A Script MIB based Java Policy Management System: Design and Implementation Report

Frank Strauß

Computer Science Department
Technical University Braunschweig
Bültenweg 74/75
38106 Braunschweig, Germany
strauss@ibr.cs.tu-bs.de

September 2001

Abstract

Tasks 6.1 – 6.3 of the joint Jasmin Project between the Technical University of Braunschweig and NEC C&C Research Laboratories scheduled for the time from January to August 2001 has been concerned with the design and implementation of a policy management system based on the Jasmin Script MIB implementation.

While Deliverable 6.1 outlined the requirements and a general architecture in May 2001, this report documents the architecture and usage of the prototyped Java class packages for developping policy scripts and their adaption to DiffServ router configuration. An example is presented and final conlusions denote the deficiencies and probable future work.

This report is heavily based on the concept report “Concept of a Script MIB based Policy Management System” [1] and the draft paper “Using the Script MIB for Policy-based Configuration Management” [2] submitted to NOMS-2002.

1 Introduction

Policy based management is not a new area of research and engineering. It has been addressed in the past from the research point of view by several research groups, workshops, and conferences [3, 4] and from the vendors’ point of view by some specific engineering and implementation efforts, e.g. [5].

The goals of this project are somewhere in between: Based on some research knowledge from the past few years and based on the knowledge from the work that has been done so far in the IETF Policy Framework (POLICY), Configuration Management with SNMP (SNMPCONF), and Resource Allocation Protocol (RAP) working groups, a policy management system was outlined and implemented in a prototype fashion. Its basis is the IETF Script MIB [6] infrastructure that has been implemented in previous phases of this project [7]. The Script MIB functionality is used to transfer and control the execution of policy 'scripts'. The execution of a policy script is realized by the existing Java runtime engine for the Jasmin Script MIB agent [8] but with some additional policy supporting class libraries. This project focuses on the design and implementation of these class libraries and their implementation and evaluation. Note that in the beginning it was planned to develop a new policy definition language, which turned out to be out of scope for the timeline of the project.

1.1 Domain: DiffServ

The targeted policy domain of this project is the configuration of DiffServ [9] routers. However, the general architecture will not be limited to DiffServ, but the examples and the elements that have been modeled and implemented are concerned with the configuration of DiffServ TCB elements like classifiers, meters, markers, queues, and schedulers.
Interfaces for DiffServ configuration are being developed and implemented by different groups. The IETF DIFFSERV working group [10] is developing an SMI MIB [11] and an SPPI PIB [12] for managing DiffServ routers via SNMP and COPS-PR. We decided to use the conceptional view of the DIFFSERV-MIB to develop our DiffServ class model.
Two free DiffServ implementations for the Linux kernel have been regarded valuable for evaluating the implementations being developed during this project. The first one has been integrated with the official Linux-2.4 kernel release. The other one has been implemented in a joint project between NEC C&C Network Product Development Laboratories and the University of Bern. We decided to use the first one, since it's available with unmodified Linux kernels and two projects are underway to implement the DIFFSERV-MIB for this implementation, so that later on a MIB based driver can be developed.

2 Design

This section first presents a number of requirements for the policy management system. These requirements had to be fulfilled during the desing and implementation process. Then Section 2.2 describes the architecture of the policy management system.

2.1 Requirements

This section describes a number of requirements that had to be achieved for the policy management system, although they might be common to all approaches in policy based management.

1. A policy rule condition must allow read access to the attributes of zero, one or multiple elements. This has to be done in a way, so that the according action can unambiguously reference those elements that matched the condition. Similarly, the attributes of the event that triggered the rule must be accessible. This means, we need a concept of free variables in the event and condition definitions that are bound by the runtime system to element instances when passed to the condition and action. These variables can be declared implicitly in the events and conditions or explicitly.

2. There must be a construct to specify the value space in which the free variables of conditions are evaluated. This may span all instances of elements within a certain table or even all instances of a class among a number of managed agents.

3. A class that models a certain element must support a number of accessor methods that allow a policy author to retrieve and manipulate an element in a comfortable way. E.g., counter retrieval functions should implicitly support rate computations, and SNMP RowStatus handling should be hidden by methods to construct and destruct element instances. This is not a requirement for the policy engine architecture itself, but for the design of domain specific elements.

4. So far, at least three types of time events are required: Periodic events that trigger continuously at a given period. Calendar events that trigger periodically at points in time specified by calendar-type attributes (month, day, weekday, hour, minute), and one-shot events that trigger exactly once at a point in time specified by calendar-type attributes. These three types are motivated by the Schedule MIB [13].

5. Another type of event is based on the reception of external notifications like SNMP traps/informs or COPS-PR state reports. These notifications should be mappable to domain specific events. Details of the initiating notifications should be accessible through accessor methods of the events.

6. The policy runtime engine must support a mechanism to report errors and optional tracing/debugging information so that users can monitor the policy engine and the authors of policies can test and debug their policy code.

7. The access to elements in conditions and actions may fail. The policy runtime engine must be able to handle these situations in a way that accordingly written

policy code can catch the error conditions and bring the affected element to a determined state.

8. It must be possible to store and execute multiple policies independently. Their code must not share any name space. However, avoiding side effects by multiple policies or policy rules acting on common elements is the responsibility of the policy author(s).

9. A security mechanism is required to differentiate which users have access to which operations on which policies. This is regarded as a very sensible aspect. Building on an existing security mechanism could be helpful.

10. Ideally, the policy programming interfaces of domain specific elements are independent of the underlying management interfaces. This means a policy acting on an element does not have to care about the question whether an underlying device is managed via SNMP, COPS-PR, a command line interface or an API.

11. In theory, policies declare behaviors of elements dependent on conditions. However, a programmatic policy system has to work in a deterministic sequential fashion; especially complicated actions must contain a bunch of code instead of just the goal that is about to be reached. The notation of policies should retain the declarative fashion of policies as much as possible.

12. It should be possible to avoid redundancy in a way that policies or policy groups sharing rules, and rules sharing conditions or actions can be built by referring common code instead of copying code fragments. This allows to increase reusability and to avoid some errors.

13. A communication mechanism between active policies based on shared memory or messages would also help to reduce redundancy. For example, one policy can determine a responsible person to which a number of other policies send reports in case of errors.

14. It could be useful to pass arguments to policies when they are activated. Although all parameters of a policy behavior should be defined within the policy code, arguments could be useful to turn debugging on and off or to trace a policy in read-only mode.

2.2 Architecture

In contrast to other research and development approaches, this project builds on the IETF Script MIB architecture as an infrastructure for transferring policies and managing their enforcement. The Script MIB has a number of functions that are required by policy systems in general. Furthermore, some of the requirements listed in Section 2.1 can be met at low costs in the Script MIB context:

- The Script MIB architecture supports pushing and pulling mechanisms for transferring scripts from a SNMP command generator or a script repository to Script MIB agents. Controlling the execution of scripts is also supported, including starting, suspending and resuming, terminating, controlling maximum run times, passing arguments to scripts, etc.
- SNMP security based on SNMPv3 and the user based security model and on the view based access control model is fully applicable to the Script MIB. It is reasonable to build on SNMP security and the Script MIB to achieve a homogeneous security setup.
- Logging and tracing of scripts is a general functionality that is useful for scripts as well as for policies. Although, the current Script MIB does not support access to logging data, the Jasmin implementation supports logging via the SMX interface between runtime systems and the Script MIB core agent.

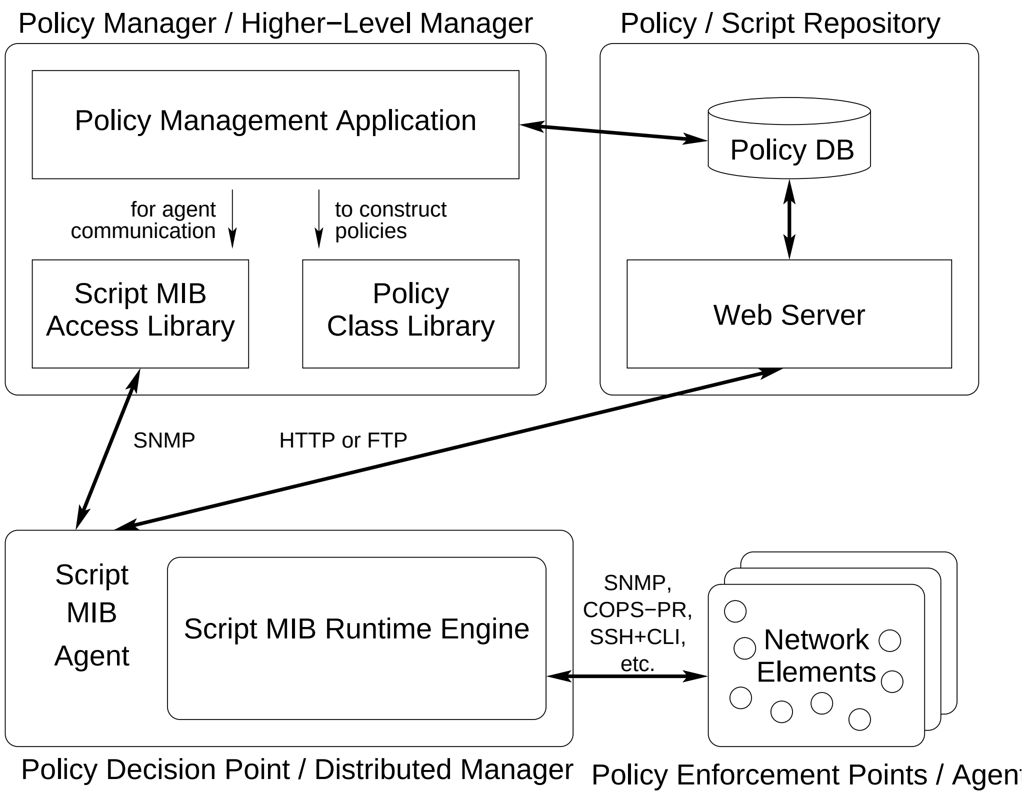
Policy Manager / Higher−Level Manager
Policy / Script Repository
Policy Management Application
Policy DB
for agent communication
to construct policies
Script MIB Access Library
Policy Class Library
Web Server
SNMP
HTTP or FTP
Script MIB Agent
Script MIB Runtime Engine
SNMP, COPS−PR, SSH+CLI, etc.
Network Elements
Policy Decision Point / Distributed Manager
Policy Enforcement Points / Agen

Figure 1: Architecture of a policy management system based on the JASMIN Java runtime engine.

The general architecture of our Script MIB based policy system approach is shown in Figure 1.
Policies are represented as scripts written in a language supported by a runtime engine. What turns a script into policy code is the use of specific class libraries that are usually registered with the runtime engine through the Script MIB language extension facility. These general scenario of policy scripts which use these libraries is shown in Figure 2.

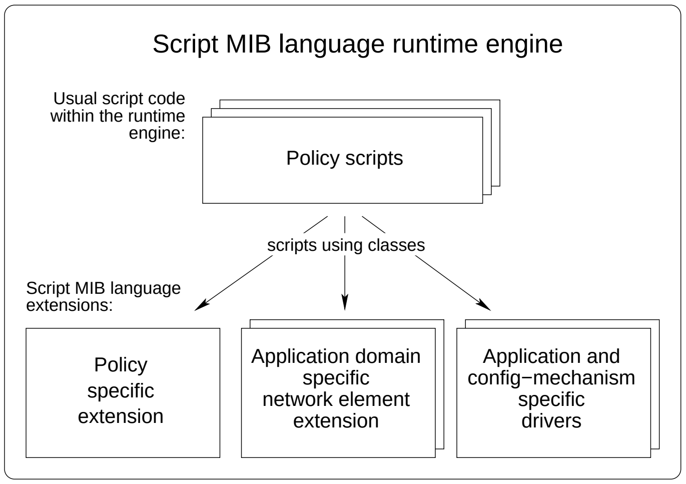
Script MIB language runtime engine
Usual script code within the runtime engine:
Policy scripts
scripts using classes
Script MIB language extensions:
Policy specific extension
Application domain specific network element extension
Application and config−mechanism specific drivers

Figure 2: The standard Script MIB runtime engine is executing policy ‘scripts’ that use policy-supporting language extensions.

1. A general policy management language extension provides interfaces to derive and implement policies, rules, conditions, actions, network elements, event generators and events.

2. Domain specific language extensions provide abstract interfaces to network elements of a specific policy application domain. They allow policy scripts to retrieve element attributes and event notifications and to correlate them to make policy decisions, so that they can in turn be used to configure network elements.

3. Drivers realize the mapping between the domain specific interfaces and the underlying device-level mechanism to actually configure the network elements.

2.3 Implementation

A prototype implementation of these libraries has been developed based on the Jasmin Script MIB implementation [8] with the Java runtime engine. This prototype is described in the following sections, while Section 2.4 gives a simple policy script example. A class diagram of the libraries as well as the example is shown in Figure 3.

All classes are documented using JavaDoc. The output is available from the Jasmin web pages [8] and enclosed with the Jasmin software distribution.

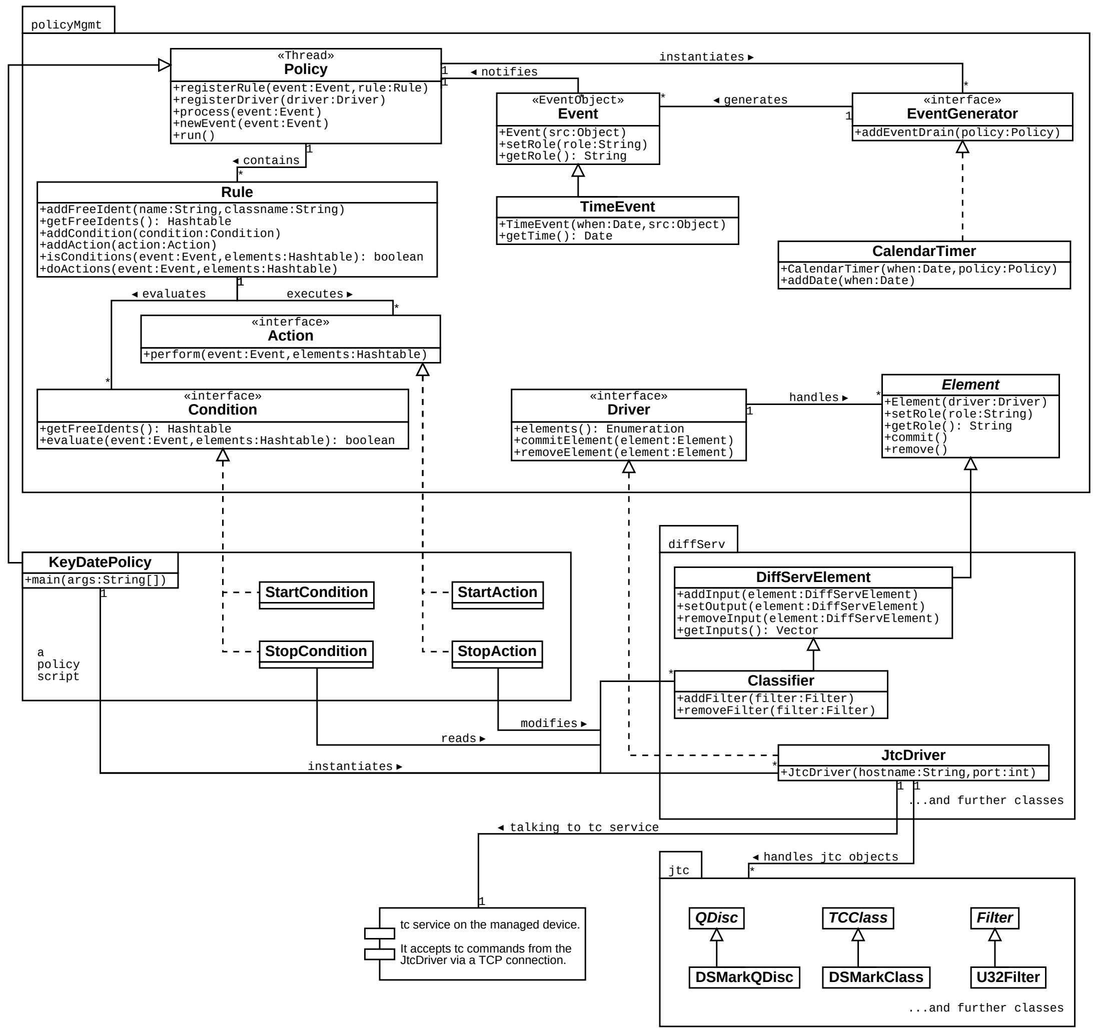
policyMgmt
«Thread»
Policy
+registerRule(event:Event,rule:Rule)
+registerDriver(driver:Driver)
+process(event:Event)
+newEvent(event:Event)
+run()
instantiates ▸
◂ notifies
«EventObject»
Event
+Event(src:Object)
+setRole(role:String)
+getRole(): String
◂ generates
«interface»
EventGenerator
+addEventDrain(policy:Policy)
◂ contains
Rule
+addFreeIdent(name:String,classname:String)
+getFreeIdents(): Hashtable
+addCondition(condition:Condition)
+addAction(action:Action)
+isConditions(event:Event,elements:Hashtable): boolean
+doActions(event:Event,elements:Hashtable)
TimeEvent
+TimeEvent(when:Date,src:Object)
+getTime(): Date
CalendarTimer
+CalendarTimer(when:Date,policy:Policy)
+addDate(when:Date)
◂ evaluates
executes ▸
«interface»
Action
+perform(event:Event,elements:Hashtable)
«interface»
Condition
+getFreeIdents(): Hashtable
+evaluate(event:Event,elements:Hashtable): boolean
«interface»
Driver
+elements(): Enumeration
+commitElement(element:Element)
+removeElement(element:Element)
handles ▸
Element
+Element(driver:Driver)
+setRole(role:String)
+getRole(): String
+commit()
+remove()
KeyDatePolicy
+main(args:String[])
a policy script
StartCondition
StartAction
StopCondition
StopAction
diffServ
DiffServElement
+addInput(element:DiffServElement)
+setOutput(element:DiffServElement)
+removeInput(element:DiffServElement)
+getInputs(): Vector
Classifier
+addFilter(filter:Filter)
+removeFilter(filter:Filter)
modifies ▸
reads ▸
instantiates ▸
JtcDriver
+JtcDriver(hostname:String,port:int)
...and further classes
◂ talking to tc service
◂ handles jtc objects
jtc
tc service on the managed device.
It accepts tc commands from the JtcDriver via a TCP connection.
QDisc
TCClass
Filter
DSMarkQDisc
DSMarkClass
U32Filter
...and further classes
1
*

Figure 3: Class diagram of (a) the policy management package `policyMgmt`, (b) the DiffServ domain specific package `diffServ`, and (c) the Linux tc specific driver `jtc`. A policy script (d) `KeyDatePolicy` makes use of these components.

Please note that the current prototype implementation does not include implementations of all DiffServ elements and the implemented classes are not necessarily complete, e.g., various DiffServ droppers and meters are not implemented and most elements do not yet support deletion.

2.3.1 The policyMgmt Package

The `policyMgmt` package contains the classes and interfaces `Policy`, `Condition` and `Action` that are usually implemented by policy scripts: A `Policy`-derived class represents the main class of a policy script that is executed when the policy is started. It registers a number of newly instantiated `Rule`s. Each rule in turn registers instances of implementations of the `Condition` and `Action` interfaces.
Two general condition classes are provided by the `policyMgmt` package: `AlwaysCondition` simply matches unconditionally and can be used by rules that shall be executed based on a triggering event without further conditions. `IsRepeated(cond, n)` is a condition that can be used to evaluate to true only if an underlying condition `cond` evaluated to true `n` times. Besides these generic conditions, application specific conditions and actions can be provided by the corresponding application specific class package. Finally, policy specific condition and action classes can be programmed individually for a policy script, i.e., they can benefit from the whole flexibility provided by the Java programming language.
The evaluations of rules is always initiated by `Event`s, which are triggered by `EventGenerator`s. Similar to conditions there is a general event generator class implemented within the `policyMgmt` package: `CalendarTimer` is a time based event generator that generates `TimeEvent`s at specific points in time.
The abstract class `Element` is the parent of all network elements modeled by domain specific packages. Roles can be assigned to Elements so that they can be used in conditions to select subsets of elements that should be affected by a rule. All elements are handled through a `Driver` interface. This interface allows to synchronize a policy's notion of element states with the configuration of the underlying network devices, i.e., it allows to fetch elements, commit modified elements, and remove elements.

2.3.2 The diffServ Package

The application domain targeted in this project is the configuration of DiffServ nodes. Hence, the `diffServ` package contains element classes to represent DiffServ classifiers, filters, meters, actions, droppers, queues, schedulers, etc. The methods of these classes allow the policy developer to create, delete and modify the elements.
The classes have been modeled based on the DIFFSERV-MIB data model, i.e., these data path elements can be plugged together (with certain limitations) through methods provided by the common parent class `DiffServElement`, which in turn is a child class of the `policyMgmt.Element` class.
Where apprørpiate, the package defines class using inheritence, e.g. `DSCPMarker` and `DropAction` are derived from `diffServ.Action` which is derived from `diffServ.DiffServElement` and `MFFilter` is derived from `diffServ.Filter` which is also derived from `diffServ.DiffServElement`.

2.3.3 The jtc Package and the JtcDriver class

The class `JtcDriver` (which is currently contained in the `diffServ` package) represents the adapter between the protocol independent classes of the `diffServ` package and the device specific configuration mechanism. In the current prototype we have implemented support for the Linux 2.4 “tc” (traffic conditioning) subsystem through the Java class package `jtc` which represents the tc data structures, namely queueing disciplines (`qdisc`), classes, and filters. This allows the `JtcDriver` to manage tc configurations mapped from the model supported by the `diffServ` package.
In order to actually write tc configuration to the kernel, we have implemented a simple TCP service on the managed Linux DiffServ node, that accepts tc commands sent by the `JtcDriver`. Upon instantiation, the `JtcDriver` connects to the server. Calls to the `elements()`, `commitElement()`, and `removeElement()` methods of the `Driver` interface lead to appropriate mapped tc commands sent by the `JtcDriver`.

2.4 Example

The following example contains a policy script that contains a pair of rules: a `StartRule` that gets triggered by a `CalendarTimer` at a specific point in time, e.g. at 23:00 on December 31, and a `StopRule` that gets triggered some time later, e.g. at 2:00 on January 1. The conditions evaluate unconditionally to true in both cases. The `StartAction` doubles a specific bandwidth parameter and after the critical time period the `StopAction` resets it to the start value. This might be reasonable to allow mobile phone traffic to carry the expected increased bandwidth demand during that time period. Figure 4 shows the essential parts of a Java policy script that supports this scenario.

```
// imports ...

public class KeyDatePolicy extends Policy {

    int         defaultRate;
    TCB         tcb;
    TokenBucket tokenBucket;
    // ...

    public class StopCondition implements Condition {
        public boolean evaluate(Event event, Hashtable elements) {
            // here we could check whether the required bandwidth
            // is back to a normal level. if not, we could return false.
            return (true);
        }
        public Hashtable getFreeIdents() { return new Hashtable(); }
    }

    public class StartAction implements Action {
        public void perform(Event event, Hashtable element) {
            defaultRate = tokenBucket.getRate();
            tokenBucket.setRate(defaultRate * 2);
            try { tcb.commit(); } catch (IOException e) {}
        }
    }

    public class StopAction implements Action {
        public void perform(Event event, Hashtable element) {
            tokenBucket.setRate(defaultRate);
            try { tcb.commit(); } catch (IOException e) {}
        }
    }

    public KeyDatePolicy(String[] args) {

        // ...

// setup the driver and the general DiffServ config...
JtcDriver driver = new JtcDriver("fosters", 10101);
tcb = new TCB(driver);
// ...

// setup the start and stop calendar timers...
Date startDate = formatter.parse("31.12.2001 23:30:00");
Date stopDate  = formatter.parse("01.01.2002 02:00:00");
CalendarTimer startTimer = new CalendarTimer(this, startDate);
CalendarTimer stopTimer  = new CalendarTimer(this, stopDate);
startTimer.start();
stopTimer.start();

// setup the policy rules
Rule startRule = new Rule();
Rule stopRule  = new Rule();
startRule.addCondition(new AlwaysCondition());
startRule.addAction(new StartAction());
stopRule.addCondition(new AlwaysCondition());
stopRule.addAction(new StopAction());
this.registerRule(startTimer, startRule);
this.registerRule(stopTimer, stopRule);

// ...
    }

    public static void main (String[] args) {

        KeyDatePolicy policy = (new KeyDatePolicy(args));
        policy.start();
// ...
        policy.join();
// ...
    }
}
```

Figure 4: `KeyDatePolicy.java` – A simple policy script that assigns addition bandwidth during a specified time period.

3 Conclusions

3.1 Met Requirements

Most of the requirements mentioned in Section 2.1 have been met by our implementation:

Policy condition classes can be written so that they determine the identifiers of network elements to be used in conditions and actions (1). The method to retrieve these identifiers can be implemented individually (2). Network elements for the DiffServ application domain have been implemented as parts of the `policyMgmt.diffServ` package (3). A `CalendarTimer` has been implemented as a typical time based event generator (4). Other timers are not yet implemented, especially a timer that allows times to be specified by expressions with wildcards and ranges would be useful.

Event generators based on received notifications, e.g. SNMP traps, have not yet been implemented. However, the policy library is designed so that an `Element` can be the trigger of an `Event` (5). A `Debug` class allows to add debugging messages to the policy scripts. Furthermore, policies can be run from a standard JVM without a Jasmin runtime engine. This makes policy development and debugging easier (6).

Special treatment of failed execution of policy conditions and actions has not been implemented, however exceptions can be thrown and caught as usual (7). Based on the Script MIB, policies represent separate scripts that don't share name spaces (8).

Security w.r.t. creating, reading, removing, and controlling policies is given by the Script MIB access control based on VACM. No additional explicit security mechanisms are required for the policy management system (9). The abstraction of application domain packages allows policies to be written in a way independent from underlying configuration protocols. Only drivers, in our prototype the `JtcDriver` class and the `jtc` package, have a notion network device specific data (10).

The object-oriented class model of the `policyMgmt` package reflects the elements and associations of a declarative representation of policies. Usually, inidividual Java code is only used at policy startup and within the `Condition.evaluate()` and `Action.perform()` methods (11). Code within a policy can be shared, e.g. by implementing methods used by similar conditions or actions. However, between policy scripts code can only be shared on source code level (12).

Communication between policies is not explicitly supported. However, usual IPC techniques, e.g. sockets or Java RMI, can be used (13). Since policies are usual scripts to the Script MIB, it's possible to pass arguments via the `smLaunchArgument` MIB object and parse them as command line arguments within the Java code (14).

3.2 Future Work

A significant drawback of the presented approach, which is purely based on the Java programming language, is the expense that has be spent even for very simplistic or standard policies. For a single policy a minimum of three classes have to be instantiated. Usually, there are some more objects that have to be handled and some classes have to implemented.
This problem could be addressed by a specific policy description language and a compiler that maps such policies to Java code of the current form. This general idea has already been mentioned in the concept report [1], but it could not yet be realized within the given time frame. An interesting approach would be to investigate the mapping of the Ponder policy specification language [14] to our architecture.

References

[1] F. Strauß. Concept of a script mib based policy management system. Technical report, http://www.ibr.cs.tu-bs.de/projects/jasmin/policy-concept.pdf, may 2001.

[2] P. Martinez, M. Brunner, J. Quittek, F. Strauß, J. Schönwäalder, S. Mertens, and T. Klie. *Using the Script MIB for Policy-based Configuration Management*. TU Braunschweig, NEC Network Laboratories, 2001. (submitted for presentation at NOMS 2002).

[3] M. Sloman, J. Lobo, and E.C. Lupu, editors. *Policies for Distributed Systems and Networks - Policy 2001 Workshop Proceedings*, Bristol, 2001. Springer.

[4] *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management*. Boston, May 1999.

[5] J. Nicklisch. A rule language for network policies. In *Policy 1999 Workshop Proceedings*, http://www-dse.doc.ic.ac.uk/events/policy-99/pdf/26-Nicklisch.pdf, 1999.

[6] D. Levi and J. Schönwälder. Definitions of Managed Objects for the Delegation of Management Scripts. RFC 2592, Nortel Networks, TU Braunschweig, May 1999.

[7] F. Strauß, J. Schönwälder, and J. Quittek. Open Source Components for Distributed Internet Management. In *Proc. 7th IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, May 2001.

[8] J. Quittek, J. Schönwälder, and F. Strauß. *Jasmin - A Script MIB Implementation*. TU Braunschweig, NEC Network Laboratories, http://www.ibr.cs.tu-bs.de/projects/jasmin/, 2001.

[9] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, Torrent Networking Technologies, EMC Corporation, Sun Microsystems, Nortel UK, Bell Labs Lucent Technologies, December 1998.

[10] DIFFSERV Working Group. *Differentiated Services Working Group Charter*. IETF, http://www.ietf.org/html.charters/diffserv-charter.html, 2001.

[11] F. Baker, K. Chan, and A. Smith. *Management Information Base for the Differentiated Services Architecture*. IETF DIFFSERV Working Group, http://www.ietf.org/internet-drafts/draft-ietf-diffserv-mib-09.txt, March 2001.

[12] M. Fine, K. McCloghrie, J. Seligson, K. Chan, S. Hahn, C. Bell, A. Smith, and F. Reichmeyer. *Differentiated Services Quality of Service Policy Information Base*. IETF DIFFSERV Working Group, http://www.ietf.org/internet-drafts/draft-ietf-diffserv-pib-03.txt, March 2001.

[13] D. Levi and J. Schönwälder. Definitions of Managed Objects for Scheduling Management Operations. RFC 2591, Nortel Networks, TU Braunschweig, May 1999.

[14] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. Technical report, http://www-dse.doc.ic.ac.uk/ mss/Papers/Ponder-summary.pdf, aug 2000.