Programming Systems for Autonomy

Alexander V. Konstantinou, Yechiam Yemini
Computer Science Department
Columbia University
New York, NY 10027, USA
{akonstan, yemini}@cs.columbia.edu

Abstract

This paper describes a new approach to programming autonomic systems. Autonomic functions are integrated into element objects at design time using a special language called JSpoon. JSpoon extends element classes with management attributes representing configuration, performance, status and fault information. The JSpoon compiler generates respective code and interfaces to instrument the data in a common Modeler repository, provided by NESTOR [28]. JSpoon programs access and manipulate management data without distinction between "agent" and "manager" roles. JSpoon further supports integration of plug-in knowledge modules that can interpret and control element operations. These knowledge modules are used to incorporate autonomic operations with elements. This design-time approach offers several substantive advantages over current alternatives. Management is integrated with the element development life-cycle. Instrumentation is compiler-generated and may be flexibly designed by element developers, while being consolidated into a unified global management data model. Knowledge modules can be seamlessly integrated with third party elements augmenting these elements with the logic for autonomic behavior.

1. Introduction

Autonomic computing has been proposed[16] as an approach to reducing the cost and complexity of managing Information Technology (IT) infrastructure. An autonomic system is one that is self-configuring, self-optimizing, self-healing and self-protecting. Such a system requires minimal administration, mostly involving policy-level management. To effect such autonomic behavior, a system must instrument its operational behavior and external interactions with other systems. It needs to represent this information in a model which admits automated interpretation and control, incorporating knowledge on how to automate management actions.

Presently, systems are constructed with ad-hoc instrumentation of Managed Information Bases (MIBs)[6] and configuration files. The information required for autonomic behavior is typically buried in design documents, operations manuals, code structures, run-time systems, and runtime environments. Management of such services involves use of proprietary management tools and protocols which have been developed to present low-level configuration and performance information to human operators. It is the responsibility of these expert operators to acquire the knowledge model needed to interpret the meaning of this information and effect configuration control. These ad-hoc forms of manageability are typically constructed a-posteriori to system design, implementation and maintenance, requiring complex adaption in order to track system evolution.

This paper describes a new approach to programming autonomic systems. Autonomic element objects are extended at design-time element with specialized management attributes. These management attributes are processed by a compiler which generates the management data model, and element object instrumentation. This approach essentially unifies the traditional management roles of element, agent, and manager under a common data model layer. Autonomic elements access the shared configuration space through a set of language and runtime services. Semantic knowledge is introduced as an extension to the data model schema in the form of plug-in modules. The plug-ins perform tasks such as constraint verification, change propagation, and fault-analysis. In this manner, the knowledge needed for autonomic behavior can be independently created and incorporated.

The next section outlines the requirements and structure of an architecture for building managed autonomic elements. Section 3 presents *JSpoon*, a language for programming autonomic element configuration and performance variable declarations and accesses. Section 4 discusses the JSpoon runtime environment services for object persistence, remote access, knowledge plug-ins, and compilation.

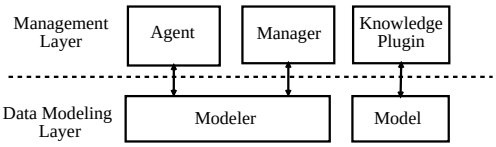

Figure 1. Autonomic Mgmt. Architecture

Section 5 introduces a plug-in module example in the form of the NESTOR propagation rule interpreter. The last two sections present previous work and conclusions.

2. Autonomic Management Architecture

In order to support autonomic behavior, an autonomic programming architecture must satisfy certain basic requirements: (1) support the representation of element configuration and performance properties used to control and monitor element behavior, (2) express relationships between different autonomic elements, (3) control access to configuration properties to assure consistent views, (4) enable autonomic elements to discover, access and control the configuration of other dependent elements, (5) provide publish-subscribe interfaces for management event notification, and (6) enable element configuration persistence and recovery.

Our approach organizes autonomic systems into a two-layer architecture as depicted in Figure 1. At the bottom layer, the *Modeler* provides a consolidated element data repository, including configuration, relationship, state and performance attributes as well as their behavior events. The Modeler also provides interfaces to access and manipulate the managed data. This enables the management knowledge layer, above, to access a unified data model, interpret its behavior and activate autonomic control functions. The knowledge layer supports representation of the data model semantics and encoding of domain-specific knowledge. This two layer architecture does not distinguish among elements, agents and managers in terms of access. These roles are implemented by programs that manipulate the data model stored in the modeler and may also incorporate instrumentation and manipulation models their own using JSpoon.

Autonomic management instrumentation variables can be assigned to one of three basic categories[9][23], with associated access patterns. *Configuration properties* control the behavior of the autonomic element and must therefore be protected in regards to concurrency and semantic content. *Performance properties* export element performance measurements and operational state, cannot be locked, and may only be set by the element owning the object. *Relationships* express dependencies to other autonomic elements.

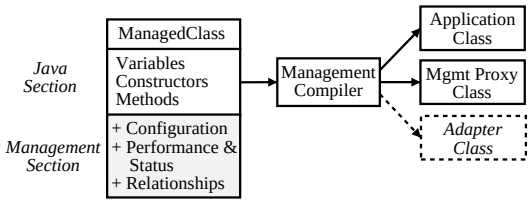

Figure 2. Managed Object Declaration

Based on these patterns, it is possible to design a restricted instrumentation declaration language that can automate the creation of object management instrumentation, and support composable configuration semantics. Figure 2 depicts a management annotated object whose class declaration is extended with an additional management section.

In the two-layered architecture the Modeler acts as the management data and schema repository. Access to the Modeler must be controlled to maintain consistency and support recovery in case of failure. The Modeler provides object life cycle management, distributed transaction coordination, and object persistence services. Autonomic elements use a common language for accessing these modeler functions. This language-based approach simplifies element implementation, and enables compile-time checks, and optimizations.

The autonomic management architecture supports extensibility of the instrumentation schema language through the use of knowledge modules. Knowledge modules enhance the instrumentation data model with semantic information. For example, a fault-management plug-in can add a language[3] for expressing propagation of problems over relationships in order to support codebook event correlation[24]. Similarly, a configuration automation system may support a language for expressing configuration constraints and change propagation rules that are maintained by a rule interpreter[28].

3. The JSpoon Language

JSpoon is a language and runtime environment for integrating data modeling layer access into Java-based managed services and managers. The JSpoon language extends Java with declarations of management class and instance variables, as well as event subscription and atomicity primitives. The JSpoon runtime environment, covered in the next section, is responsible for supporting persistence, synchronization, remote access, and knowledge plug-in services.

The set of variables declared in JSpoon forms the *management section* of the Java object, which is exported to other JSpoon programs. There are two types of JSpoon variables: *configuration*, and *performance* variables. Con-

```
public class NtpServer extends Thread {
  protected DatagramSocket sock;

  config key int port = 123;
  config boolean active = true;
  instrument counter long reqCount = 0;

  public NtpServer() throws ... {
    sock = new DatagramSocket(port);
  }

  public void run() {
    while(active) {
      sock.receive(packet);
      reqCount++;
      // process request ...
    }
  }
}
```

Figure 3. Variable Declaration Example

figuration variables are used to control program behavior, whereas performance variables instrument program status. Configuration variables may be persisted across Java Virtual Machine (VM)invocations. Access is controlled by the JSpoon runtime environment to support transaction semantics. Atomic access is expressed using the JSpoon language locking constructs. In contrast, performance variables may only be updated by the owner of the object, are not persistent, and may not be accessed transactionally.

Figure 3 provides an example of a JSpoon object class. The class implements a Simple Network Time Service (SNTP)[20]. SNTP is a UDP-based protocol for querying time servers over an Internet Protocol network. As illustrated in this code fragment, the SNTP service implementation is written in standard Java extended with JSpoon declarations. The class encapsulates a regular Java instance variable called `sock` which maintains the server's UDP binding. In addition, three management instance variables are declared in the extended syntax of JSpoon: `port` and `active` (configuration), as well as `reqCount` (performance).

The `port` instance variable stores the NTP service UDP port number. It is a configuration variable, as identified by the JSpoon modifier `config`, of primitive type `int` initialized to the default SNTP UDP port 123. The `key` variable states that the value must be unique in all instances of the class within the Java VM. The second configuration instance variable, `active`, controls termination of the server process. The `reqCount` performance variable, as identified by the `instrument` modifier, counts the number of time queries received.

Modifier	To	Description
`computed`	i	Evaluate on-demand; not stored
`counter`	i	Monotonically increasing value
`final`	c	Assign-once (Java semantics)
`key`	c	Unique object identifier
`static`	c, i	Property of class; not instance
`transient`	c	Non-persistent variable

Figure 4. JSpoon Declaration Modifiers

From the service's perspective, the declaration and usage of JSpoon management variables is similar to that of regular Java variables. There are two main differences: (1) management variables are exported through the JSpoon runtime environment to external management processes, and (2) the types of operations permitted on the management variables may be restricted based on the JSpoon modifiers. For example, an external manager may change the value of the `active` configuration property, thereby effecting termination of the service. Similarly, the operations allowed on the `reqCount` *counter* performance variable are restricted to monotonically increasing value updates.

JSpoon modifiers may only be applied to class and instance variable declarations. The JSpoon compiler will generate a syntax error if JSpoon modifiers are used in other contexts, such as in method argument or variable declarations.

3.1. JSpoon Modifiers

JSpoon configuration and performance variable declarations may be specialized with the modifiers listed in Figure 4.

Computed variables have their value evaluated on-demand using a Java expression. For example, the computed free memory performance variable shown below is associated with a Java expression for obtaining the free memory in the Java VM.

```
instrument computed long freeMemory =
  Runtime.getRuntime().freeMemory();
```

Computed variable expressions are bound to the scope of their declaration.

Numeric performance JSpoon variables marked with `counter` are restricted to a monotonically increasing value. The counter marker may be applied to the primitive number types, and objects implementing the `java.lang.Comparable` interface. The only operations permitted on primitive numeric types are `++` (increment by one), and `+=` (increment by value, where value ≥ 0). Numeric object types are immutable, therefore the

only allowed operation is assignment (=) where the assigned value compares-to greater, or equal to the previous value.

Configuration variables may be marked as `final` to specify that they may only be assigned once and cannot not be modified by the program or an external entity. It is possible to change persistent final variables in between program invocations. Typically, variables will be declared final if the program is not able to adjust its behavior after startup. For example, the user ID under which a Java process should be executed may be declared as a final variable.

Certain configuration variables may uniquely identify an object instance. For example, in the SNTP UDP server of figure 3, the port variable is defined as a `key` variable since only a single instance of `NtpServer` can bind to the same unicast UDP port. When multiple configuration variables are declared as keys their combination uniquely identifies the object.

The `static` JSpoon modifier associates its configuration or performance variable with the enclosing class, not an instance as is the default.

Configuration variables may be marked `transient` to indicate that the JSpoon runtime should not maintain their values across Java VM invocations, or when the object is serialized. Persistence requires additional changes in the way Java programs manage the life-cycle of their objects, and is covered in the next section. By default, all configuration variables are persistent. Instrument variables are always transient, and can only be persisted through explicit assignment to a configuration variable.

Unlike Java class and instance variables, JSpoon variables are not associated with an access modifier such as `public`, `protected`, or `private`. All JSpoon variables are publicly accessible via the JSpoon runtime. A role-based security policy may be configured at the modeler-layer, or through a security knowledge plug-in, and will not be presented in this paper.

3.2. JSpoon Types

JSpoon variables are strongly typed in the Java type system. All Java primitive types are allowed in JSpoon variable declarations. Java object types may be used if, and only if, they represent immutable objects, and are serializable. Since here is no Java marker interface identifying objects as immutable, the JSpoon compiler includes a list of immutable standard Java library object classes. User-defined classes employed in JSpoon variable declarations must implement the `jspoon.Immutable` marker interface and all their non-JSpoon variables must be declared as `final`.

JSpoon extends the Java type declaration system with support for enumeration types. The example in figure 5 shows an enumeration declaration of the type `Status` with

```
enum Status { stopped, running };
instrument Status status;

public void method() {
  status = Status.stopped;
  // ...
  if (status == Status.running) ...
}
```

Figure 5. Enumeration Type Example

two enumerated values. An enumeration type is *conceptually* mapped into a Java inner class of the same name, with immutable instances for each enumeration value. These instances are available as static final variables of the enumeration class. As shown in the example, an enumeration variable may be assigned or compared for reference equality with the static enumeration instances.

JSpoon relationship declarations are used to establish an association between two Java objects. Unlike simple references, relationships can be navigated in both directions. Each end-point in the binary association is associated with a role (variable identifier), and multiplicity (to-one, to-many-set, or to-many-sequence).

A sample relationship declaration is shown in figure 6. Instances of the class `HttpServer` are associated with a set of `HttpThread` objects through the `threads` variable role. The same relationship must also be declared in the `HttpThread` class, with the role names reversed. In this case, the `serves` variable role is declared with a to-one multiplicity. Variables identifying to-one relationship roles may be used as normal Java object reference variable. To-many relationship variables support accessor methods for retrieving, adding, removing and setting their membership.

3.3. Events

JSpoon programs may subscribe to receive notification of modeler events. The modeler generates six basic types of events: class `load`, class `unload`, `create` object, `delete` object, attribute `get`, and attribute `set`.

JSpoon supports event subscriptions as a language construct. The `subscribe` keyword takes an event condition expression as its first argument, executing an associated statement when a matching event is found. An optional event declarator may be provided if the contents of the matching event need to be processed within the statement. Subscriptions return a JSpoon lease object used to manage their life-cycle.

Figure 7 lists two event subscription examples. The first subscription matches assignment of `NtpServer` port instance variables to non-standard SNTP ports. The event

```
public class HttpServer
  extends Thread {
  relationshipset threads, HttpThread
                  serves;
  public void run() {
    while(true) {
      Socket s = sock.accept();
      threads.add(new HttpThread(s));
    }
  }
}
public class HttpThread
  extends Thread {
  relationship serves, HttpServer,
               threads;
  public void run() {
     // Process HTTP request ...
     serves = null;
  }
}
```

Figure 6. Relationship Type Example

```
JSpoonLease lease =
  subscribe evt:
   (NtpServer.port != 123) {
     evt.object.active = false;
}

NtpServer mySrv = ...;
lease = subscribe
  ( (mySrv.port == 123) &&
    (! mySrv.active) ) {
   mySrv.active = true
}
```

Figure 7. Event Subscription Example

```
atomic(lock-timeout) {
  z = x + y;
}
```

Figure 8. Atomic Action Example

handler terminates the SNTP service by setting its `active` attribute to false, by retrieving the object reference from the event object. The JSpoon compiler employs type-inference to determine that the enclosed object will be of type `NtpServer`, thereby saving the need for an explicit cast. JSpoon programs may also subscribe for events on specific object instances, as is shown in the second example.

Basic events are associated with a type (load, unload, ...) which may be queried using the `type` attribute. Other event attributes include `object` (object), `transactionID` (source), and `value`.

The subscribe construct defines a generic Event-Condition-Action (ECA) mechanism, raising the issues of termination and confluence[26, 1]. Using established ECA rule management techniques, the JSpoon compiler analyzes event subscription expressions and generates a dependency graph. This graph is used by the JSpoon environment to identify and monitor possible cycles or order-sensitive rule evaluations. Future work will investigate a limited expression and statement language to enable full compile-time cycle and ambiguity detection.

3.4. Atomic Operations

JSpoon expresses concurrency control similarly to the way that Java handles thread synchronization on object methods and variables. JSpoon programs requiring atomic access to multiple configuration attributes must perform these accesses within an `atomic` block. The atomic block provides JSpoon programs with atomicity, consistency, isolation, and durability (for persistent properties) in accessing management attributes.

An atomic block example is listed in Figure 8. The JSpoon program encloses its access to the configuration properties `x`,`y`, and `z` in a `atomic` block to assure consistent change. The JSpoon runtime will generate the required read lock requests for `x` and `y`, as well as a write lock request for `z`.

The obtained locks are owned by the thread of execution. Nested transactions are supported through syntactically nested blocks, or through invocation of methods containing atomic blocks. The effected changes are committed at the end of the block, and the acquired locks are released, unless this is a nested transaction.

Lock requests may fail due to a communications failure, detection of a deadlock, or lock acquisition timeout. These errors are signaled in the form of exceptions. If an exception is thrown in an atomic block, then the values of all *management* configuration attributes will be restored to their previous values. The syntax of the atomic block supports the optional specification of a lock acquisition timeout.

Atomic blocks may also be used to group the generation of performance variable update events. Performance variable updates performed within an atomic block do not generate modeler update events until the end of the block. This mechanism may be used to provide consistent views of performance variables and to reduce the overhead of synchronization between the service thread and the JSpoon event

monitoring thread.

4. The JSpoon Runtime Environment

The JSpoon runtime environment is responsible for providing object life-cycle services, and exporting management information to remote management processes. This section will outline the managed object persistence, synchronization, remote access and event generation services.

4.1. Persistence

The JSpoon runtime environment supports persistence of object configuration variables. In order to enable persistence, classes must implement the `Persistent` marker interface. Persistent objects must be assigned a unique *object identifier* (OID) in order to support retrieval when instantiated within the Java program. Objects which have key variables, all of which are final and persistent, can be automatically assigned a unique OID. In all other cases, the programmer is responsible for assigning the unique OID.

The JSpoon compiler modifies the signature of persistent class constructors to include an additional argument of type `jspoon.JSpoonOID` and to throw the `PersistenceException` exception. The constructors are also modified to include the necessary hooks for retrieving previously persistent variable values. It should be noted that the persistent values will override any default variable values specified in the declaration section. At construction time, the programmer must provide the additional OID parameter to establish the unique identity of the object.

Figure 9 shows the `NtpServer` class after it has been marked to support persistence, and a simple simple `TimeDaemon` application which uses a single instance of the `NtpServer` class. At construction time, the additional OID argument must be provided to establish the object's identity. A static identifier can used is this case because the application is limited to creating a single instance of the persistent class.

Every JSpoon process must be assigned a unique *service identity* (SID). Service IDs are required in order to prevent multiple instances of a program from owning the same persistence repository data. Persistence repositories must support locking to prevent concurrent binding of multiple JSpoon programs using the same SID. A lease-based mechanism is employed to support releasing of resources following a service failure.

A fully qualified object ID contains the service ID as well as the location of its persistent repository. It is possible to set the default service ID and repository location of a JSpoon process in order to simplify construction of OID objects. Services can have multiple SIDs and connect to multiple persistence repositories.

```
public class NtpServer extends Thread
  implements jspoon.Persistent {
  // ...
}
public class TimeDaemon {
  public static void main(...) {
     JSpoonOID oid = new JSpoonOID
       ("jspoon:NtpServer#Singleton");

     NtpServer srv =
       new NtpServer(oid);
  }
}
```

Figure 9. Persistence Example

An object ID may be represented as a URI[4] of the form:

`jspoon://`*userinfo*`@`*host*`:`*port*`/`*serviceID*`#`*OID*

For persistent objects with final key attributes the OID may be expressed as:

className`?`*key1*`=`*value1*`,`*key2*`=`*value2* ...

The Java language supports objects serialization as a mechanism for storing and transmitting object state. The JSpoon compiler modifies the `writeObject` and `readObject` methods of serializable objects to include marshaling and unmarshaling of variables in the management section, and code for binding the deserialized object into the local JSpoon runtime environment.

4.2. Synchronizing States

JSpoon class configuration variables may be changed by remote managers. A change in the configuration variable will be reflected in program behavior at the next point at which this variable is read. For example, the `active` variable in the program of figure 3 is consulted every time a request is serviced. If the server is idle, then it will remain blocked on the socket `receive()` method, and will not terminate until a datagram has been received and processed. In order to increase responsiveness, the program may set a socket timeout to establish an upper bound on its delay to respond to a change in the `active` configuration.

The JSpoon runtime environment monitors application access to configuration variables, and can detect if the program has failed to read a changed configuration value after a certain period. Based on its configuration, the environment can elect to restart the server. Since the Java VM does not support external thread termination (kill), it is recommended that JSpoon program threads consult the static `jspoon.JSpoonThread.terminate` boolean variable, and cleanly terminate execution when it has been

```
public class NtpMonitor {
  public static void main(...)
    throws ... {

    JSpoonOID[] oids =
      jspoon.JSpoonNaming.list
      ("jspoon://localhost/NtpServer");

    NtpServer server = (NtpServer)
      jspoon.JSpoonNaming.lookup
      (oids[0]);

    while(true) {
      System.out.println
        (server.reqCount);
      Thread.sleep(5000);
    }
  }
}
```

Figure 10. Manager Example

set to true.

There are cases in which configuration variables are consulted only at program startup. Examples include network service port numbers, persistent storage directories, and others. To support dynamic reconfiguration, programs should embed explicit synchronization code. For example, main loop of the example from figure 3 can be rewritten as:

```
while(active) {
  sock.receive(packet);
  if (port != sock.getLocalPort()) {
    // close & reopen socket
  }
}
```

4.3. Remote Access

JSpoon programs requiring access to management attributes of remote objects must first obtain a local copy (view). This is performed by using the static methods of the `jspoon.JSpoonNaming` class. The `list` method supports query of objects based on a query URI. In the example of figure 10 the manager requests the OIDs of all instances of the `NtpServer` class. The example code assumes that at least one OID is returned, and then invokes the lookup method to obtain a local view of the JSpoon object. Subsequently, the JSpoon object can be accessed in the manner illustrated in previous examples.

If the listing and lookup methods are enclosed in an atomic block, the JSpoon runtime will obtain appropriate locks to assure that the results remain stable. In the listed example, had the listing and lookup methods been enclosed in an atomic block, the runtime would lock `NtpServer` object creation and remove effects.

4.4. Managed Java Library Objects

The JSpoon environment provides managed versions of standard Java library objects. JSpoon managed objects follow a naming convention of appending the jspoon prefix to the full class name of the instrumented class. In this manner, the `jspoon.net.JSpoonDatagramSocket` class supports configuration and performance instrumentation of UDP datagram sockets. Use of managed objects can greatly increase the management flexibility of Java programs. For example, use of the managed socket class in the `NtpServer` would enable managers to configure socket options such as the traffic class.

4.5. Introspection & Knowledge Plugins

Knowledge plug-ins extend the capabilities of the basic JSpoon schema. For example, a simple constraint knowledge plug-in may add support for type range restrictions. A JSpoon autonomic element may use this plug-in to further restrict the values of port number configuration attribute to the range $[1024..65535]$.

JSpoon knowledge plug-ins are enabled in the form of schema extensions to JSpoon classes. The schema of a JSpoon class is represented with a meta-object that is also a persistent JSpoon class of type `JSpoonClass`. `JSpoonClass` defines a to-sequence relationship to instances of the JSpoon class `JSpoonSchemaExtension`. Schema extensions consist of a knowledge plugin URI and an opaque object. A schema extension may be bound to multiple instances of `JSpoonClass`. A UML class diagram showing parts of the JSpoon meta-schema classes is shown figure 11.

When the JSpoon runtime environment loads a JSpoon class, it attempts to retrieve the class meta-object from persistent storage. The schema extension relationship of the meta-object is queried and the runtime environment attempts to download any knowledge module proxy plug-ins that are not already installed based on the URI. This capability depends on the Java cross-platform and security features. Runtime changes to the meta-object schema extension relationship can also trigger the loading or unloading of knowledge plug-ins.

At load-time, plug-ins use the JSpoon runtime event subscription API to request notification of object management events. Plug-in subscriptions provide synchronous notification of pending changes, enabling the knowledge module to abort invalid changes. Plug-ins receive two additional

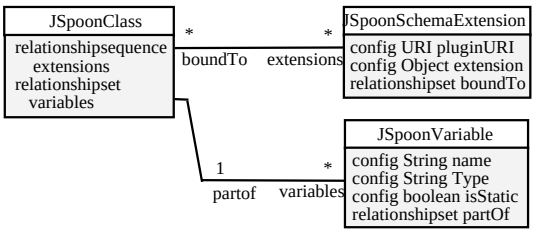

Figure 11. JSpoon Meta Schema

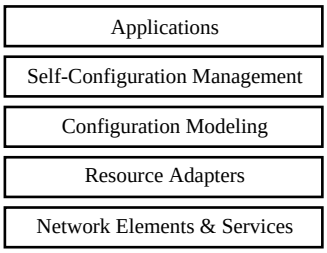

Figure 12. NESTOR Architecture

events triggered by the creation and closing of a transaction, or nested transaction. These events enable evaluation of postconditions.

The knowledge schema extension mechanism is very powerful, but may introduce cyclical computations. A cyclical computation may be triggered by the interaction of two knowledge plug-in modules that propagate changes. Because the periodicity of the cycle may be very large, it may be impossible to identify in a reasonable time. Therefore, in absence of additional information on propagation paths, the JSpoon runtime system defines a maximum knowledge module iteration count.

4.6. JSpoon Compilation

JSpoon-enhanced classes are compiled into Java VM bytecode class files. The language was designed to support JSpoon-to-Java source-to-source compilation as an intermediate step. The JSpoon compiler generates two identically named Java classes for every JSpoon class. The element class contains both the management and non-management sections. The management class contains only the management variables and is used by remote elements. Both classes depend on the JSpoon runtime for view maintenance.

Management variable accesses by JSpoon programs are transformed into corresponding accessor methods. Left-hand-side (assignment) updates and right-hand-side reads are replaced by *set* and *get* method invocations. The actual management attributes are declared as private transient Java instance variables with mangled names. Configuration variable accessors invoke JSpoon runtime methods for locking and logging. Performance variable accessors invoke JSpoon event generation methods. A static block is also created to notify the runtime of a class loading event.

Atomic blocks are compiled into JSpoon runtime environment method invocations to create a transaction (potentially nested), commit the transaction at the end of the block, or abort if an exception has been thrown. The JSpoon compiler attempts to optimize lock acquisition through static code analysis to batch lock requests, and minimize lock upgrades. Batched lock requests are sorted by object ID to reduce the likelihood of deadlock. If the compiler can determine that an atomic block does not include access to configuration attributes, then no transaction is generated, only the events are batched.

The JSpoon compiler may optionally generate proxy objects for exporting the management schema to existing Java and XML-based persistence and management APIs and protocols. For example, the compiler may generate MBean objects conforming to the JMX[25] architecture to support interaction with JMX managers. The compiler may also generate an XML schema, and protocol proxy for XML management standards such as XMLCONF[10].

5. NESTOR Autonomic Knowledge Plug-in

NESTOR[28] is an architecture for network management automation developed at Columbia University. NESTOR supports automated management of existing network elements and services through a layer of resource adapters. NESTOR has been demonstrated in the automation of network configuration[28], network security [18], multimedia QoS, and Active Network applications[17]. The NESTOR prototype implementation has been released to industry research labs and has been applied as a platform for development of a distributed firewall[5].

The overall NESTOR architecture is depicted in Figure 12. The element and resource adapter layers correspond to the autonomic element program. NESTOR adapters are used to interface with non-JSpoon instrumented network service. The NESTOR configuration modeling layer is an implementation of the autonomic modeling layer, and provides a distributed object repository that supports distributed transactions, leasing and event notification. The NESTOR self-configuration management layer corresponds to the autonomic knowledge layer. The NESTOR configuration constraint and change propagation managers can be encapsulated as a JSpoon autonomic knowledge module plug-in.

A sample NESTOR propagation rule is shown in Figure 13. The rule states that the Maximum Transfer Unit of a UDP-based web-radio streaming service should be set to

```
com::acme::WebRadio->allInstances
->forAll(a : WebApp | a.mtu =
  servedBy.networkInterfaces
  ->select(i | not i.isLoopback)
  ->collect(i | i.mtu)->min())
```

Figure 13. NESTOR OCL Propagation Rule

the minimum MTU of the network interfaces of its execution environment.

6. Previous Work

The advantages of representing network element and service configuration using an object-relationship model have been established by several research projects and management standards [27, 9, 8, 15]. The categorization of management variables has been previously discussed in network and services management architecture research [12, 21, 23, 19]. This paper contributes to this research by specifying a language and mechanism for declaring restricted management variables as part of the service object class, and compiling them into a management object following matching accessor design patterns.

Previous work in automated service configuration[28, 17, 22, 14], fault root-cause analysis[24], and self-healing[11, 13] has depended on existing service instrumentation. Although these approaches have demonstrated practical automation capabilities, they require complex coordinated design and evolution of management systems and managed elements. For example, the management system vendor must be able to access the instrumentation of the managed element, construct a data model and a knowledge model to interpret its operational meaning and control its configuration in a manner consistent with operational procedures conceived by the element vendor. This information is often not readily available and may change as the managed element continues to evolve. Therefore, there is a need to simplify the process through which element designers export instrumentation and modeling information concerning the element's operations management, while enabling multiple knowledge models to be seamlessly integrated and applied to the element to provide autonomic operations. This work provides an opportunity to greatly improve the efficacy of these results through a rich instrumentation layer, and an generic architecture for providing automation knowledge modules.

JSpoon complements existing Java-based management architectures such as JMX[25] and JavaBeans. Autonomic element programmer's benefit from the close integration between element and management code, while benefiting from the compiler-generated management system exports. Related research includes a system for automated management of EJB component interfaces[2]. The work presented proposes a more general instrumentation automation approach, at the cost of requiring programmer cooperation. Automated SNMP-based management instrumentation has been previously demonstrated in non-imperative languages such as the NetScript[7] data-flow language. This paper presents an architecture for automating instrumentation of Java, a general-purpose imperative language.

7. Conclusion

The JSpoon approach to autonomic management offers several substantive advantages over current alternatives. First, all information needed for management is consolidated with element design and can be maintained through its life-cycle evolution. Second, instrumentation, data models, knowledge model and their bindings can be generated and managed through compiler support and static-time validation. Third, knowledge modules can be seamlessly incorporated with elements by vendors of autonomic computing products, independently of the element vendors enabling synergistic evolution of products. Fourth, instrumentation, data and knowledge models can be unified across multiple elements greatly simplifying the task of providing autonomic self-managing capabilities of large composite systems.

Acknowledgments

Research sponsored in part by DARPA contract DABT63-96-C-0088.

References

[1] A. Aiken, J. M. Hellerstein, and J. Widom. Static analysis techniques for predicting the behavior of active database rules. *ACM Trans. Database Syst.*, 20(1), 1995.

[2] N. Almasri and S. Frnot. Dynamic instrumentation for the management of EJB-based applications. In *Systmes composants adaptables et extensibles*, Grenoble, France, 2002.

[3] S. M. Arts. *MODEL Language Reference Manual*, 1996.

[4] T. Berners-Lee, R. Fielding, U. Irvine, and L. Masinter. Uniform Resource Iidentifiers (URI): Generic syntax. Technical Report RFC 2396, IETF, 1988.

[5] J. Burns, P. Gurung, D. Martin, S. Rajagopalan, P. Rao, D. Rosenbluth, and A. Surendran. Management of network security policy by self-securing networks. In *DARPA Information Survivability Conference and Exposition (DISCEX II)*, Anaheim, California, 2001.

[6] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). Technical Report RFC 1067, IETF, 1988.

[7] S. DaSilva. *Netscript: A Language System for Active Networks*. PhD thesis, Columbia University, 2002.

[8] Distributed Management Task Force (DMTF). Common Information Model (CIM) specification. Technical Report Version 2.2, DMTF, June 1999.

[9] A. Dupuy, S. Sengupta, O. Wolfson, and Y. Yemini. Netmate : A network management environment. *IEEE Network Magazine (special issue on network operations and management)*, 1991.

[10] R. Enns. XMLCONF Configuration Protocol. Technical Report draft-enns-xmlconf-spec-00, IETF, 2003.

[11] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, pages 27–32, Charleston, S.C., 2002.

[12] M. Garschhammer, R. Hauck, H.-G. Hegering, B. Kempter, M. Langer, M. Nerb, I. Radisic, and H. Rlle. Towards generic service management concepts - a service model based approach. In *7th IFIP/IEEE Symposium on Integrated Management (IM 2001)*, pages 719–732, Seattle, WA, USA, 2001.

[13] I. Georgiadis, J. Magee, and J. Kramer. Self-organizing software architectures for distributed systems. In *ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, Charleston, S.C., 2002.

[14] G. Goldszmidt and Y. Yemini. Distributed management by delegation. In *The 15th International Conference on Distributed Computing Systems*, Vancouver, British Columbia, 1995. IEEE Computer Society.

[15] S. Judd and J. Strassner. Directory-Enabled Networks : Information model and base schema. Technical Report Version 2.0.2-2, DEN Ad Hoc Working Group, 1998.

[16] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1), 2003.

[17] A. Konstantinou, Y. Yemini, and D. Florissi. Towards self-configuring networks. In *DARPA Active Networks Conference and Exposition (DANCE)*. IEEE Press, 2002.

[18] A. V. Konstantinou, Y. Yemini, S. Bhatt, and S. Rajagopalan. Managing security in dynamic networks. In *13th USENIX Systems Administration Conference (LISA'99)*, Seattle, WA, USA, 1999.

[19] M. Mansouri-Samani and M. Sloman. Monitoring distributed systems. *IEEE Network*, 7(6):20–30, 1993.

[20] D. Mills. Simple Network Time Protocol (sntp). Technical Report RFC 2030, IETF, 1996.

[21] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbinger, G. Johnson, M. Modvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[22] L. Ricciulli, P. Porras, P. Lincoln, P. Kakkar, and S. Dawson. An adaptable network control and reporting system (ancors). In *DARPA Active Networks Conference and Exposition (DANCE)*, California, USA, 2002.

[23] A. Schade, P. Trommler, and M. Kaiserswerth. Object instrumentation for distributed applications management. In *IFIP/IEEE International Conference on Distributed Platforms*, pages 173–185, 1996.

[24] SMARTS. *InCharge*, 1997.

[25] Sun Microsystems. Java Management eXtensions instrumentation and agent specification (v.1.2). Technical report, 2002.

[26] L. Van Der Voort and A. Siebes. Termination and confluence of rule execution. In *2nd International Conference on Information and Knowledge Management (CIKM 93)*, Washington, DC, 1993.

[27] Y. Yemini, A. Dupuy, S. Kliger, and S. Yemini. Semantic modeling of managed information. In *Second IEEE Workshop on Network Management and Control*, Tarrytown, NY, 1993.

[28] Y. Yemini, A. Konstantinou, and D. Florissi. NESTOR: An architecture for NEtwork Self-managemenT and ORganization. *IEEE JSAC*, 18(5), 2000.