JPolicy: A Java Extension for Dynamic Access Control

Tim Owen Ian

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. insupplyi8gcon([26)998.739faci0.4]TJ-407.980T47(rather)Tj-en7602sTd(R

Services such as Google, Amazon and Ebay indicates that policy-based control of service use is already of interest. In

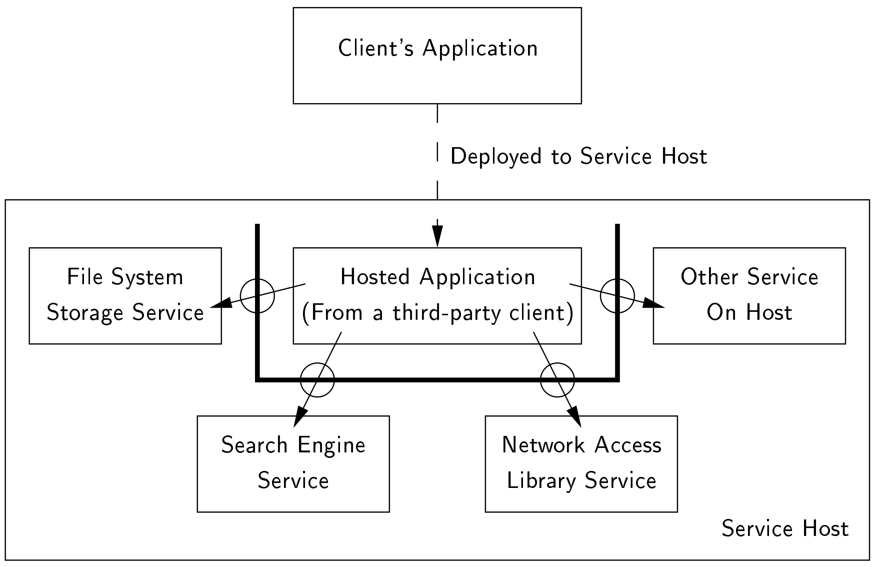
Client's Application
Deployed to Service Host
File System Storage Service
Hosted Application (From a third-party client)
Other Service On Host
Search Engine Service
Network Access Library Service
Service Host

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 classenheof

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 aof10.2Tj63980Td[

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 [illegible]

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 t8r a yes/no flag, so e.g. a *earchEngine* 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.

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 tTj51.71990Td[(d8)[illegible]oceeding:

th8(scop)-2999.9]TJ18.7203nefit0Td(A9)Tj15.96

[illegible]]TJ38.4001..12Tf1064.8

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 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 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

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

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

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

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

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/KIC Journal of Communications and Networks (JCN)*, March 2001.

[2] Amazon. *Web ervices*, 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 ymposium on Operating ystem Principles*, 1983.

[4] Luca Cardelli. Abstractions for mobile computation. In *ecure 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 oftware Engineering Research and Practice (ERP '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 IGPLAN- IGACT ymposium 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 cience*, 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 IGPLAN Notices*, pages 59–69, June 2001.

[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 ystem ecurity*, 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.

[30] I. Wakeman, A. Jeffrey, T. Owen, and D. Pepper. Safetynet: A language-based approach to programmable networks. *Computer Networks and I DN ystems*, 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 ervices Activity*, 2003. Online specification documents `http://www.w3.o`