a particular resources. KeyNote provides a flexible algebra for specifying logical condition expressions over string and numeric *attributes*. Each attribute value is known prior to the evaluation of policy. Hence, because the value is fixed before **any** policy is considered, there is no opportunity for policy to defer to external evaluation.[1] The Akenti system [24] also assumes attribute values are known prior to policy evaluation (Akenti's author indicate that run-time evaluated conditions are also on the horizon). Conditions are encoded in Akenti through *Use Certificate*s. These certificates are similar in form and function to condition expressions of KeyNote, but are restricted to the vocabulary of the domain in which they exist.

Some approaches simplify evaluation by fixing the set of conditions available to policy. Both the MSME [19] and the Security Policy System (SPS) [28] limit conditions to only those needed by each target domain. In the case of SPS (which governs IPsec communication), ports and end-points largely dictate where access is granted. Because SPS is built on the more the general IETF Policy Framework [17], SPS can be extended to include run-time conditions. MSME similarly can support externally evaluated conditions through the extension of the evaluation infrastructure. Note that because these systems assume policy conforms to a predefined schema, all conditions must be known *a priori*. Run-time conditions are supported by fixing the kinds of conditions that policy may use, rather than fixing values before evaluation (as seen in the systems above). Because support for each condition can be built directly into the enforcement infrastructure, no general condition framework is necessary.

In more general settings, anticipating all possible conditions that a policy may use is not feasible. For example, the Generic Authorization and Access-control API [20] (GAA-API) defines interfaces for a general purpose policy infrastructure. Conditions are arbitrary functions that are defined by the type, name, and governing authority. Conversely, in Ismene, very little is assumed by the policy infrastructure [14]. Unlike GAA API, Ismene policies do not identify which authority, if any, should govern the evaluation process. Both of the systems leave the vast majority of condition evaluation to the supporting implementation. Each system provides an *upcall* interface, but by an large, neither mandates how how the requirements for secure evaluation are identified or addressed.

It is interesting to note that possibly the most flexible policy language, PolicyMaker, does not explicitly support externally evaluated conditions. PolicyMaker views policy, called filters, as having "the full complexity and expressiveness of general programs" [5]. Blaze continues by stating that, "There is no need [for the PolicyMaker policy infrastructure] to open files or interact with the network." This paper seeks to extend the PolicyMaker model model not by allowing programatic policy, but by deferring complex run-time operations to externally evaluated conditions. The remainder of this paper considers the properties, requirements, implementation of such conditions.

3 Policy and Context

Policy infrastructures evaluate and enforce policy. Note that in practice the policy infrastructure may span services and applications. For example, Blaze et. al. describe an infrastructure used to implement IPsec Policy [7]. The approach integrates KeyNote [4] (policy evaluation platform) with an OpenBSD implementation if IPsec (enforcement platform). We refer to the combination of a policy decision point (PDP) and policy enforcement point (PEP) as the policy infrastructure. Note that this does not mandate that both services coexist within the same host.

A generic condition environment is depicted in Figure 1. In response to some attempted action, the policy infrastructure evaluates policy. A condition used by the policy is evaluated by the infrastructure by extracting state (possibly through a parameterized function) from a local or remote

[1] The authors of the KeyNote system have considered extending this model to include *active attributes* [10]. These attributes would call external function (code) at the time a condition expression is evaluated.

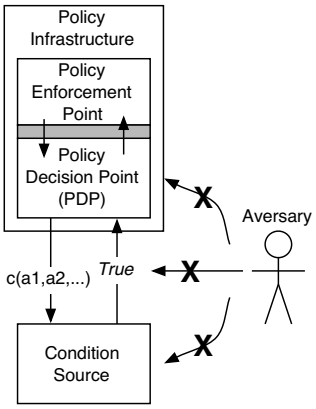

Figure 1: *Condition Evaluation* - policy infrastructure extracts information from (internal or external) condition sources via parameterized function. This information is used to determine how policy is enforced by the Policy Enforcement Point (PEP). An adversary may attempt to influence policy by manipulating the PEP, the PDP, or the condition source.

condition source. This state may be interpreted or transformed by the policy infrastructure to arrive a condition result. An *adversary* attempts to alter condition results by manipulating the environment (e.g., by altering messages passed between the source and PDP).

Conditions are simply functions that are used to measure context. Every condition is a function of zero or more arguments ($a_1, a_2, \ldots, a_n$). Each parameter is a static or variable value (i.e., identified by the policy infrastructure at run-time). Throughout, it is assumed that all conditions are Boolean (i.e., results if restricted to *true* or *false*). Non-Boolean conditions (i.e., functions with continuous or discrete output) can lead to complex policy evaluation (in both the intuitive and complexity-theoretic sense). However, it is expected that the evaluation of conditions defined over these non-binary conditions would not be qualitatively different than the procedures defined throughout.

Note that policy may require the evaluation result be further qualified. For example, where the result of evaluation is positive, the result can provide additional information (e.g., a cryptographic algorithm that must be used to encrypt session traffic). Where the result is negative, further detail information could be similarly provided (e.g., condition failed because missing credential). The PolicyMaker system supports the inclusion of additional context by supporting annotations [5].[2] Support for such features would represent simple extensions to the framework described in this paper. However, for ease of exposition, exploration of annotations is deferred to future work.

[2]These annotations are not strictly supported by the condition evaluation, but rather are added when a condition equation evaluates to *true*.

Conditions are mostly frequently used to construct policy rules. Each such rule represents a logical expression of conditions and Boolean operators. Subject to rule processing discipline (e.g., rule ordering), the policy rule is applied where the condition expression evaluates to true. For example, a traditional firewall rule:

`SRC=192.168.7.8:22,DST=192.168.7.27:*` $\rightarrow$ `accept`

follows this model. The rule states that all traffic that satisfies the condition expression (i.e., where source address/port, and destination address are equal to the identified values) should be allowed (i.e., defines an accept policy). Other policies languages generally work analogously, where the form of policy, the semantics of the expressions, and the range of supported conditions may differ. Other works have focused on the preceding issues of form and semantics, but have largely ignored supporting conditions. This paper is devoted to this last aspect of policy: *what kinds of conditions are needed, how are they implemented, and how do we make them secure.*

3.1 Condition Properties

Conditions are defined by their evaluation algorithm. Each such algorithm is classified by the kinds of information it acquires and how, where, and when it is acquired, the how the received information is used. Gleaned from the collection of conditions used by the Ismene test-bed applications [12], the following properties characterize these algorithms:

- *local/remote* - As seen in traditional policy systems, local conditions are evaluated without any external input. For example, an condition `hostname(bob)` tests whether the local hostname is "bob". Clearly, this does not need external information, and can be evaluated through locally configured values.

- *data/computation* - Data driven conditions simply test the existence or value of some known state variable. Conversely, computation-driven conditions implement (sometimes complex) algorithms for computing condition results. For example, a `valueEquals(a,b)` condition tests whether the values are equal. Conversely, the condition `sumValueGreaterThanZero(a,b)` performs some computation on the state prior and tests some aspect of the result. This distinction becomes interesting when the computation becomes complex, e.g., where multiple data sources must participate in the computation.

- *stateful* - Stateful conditions modify (rather than just acquire) state during evaluation. That is, the act of evaluation modifies some local or distributed state. This is useful where the access is *consumptive*. For example, some digital rights systems use

an access counter to restrict the number of times a particular object is used. A stateful condition `accessCounter(counter-id,threshold)` would test to see if the counter has remaining accesses, and if so increments the associated counter. Hence, this condition could be used as a means of restricting access to the resource (see Section 5).

A related property is *idempotence*. Idempotence ensures that the same evaluation returns the same value no matter how many times it is called. One can think of data driven conditions as being idempotent: the act of obtaining the value has no effect on the value itself. Note that external factors may alter the underlying value (e.g., changing network conditions). Because the act of evaluating the condition does not affect the value, it is idempotent.

- *synchronous/asynchronous* - synchronous conditions are evaluated at the point at which the policy infrastructure requests it. Conversely, asynchronous conditions cannot be evaluated immediately. These properties are very similar to non-blocking socket behavior: the policy infrastructure is free to proceed with other tasks while the evaluation completes. However, this requires that the application using the condition have ability to be non-blocking.

  Like many aspects of policy, the semantics of a asynchronous evaluation are subject to interpretation. One could be optimistic and allow access on the assumption that the evaluation would be positive (and revoke access later if it proves not to be so), or pessimistic and delay or prevent access until the condition evaluation process is completed. The selection a particular meta-policy for asynchronous evaluation is left to the policy infrastructure.

  An example asynchronous condition is `isDNSAuthority(dom,srv)`. This condition tests whether the server *srv* is authoritative for the DNS domain *dom* as culled from the `whois` service. In most applications, because it would be highly undesirable to block all operations until the `whois` lookup is completed, it is advantageous to allow the condition to complete asynchronously.

Clearly, these categories are not mutually exclusive nor exhaustive. For example, a condition may use the result of stateful acquisition of remote values that are used as input to computation. Section 4 considers the degree to which the condition definition (as classified along these dimensions) dictates the ways the condition, and ultimately the policy, can be evaluated.

The semantics of a condition are often subtle. For example, consider a condition `timebetween(9am,5pm)`. How a particular policy infrastructure views time is largely defined by its security requirements. Where the policy only needs a local (and potentially insecure) timing source, the condition could be evaluated locally. If, however, the timing needed to be synchronized across multiple enforcement points (and secure), it would have to be acquired by an external timing source. Note that because the desired semantics may differ from environment to environment, support for both conditions may be required (e.g., both `localtime` and `networktime` would made available).

3.2 Condition Security

One critical aspect of a condition definition not encompassed in the preceding discussion are the condition security requirements. Security is really a matter of environmental interpretation: each environment will place a unique set of requirements on the source and methods of condition evaluation. This is best illustrated by example. Consider the `timebetween()` condition as used in two application environments, a multiplayer game and an online trading application.

For this example, it is assumed that users in the multiplayer game are restricted to particular times defined by policy. Hence, in addition to testing the appropriate credentials, the authorization policy would test to see that user is allowed to participate at the current time. Conversely, `timebetween` is used in the trading application to ensure that every transaction occurs during normal trading hours (e.g., 9am-5pm).

Now consider the security requirements of each of these environments. In the online game, it is likely that the only requirement is for authenticity (i.e., to prevent the player from forging timing information and gaining access during restricted times). The same condition will have much stronger security requirements in the trading application. To protect all participants, the trading application will require some after-the-fact evidence that the transaction occurred at a normal time (non-repudability). Other factors may contribute to the security requirements. For example, existence of a transaction (and not necessarily its contents) may be valuable information. Hence, the evaluator may require anonymity and/or confidentiality.

Note that many sources of condition information (the timing source in the previous example) will have their own *authorization policy*. Hence, where complex conditions are employed, it is important to consider how these policies are defined, and at the end-points, used. Such organization can lead to recursive policy evaluation. We see the evaluation of the interaction between condition evaluation infrastructure and the supported policy systems as a key area of future investigation.

A key question is whether an condition can be designed in such a way that it will be able to address a large number

of security requirements. One of the key design goals of the the Antigone condition API is to support just this model. However, as seen in Section 5, one cannot anticipate all possible security requirements.

The condition schema defined in Section 4.1 is designed to specify the following security requirements/properties. These properties came about from a study of the conditions used by the AMirD multiparty file-system mirroring and other example applications [12]. While we acknowledge that many other properties and definitions exist, we argue that these are suitably representative to develop an understanding of condition security. However, the schema may be extended as need to encompass additional properties.

- *confidentiality* - An adversary must not be able to ascertain (with some fairly high probability) the condition or parameters being evaluated. Generally, this requires that the content of communication be made inaccessible (e.g., via encryption).

- *integrity* - The adversary must not be able to alter the results of an evaluation. For this property to be preserved for a remote condition, the parameters must be faithfully communicated to the remote entity, and the result must be returned without modification.

- *authenticity* - The evaluator must be able to ascertain the origin of the evaluation result. If the source of condition evaluation is not authentic, the policy infrastructure is subject to manipulation. For example, either as a man or in middle or by masquerading as the source, an adversary could intercept and alter results to allow itself or cohorts access. Where such an attack is mounted to attempt a denial of service, it could prevent any legitimate activity could be prevented by blocking all access to the protected resources.

- *non-repudability* - A condition source must not be able to claim that it did not assert a returned result. This is important where access to highly valued resources are being governed by authorization policy. For example, the trading example above will accept or reject transactions based on the evaluation result received from the timing source. Because this acceptance or rejection may have serious legal or financial ramifications, some evidence of the correct evaluation is necessary. Should a dispute arise, the trading service (application) would want demonstrate that if correctly evaluated policy. Assuming non-repudability is provided, if the timing source incorrectly evaluated the condition, then it could not later deny this fact (and would be culpable).[3]

- *anonymity* - The identity of the evaluator should not be known by the condition source. Moreover, an adversary on the network should not be able to reasonably ascertain the existence of the evaluation or identity of the evaluator.

  Note that in some instances, achieving anonymity is difficult (more than simply encrypting communication). For example, again consider the `timebetween()` condition. If a well known timing source is used, the existence of communication between the evaluator and the timing source may expose the evaluation, i.e., condition evaluation is the only reason that the evaluator would communicate with the timing source.

While these properties primarily address security requirements of remote evaluation, they also may be important in local environments. For example, any system may wish to prevent an adversary from manipulating locally stored system state.

3.3 Evaluation

Conditions are often characterized simply as functions. Each condition maps a set simple inputs onto a set of simple outputs. However, such characterizations make a number of assumptions about the behavior of its evaluation. The condition is assumed to have valid input that is always available (one can think of persistent state as input). The abstraction of function fails to capture the fact that conditions are not always mathematical operations, but must be sensitive to the constantly evolving environment in which they exist.

Possibly a more useful characterization of a condition is that each represents a *program*. This extended view embraces the dynamicity of the environment: conditions can fail because of insufficient or unavailable resources, have invalid input, or simply take too long to evaluate. This view of condition places additional requirements on the authorization policy infrastructure. That is, the infrastructure must accept and carefully consider how authorization policy is evaluated in the presence of such failures. Note that blanket policies such as “treat every failure as a negative response” may provide a means by which an adversary can manipulate policy evaluation. Note that the these meta-policies are often a function of the semantics of the policy language, and are not necessarily defined by the policy infrastructure.

Note that a perfectly implemented application can be co-opted by a poorly implemented condition. For example, consider the trading example in the preceding section. Assume that every transaction is governed by a policy that uses the `timebetween()` condition. Any adversary that wishes to prevent transactions from occurring can DOS the timing source. Note that this vulnerability is not a function

[3] To ease exposition, the above description simplifies secure transaction timing. In practice, the transaction itself must be tied to the timing information (in some cryptographically strong way).

of the trading application, but is a function of the policy that governs it.[4] Such dependencies can be overlooked (or be unknown) at the time applications are built. Hence, it is incumbent upon the developers of the condition implementations to anticipate and address the needs of the target applications.

Policy infrastructures are increasingly allowing applications to provide the conditions upon which policy decisions are made. Systems like Antigone [15] and the Generic Authorization and Access-control API (GAA-API) [20] provide generic *upcall* interfaces to which condition implementations are built. Applications register at compile or run-time the set of conditions to be supported by the application. The policy infrastructure passes the relevant state through the upcalls. The condition specific code is executed and results returned to the policy infrastructure. Because these upcalls are essentially programs, they are free to implement a wide range of functions.

The following considers broad classes of condition implementations. These design pattern represents type of conditions observed in existing policies, or those as designed to support the Antigone project.

Condition function - conditions of this type are simply computable functions. Often using state that is internal to the application, these conditions simply test some property of the environment. For example, consider the ubiquitous condition `username(name)`. This condition tests whether the local identity name is equal to the parameter string. Such conditions are encoded, for example in KeyNote, as equations, e.g.

```
(username == "bob")
```

The key aspect of these conditions is that they can be implemented directly in the application or the policy infrastructure. This pattern is representative of the vast majority of conditions seen in policy systems.

Local evaluation - locally enforced conditions perform some evaluation function by extracting and manipulating state on the local host, but external to the process address space. The distinction between local evaluation and condition function is useful because process external information is subject to external forces. Unseen adversaries can manipulate local resources and state (i.e., modify the `/etc/password` file on a UNIX system). Hence, the threat models appropriate for the condition classes are fundamentally different. Moreover, depending on the nature of the condition, the ways in which conditions are implemented are likely to be very different.

[4]Note that additional systems engineering (e.g., redundancy) of the condition evaluation infrastructure, rather than the application or service is the only means by which these issues can be dealt with. In this case, alternate timing sources can be used to mitigate this attack.

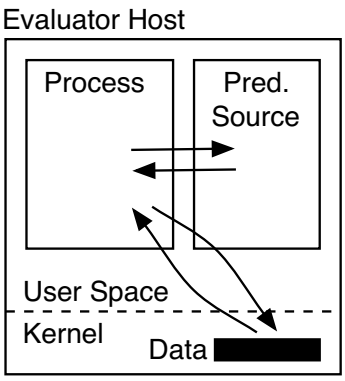

Figure 2: *Local Evaluation* - The condition is implemented by a call to some local host service. This information can be accessed using local services (e.g., via IPC) or by extracting information directly from the kernel (e.g., via system call).

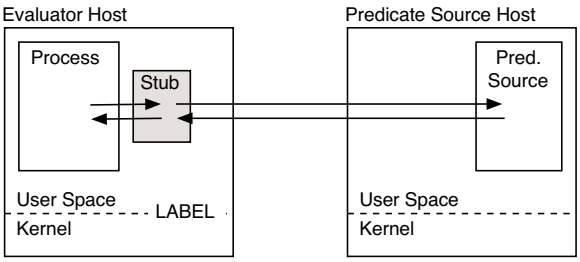

Figure 3: *remote method invocation* - The condition is implemented as a call to remote function. Existing mechanisms (e.g., RPC, CORBA) provide functional semantics, and can readily be used to implement policy conditions.

Depicted in Figure 2, each conditions extracts state from the local environment. In the case where state is held or computed by a local service (e.g., monitor process), some IPC mechanism is necessary. The means and format of such communication is dependent on the service and semantics of the condition. Other mechanisms allow state to be retrieved from the operating system itself (e.g., via system call).

A canonical local evaluation condition is `system-load(component,thresh)`. This condition tests whether a threshold load on some aspect of the system has been reached. As such, it can be used as a form of admission control in authorization policy: access is granted only where the system has sufficient resources to support it. This would implemented by polling a monitoring process or directly extracting it from the kernel via system call.

Remote method invocation - these conditions simply poll external services. One can view these condition implementations as traditional remote procedure calls [3]. Figure 3 il-

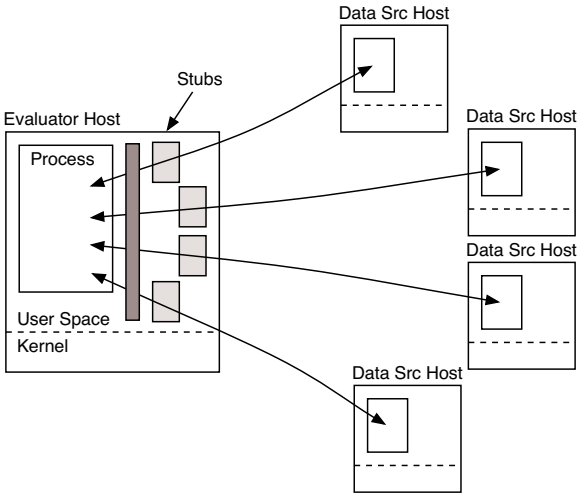

Figure 4: *complex condition evaluation* - Each condition is implemented by some distributed algorithm or protocol. These conditions are frequently designed to support specific authorization policy needed to govern the target application or domains.

lustrates the common RMI condition design. The condition marshals and transmits input parameters to a remote service through a *stub* function, and receives and unmarshals the results. Frameworks supporting this design common in contemporary distributed systems (e.g., CORBA [25], SOAP [8], Java RMI [9]). A key advantage of using these frameworks is that they are often implement their own security infrastructure. Note that the security requirements of RMI methods are very different from previous designs (i.e.,, remote vs. local threats).

It is worth acknowledging that while the preceding discussions have implied the remote entities that "perform" the condition evaluation operation, but this is not necessarily the case. In practice, conditions may make use of poll externally available interfaces to acquire the information needed to perform evaluation. This information is manipulated in some way to implemented the condition semantics. The distinction here is that the remote service being polled need not necessarily be aware it is part of an evaluation process. This is best illustrated by example.

Consider yet again the `timebetween(start,end)` condition. In practice the condition is most likely implemented by polling a remote timing source. The timing source simply returns the current time. The receiving stub will translate the current time into a result: in this case, determining if the current time is between *start* and *end*. Because time is returned, the same service can be used to implement any number of conditions (e.g., `isWeekday()`).

complex condition - conditions often require implementations that are hybrids of local, remote, and functional conditions. Such implementations can represent more extensive programs which may involve interactive protocols involving many end-points. Depicted in Figure 4, the conditions can be presented to the policy infrastructure as a single stub function. However, beneath this simple veneer lies complex logic which coordinates the data and protocols needed to evaluate the condition. Note that it may be possible to for an implementation to be (at least partially) constructed from more basic conditions. This is similar in philosophy to the component systems and protocol stacks [2], and is illustrated in the following example.

A lock algorithm is a good example of a complex condition. Consider a condition `hasLock(lock-id)`, where *lock-id* is some lock passed between a number of peers. The condition returns *true* the local host currently has or is able to obtain the lock *lock-id*. An implementation of the condition initially tests the local environment to see if it already has the lock (e.g,. through, for example, a local or condition function `hasLockLocal(lock-id)`). If not, it will attempt to acquire the lock by performing a distributed computation (and possibly talking to every other peer in the system). If the lock protocol is successful (in the sense that the local entity obtains the lock), then the condition returns true.

4 Condition Framework

A key observation gleaned from the classification presented in the preceding section is the degree to which conditions share common properties. We exploit these commonalities in designing a general purpose condition service and associated API. The Antigone Condition Framework (ACF) specifies and directs the use of conditions through common artifacts and processes. The remainder of this section considers in detail the design and use of this framework and concludes with a discussion of the advantages, limitations, and opportunities afforded by its architecture.

ACF supports the creation and evaluation of policy conditions. Depicted in figure 5, the life-cycle of a condition in ACF is defined by three central processes: condition *specification*, *instantiation*, and *evaluation*. Initially, infrastructure developers will define the sets of conditions that are made available to the policy issuers. This includes the creation of an implementation as well as *specifying* a defining Antigone Condition Document (ACD). This document is used to detect incorrect usage when the policy is issued (*instantiation*). The condition is *evaluated* implementations made available to the ACF when policy is used.

Note that while this framework is targeted to the Antigone/Ismene system, it is policy agnostic: that is, any policy language is free to use conditions built within this framework. However, the policy infrastructures often must modify the evaluation process to embrace run-time condition evaluation.

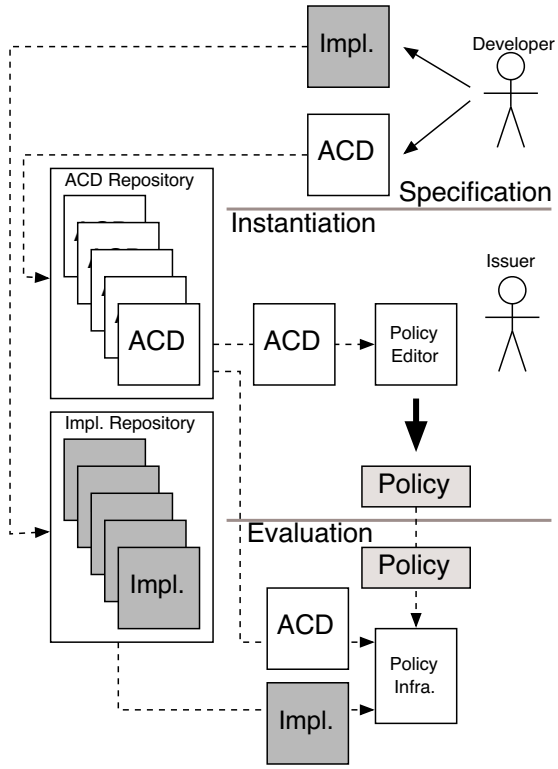

Figure 5: Condition Life-cycle - conditions are specified in Antigone Condition Documents (ACD) and implemented during condition *specification*. The ACD is subsequently used to validate issued policy (during *instantiation*), and ultimately to direct the use of the condition (during policy *evaluation*).

```
<!DOCTYPE condition [

<!-- General Section -->
<!ELEMENT condition ( name, version,
                      secpolicy?, parameter* )>
  <!ATTLIST mode (synch|asynch) "synch">
<!ELEMENT name (#PCDATA)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT implref (#PCDATA)>

<!-- Security Policy Section -->
<!ELEMENT secpolicy (service,authority*)>
  <!ATTLIST secpolicy confidentiality (T|F) "T">
  <!ATTLIST secpolicy integrity (T|F) "T">
  <!ATTLIST secpolicy authenticty (T|F) "T">
  <!ATTLIST secpolicy nonrepudation (T|F) "F">
  <!ATTLIST secpolicy anonymity (T|F) "F">

<!ELEMENT service (#PCDATA)>
<!ELEMENT authority (name?,encoding,credential)>
<!ELEMENT encoding (#PCDATA)>

<!-- Parameters Section -->
<!ELEMENT parameter ( order, const,
                      value, encoding? )>
  <!ATTLIST parameter ( string | boolean |
                        float | integer |
                        encoded ) #REQUIRED>
  <!ATTLIST const type (T|F) #REQUIRED>
<!ELEMENT order (integer)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT encoding (#PCDATA)>

<!-- Base Types -->
<!ELEMENT integer (#PCDATA)>

]>
```

Figure 6: Antigone Condition Document DTD - the ACD DTD defines the basic condition type, a security policy, and a set of expected parameters. This information is used at policy issuance and evaluation to ensure correct condition usages, as well as a roadmap for execution.

An initial version of ACF is under construction. As is true of any large framework, any number of issues will emerge as the implementation matures. For this reason, this section centrally focuses on the means and use of ACF interfaces. We plan to complete the construction and report on performance and usability issues in the near future.

4.1 Specification

The specification process defines the semantics of a condition through through an implementation. The ACD defines how a policy must communicate with this implementation. The ACD is simply a XML document conforming to the ACD document type definition (DTD). Presented in Figure 6, this DTD is comprised of three main sections, a general section, a security policy, and the parameters section.

The general section specifies the high level attributes of the condition. This includes a definition of the name and version information, as well its mode of operation. The mode determines whether the condition is synchronous or asynchronous (see Section 3.1), the implref identifies the implementation (see Implementing Conditions 4.4 below).

The security policy identifies the security requirements that are relevant to the condition. The requirements are represented through a set of flags indicating what specific properties are of interest. The service element is an opaque string used to identify which security infrastructure should be used to implement security (e.g., IPsec). Finally, the set of authorities used to perform evaluation, if any, are encoded as indicated by the element definition. Note that because local evaluation conditions do not consult external sources, security policy is optional.

Finally, the parameter section defines the expected input. This includes the obligatory ordering and type enumeration, as well as constant and encoding definitions. The *const* element indicate whether the input value is fixed by policy definition or asserted at run-time (e.g., like KeyNote attributes). Where the type is not atomic, encoded forms are supported (e.g., PKCS#12 encoded certificates).

An ACD is created by partially instantiating the objects defined in the DTD. For example, Figure 7 presents an ACD for a `hasToken` condition. This condition takes three

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE note SYSTEM "acd.dtd">

<condition mode="asynch">

  <!-- Security Policy -->
  <name>hasToken</name>
  <version>1.0</version>
  <implref>libhtok.1.0.so</implref>

  <!-- Security Policy -->
  <secpolicy confidentiality="T"
             integrity="T"
             authenticty="T"
             nonrepudiation="F"
             anonymity="F">
    <!-- Note that run-time policy evaluation
         will insert authorites -->
  </secpolicy>

  <!-- Parameter List -->

    <!-- Token ID -->
    <parameter type="string" const="T">
      <order>1</order>
      <value>tokenid</value>
    </parameter>

    <!-- Token Authority Certificate -->
    <parameter type="encoded" const="F">
      <order>2</order>
      <value></value>
      <encoding>PKCS#12</encoding>
    </parameter>

    <!-- Wait Time -->
    <parameter type="integer" const="T">
      <order>3</order>
      <value>60</value>
    </parameter>

</condition>
```

Figure 7: Example ACD - the `hasToken` condition attempts to acquire a named token from the authority identified in the parameter list.

arguments, a token identifier, a certificate, and a wait timer. The corresponding implementation attempts to acquire token associated with the token identifier from a service defined in the certificate. If a signed response is received prior within the wait time (seconds), *true* is returned if the token is acquired, and *false* otherwise. The condition fails if no response is received in the allotted time.

4.2 Instantiation

A policy editor is an application used to issue policy [26]. Such editors are responsible for validating that the policy is consistent with the policy representation, and is self consistent (e.g., is enforceable [13]). For example, consider the following call to the `hasToken()` included as part of a larger Ismene policy:

```
hasToken(tigertoken, $cert)
```

A policy editor making use of the ACF would begin by looking up the ACD for `hasToken()` (defined in Figure 7). The ACD indicates the three parameters are used. The first parameter is a string constant defined by the policy. The *tigertoken* string fulfills that definition and the parameter is accepted as valid. The second parameter is marked as a run-time attribute (i.e., const="F"). Because the second parameter is marked as a run-time ('$'-symbols indicate attribute replacement in Ismene), the second parameter is accepted. One might assume that because definition is missing the last argument that the condition would be rejected. However, because the ACD defines a default value for the last parameter (i.e., in the value element), it is automatically inserted. Because all parameters are consistent with the definition, the condition is deemed correct.

In practice, policy editors will use libraries of ACDs to interpret and verify issued policy. It is incumbent upon the administrators to provide ACDs for all of relevant conditions. For example, an LDAP directory [27] of ACDs XML documents could be maintained by the policy issuers. The design and maintenance of these repositories is outside the scope of the current work.

4.3 Evaluation

The policy infrastructure calls the `ACF_evaluate` function to execute the evaluation process. This call (and the ACF-specific return enumerate type) are defined:

```
typedef enum { ACF_false  = 0,
               ACF_true   = 1,
               ACF_fail   = 2,
               ACF_asynch = 3
             } ACF_result;

ACF_result
ACF_evaluate( char *func, int argv,
              char *argc, void *context );
```

Note that all parameters are passed as null terminated strings (as in standard C programming `argc/argv` style. Parameters such as credentials must be encoded prior being passed to the ADF framework (e.g., PKCS#12 encoding of certificates). This has the advantage that the policy infrastructure need not be share domain specific structures (e.g., internal representations of Kerberos certificates). Where necessary, the ACD specifies the expected encoding scheme in the `encoding` element in the parameter section.

The evaluate function returns one four return values. `ACF_true` or `ACF_false` are returned where the condition successfully evaluates to *true* or *false*, respectively. `ACF_fail` is returned when the condition cannot be successfully evaluated. The policy infrastructure must decide how to handle these errors. `ACF_asynch` is returned whenever the called condition is asynchronous.

Asynchronous conditions are handled by threads created during the initial evaluation call. The policy infrastructure

registers an opaque `context` object with the ACF through in the evaluation call. When the evaluation is completed, the framework signals the completion of the evaluation via callback to the policy infrastructure. The callback supplies a return value (i.e., true, false, fail) and the originally registered context object. The policy infrastructure maps the context object back onto the suspended operation.

The evaluate function initially acquires the ACD associated with the `func` parameter from a ACD repository. The parameters are validated as in the instantiation (with the exception that all non-const values are now instantiated). A new ACD with the new parameters values and authorities is created. The registered implementation is called with a single argument, the ACD.

4.4 Implementing Conditions

Conditions are implemented as threads. Each such thread accepts the singular ACD object, and interprets the parameters accordingly. A number of utility functions are being created to ease the process of condition creation (e.g., parameter extraction, credential decoding). What the condition does, how it does it, and the resources it uses is entirely up to the developer.

The ACF does not provide a specific security infrastructure: all details of how the security policy is enforced is left to the condition implementation. The reasoning for this decision is simple, we cannot possibly anticipate all the possible infrastructures and approaches to providing security.[5] Hence, we defer issues to the implementation. It is important note that does not mean that condition security is fixed service. Inasmuch is possible, each implementation should be able to deal with the the security policies communicated by ACDs. This adds additional flexibility, where the particular environment can dictate the security needs by modifying the ACD, rather than the implementation.

Note that in practice, many conditions are implemented as external programs (i.e., shell scripts [5]). ACF supports these implementation by creating a thread that, after extracting the parameter values, simply forks the appropriate process and sleeps until that process terminates.

One of the central issues associated with implementations is the means by which they are loaded. Requiring that ACF be linked against every possible condition implementation is problematic. Firstly, this would require that every condition be known *a priori*, which is exactly the kind of assumption ACF is designed to avoid. Secondly, even if one knew (and could acquire) beforehand the implementations of every condition, the resulting executable would be huge. Finally, this requires that the ACF be rebuilt every time a new condition is introduced.

[5] We expect to explore standard security services as more experience with the framework is garnered.

For all of these reasons, we determined that it is imperative that condition implementation be *dynamically loadable*. To accomplish this, ACF is being built with a condition loader facility. This uses the UNIX dynamic load functions [18] (e.g., `dlopen`, `dlsym`) to open and read local libraries. Each condition implementation is provided in a shared library whose name is specified in the implref element of the ACD. The library exports a single symbol, a function with the condition name that receives a single character pointer (the ACD).

We intend to investigate more flexible distribution methods for condition implementations in the future. For example, the ACF could download implementation from a central (and presumably authenticated) repository when an implementation is not locally available. However, such methods must be carefully designed to avoid introducing new vulnerabilities (e.g., possibility of DoS).

5 Discussion

The ability to integrate complex and distributed condition evaluation within authorization policy opens the door to new uses of policy. For example, consider the `hasToken` condition in the previous section. Applications which are regulated with this condition automatically implement token-based operation. Hence, through condition evaluation, authorization policy can define application behavior. This *policy-oriented programming* enables application features to be transparently added through the use of context.

Aspect-oriented programming [11] seeks to implement high-level features through object technologies (e.g., inheritance). Policy-oriented programming differs not only in approach, but by whom and when application behavior is delegated. Policy approaches allow domain administrators (issuers), rather than developers to decide which features an application will implement. Moreover, based on the semantics of the policy representation and evaluation, judicious use of condition allows the developer to clearly specify how, when, and by whom these features are used. Moreover, policy base programming occurs at a finer-grain: features are applied to individual actions, rather than on the application as a whole.

State maintenance in condition evaluation can complicate policy. For example, consider an authorization policy rule representing the conjunction of two stateful conditions `hasToken(a)` and `hasToken(b)`. Now assume in a particular evaluation, an evaluator obtains the `a` lock, but not `b`. The operation would be rejected. The evaluator would hold the lock `a` but not perform the association action (which we assume releases the lock). Moreover, if another evaluator obtained the lock `b`, but could not obtain `a`, a deadlock would occur.

Ryutov and Neuman [20] address the state maintenance problem by introducing *pre-*, *mid-*, and *mid-conditions*. These conditions essentially identify the operations that must occur at different phases of the action. Hence, the underlying code can manipulate state as is necessary for the application. We are currently looking at a condition-centric version of this approach. The modified approach informs condition implementations that the action/rule to which they are attached was rejected, failed, or completed successfully. Each implementation is expected to perform the appropriate processing based on the result information.

While this work has discussed the security needs of condition evaluation, the infrastructure described in the preceding section does not indicate how these needs are addressed. The reason for this is two-fold. Firstly, building a security infrastructure that meets the needs of all possible environments is impossible. Hence, specifying specific technologies (e.g., PKI, AES) is inherently limiting. For this work, we have chosen to focus on the interfaces and definition of conditions, rather than their implementation.

Secondly, security requirements often can only be met through integration with existing security services. One illustrative application is a shopping application. An authorization policy governing the purchase action would indicate that the credit card purchase must be accepted by the Secure Electronic Transactions (SET) protocol. Because it is unlikely that any general purpose infrastructure will implement SET, it must be implemented within the condition (or application). Because many such dependencies exist in real applications, attempting to construct a single framework that addresses all conditions is not realistic.

6 Conclusions

In this paper, we have considered traditional and extended views of policy conditions. In its extended form, we view conditions as programs, rather than as expressions defined over a fixed set of attributes (as one would see in contemporary policy systems). However, we must acknowledge the additional security and infrastructure requirements that this view introduces. Our taxonomies of condition type, evaluation method, and security requirements show that conditions are largely defined along similar axises. Because conditions share similar properties, we can contemplate general-purpose facilities.

We have designed the novel Antigone Condition Framework (ACF). This framework implements a general-purpose condition specification, implementation, and evaluation service. In ACF, conditions are defined by XML documents called Antigone Condition Documents (ACD) and implemented by dynamically loaded libraries. At runtime, ACDs are used to validate and initiate condition evaluation, and to reference (and potentially acquire) condition implementing libraries. ACF is a general purpose framework. Subject to semantic restrictions, any policy infrastructure can be augmented with ACF conditions. Hence, the ACF can be used to expand existing policy with more flexible context.

The extended view of conditions affords new ways of leveraging policy. *Policy based programming* allows policy issuers to augment existing applications with new features through the specification of authorization policy (e.g., implement distributed locking through condition evaluation). Hence, issuers are able to use *late-binding* to add environment-specific application behavior. Moreover, the applications need not be aware of the added functionality.

The current ACF implementation is a very rough prototype. In the near future, we intend to complete the code and experiment with many different kinds of conditions. As part of this process, we will refine the schema and consider the design of directory services used to store ACDs and implementations. Other works will investigate how policy-based programming is used to support flexible environments. It is through these works that we hope to expose the semantic depth, and hence the value, of policy context.

References

[1] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, MITRE Corperation, Bedford, MA, 1973.

[2] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16(4):321–366, November 1998.

[3] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. In *Proceedings of the ACM Symposium on Operating System Principles*, page 3. Association for Computing Machinery, 1983.

[4] M. Blaze, J. Feigenbaum, John Ioannidis, and A. Keromytis. The Role of Trust Management in Distributed Systems Security. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, volume 1603, pages 185–210. Springer-Verlag Lecture Notes in Computer Science State-of-the-Art series, 1999. New York, NY.

[5] M. Blaze, J. Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, November 1996. Los Alamitos.

[6] M. Blaze, J. Feignbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust Management System - Version 2. *Internet Engineering Task Force*, September 1999. RFC 2704.

[7] Matt Blaze, John Ioannidis, and Angelos D. Keromytis. Trust management for IPsec. *Information and System Security*, 5(2):95–118, 2002.

[8] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Frystyk Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.2, June 2002. `http://www.w3.org/TR/soap12-part1/`.

[9] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI performance and object model interoperability: Experiments with Java/ HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–955, 1998.

[10] John Ioannidis. *Personal communication*, December 2002.

[11] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[12] P. McDaniel. *Policy Management in Secure Group Communication*. PhD thesis, Univeristy of Michigan, Ann Arbor, MI, August 2001.

[13] P. McDaniel and A. Prakash. Antigone Secure Group Communication System. *NASA Tech Briefs*, 2001. (*to appear*).

[14] P. McDaniel and A. Prakash. Methods and Limitations of Security Policy Reconciliation. In *2002 IEEE Symposium on Security and Privacy*, pages 73–87. IEEE, MAY 2002. Oakland, California.

[15] P. McDaniel, A. Prakash, and P. Honeyman. Antigone: A Flexible Framework for Secure Group Communication. In *Proceedings of the 8th USENIX Security Symposium*, pages 99–114, August 1999.

[16] P. McDaniel, A. Prakash, J. Irrer, S. Mittal, and T. Thuang. Flexibly Constructing Secure Groups in Antigone 2.0. In *Proceedings of DARPA Information Survivability Conference and Exposition II*, pages 55–67. IEEE, June 2001.

[17] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. Policy Core Information Model – Version 1 Specification. *Internet Engineering Task Force*, February 2001. RFC 3060.

[18] Unix Man Page. `dlopen` man page. Linux Programmers Manual, Section 3.

[19] G. Patz, M. Condell, R. Krishnan, and L. Sanchez. Multidimensional Security Policy Management for Dynamic Coalitions. In *Proceedings of Network and Distributed Systems Security 2001*. Internet Society, February 2001. San Diego, CA, (*to appear*).

[20] T. Ryutov and C. Neuman. Representation and Evaluation of Security Policies for Distributed System Services. In *Proceedings of DARPA Information Survivability Conference and Exposition*, pages 172–183, Hilton Head, South Carolina, January 2000. DARPA.

[21] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[22] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[23] Ravi S. Sandhu and Pierrangela Samarati. Access Control: Principles and Practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[24] Mary Thompson, William Johnston, Srilekha Mudumbai, Gary Hoo, Keith Jackson, and Abdelilah Essiari. Certificate-based Access Control for Widely Distributed Resources. In *Proceedings of the 8th USENIX Security Symposium*, pages 215–228, August 1999.

[25] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1994.

[26] A. Westerinen, J. Schnizlein, J. Strassner, Mark Scherling, Bob Quinn, Jay Perry, Shai Herzog, An-Ni Huynh, Mark Carlson, and Steve Waldbusser. Policy Terminology (*Draft*). *Internet Engineering Task Force*, march 2001.

[27] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *Internet Engineering Task Force*, March 1995. RFC 1777.

[28] J. Zao, L. Sanchez, M. Condell, C. Lynn, M. Fredette, P. Helinek, P. Krishnan, A. Jackson, D. Mankins, M. Shepard, and S. Kent. Domain Based Internet Security Policy Management. In *Proceedings of DARPA Information Survuvability Conference and*

Exposition, pages 41–53, Hilton Head, South Carolina, January 2000. DARPA.