JPolicy: A Java Extension for Dynamic Access Control

Tim Owen Ian Wakeman Julian Rathke

Computer Science Technical Report 01/2004
Department of Informatics, University of Sussex, UK

January 2004

Abstract

The development of remote execution platforms, where service hosts accept code from third party clients and run it on their behalf, provides a powerful alternative to the RPC model of clients invoking remote services over a network. A major concern for service hosts is to control access and usage of their own services and resources, whether the client code is being executed by the service host itself or operating remotely and using RPC. We introduce a set of extensions to Java that enables service hosts to control how their services may be used by client Java programs, according to a specified policy. We separate the policies from the services being controlled, rather than fixing the policy at compile-time or load-time. A novel aspect of this work is that the policies allow dynamic access control to services: the availability of service functionality can vary during execution. Furthermore, we support the changing of policies at run-time without modification to service code or clients. We describe our implementation of these language extensions and show how the system enables service providers to enforce policies that protect their resources and services from undesirable client code behaviour.

1 Introduction

While the traditional model of the Web involves users manually interacting with a visual interface, the development of Web Services [32] is motivated by the desire to allow programmable access to services. But from the programmer's point of view, Web Services essentially involve just RPC [3] invocations: the client code packages together a request and optionally some data and sends it to the server. The server executes its own code to service the request and may return some data as a result.

Although this client-server mode of web service interaction is a powerful extension to the monolithic model, it can involve a considerable amount of network traffic for applications that require intensive dialogue with a remote service. Furthermore, clients with limited resources or connectivity are not always the most appropriate place to execute code that interacts with a remote service.

The basis of the Remote Evaluation [26] model (also termed Remote Execution, or just RE) is that service providers accept programs from third-party clients and host the execution of the code themselves. This extends the RPC style by allowing code to be packaged and sent to a server, rather than simply supplying data with a request for the server to execute its own code. Such a facility enables programmers to deploy their own code to perform useful work with a remote service, with the benefit of placing the code closer to the services that it requires. Furthermore, clients with limited resources can deploy code to be executed remotely then disconnect or switch off until such time as the results of the computation are required.

In either of the above scenarios a major concern for service hosts is how to control access and usage of their own services, whether the code accessing a service is being executed by the service host itself, or operating remotely and using RPC. For example, the Amazon Web Service API[2] enables RPC-like programmable access to the company's online shop, but specifies some restrictions on how client code may use the service. According to the access agreement, client code may not perform requests faster than once per second (although this is a guideline and is not actively enforced). Similarly, the Google Web Service API[15] limits client programs to 1000 query requests per day (which is enforced: an exception is thrown if the limit is exceeded). The Ebay API[10] offers a range of different limits depending on the level of membership — a typical basic limit is that programmers can make upto 5000 calls per day in test mode and 50 calls per day in production mode, and maintain 8 simultaneous connections.

We believe that any increased adoption of programmable services and hosting of third-party code will require mechanisms that allow hosts to control the usage of their services. The informal policies used by Web

Services such as Google, Amazon and Ebay indicates that policy-based control of service use is already of interest. In the RE scenario, where service providers host third-party client code, then the need for access control and resource rationing becomes particularly important. Since hosted code will be relying upon the host for access to more general services such as disk storage and network functionality, more sophisticated policies and policy control are required.

There are various ways in which a program may exhibit undesirable behaviour, but we are particularly interested in the external dependencies of hosted programs: the services that they use via programming APIs to libraries. Figure 1 shows an application written by a third-party deployed to be run on a service host. The application code relies on external host-provided services to be available in its execution environment. These typically include disk and file access, network functionality and database use, for example. Higher level services such as Google's search facility or other network-accessible services are also used through an API, so these too form part of the external environment in which a program executes.

We consider memory consumption and CPU usage as essentially operating system (or virtual machine) concerns. While these issues are clearly important to the overall behaviour of code, those resources are typically not accessed explicitly by a program in the way that library services are — rather, they are implicit in its execution. Our work will not focus on CPU and memory usage and we consider this to be an interesting but orthogonal issue.

If we assume that host resources and services are provided to a program through an API, then a typical level of granularity for control is an individual method. In Figure 1 the small circles show the points at which client code makes calls out to host services. Hence, controlling these API method invocations to external service libraries allows us to impose policies on program behaviour.

In this paper, we describe how programming language support for fine-grained controls on the behaviour of programs enables service providers to protect access to their services and resources. Our design shows that

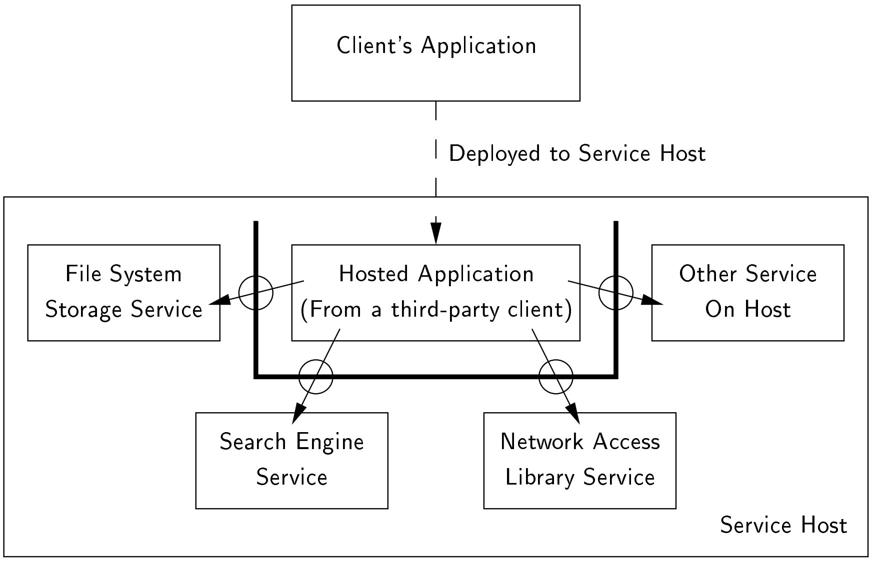
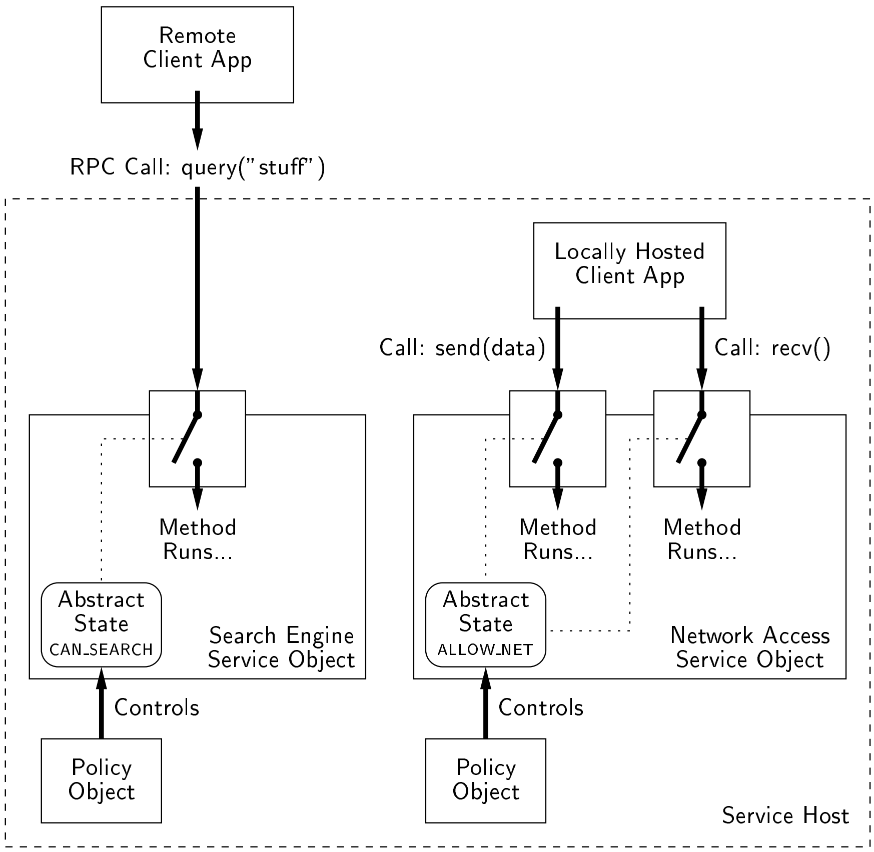


Figure 1: Remote Execution on a Service Host

the application of policies to control service use can be achieved without necessarily requiring client code to be aware of this policy control. We introduce a language construct for specifying service usage policies such as those examples mentioned above, and demonstrate a prototype implementation that extends Java with the ability to control how client code can use a service. This technique applies equally well to controlling service use in the traditional RPC style and in the RE situation where client code is being hosted by the service provider. In the remainder we refer to this Java extension as *JPolicy*. The JPolicy language is a modest extension of Java with a small number of extra constructs to support policy-based control of services.

The type of scenarios we aim to support with our system are exemplified in Figure 2. Service providers implement their services in Java as classes and interfaces in the normal way. The functionality provided by the service is specified by the service API, which is the set of methods that clients can call. Clients can access the service either remotely, using RPC, or locally if the service host is prepared to accept third-party client code and execute it on the host. We assume that each client gains access to a particular service through an object instance that represents that service, provided




```
class SearchEngine {
  Vector query(String searchTerm) when CAN_SEARCH { ... }
}

policy BoundedQueries for SearchEngine {
  CAN_SEARCH when { /* total calls to query method < bound */ }
}

policy TimeLimitedSearch for SearchEngine {
  CAN_SEARCH when { /* time of day is between 9 and 5 */ }
}

class NetworkAccess {
  void send(Object data) when ALLOW_NET { ... }
  Object recv() when ALLOW_NET { ... }
}
```

Figure 2: Example JPolicy Services and Policies

to them as their unique entry point.

Since the functionality of a service object is accessed through its methods, we can control the behaviour of client programs that use the service by enabling or disabling the *availability* of each service API method dynamically. This is depicted in the diagram of Figure 2 using the “switch” metaphor. For example, we can impose a rate-limiting policy on a SearchEngine service by switching off the availability of the query method after a limited number of invocations has been reached, then switching it back on after some delay.

A novel aspect of the JPolicy language lies in the separation of policy from service code. Many existing policy control or resource access control systems rely on a tight coupling of (compiled) service code and exact access control specifications [9, 7, 21, 5]. Decoupling policy from code provides some immediate benefits: servers are not obliged to recompile or rewrite API code to account for any changes in policy and versioning of policy controlled services is achievable within a single policy control framework.

The technology which underpins this decoupled approach to variable method availability is based on a notion of *abstract state*, in which a method invocation proceeds only when a service object's *abstract* state permits it. We may like to think of these as named properties of the underlying states of the policy. For example, a service offering a `print` method may offer this functionality under various circumstances, or states, described exactly in a sophisticated underlying policy. From the service code's perspective all that is relevant though is whether the method is available when called. This might lead us to an abstract state named *Printable* which is intended to describe those collection of underlying states in which the `print` method is available. By only allowing the service code to refer to abstract states we enforce a separation of *policy* from the service object itself, so that the policy is not hard-wired into the service.

The extensions to Java that we include in JPolicy enable this policy-driven control of method availability:

- Instance methods can be annotated with a *when* clause, that specifies an abstract state name. The presence of this clause makes the method *modal*, in the sense that the invocation of a modal method on some object can only proceed if the object is currently in the abstract state named in the clause.

- Each object of a class that contains modal methods maintains a notion of which abstract states it is currently in. For example, the NetworkAccess class in Figure 2 has modal methods naming an abstract state called ALLOW_NET and each NetworkAccess service object will record whether it is presently in the abstract ALLOW_NET state or not.

- The transitions between being in some abstract state and not are driven by an external policy rather than the object itself. The language supports the definition of policies, that determine when an object should enter or leave an abstract state. The conditions under which these transitions occur can be based on a variety of factors, as explained in Section 4.

- Client code that attempts to invoke a modal method will only proceed if the target object is in the correct abstract state. There are a number of ways in which clients can handle modal method calls: either execution blocks until the invocation proceeds, or blocking can be avoided by trying alternative code or skipping the call. If left unspecified the default behaviour for clients is to block on unavailable methods.

The following sections describe these extensions in more detail, and explain how they allow service access to be controlled in a fine-grained manner.

The remainder of the paper is organised as follows. In the next section we elaborate on the notion of modal methods and abstract states. The consequences of policy control for client code is discussed in Section 3. In Section 4 we present our model for policy specification and how these are linked with services. We follow this with a description of our JPolicy implementation and discussion of its run-time behaviour in Section 5. Finally, we conclude with sections containing discussions on related and further work.

2 Modal Methods and Abstract States

The primary purpose of the JPolicy extensions is to enable service hosts to control when the methods in their Java services can be called. Since the functionality of a service is accessed by invoking its methods, then we can enable or disable the use of particular service functionality by selectively blocking the invocation of a method in that service. This allows the service host to control what facilities client code can access, and when it can be accessed, assuming clients have no other means to use services in their hosted environment.

In JPolicy, the level of granularity for access control is individual methods, which allows fine-grained management of the functionality available to client code. For example, a host service that provides network access could be controlled with a policy that allows outgoing network connections to be made but not incoming, by enabling one service method and not another.

Our basic approach to enabling or disabling certain methods is to intercept calls from clients and decide whether to continue the invocation or not. Therefore, such methods can be thought of as having two modes of operation: the normal Java mode where an invocation proceeds immediately and a disabled mode where an invocation does not proceed but is blocked instead. These *modal* methods are distinguished in the JPolicy language by an annotation on the method's definition: a *when* clause. For example, the `SearchEngine` service class from Figure 2 has a modal method called `query` with the following signature:

```
Vector query(String searchTerm) when CAN_SEARCH
```

The presence of a when clause means that this is a modal method, and hence invocations of the method may or may not proceed immediately. Conversely, methods without a when clause are non-modal and behave exactly as normal Java methods.

The content of a when clause is the declaration of an *abstract state* name, which is specific to the class in which it appears. The intent of this is to declare that there is some abstract state of the object in which the associated modal method should be enabled. This abstract state is simply a yes/no flag, so e.g. a *SearchEngine* object is either in the `CAN_SEARCH` abstract state or it is not. The actual meaning of the abstract state, in terms of when individual objects are in the state is determined by the policy associated with an object — this is explained below in Section 4.

Since a class may contain several modal methods, each with a when clause potentially naming different abstract state names, the result is a set of independent abstract states. Each object of that class maintains a notion of its current combined status: which abstract states it is presently in, and which it is not. This status then controls the availability of methods, because the decision to allow a method call of a modal method to proceed is determined by whether the target object is in the relevant abstract state at that moment. A related issue here is that, in our current design, only instance methods can be modal — we do not allow Java's `static` methods to have when clauses. This is because the abstract state is a property of individual objects, but static methods are not invoked with respect to any specific object. We could extend the language to allow static modal methods, by associating the static abstract state with the class itself, in a similar way to Java synchronization on static method calls where the class's lock is used.

In terms of the diagram in Figure 2, the abstract state is the "switch" that manipulates the handling of method invocations. As shown in the `NetworkAccess` service, more than one modal method's when clause can refer to a particular abstract state. In that case, the availability of all those modal methods is tied together: either they are all enabled or all disabled. It is the task of service programmers when adding when clauses to make this decision about which modal methods should be controlled by which abstract state names.

An important feature of our design is that objects do not control the status of their set of abstract states, rather this is the responsibility of separate policies. An object enters or leaves an abstract state when its associated policy dictates. We discuss the definition and use of policies in Section 4.

3 Client Code Implications

As explained in the previous sections, the existence of modal methods in JPolicy means that the behaviour of a method invocation depends on the method's current mode, which is determined by the abstract state of the target object. Clearly, this has implications for client code that calls modal methods. The service API that is presented by the set of method signatures shows which methods are modal, as indicated by the presence of when clauses. This information is important to client programmers, in a similar way to the existence of a `throws` clause in a method signature: it warns the programmer that an invocation of a modal method may not proceed. This acts as a reminder to clients that the services they wish to use are subject to access controls.

In JPolicy, the effect of calling a modal method that is switched off (i.e. because the target object is not in the required abstract state) is to block the calling thread. The caller is only unblocked if the target object enters the relevant abstract state again, at which point the method call proceeds. This behaviour allows service hosts to completely deny service functionality to clients, if they choose to impose a policy that never puts the object into certain abstract states.

Our design does not allow clients to explicitly examine the current abstract states of an object, therefore we have introduced a construct which provides client code with alternatives to blocking. Instead clients can attempt an invocation of a modal method and take evasive action if the call does not proceed. This is analogous to exception handling constructs in Java — indeed, we use similar syntax. The following example of client code shows one form of this construct:

```
try Vector results=searchService.query("some terms") {
    // code block using the results value
}

// Further code (results not in scope here)
```

In this syntax, the newly declared local variable `results` is assigned the result of the modal method call if the invocation proceeds immediately, and its scope is only within the following code block. If the `query` method is switched off due to the current abstract state of the `searchService` object then the caller will not block, but the method call and code block will be skipped and execution continues after the block. Another use of this construct is to allow a second block of alternative code to be executed in the case of the modal method call not proceeding:

```
try Vector results=searchService.query("some terms") {
    // code block using the results value
}
else {
    // alternative code block (results not in scope here)
}
```

Here, the alternative block is executed if the `query` call cannot proceed immediately. The third variation of this construct simply includes a millisecond timeout clause to the modal method invocation attempt:

```
try for 100 Vector results=searchService.query("some terms") {
    // as before
}
```

In this form, if the object's abstract state changes to re-enable the method within the specified time of 100 milliseconds then the call will proceed. The benefit of this clause is that it provides clients with an option midway between immediate skipping and indefinite blocking.

4 Control Policies

The previous sections detail the way in which service providers can write their services so that they can be policy-controlled, and how client code can deal with using services that are controlled by policies. This section shows what these policies actually look like and how they are linked to the service objects under their control.

As explained above, the central concept in our work is the abstract state of an object, which is reflected in the set of named abstract states that appear in modal method when clauses. The availability of a modal method is dictated by the current status of the target object's associated abstract state. As Figure 2 shows, the role of policies in JPolicy is to cause changes in the abstract state of objects, whereas the service objects themselves only examine the status and do not change it. This separation of concerns means that policies are not hard-wired into the service code itself. Abstract states act as the intermediary: individual policies determine when to change an object's abstract state, and the object reads this status when deciding whether to allow a method invocation to proceed. Consequently, the job of the policy declaration construct we have included in JPolicy is to define exactly when an object is in an abstract state and when it is not.

4.1 Policy Specification

In the JPolicy language we extend the Java syntax with a top level construct for specifying a policy. Therefore a compilation unit of Java code contains a list of class, interface and policy definitions. Our model of a policy is in the form of a labelled transition system — essentially a finite state automaton, consisting of a set of concrete states with transitions between them. The policy definition declares the name of the class for which it can be used, then specifies the sets of its concrete states that correspond to each abstract state of objects of that class. The general form of a policy specification is as follows:

```
policy PolicyName for ClassName {
  -> initialConcreteState
  transition concreteState1 -> concreteState2 when (conditionA)
  transition concreteState2 -> concreteState3 when (conditionB)
  ...
  ABSTRACT_STATE_X when { list of concrete states }
  ABSTRACT_STATE_Y when { list of concrete states }
  ... // for each named abstract state declared in ClassName
}
```

```
class SearchEngine {
  Vector query(String searchTerm) when CAN_SEARCH { ... }
}

policy BoundedQueries (int bound, int interval) for SearchEngine {

  int credits = bound;

  -> some;

  // Every time we call the method, the counter decrements...
  transition some -> some when (query)
      { credits = credits-1; }

  // Until none are left...
  transition some -> none when (credits <= 0);

  // Then the counter is replenished at next time interval
  transition none -> some when ((TimeService.now % interval)==0)
      { credits = bound; }

  CAN_SEARCH when { some }
}
```

Figure 3: Example Policy Specification

The particular concrete states and transitions of these automata reflect the specific nature of each policy. For example, suppose we are specifying a Google-like Web Services policy that a certain modal method in a service can only be invoked a limited number of times in some time period. The policy may have two concrete states, to represent whether the call limit has been reached or not, and transitions between these based on a method call counter and a time event. Figure 3 shows how this policy can actually be written using the JPolicy syntax.

To assist in the construction of policies such as the call limiter outlined above, which involves counting, policy specifications can include local variables that may be updated using a limited expression language. Without this facility, policies that need to implement counters would have to specify states to record a count total, which becomes tedious and repetitive.

Figure 3 shows a local variable named `credits` that counts how many calls can still be made before the limit is reached. Updating of local variables, such as incrementing one used as a counter, is enabled by the addition of an optional clause in transition specifications. This clause, shown in braces at the end of a transition declaration, simply lists policy variable updates — it is not arbitrary Java code.

A further enhancement of the policy construct is that it can be parameterised by values, much like the way a Java class can be parameterised by having its constructor declare a list of formal parameters. A policy's named parameters can be used to initialise its local variables. In Figure 3 the `BoundedQueries` policy has been parameterised by the call limit and the time interval, rather than having these hard wired into the transitions.

The descriptions above outline the general form of the policy construct, but the utility and expressiveness of policies is determined mainly by the content of the boolean conditional expressions used to label transitions. In Figure 3, the `BoundedQueries` policy illustrates the use of most of the terms that can be referred to in condition expressions. In some cases, the condition expressions in a policy make reference to aspects of the object that the policy is controlling, which is called the target object. The terms used in transition expressions are:

Object instance methods The event of an instance method being invoked on the target object can be used to trigger a transition. The first transition in the `BoundedQueries` policy is an example of this: it names the `query` method in its condition, in order to count how many times the method is invoked.

Static fields The value of a static field of some arbitrary class can be used to gain access to external services in the wider environment. For example, the policy in Figure 3 makes use of a service that presents the current time as a one second counter field.

Object instance fields Although not used in the `BoundedQueries` policy example, any instance field of the target object can be referred to. This enables policies to use the internal state of service objects to guide transitions.

Policy local variables Policies can declare a number of mutable local variables, and these can be referred to in transition conditions. These variables can be updated by assigning new values when a transition occurs.

Policy parameters The named parameters of a policy can be considered as unchanging local variables, like `final` method parameters in Java. The example policy uses the `interval` policy parameter in its third transition expression.

Constant values These are simple literal values, such as integer constants.

With the exception of instance methods, the condition expression syntax allows these terms to be combined using a simple set of logical, conditional and arithmetic operators. A transition may only use a method name in its condition if it is the only term in the expression, because it does not make sense to apply expression operators to a method call event. An important property of the transition conditions is that they are pure expressions in the sense that they do not cause any side effects.

4.2 Policy Attachment

Once policy specifications have been written for service classes, the connection between a policy and a particular object is made by our final language extension: a simple policy assignment construct. This infix operator connects an object of some class with a policy for that class, specifying arguments if the policy is parameterised:

```
searchService policy BoundedQueries(250, 10)
```

where `searchService` refers to an object of type `SearchEngine` and the policy named `BoundedQueries` is defined as one for `SearchEngine` objects.

In a remote execution environment, service hosts will typically need to be able to assign policies to the service objects they give to clients, but deny clients the ability to change the policy associated with these objects. Clearly, if clients could replace the policies attached to service objects then they can subvert the host's control on service usage — thus defeating the purpose of policy-based control. In JPolicy, we can prevent this by restricting the ability to change an object's policy to the service provider only. This is achieved by using a Java interface type for the client's view of a service, and constraining the semantics of the policy assignment construct so that it can only be used to change the policy of an object that is handled through a variable of class type. Clients do not know the name of the class that implements each service, they only see an interface through which to view the service. Therefore, the restriction on policy assignment to class type variables only means that client code cannot replace the policy on a service object.

Because the concept of an abstract state provides a level of separation between policies and the objects they control, we can dynamically change the policy that is associated with an individual object. Using the construct just described, a different policy can be applied to an object without making any changes to the object at all, or even informing it that a policy change has occurred. And since the policy details are encapsulated in a separate entity, service code does not need to be re-compiled when a policy change takes place.

5 Implementation Details

We have implemented a compiler for the JPolicy language described above. Since the language is designed as an extension to Java, the compiler translates the syntax extensions into pure Java. The resulting Java source can then be compiled to bytecode or native executables using standard Java compilers. See the Appendix for a full listing of the Java code generated for the example class and policy shown in Figure 3.

The translation approach is straightforward: policy specifications are converted into Java classes, service classes gain a field that links to a policy object, and modal methods are implemented by synthesizing two wrapper methods that guard access to the actual method code. Client code requires very little translation, primarily because the use of a when clause is not visible in the translated Java method signature. Indeed, clients that do not use the non-blocking `try` construct described in Section 3 can actually be compiled using a standard Java compiler, and still be successfully linked against service code written in JPolicy. This is a useful feature in an environment where it is unrealistic to expect all clients to be implemented in a non-standard language. Service hosts can develop policy-controlled services, while retaining backwards compatibility for Java clients that are unaware of the use of policies.

Any JPolicy class containing at least one modal method is augmented with an extra instance field named `currentPolicy` that provides the link to a policy object. The storage of the current set of abstract states is actually held in the policy object itself, rather than instances of the class being controlled. The reason for this implementation design is that one policy object may control several target objects, so we avoid duplication by maintaining one set of abstract states in the policy. Target objects use their link to the policy to check the current status when handling a method invocation. The policy attachment construct shown in Section 4.2 is simply translated into an assignment of a policy class instance to the `currentPolicy` field of the controlled object. The creation of new policy objects for this construct is handled by the policy itself, using a factory method.

Each modal method is compiled by erasing the when clause, renaming the method with a suffix and marking the method with `private` visibility. We then generate two wrapper methods: one with the original method name, signature and visibility, and a second with similar signature and visibility but renamed and with an extra parameter. The purpose of these wrapper methods is to implement the method access control before calling the original method body. One wrapper implements the blocking action that waits until the abstract state allows the call to proceed. The second wrapper implements the non-blocking invocation attempt discussed in Section 3, and throws an exception if the call cannot proceed. The general pattern of translation for a modal method such as:

```
public Vector query(String searchTerm) when CAN_SEARCH {
  // original method body
}
```

is to generate this set of three Java methods:

```
private Vector query_ORIGINAL(String searchTerm) {
  // Notify the currentPolicy that the method has been
  // invoked, then...
  // execute the original method body
}

public Vector query(String searchTerm) {
  // Block waiting for the object to be in the CAN_SEARCH
  // abstract state, then...
  return this.query_ORIGINAL(searchTerm);
}

public Vector query_ATTEMPT(int timeout, String searchTerm)
    throws MethodUnavailableException {
  // Wait at most timeout milliseconds for the object to be
  // in the CAN_SEARCH abstract state, then...
  if ( /* object is now in the abstract state */ )
    return this.query_ORIGINAL(searchTerm);
  else
    throw new MethodUnavailableException();
}
```

We use standard Java synchronization features to implement the waiting for abstract states — this avoids a busy wait loop by putting the calling thread to sleep until the policy object notifies the thread that the abstract state has changed. Since the set of abstract states is actually stored in the policy object, the service object uses its `currentPolicy` instance field to request the current status of an abstract state.

The first of the two generated Java wrappers has an identical signature to that of the original method written in the JPolicy language (once the when clause has been erased). This means that client code can actually be written in plain Java, by ignoring the when clauses and calling the modal methods without knowing anything about the JPolicy language. Therefore, we do not need to assume that clients will be written in JPolicy and compiled using our compiler. The policy-based control of method invocation is enforced completely on the service side of the call, rather than trusting the client not to circumvent the controls.

However, if client code is written in the JPolicy language and does make use of the invocation attempt `try` syntax described in Section 3, then a call attempt such as:

```
try Vector results=searchService.query("some terms") {
  // success code block
}
else {
  // alternative code block
}
```

is translated into the following Java code:

```
try {
  Vector results=searchService.query_ATTEMPT(0, "some terms");
  // success code block
}
catch (MethodUnavailableException e) {
  // alternative code block
}
```

The descriptions above cover the translation of client and service code that uses the extensions in JPolicy. The remaining construct is the policy definition itself — these are converted into a Java class, with fields for each local variable and parameter, and a `BitSet` to record the current status of each abstract state. The translated class implements the labelled transition system, with an instance field to record the current concrete state and a method to implement each transition.

Much of the policy implementation uses an event-driven approach: events such as methods being invoked or fields changing values cause handler methods in the policy to be called. These handlers dispatch transitions as appropriate, based on evaluating the condition expressions. Events are sent to the policies by inserting code in methods and augmenting field update code so that policy objects are notified when a method is invoked or a field value is changed.

As transitions cause the current concrete state to change, the mapping between concrete states and each abstract state is used to make any changes to the current abstract states. When an abstract state does change, target objects are notified so that any blocked method invocations can make use of this information.

Our translation strategy for modal methods involves a level of indirection, since clients call one of the wrapper methods which first performs an abstract state check before forwarding the call to the actual method implementation. In the case where the target object is in the required abstract state then the call proceeds, just as for non-modal methods, but it is desirable to minimise this overhead for invocations of modal methods. We have made some basic measurements of the overhead, by comparing execution times of an indirect call against a direct call of the actual method (bypassing the abstract state check by removing the `private` visibility). The results indicate an approximate performance reduction of 5% for the invocation of a modal method compared to the non-modal case, using the Sun Java VM v1.4.1 on Linux.

6 Related Work

The concept of Remote Execution (RE) of third-party code has appeared in various application areas, from Java applets to mobile code and agents[4, 13, 28] and active networks[1, 18, 20, 25, 30, 31]. Naturally, one of the major concerns with RE is that host owners are accepting code from potentially unknown and untrusted third parties, and therefore wish to protect their machines from malicious, greedy or poorly-written programs. There are a number of existing techniques that address this issue of program behaviour control:

Sandboxing is used to control how a program can access resources in its execution environment. The Java Security Manager architecture[14, 27] is founded on this approach, where calls to critical library methods in the API such as creating network sockets or using the filesystem are checked at run-time against a security policy of access rights. Our work is an extension of this form of control, where we allow arbitrary methods to be controlled rather than the fixed set that are hard-wired into the Java model. A further limitation of the Java Security Manager is that the security policies are "all or nothing" in the sense that a policy either allows access or denies it — a decision that remains during execution. We improve upon this by enabling more expressive dynamic policies, where method accessibility can vary over time depending upon factors such as time, user reputation, current system load or past usage of the service's methods.

Safe Languages can be designed so that undesirable program behaviour is limited by the language itself and enforced by the compiler — once the program has successfully been type-checked, then it is deemed to satisfy the definition of safety designed into the language. This language design approach has been applied to higher level languages intended for programmers[18, 25, 28, 29] and lower level assembly and bytecode languages[21, 24]. Again, the allowable access policies are usually limited by the system design or are compiled into individual programs, and hence are set once at compile-time or load-time. The Vault language[9] permits dynamic access control, rather than a fixed allow/deny permission, but the policy is still encoded into the program source — changing policies requires the program to be re-compiled.

Module Thinning is a technique for limiting the possible activities of a program by means of reducing the visibility of services it can access.

This approach has been used in Active Network systems[1, 20] to limit the access of mobile code to resources on the network node. Work on mobile Java code agents[17] protects services by narrowing the view of a service interface, which prevents client code from linking to certain methods. When a program is dynamically linked before execution, all external dependencies are matched up with the library modules that provide these facilities. A service host can use security policies to control the linking process and thereby deny access to particular services, or perhaps link against different implementations of a library depending on the required level of functionality. Here, a limitation is that the client program sees a fixed view of the available resources - once it has been loaded and linked, the visibility cannot be altered dynamically.

Code Rewriting allows hosts to impose a security policy on the behaviour of incoming programs, by modifying the binary executable[22] or bytecode itself before execution starts. One example of this approach is the SASI system[11] where policies in the form of security automata[23] determine the modification of a binary executable or Java bytecode program. This has the benefit of allowing clients to write code in any source programming language, and avoids the need to send higher level program code to the service host. The drawback of such systems is that a fixed policy is typically woven into the program code statically, when the rewriting is performed.

A common limitation in many of the techniques just described is that the host's security policy typically specifies what can and cannot be accessed, then this is built into the program or environment. Hence accessibility is determined once and cannot change during program execution. Our work is motivated by the observation that many useful security and access policies require more flexibility than a fixed allow/deny rule. In particular, method availability should be variable over time, with the policy depending on factors such as service use and the wider environment. Moreover, there should be a separation between the service code and the possibly many policies that could be applied to controlling that service.

As stated in the introduction, the specify resources of memory and CPU time are ones we consider an operating system or virtual machine concern. Nevertheless, there is research into resource-bounded programming languages[9, 19] that attempts to determine statically the memory usage and execution time bounds of programs, but this requires significant language restrictions. We consider this area of research to be an interesting complement to our work, but one motivation for our system design is that we cannot assume all client code will be written in special purpose languages.

The issue of defining and applying policies to the control of systems is well studied, such as the management policy language Ponder[8] and work on Role-based Access Control[6]. These systems focus on the higher level management concerns: the specification and analysis of policies, typically using a declarative language. Allied to this, the field of security includes much research on access control, e.g. [16], and capabilities based on authentication, i.e. who is allowed to perform certain operations. Our interest is in the lower level aspects of how policies are implemented, in the specific domain of controlling program behaviour at the method invocation level. This allows dynamic access control based on factors other than user authentication, such as time, reputation or service usage patterns.

7 Conclusions and Further Work

We have designed and implemented JPolicy: a set of language extensions for Java that enables programmers to control how the functionality in their services can be accessed by client code, using separate policy specifications that dynamically vary the availability of methods. The design described in this paper explains how service providers can use this language to specify fine-grained policies on how services can be accessed at run-time.

The primary outcome of this work is the successful application of policies to control program behaviour, in particular the novel ability of our system to enable control policies to be dynamically changed at run-time. We achieve this without altering the service code being controlled or the client code that is accessing the service. Furthermore, our design does not require the clients to be written using the extended language — client code in plain Java can still use policy-controlled services.

Our design involves a relatively simple and intuitive extension to the Java programming model, whereby programmers annotate those methods for which access control is required. The policies that control this access are specified using the familiar model of a state machine, which enables a concise representation of the required access control. As explained in Section 5 the JPolicy language has a straightforward implementation that maps the extensions into Java and incurs only a small run-time overhead to implement the access control checks.

There are a number of areas with scope for further work. Integration of our system with other research is possible in two notable areas: policies and resource accounting. It would be interesting to use the Ponder language[8] which is designed for specifying system management policies, and construct a back-end for the Ponder compiler that translates these higher-level policies into the representation that we describe in Section 4. The XenoServer platform[12] offers an environment for hosting third party client code, whereby resources such as memory and CPU time are accounted for and charged to the client. This is complementary to our work on controlling the use of services, as it deals with those aspects of program behaviour and resource usage that we do not address. We are in contact with the XenoServer team, with a view to deploying our system as an execution environment on that platform.

The JPolicy system forms part of a larger research effort into supporting third party code hosting, so we would like to extend our initial prototype system to deal with client code deployed remotely. Since our current design allows the use of clients written in Java, we can exploit remote class loading to transfer code and apply policy-based control to the services used by the hosted client. In the wider project, we are also examining to what extent static checking of access controls can be applied instead of dynamic run-time checks. Initial work on a type and effect system shows promising results, with the potential to feed into the JPolicy system design.

Appendix: Generated Code

The following simple service class and policy (from Figure 3) are used to demonstrate the form of the generated Java code. The JPolicy source code is:

```
class SearchEngine {
  Vector query(String searchTerm) when CAN_SEARCH {
    return new Vector();
  }
}

policy BoundedQueries (int bound, int interval) for SearchEngine {
  int credits = bound;
  -> some;

  transition some -> some when (query) { credits = credits-1; }

  transition some -> none when (credits <= 0);

  transition none -> some when ((TimeService.now % interval)==0)
      { credits = bound; }

  CAN_SEARCH when { some }
}
```

From this source, the following Java code is produced by our compiler:

```
class SearchEngine extends java.lang.Object {
  static SearchEngine.Policy defaultPolicy = new SearchEngine.Policy();
  public SearchEngine.Policy currentPolicy = SearchEngine.defaultPolicy;
  Vector query(String searchTerm) {
    if ( ! (this.currentPolicy.get(0)))
      synchronized (this.currentPolicy) {
        while ( ! (this.currentPolicy.get(0)))
          try { this.currentPolicy.wait(); }
          catch (InterruptedException CAUGHT_EXCEPTION) { }
      }
    else { }
    return this.query_ORIGINAL(searchTerm);
  }
  Vector query_ATTEMPT(int timeoutMillis, String searchTerm)
    throws MethodUnavailableException {
    if (timeoutMillis != 0 &&  ! (this.currentPolicy.get(0)))
```

```
      synchronized (this.currentPolicy) {
        try { this.currentPolicy.wait(timeoutMillis); }
        catch (InterruptedException CAUGHT_EXCEPTION) { }
      }
    else { }
    if ((this.currentPolicy.get(0)))
      return this.query_ORIGINAL(searchTerm);
    else throw new MethodUnavailableException();
  }
  private Vector query_ORIGINAL(String searchTerm) {
    this.currentPolicy.query_METHOD_CALLED();
    return new Vector();
  }
  static class Policy extends java.lang.Object {
    public boolean get(int state) {
      return true;
    }
    public void query_METHOD_CALLED() { }
  }
}

public class BoundedQueries extends SearchEngine.Policy
  implements TimeService.now_LISTENER {
  private int credits;
  private java.util.BitSet abstractStates = new java.util.BitSet(1);
  private int concreteState;
  private final SearchEngine TARGET;
  private final int bound;
  private final int interval;
  private synchronized void DO_TRANSITION_0() {
    {
      this.credits = this.credits - 1;
    }
    if (this.credits <= 0) {
      this.DO_TRANSITION_1();
      return ;
    } else { }
  }
  private synchronized void DO_TRANSITION_1() {
    TimeService.ADD_LISTENER_FOR_now(this);
    { }
    this.concreteState = 1;
    this.abstractStates.clear(0);
```

```
    this.notifyAll();
  }
  private synchronized void DO_TRANSITION_2() {
    TimeService.REMOVE_LISTENER_FOR_now(this);
    {
      this.credits = this.bound;
    }
    this.concreteState = 0;
    if (this.credits <= 0) {
      this.DO_TRANSITION_1();
      return ;
    } else { }
    this.abstractStates.set(0);
    this.notifyAll();
  }
  public synchronized void query_METHOD_CALLED() {
    if (this.concreteState == 0) this.DO_TRANSITION_0(); else { }
  }
  public synchronized void TimeService_UPDATED_WATCHABLE_now() {
    if (this.concreteState == 1 &&
        ((TimeService.now % this.interval) == 0))
      this.DO_TRANSITION_2(); else { }
  }
  public static BoundedQueries makePolicy(SearchEngine TARGET,
                                          int bound,
                                          int interval) {
    return new BoundedQueries(TARGET, bound, interval);
  }
  public boolean get(int state) {
    return this.abstractStates.get(state);
  }
  private BoundedQueries(SearchEngine TARGET,
                         int bound,
                         int interval) {
    this.TARGET = TARGET;
    this.bound = bound;
    this.interval = interval;
    this.credits = this.bound;
    this.concreteState = 0;
    this.abstractStates.set(0);
    if (this.credits <= 0) this.DO_TRANSITION_1(); else { }
  }
}
```

References

[1] D. S. Alexander, Paul B. Menage, W. A. Arbaugh, A. D. Keromytis, K.G. Anagnostakis, and J. M. Smith. The Price of Safety in an Active Network. *IEEE/KICS Journal of Communications and Networks (JCN)*, March 2001.

[2] Amazon. *Web Services*, 2003. Online document `http://www.amazon.com/gp/aws/landing.html`.

[3] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*, 1983.

[4] Luca Cardelli. Abstractions for mobile computation. In *Secure Internet Programming*, pages 51–94, 1999.

[5] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *International Conference on Software Engineering Research and Practice (SERP '02)*, June 2002.

[6] M. Covington, M. Moyer, and M. Ahamad. Generalized role-based access control for securing future applications. In 23rd National Information Systems Security Conference, Baltimore, MD, October 2000.

[7] Karl Crary and Stephanie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 184–198. ACM Press, January 2000.

[8] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder Policy Specification Language. *Lecture Notes in Computer Science*, 1995:18–38, January 2001.

[9] Robert DeLine and Manuel Fähndrich. Enforcing High-Level protocols in Low-Level software. In *Proceedings of PLDI-01*, volume 36(5) of *ACM SIGPLAN Notices*, pages 59–69, June 2001.

[10] Ebay. *Developers Program*, 2003. Online document `http://developer.ebay.com/DevProgram/developer/faq.asp`.

[11] Ulfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.

[12] K. A. Fraser, S. M. Hand, T. L. Harris, I. M. Leslie, and I. A. Pratt. The XenoServer computing infrastructure. Technical Report UCAM-CL-TR-552, University of Cambridge, Computer Laboratory, January 2003.

[13] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[14] Li Gong. *Java 2 Platform Security Architecture*. Sun Microsystems, 2002. Online specification `http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html`.

[15] Google. *Web APIs*, 2003. Online document `http://www.google.com/apis/`.

[16] Robert Grimm and Brian Bershad. Separating access control policy, enforcement, and functionality in extensible systems. *ACM Transactions on Computer Systems*, 19(1):36–70, February 2001.

[17] Daniel Hagimont and Leila Ismail. A protection scheme for mobile agents on Java. In *Mobile Computing and Networking*, pages 215–222, 1997.

[18] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A programming language for active networks. *ACM SIGPLAN Notices*, 34(1):86–93, 1999.

[19] Martin Hofmann. A type system for bounded space and functional in-place update–extended abstract. *Nordic Journal of Computing*, 7(4):258–289, Autumn 2000. An earlier version appeared in ESOP2000.

[20] Paul Menage. RCANE: A Resource Controlled Framework for Active Network Services. In *Proceedings of the First International Working Conference on Active Networks (IWAN '99)*, volume 1653, pages 25–36. Springer-Verlag, 1999.

[21] J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, January 2002.

[22] R. Pandey and B. Hashii. Providing fine-grained access control for mobile programs through binary editing. Technical Report TR-98-08, UC Davis, 1998.

[23] Fred B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[24] Beverly Schwartz. Introduction to spanner: Assembly language for the smart packets project. Technical report, BBN-TM-1220, September 1999. `http://www.ir.bbn.com/~bschwart/publications/TM1220.pdf`.

[25] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, R. Dennis Rockwell, and Craig Partbridge. Smart packets: applying active networks to network management. *ACM Transactions on Computer Systems*, 18(1):67–88, 2000.

[26] J. Stamos and D. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4), October 1990.

[27] Sun Microsystems. *Java Authentication and Authorization Service (JAAS) Reference Guide*, 2001. Online specification `http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRefGuide.html`.

[28] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, 1997.

[29] Dennis Volpano and Geoffrey Smith. Language issues in mobile program security. *Lecture Notes in Computer Science*, 1419:25–43, 1998.

[30] I. Wakeman, A. Jeffrey, T. Owen, and D. Pepper. Safetynet: A language-based approach to programmable networks. *Computer Networks and ISDN Systems*, 36(1):101–114, 2001.

[31] D. Wetherall, J. Guttag, and D. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols, 1998.

[32] WWW Consortium (W3C). *Web Services Activity*, 2003. Online specification documents `http://www.w3.org/2002/ws/`.