Rule-Based Building-Block Architecture for Policy-based Networking*

Yasusi Kanada
Systems Development Laboratory, Hitachi, Ltd.
Totsuka-ku Yoshida-cho 292, Yokohama 244-0817, Japan
E-mail: kanada@sdl.hitachi.co.jp

Brian J. O'Keefe
Hewlett-Packard Company
3404 East Harmony Road, Ft.Collins, CO 80528-9599, USA
E-mail: brian.okeefe@hp.com

Abstract: *We developed two rule-based building-block architectures, i.e., pipe-connection and label-connection architectures, for describing complex and structured policies, especially network QoS policies. The latter is focused on in this study. The relationships or connections between building blocks are specified by the dataflow and control flow between them. The dataflow is specified by tags, including virtual flow labels (VFLs), which are data attached to "outside packets". The control flow can be classified and specified by four control structures: concatenation, parallel application, selection, and repetition. We have designed fine-grained and coarse-grained building blocks and methods for specifying dataflow and control flow in differentiated services (Diffserv), and implemented the coarse-grained ones in a policy server. Two cases of building-block use are described, and we concluded that there are five advantages of building-block-based policies, i.e., expressibility, uniform semantics, simplicity, flexibility, and management-task-oriented design. We also developed techniques for transforming building-block policies into executable ones, which are called policy division and fusion.*

1. Introduction

Policies can control computer systems, networks, or even systems that contain human beings as their components. In this paper, we focus on policy-based networks with relatively low-level policies that a computer can translate into network node configurations, and we use network QoS examples. However, the concepts described in the present paper are generic, and can be applied to other applications such as access control or security (encryption/decryption) control.

Policy-based networks are networks that are controlled by policies. Policies are defined by users, a network administrator, or operators, which are managed by policy servers, and deployed to network nodes. A policy is usually described as a sequence of condition-action rules:

if (condition$_1$) action$_1$, if (condition$_2$) action$_2$, …, if (condition$_n$) action$_n$.

A policy is executed as follows: the left-most rule whose condition is evaluated to be true is selected, and the corresponding action is executed.[1] Only one of the actions within the policy will be executed, even if subsequent rules have a true condition.[2]

Policy-based networking technologies have been developed to reduce the complexity in configuration of a network and its nodes. Policies are replacements of vendor- and device-dependent configuration commands, and they are derived from SLSs (service-level specifications). Eash SLS is the specification of a QoS (i.e., availability, throughput, delay, etc.) for a customer, and an SLS is derived from an SLA (service-level agreement), which is a contract between a network operator and a user or between two or more network operators.

Policy-based networking has three major characteristics. First, it is easy to create, to modify, and to delete policy rules dynamically without interfering with other rules as far as the rules do not interfere with each other (i.e., if they have been defined ideally). A rule is a fine-grained module that can be added, deleted, or modified independently of other rules. Second, the amount of network configuration tasks can be reduced by using policies, because one policy can be used for policy targets which are of various types, i.e., network nodes or interfaces, and which have been developed by a variety of vendors. Third, heterogeneous networks can be controlled according to a unified set of policies that follow the IETF (Internet Engineering Task Force) standards. Policies are modeled by the Policy Framework Working Group of the IETF [26, 31]. A protocol called COPS (Common Open Policy Services) [9], its usage called COPS-RSVP [11] and COPS-PR [6], and data formats conveyed by COPS-PR, which are called PIBs (policy information bases), e.g., Diffserv PIB [7], are also standardized by the IETF. Policies for systems management

* A (probably final) version of this paper was published in the Journal of Network and Systems Management (Vol. 11, No. 3, September 2003).

[1] In some applications, a more complex policy model is required. We have used this model in the present paper for the sake of simplicity.

[2] These semantics may seem to be too restrictive. However, this constraint is at least required in the pipe-connection model described in Section 2.2, and more complicated policies can be constructed using such primitive policies by policy combination.

are modeled in the OSI (Open Systems Interconnection) [13].

In addition to the first characteristic mentioned above, i.e., modularity, the rule-based architecture results in two additional characteristics. First, SLA or SLS, which are also declarative, can be more easily translated into policies because rule-based policies are declarative. Second, by using technologies developed for rule-based languages such as OPS5 [10] or Prolog, declarative policies can be regarded as executable programs with unified semantics. Policies must be executable when deployed to network nodes. These characteristics are more extensively explained by Kanada [16].

In policy-based networks, two or more policies often work in cooperation, and a high-level policy may have to be decomposed into two or more low-level policies. For example, in a QoS-assured network service such as Diffserv (Differentiated Services) [5], packet flows from service subscribers are classified and policed (i.e., limited to a certain bandwidth) at an edge router, and queued and scheduled in each router that the flow passes through. Thus, policies for classification, policing, and queuing/scheduling must cooperate to assure QoS. If the service is typical Diffserv, the policy for classification specifies the class or the DSCP (Diffserv Code Points) [29] of the flow, and the policy for queuing/scheduling specifies the testing of the DSCPs to determine the algorithms and parameters for queuing and scheduling required by packets in that class. These policies can be regarded as components of a network-wide QoS policy. There should, therefore, be means of structuring the policies, i.e., building blocks (or components) that the policies should be constructed from and the relationships between the building blocks.

Conventionally, two or more cooperating policies are described as separate policies with no explicit means for cooperation. They cooperate implicitly and in an ad-hoc way. This works well for many applications and for a limited but wide range of situations. For example, in Diffserv, separate policies work well when edge and core functions are deployed to different network interfaces and the functions are simple. However, there are two major reasons why the concept of policy combination [17], or an explicit and systematic method of cooperation, is required. One reason is that policy description should be more expressive and flexible. It is needed to describe the relationship between policies when the relationship is not predefined. If it is required to specify the relationship in/by the policies for the sake of expressibility and flexibility, policy combination is required. The other reason is that complexity and ambiguity in policy description should be reduced. If the relationship is implicit, it may become ambiguous and may easily cause misunderstandings. In addition, because no information can be passed from a deployed policy (i.e., a policy that is being executed) to another, the same data may have to be computed two or more times when they use the same data. For example, in Diffserv, a classifier, which is a list of conditions in the policy and classifies the packets, may be duplicated to each policy. This duplication may cause serious semantic problems [18] and bugs (errors in policies).

There are also two more concrete reasons or advantages in the use of building blocks. One advantage is reusability. If the values of every constant and variable in a building block are fixed, the building block has fewer chances of being reused. In contrast, if we can give the values as parameters to the building blocks, we can maximize reusability. There are examples of parametric building blocks are shown in Section 4.1. The other advantage is the possibility of optimization. Building-block structures are required for optimizations of policies because an optimization is a transformation of the structure of a policy [16].

Moffett and Sloman [25] analyzed policy conflicts, and Lupu and Sloman [22] developed methods for handling them. Conflicts are types of relationships between policies. They are usually negative relationships because they cause inconsistencies among policies and because they accidentally and implicitly occur. In contrast, the concept of policy combination was developed for explicitly specifying a *positive* relationship between policies.

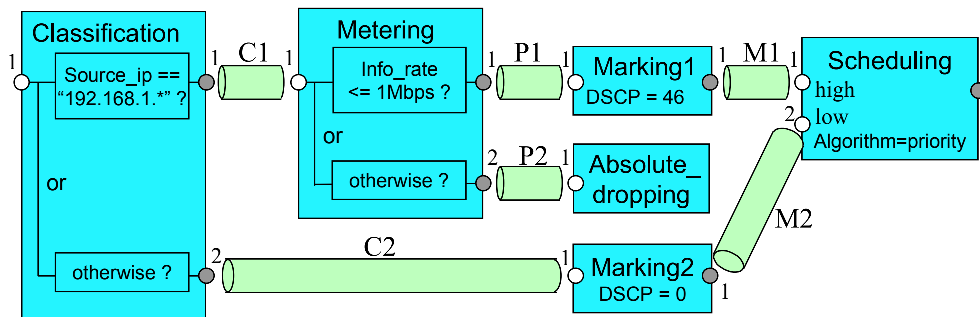
Section 2 of this paper describes two rule-based building-block architectures for policies, or two policy-combination architectures, i.e., the pipe-connection and the label-connection architectures. Dataflow and control flow between building blocks are classified in Section 3. There are examples of building blocks, including ones that have been implemented in a policy server in Section 4. Techniques for the transformation of building-block policies into executable ones, i.e., policy division and fusion, are also briefly described in Section 5. Related work is reviewed in Section 6.

2. Two Rule-based Building-Block Model Architectures

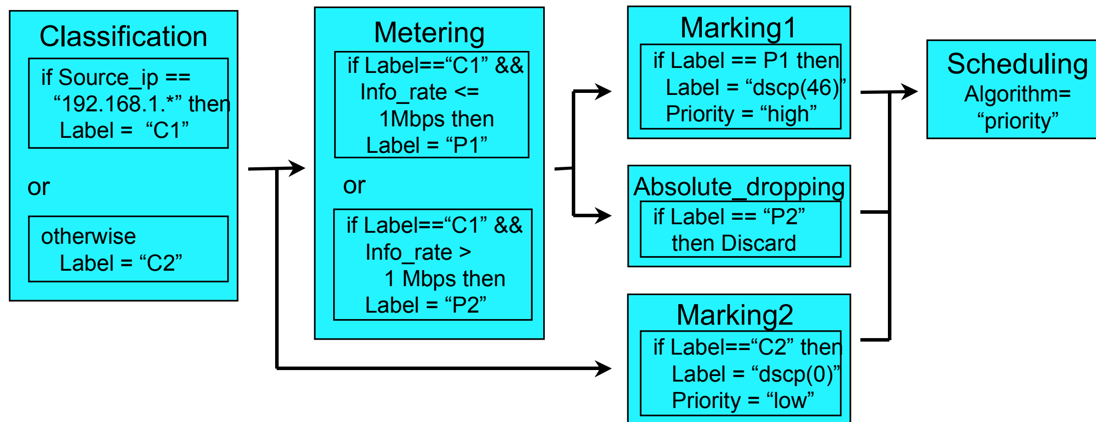
In this section, two policy architectures are introduced after common concepts are explained. These architectures were originally proposed and compared in previous work [15, 16].

2.1 Structure of Building Blocks

In both architectures, a policy or policy rule consists of building blocks and the connections between them. A building block is a rule or a list of rules and is executed as follows. A rule is selected if the input packet matches the condition specified in the rule. Then the action specified in the rule is executed and an output packet is generated. If no condition in the rule set matches the input



(a) A policy using pipe-connection architecture



(b) A policy using label-connection architecture

Figure 1. Policies using two building-block architectures

packet, no action is taken and no packet is outputted. Thus, a building block inputs a stream of packets, or a flow, filters it, and either splits it into multiple flows or merges multiple flows into one.[1]

A network node function, or a whole network node, can be modeled as a building block or a collection of building blocks. A building block has input ports and output ports. Building blocks are combined by connecting each input port and output port. Consequently, the function of the network node can be represented by a graph, usually a DAG (directed acyclic graph), in which the vertices represent the building blocks. The whole network can also be modeled by using building blocks. Each function in the network domain can be represented by a graph. The task of a policy server is to input a high-level policy, to construct a graph, to decompose this graph into subgraphs or to transform this graph into lower-level graphs, and to deploy each subgraph to each router in the domain. The edges between the subgraphs are mapped to the lines between the network nodes.

[1] The rule is not necessarily selected when a packet arrives. If the condition can be evaluated before the packet arrives, the rule can be selected beforehand. For example, if the policy controls a control-plane function such as routing, the condition can usually be evaluated beforehand.

The function of each building block may be described by using smaller building blocks, i.e., fine-grained policies, or they may be implemented by using sequences of executable control commands, such as CLI (command-line interpreter) commands or SNMP messages.

2.2 Pipe-connection Architecture

The pipe-connection architecture is explained by using the example in **Figure 1**(a). This example is a Diffserv router configuration policy. This policy is a device-level policy and can partially implement a network-level policy.

In the pipe-connection architecture, each building block has a fixed number (usually one) of input and output ports. Each input or output port has a port identifier. Port identifiers can be numbers or alphanumeric identifiers, but they have been assumed to be ordinal numbers here.

Building blocks are connected by pipes. Pipes represent data, which are usually data streams. Pipes are uniquely identified by their tags. The beginning of a pipe is connected to an output port of a building block. The end of the pipe is connected to an input port of another building block. In the example shown in Figure 1(a), all the pipes represent containers (i.e., pipes) for packet streams. Conceptually, a packet stream flows in each

pipe. When a packet flows into a building block as an input, one (or zero) packet flows out from the building block as an output. Packets enter a building block through an input port and exit through an output port. In the majority of the building blocks, the number of packets is preserved, i.e., packets are not implicitly duplicated, merged, or discarded; a packet is outputted to only one of the output ports and two packets that come through the same or different input ports are never merged into one packet.[1]

Figure 1(a) shows six building blocks: Classification, Metering, Marking1, Absolute_dropping, Marking2, and Scheduling. The Classification and Metering building-blocks contain two sub-blocks, or rules. Other building-blocks can contain only one sub-block. The Classification and Metering building-blocks are connected by a pipe called C1, which connects output port 1 to input port 1. The Classification building-block has two output ports. Each packet that flows into this building-block flows out from only one of these output ports. The Absolute_dropping building-block has no output ports; packets that flow into a Absolute_dropping building-block never flow out. The Scheduling building-block has two input ports, but other building-blocks have one input port.

This architecture can be properly represented using a logic-based concurrent programming language such as GHC [32], Concurrent Prolog [30], or Parlog [8]. Thus, we can provide uniform semantics to policies in this architecture. These languages are descendants of Prolog, which is based on forward-chaining logical inference. However, parallel or pipelined programs can be described using these languages, as opposed to Prolog, which is a sequential programming language. The capability of describing pipelined processing is better suited for describing stream processing. Therefore, these languages are especially suited for describing low-level network policies. By using a logic-based language, a program or a policy can be both declarative and executable.

In a logic-based concurrent language, each rule consists of a guard and a body:

rule_name(Parameters) :– Guard | Body.

A guard is similar to a condition list and a body is similar to an action list. A rule whose guard is evaluated to be true is selected. As both a guard and a body are a list of predicate (i.e., policy) calls, they consist of building blocks, which may be nested.

The above-mentioned languages are suited to describing data stream processing. Therefore, pipe-connection models can be expressed directly. Kanada [15] defined a language for this architecture, SNAP (Structured Network programming by And-Parallel language). In SNAP, each building block is represented by a predicate, which consists of clauses (i.e., rules). Building blocks are connected by logical variables. This means a logical variable is used as a pipe.

2.3 Label-connection Architecture

The policy using the label-connection architecture is explained through Figure 1(b). The function of this policy is the same as that in Figure 1(a). In this architecture, each building block may have any number of input and output data. Variables, such as the Source_ip or Priority in Figure 1(b), represent input/output data. Building blocks contain one or more rules. For example, both the Classification and Metering building-blocks have two rules. The order of execution of the building blocks is partially specified; i.e., the order constraints are represented by a directed graph. In Figure 1(b), six building blocks are combined according to the arrows, which specify the order of execution. For example, the Classification building-block, which contains two filtering rules, is connected to the Metering and Marking2 building-blocks. As a result, the Metering and Marking2 building-blocks will be executed just after executing the Classification building-block. Whether the Metering or Marking2 building-block is executed depends on the conditions of the rules in the Metering and Marking2 building-blocks and on the value of the packet. If the packet matches a condition in the Metering building-block, this building block is executed.

Each rule attaches a virtual flow label (VFL), called “Label” in Figure 1(b) (explained in Section 3.1) to each packet in a flow or to the whole packet flow. Only one VFL can be attached to a flow or packet. In Figure 1(b), the rules in the Classification and Metering building blocks assign a VFL, and the Marking1 and Marking2 building blocks assign a DSCP as the label. The initial value of a VFL is a specific value, e.g, "" (an empty string).

Label-connection architecture can properly be represented by using a language for production systems similar to OPS5 or other forward-chaining rule-based languages for developing expert systems, or by using a language for active databases [23]. Thus, we can provide uniform semantics to policies in this architecture as well. In such a language, each rule is a condition-action rule or event-condition-action (ECA) rule [23].[2]

Conventional languages for production systems have no method for structuring rules (i.e., building blocks). There should, therefore, be a method for defining a collection of rules, i.e., a policy, and there must be a method

[1] This “conservation law” requires only one of the actions within a policy to be executed in the pipe-connection architecture; i.e., only one rule must be selected.

[2] We applied a restriction where only one of the actions within a policy will be executed. However, it can be removed in the label-connection architecture.

of ordering policies. A method of ordering policies is described in Section 3.2.

3. Dataflow and Control Flow in Label-connection Architecture

Kanada [16] compared pipe- and label-connection architectures. He concluded that label-connection architecture is more practical in the short term because it is closer to conventional policy architecture and has greater advantages for policy specification and deployment using the current technologies.[1] This section, thus, focuses on the label-connection architecture, where a connection between building blocks is specified by the dataflow and control flow between the building blocks. Dataflow and control flow are classified and discussed below.

3.1 Dataflow between building-block policies

To make two or more policy rules that belong to different policies cooperate, information must be transferred between them. A piece of information transferred between rules is called a *tag*. Tags can be *real*, i.e., in the packet, or *virtual*, i.e., outside the packet (see **Figure 2**). A real tag is conveyed by packets, so the two cooperating policies can exist in different network nodes. However, a virtual tag is not conveyed by the packets themselves, so the two cooperating policies must exist in the same network node in which the virtual tag value can be stored or transmitted implicitly, unless the virtual tag value is conveyed by some other means, such as wavelength or physical location, similar to Generalized MPLS (multi-protocol label switching) [4]. A tag causes explicit data dependence [21] between policy rules.

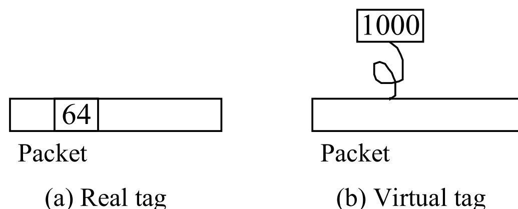


Figure 2. Real and virtual tags

Tags can be categorized as follows.

1. *Labels*: A tag may be used for selecting a rule from a policy. This type of tag is called a label. A label connects one rule to another (i.e., the label assigned by one rule specifies the next rule to be applied). The VFL is an alias of the virtual label. DSCPs (Diffserv code points) [29] are used as real labels.
2. *Attributes*: A tag may be used in the action part of the rule without selecting a rule. This type of tag is called an attribute [16]. Virtual attributes are useful for hierarchical scheduling, shaping, and policing. An example of hierarchical shaping is described in Section 4.4.

A real tag usually has an integral value; a virtual tag may have an integer or a character string. String values are used throughout this paper. The string values are usually translated into other types of data, such as integer values by the policy server (PDP) or agent (PEP).

3.2 Control flow between building-block policies

Policies change the behavior of a network in a consistent manner. Thus, a set of policies that work together may be regarded as a distributed program. In procedural programs, such as those written in C or C++, there are four relationships (control structures) between statements: concatenation (sequential application), parallel application, selection, and repetition. A set of policies can be regarded as a rule-based program. Thus, the order of application and the control dependence [1] of policies must be specified. Thus, the relationships between policies can be classified into these four types [17].

3.2.1 Concatenation

If two policies are sequentially applied, the relationship between these two policies can be called a concatenation. For example, in a Diffserv network, a policy for packet classification and marking and a policy for queuing may be concatenated and deployed to an edge router. **Figure 3**(a) shows the policy and flow diagram. There are at least two policy rules in the classification and marking policy **C**.

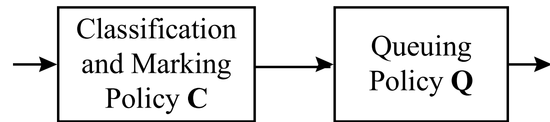
In these rules, "EF" refers to expedited forwarding [14], and "DF" refers to default forwarding or “best-effort” forwarding. Note that the two policies are ordered by the concatenation, and the two rules in these policies are connected by the DSCP. The concatenation specifies the control dependence between the policies, and the DSCP specifies the data dependence (a flow dependence [21]) between them. The DSCP is used as a real label in Rule Q1: DSCP "EF" connects Rules C1 and Q1, and DSCP "DF" connects Rules C2 and Q2. Rules Q1 and Q2 assume priority scheduling. This means a priority scheduling rule is connected to these rules. The EF traffic receives higher priority, and the DF traffic receives lower priority.

3.2.2 Parallel application

Two policies may be applied in parallel if they do not conflict. In a Diffserv network, a marking policy and a queuing policy may be applied in parallel. An example of marking and scheduling policies, i.e., **M** and **Q’**, and their flow diagram are shown in Figure 3(b). Here, the

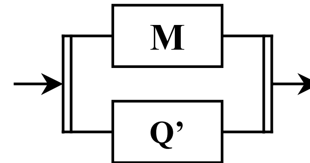
[1] The advantages described by Kanada [16] are simpler rule structures, simpler and more modular building block structures, and more flexible tag (VFL) use. The only disadvantage is less parallelism.

```
C = {  C1:  if (Source_IP == x.x.x.x) { DSCP = "EF";  },
       C2:  if (Source_IP == y.y.y.y) { DSCP = "BE";  },
       … }.

Q = {  Q1:  if (DSCP == "EF") {
                Scheduling_Priority = 6;
                Enqueue;  },
       Q2:  if (true) {
                Scheduling_Priority = 1;
                Enqueue;  },
       … }
```



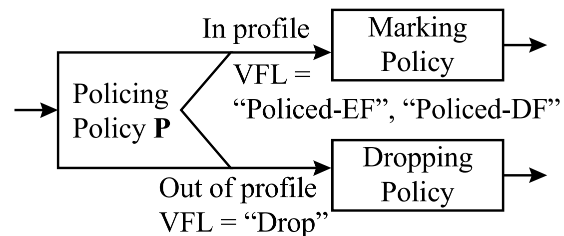
(a) Concatenation

```
M = {  M1:  if (VFL == "Policed-EF") {
                DSCP = "EF";  },
       M2:  if (true) {
                DSCP = "DF";  }}.

Q' = { Q1': if (VFL == "Policed-EF") {
                Scheduling_Priority = 6;
                Enqueue;  },
       Q2': if (true) {
                Scheduling_Priority = 1;
                Enqueue;  }}.
```



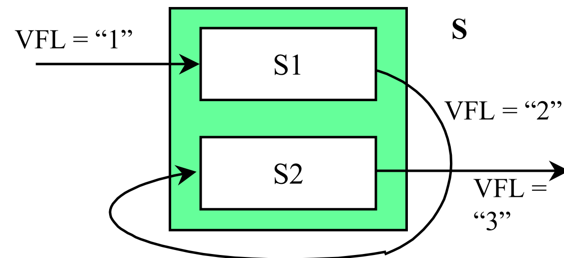
(b) Parallel application

```
P = {  P1:  if (DSCP == "EF" &&
                Information_Rate <= 2 Mbps) {
                VFL = "Policed-EF";  },
       P2:  if (DSCP == "EF") {
                VFL = "Drop";  },
       P3:  if (true) {
                VFL = "Policed-DF";  } }.
```



(c) Selection

```
S = {  S1:  if (VFL == "1") {
                VFL = "2";
                …;  },
       S2:  if (VFL == "2") {
                VFL = "3";
                …;  }}.
```



(d) Repetition

Figure 3. Examples of control flows

marking policy is separated from the classification policy. Rule M1 is applied to higher-priority packets, and Rule M2 is applied to other (lower priority) packets. The classification policy does not mark a DSCP, it just puts a VFL on the packets. Rules M1 and Q1' refer to the same VFL value. The VFL "Policed-EF" connects a rule in the classification policy to Rules M1 and Q1'.

3.2.3 Selection

If a policy outputs multiple types of application results according to the conditions of the rules, and there are multiple policies, each of which inputs each of the results, the relationship between these three policies can be called a selection. *N*-way selection, where *N* is larger than 2, is also possible.

For example, in a Diffserv network, a policy for policing may be combined with a policy for marking and a policy for absolute dropping, and these policies may be deployed to an edge router (see Figure 3(c)). Policing policy **P** in this figure can be explained as follows.

Rules P1 and P2 are applied to packets whose DSCP is "EF". This means they are higher-priority packets. The packets are passed through an information-rate meter. Rule P1 is applied to in-profile packets, and Rule P2

is applied to out-of-profile packets. Rule P3 is applied to lower-priority packets whose DSCP is not "EF". Rules P1 and P3 attach a VFL that indicates that the packet must be forwarded with higher or lower priority, and Rule P2 attaches a VFL that indicates the packets must be dropped.

Rules in the marking policy may be the same as Rules M1 and M2 explained in Section 3.2.2. A rule in the dropping policy is given as follows.

```
if (VFL == "Drop") { Absolute_drop; }.
```

This rule drops all the packets that have "Drop" as their VFL value.

3.2.4 Repetition

If a policy is repeatedly applied until a condition is met, the relationship of the policy to itself can be called a repetition. A repetition may contain two or more policies. A repetition with only one policy P1 is represented by **Figure 4**(a) and a repetition with two policies by Figure 4(b). Figure 3(d) shows the simplest case; i.e., where there is only one policy with two iterations. In this example, the VFL, "1", is used for the first iteration, and "2" is used for the second iteration. When "3" is set as a VFL, there is no rule whose condition matches this VFL in this policy, so the loop terminates. There are only two iterations in this example. However, it can be any number. If a different rule is applied for each iteration, the number of iterations is constant because the number of rules is fixed. However, as a rule can be applied repeatedly, the number of iterations is variable.

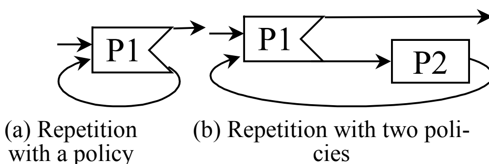


Figure 4. Two examples of repetition

4. Differentiated Service Policies Using Building-Blocks

Building blocks for Diffserv policies and methods of combining them are described in this section. Usages of the policies are also discussed. Two sets of building blocks, i.e., fine-grained ones [16], which are conceptually simpler, and coarse-grained ones [19], which are easier to use, are explained, and use cases of the coarse-grained building-blocks are presented.

4.1 Fine-grained building-blocks

Kanada [15, 16] explained five types of primitive fine-grained building-blocks for DiffServ, i.e., filtering, metering, marking, dropping, and scheduling rules. The building blocks were designed both for the label- and pipe-connection architectures. The building blocks defined here are similar to the objects defined in Diffserv MIB [3], Diffserv PIB [7], or the QoS Information Model [27]. However, the semantics can be defined more clearly by using a rule-based language.

Filtering, marking, and dropping rule sets are applied to a packet stream only once because repetitive application of these rules is unnecessary. Metering, merging, and scheduling rules can be applied to a packet stream two or more times because repetitive applications of these rules are sometimes necessary.

Rules are denoted using the following syntax: *ruleTypeName*[*parameters*]. The parameters in brackets have their own names and values. We can reuse building blocks by replacing the values of parameters.

(1) Filtering rules

Filtering rules represent a part of an MF (multi-field) or a BA (behavior aggregate) classifier. An MF classifier is a function that classifies packets by the source and destination IP addresses, IP protocol, source and destination IP ports and DSCP in the packet header. A BA classifier is a function that classifies packets only by the DSCP. The values used for the classification are specified as parameters in the filtering rule. If a packet meets the condition in the rule, it is outputted to the stream. Otherwise it is dropped. Examples of filtering rules are as follows.

```
filter[Source_IP = "192.168.1.*"].
                    // A part of an MF classifier.
filter[DSCP = 46].   // A part of a BA classifier.
```

The first example is almost identical to the Classification policy in Figure 1(a) or Figure 1(b); it can be implemented by using either the pipe- or label-connection architecture and it can be used in one of these architectures.

(2) Metering rules

Metering rules only pass traffic that conforms to the profile contracted by an SLA (service-level agreement). Metering rules can be implemented by using a token-bucket meter or some other type of meter. The average maximum information rate and the bucket size can be specified as parameters. An example of a metering rule is

```
meter[Max_information_rate = 1 Mbps].
```

This rule is very similar to the Metering policy in Figure 1(a) or Figure 1(b). The difference is that the maximum rate is specified using an equal sign (“=”) instead of inequality for the sake of syntactic uniformity.

(3) Marking rules

Marking rules write a DSCP into the DS field of the packets in the input stream. All the packets are then outputted to the output stream. The only parameter for marking rules is the DSCP. An example is

mark[DSCP = 46].

(4) Dropping rules

Dropping rules drop all packets in a stream. There are two types of dropping rules in a label-connection model: absolute dropping rules and random dropping rules. Absolute dropping rules drop all packets. Random dropping rules drop packets by using a weighted random-early-discard (WRED) or similar algorithm. An example of a random dropping rule is

randomDrop[QMin = 10 kB, QMax = 20 kB, PMax = 0.1].

The function of random dropping rules is included in the scheduling rules in a pipe-connection model, so they do not exist. There are no parameters to be specified for the absolute dropping rule,

absoluteDrop.

(5) Scheduling rules

Scheduling rules are used for merging streams through scheduling. The parameters of a scheduling rule specify the method and parameters for controlling enqueuing and dequeuing. The scheduling algorithm and its parameters can be specified in scheduling rules, which can also be used to control shaping. The maximum and minimum output rate (or both) can be specified. An example is

schedule[Algorithm = priority].

In this example, input packets are scheduled by using a priority scheduling algorithm. Input packets should have priority attributes, which are given by another scheduling rule.

In addition, a type of rule called a “merging rule” is required for the pipe-connection architecture [16]. A merging rule merges data streams coming in through two or more pipes and going out through a single pipe. A merging rule is not application-specific but generally required in this architecture. Merging rules are necessary because flows cannot be merged implicitly in the pipe-connection architecture in contrast to the label-connection architecture in which flows are implicitly merged.

Examples of compound policies using the fine-grained building-blocks mentioned above are in Figures 1 and 2. In the pipe-connection architecture, both the dataflow and control flow between the building blocks are specified by pipes. In the label-connection architecture, the dataflow is specified by VFLs and the control flow is specified by ordering the policies.

4.2 Coarse-grained building blocks

An example of coarse-grained building-blocks, which are based on the label-connection architecture, and a method for combining them are described below. Policies made of these building blocks are used [19] in a commercial policy server called OpenView/JP1 PolicyXpert [12].[1]

(1) Traffic Classifier (CL) Policy

A CL Policy classifies the packets and assigns a type of VFL called a CID (Classifier Identifier). A CL Policy is usually deployed at the edge or border interfaces of network nodes (i.e., interfaces that are connected to points outside the Diffserv domain) and applies to inbound traffic. An example of a CL policy rule is

```
if (Source_IP_address is 192.168.1.1) {
    CID = "EF_CID";
}.
```

(2) Traffic Conditioner (TC) Policy

A TC Policy meters, marks, and/or drops packets absolutely (i.e., unconditionally). TC Policies, too, are usually deployed to edge or border interfaces and apply to inbound traffic. A type of VFL called a TID (Traffic Identifier) can be assigned. An example of a TC policy rule that contains metering is

```
if (Source_IP_address == "192.168.1.1") {
  if (InformationRate <= 10 Mbps) {
          // if the bandwidth exceeds 10 Mbps,
      DSCP = 46;          // then mark 46.
  } else {
      Absolute_drop;
          // otherwise, absolutely drop the packet.
  };
}.
```

(3) Queue Control (QC) Policy

A QC Policy queues and schedules, or drops packets randomly (by using the WRED or a similar algorithm). A QC Policy is usually deployed to core interfaces (i.e., interfaces that are connected to other interfaces within the Diffserv domain) and applies to outbound traffic. A QC Policy rule can be regarded as a model of a queue or scheduler; i.e., a traffic control object. A type of VFL called a QID (Queue Set Identifier) can also be assigned. An example of a QC policy rule is

```
if (DSCP == "EF") {
  Scheduling_algorithm = "B-PQ";
        // bounded priority queuing
  Priority = 6;          // six means “high”
  Shaping_rate = 20 Mbps;
}.
```

[1] OpenView and PolicyXpert are trademarks of Hewlett-Packard Company. JP1 is a trademark of Hitachi, Ltd. PolicyXpert Version 2.0 was jointly developed by Hewlett-Packard and Hitachi.

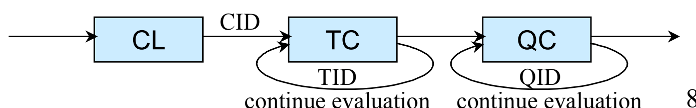


Figure 5. The evaluation order of CL, TC and QC Policies

These policies can be combined to represent a complex policy as will be discussed. However, it is important to note that, if a policy is simple, it can be described simply; i.e., a simple Diffserv function can be represented by a TC policy for each edge interface and a QC policy for all the core interfaces. The coarse-grained building blocks are, therefore, easier to use than the fine-grained ones.

When the policy to be represented is more complex, the order of policy evaluation is predefined as part of the definition of these policies. The order of CL, TC and QC Policies is predefined as outlined in **Figure 5**. This means that a CL Policy can be followed by a TC Policy, a TC Policy can be followed by a TC or QC Policy, and a QC Policy can be followed by a QC Policy or no policy. This figure also shows which type of VFL can be used for connecting rules. Only DSCPs, i.e., real flow labels, can be used for connecting TC and QC Policy rules.

There is nondeterminacy (i.e., there are alternatives) in the execution order of TC and QC Policies. To resolve this nondeterminacy, "continue evaluation" must be explicitly specified when a policy evaluation is repeated. For example, the following rule in a QC Policy is combined with another rule (which tests the QID "shape2") in the QC Policy.

```
if (DSCP == "AF11" || DSCP == "AF12" || DSCP ==
    "AF13") {
     Scheduling_algorithm = "A-BW";
     Max_queue_size = 200 packets;
     Committed_rate = 64 kbps;
     QID = "shape2";
     Continue evaluation;
    }.
```

If no "continue evaluation" statement is specified, the rule is not followed by another QC Policy rule.

4.3 Two cases of building-block use

Two cases of building-block use, which were originally described in our previous paper [19], are explained in this section. Although we will use the previously discussed coarse-grained policies for the explanations, the fine-grained policies could also be used for the same purpose.

4.3.1 Separation of subscriber and service policies

Both network services and service subscribers, i.e., end customers, can be managed by using policies. Policies for service subscribers can be separated from service policies by using CL and TC Policies as well as CIDs (shown in **Figure 6**).

In a Diffserv network, three service classes, i.e., gold, silver, and best-effort classes, can be defined. The same DSCP can be used for both gold and silver classes, but the policing rates for them, which are specified by TC Policy rules, can be different; e.g., gold traffic is policed to 1 Mbps, but silver traffic is policed to 128 kbps. Then, two different DSCPs are used, and three different CIDs, "G", "S", and "B", are used for gold, silver, and best-effort classes. Service properties can be defined by the network administrator in a service policy, which is implemented by using a TC Policy; i.e., each class of services is specified by a TC Policy rule. Subscriber properties can be defined by the network operators in subscriber policies, which are implemented by using CL Policies; i.e., each subscriber is specified by a CL Policy rule. CIDs are used for mapping or aggregating subscribers into service classes. The TC Policy can then be deployed to all inbound edge interfaces of the Diffserv network. Each CL Policy can be deployed to an edge interface and contain rules connected to the service policy rules in the TC Policy by CIDs. In Figure 6, CL Policies 1, 2, and 3 (subscriber policies) are defined, and they are deployed to three edge interfaces. There is only one TC Policy (service policy), and it is deployed to the same interfaces as the CL Policies.

When a subscriber is added or removed, the network operator can modify only the relevant CL Policy and need not modify the service policy. This separation of subscriber and service policies clearly separates the task of the network administrator from the task of the network operator without complicating the structure of policies or that of the system structure. Subscriber and service policies are separated by using VFLs, but the policies cooperate following uniform policy semantics. In this

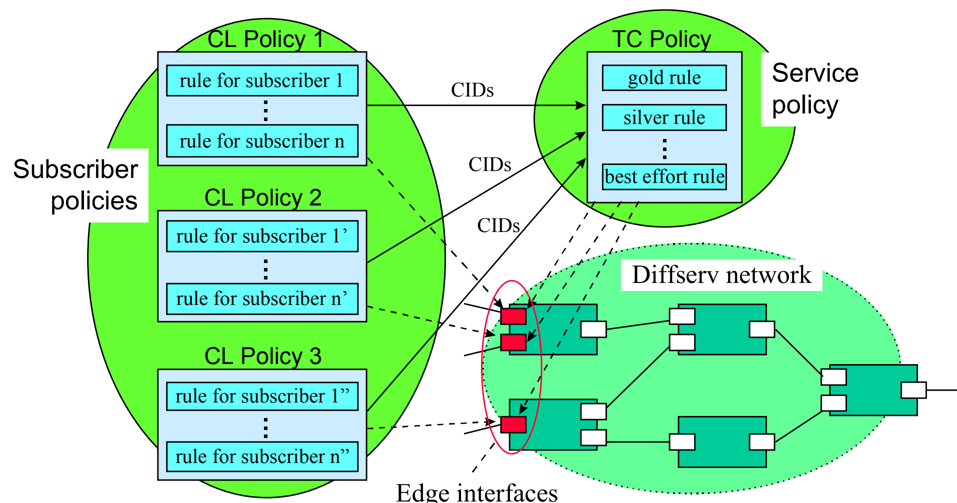


Figure 6. Separation of subscriber and service policies

example, the building blocks enable the task-oriented design of policies, i.e., it supports the tracks of both network administrators and operators. This suggests that we have chances to design task-oriented building blocks by using VFLs (or pipes), although we cannot say this is always possible.

In addition, multiple service classes that share a DSCP can be separated by using CIDs. This means that instead of using CL Policies not only to distinguish subscribers, they can also be used to distinguish service classes that share a DSCP. This makes DSCP use flexible.

4.4 Hierarchical shaper and policer

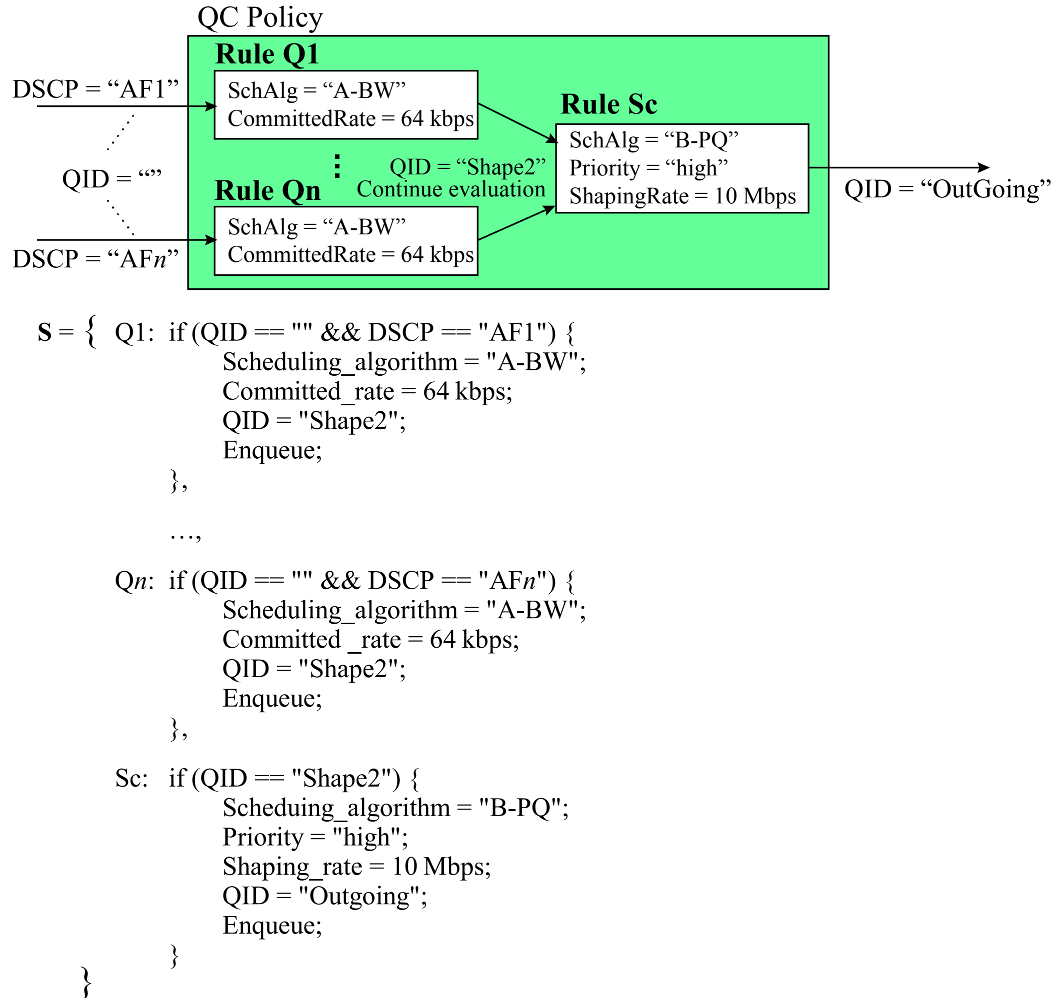


Figure 7. A hierarchical shaper

In multi-service networks, hierarchical schedulers and shapers can be used for harmonizing various types of traffic. A shaper limits the transmission rate of the traffic by delaying transmission, and a policer limits the information rate by discarding exceeded packets. These functions can be represented by using QIDs and a QC Policy. Each QC Policy rule represents a simple queuing or scheduling method. QC Policy rules can be combined by QIDs to represent a complex queuing/scheduling method.

For example, a hierarchical shaper (**Figure 7**) limits the information rate of summed traffic, in addition to limiting the information rate of each traffic flow. In Figure 7, the rate of each traffic flow is limited to 64 kbps and the rate of the summed traffic is limited to 10 Mbps. This QC Policy consists of $n + 1$ rules. Rules Q1, …, Q*n* receive input traffic and output traffic shaped at a maximum of 64 kbps by using a generic fair-queuing method called aggregated-bandwidth fair-queuing (A-BW). Here, the input traffic is assumed to have a QID value "" (the initial value), the output traffic has the QID value "Shape2", and “continue evaluation” is specified in each of the rules Q1, …, Q*n*. A-BW can be mapped to an appropriate scheduling (queuing) method implemented at the given network node. Rule Sc inputs the aggregation of the shaped traffic from Q1, …, Q*n*, and outputs traffic at a maximum of 10 Mbps by using a bounded priority-queuing (B-PQ) method.

Each rule Q1, …, Q*n* models a queue, and rule Sc models an A-BW scheduler, which is followed by a B-PQ scheduler that is not given explicitly. Other scheduling methods, i.e., strict priority queuing (S-PQ) and per-flow bandwidth fair queuing (P-BW), can also be specified in a QC Policy rule.

A hierarchical policer can be represented in a similar way to the shaper, but the details have been omitted here. Hierarchical shapers and policers can be expressed by using building blocks because these can specify their own inputs and outputs and relationships clearly. These specifications cannot be described by using conventional policies.

Note that, although hierarchical shaping, scheduling, or a policing policy is complex, a simpler function, such as a marking function or a non-hierarchical shaping or

policing function, can be represented by only a single rule. This means that this building block architecture makes it possible to describe simple policy simply, while a complex policy can be represented by using the same set of policies.

5. Transformation of Policies

To enforce a policy on network nodes, policies may have to be transformed because the set of policies (components) that the nodes can accept may be different from the set that the policy server handles. In such cases, the policy server must translate higher-level (device-independent) into lower-level (possibly device-dependent) policies. This translation can be compared to the compilation of programs written in languages such as C++. For example, in network devices, functions such as QoS or access control are usually configured by using command-line interfaces (CLIs). Commands define conditions and actions; i.e., such device functions are also controlled by using policies that consist of device-level policy rules. For example, in a Cisco router, conditions can be defined by an access control list (ACL) and referred to by an action command.

Although policy translation can be compared to program compilation, it may be much more complicated. That is, the functions of lower-level policies do not necessarily correspond to those of higher-level policies. A higher-level policy may have to be transformed into two or more lower-level policies, and two or more higher-level policies may have to be transformed into one lower-level policy. The former transformation is called a *policy division*, and the latter transformation is called a *policy fusion*. For example, a higher-level policy contains two (cooperating) functions, but no lower-level policy may contain both functions. In this case, the policy must be divided into two; otherwise, it cannot be deployed to devices. Such methods of policy division and fusion are described by Kanada [18]. This paper also revealed the problem (discussed in Section 1) where the duplication of classifiers causes serious semantic problems that makes policy transformation difficult or impossible. In addition, Kanada and Yazaki [20] showed that the complexity of and restrictions on policy division could be reduced by software and hardware integration in routers.

6. Related Work

Ao, Moffett, Lupu, Sloman, and others analyzed hierarchical policies and proposed solutions. Moffett and Sloman [25] explored the refinement of general higher-level policies into a number of more specific lower-level policies. The translation methods described in the previous section can be regarded as specific methods for this refinement. Lupu and Sloman [23] specified the parallelism and synchronization between the activities within policies by using Petri Nets and other means. These means are similar to the one described in Section 3.2.2. Ao et al. [2] proposed a hierarchical inter-policy relationship that was called superior/subordinate. A policy and its components in the present paper can be regarded as policies with superior/subordinate relationships.

The charter [28] of the NGOSS™ (New Generation Operation Support Systems) Policy-Based Management Working Group of the TeleManagement Forum listed a hierarchical policy framework, managing policies using meta-policies and other working items that related to the cooperation of policies.

Diffserv policies have been used as examples in the present paper. Low-level Diffserv policies can be represented by Diffserv PIB [6] or the QoS Device Information Model [27]. The concept of a *data path* is used in Diffserv PIB to connect components. This architecture is close to the pipe-connection architecture because a data path connects two specific components and is, thus, similar to a pipe. However, if a component has two or more input or output ports, a data path cannot distinguish them. Thus, in the QoS Device Information Model, the concept of *preamble markers*, which distinguish multiple ports, is used. A preamble marker is similar to a VFL, so, this model can be regarded as a mixture of the pipe- and label-connection architectures.

7. Summary and Conclusion

We developed two building-block architectures for policies: the pipe-connection and label-connection architectures, and we discussed the latter in detail. To describe the relationships between building blocks in this architecture, it is necessary to specify both the dataflow and control flow between the building blocks. A piece of data transferred between the building blocks is called a tag, including VFLs. The control flow can be classified and specified by four control structures: concatenation, parallel application, selection, and repetition.

We designed fine-grained and coarse-grained building blocks as well as dataflow and control flow specification methods for Diffserv, and implemented the coarse-grained ones in a policy server. The case studies qualitatively revieled that building-block-based policies have the following advantages.

- *Expressibility*: Complex functions, such as hierarchical shapers or policers, can be specified by combining primitive policy rules.
- *Uniform semantics*: All the policies follow uniform forward- or backward-chaining rule-based semantics.
- *Simplicity*: Coarse-grained policies can be designed to represent a simple policy in a simple form, while a

complex policy can be represented by using the same set of policies.

- *Less constraints*: Network services can be specified without strong constraints on the packet format. In Diffserv, multiple services that use the same DSCP can be easily defined.
- *Task-oriented design*: Building-block policies can be designed for specific management tasks; subscriber and service policies can be clearly separated in Diffserv policies.

The case studies suggest that these advantages can also be obtained in applications other than Diffserv. However, we need more experience to confirm this.

To deploy policies to network nodes, building-block policies may require complex transformations, i.e., policy division and fusion, which may be restrictive. However, the complexity and restrictions can be reduced by integrating software and hardware.

References

1. Allen, J. R., Kennedy, K., Porterfield, C., and Warren, J., “Conversion of Control Dependence to Data Dependence”*, 10th ACM Symposium on Principles of Programming Languages*, (*POPL 83*), pp. 177–189, 1983.
2. Ao, X., Minsky, N., and Nguyen, T. D., “A Hierarchical Policy Specification Language, and Enforcement Mechanism, for Governing Digital Enterprises”, *3rd IEEE International Workshop on Policies for Distributed Systems and Networks* (*Policy 2002*), pp. 38–49, June 2002.
3. Baker, F., Chan, K., and Smith, A., “Management Information Base for the Differentiated Services Architecture”, RFC 3289, IETF, May 2002.
4. Berger, L., ed., “Generalized Multi-Protocol Label Switching (GMPLS) Signaling Functional Description”, RFC 3471, IETF, January 2003.
5. Carlson, M., Weiss, W., Blake, S., Wang, Z., Black, D., and Davies, E., “An Architecture for Differentiated Services”, RFC 2475, IETF, December 1998.
6. Chan, K. H., Durham, D., Gai, S., Herzog, S., McCloghrie, K., Reichmeyer, F., Seligson, J., Smith, A., and Yavatkar, R., “COPS Usage for Policy Provisioning (COPS-PR)”, RFC 3084, IETF, March 2001.
7. Chan, K., Sahita, R., Hahn, S., and McCloghrie, K., “Differentiated Services Quality of Service Policy Information Base”, RFC 3317, March 2003, IETF.
8. Clark, K. and Gregory, S., “PARLOG: Parallel Programming in Logic”, *ACM Trans. on Programming Languages and Systems*, Vol. 8, No. 1, pp. 1–49, 1986.
9. Durham, D. (ed.), Boyle, J., Cohen, R., Herzog, S., Rajan, R., and Sastry, A., “The COPS (Common Open Policy Service) Protocol”, RFC 2741, IETF, January 2000.
10. Forgy, C. L., “*OPS5 User’s Manual*”, Technical Report CMU-CS-81-135, Carnegie Mellon University, Dept. of Computer Science, 1981.
11. Herzog, S . (ed.), Boyle, J., Cohen, R., Durham, D., Rajan, R., and Sastry, A., “COPS usage for RSVP”, RFC 2749, IETF, January 2000.
12. “HP OpenView PolicyXpert 2.0 — Users Guide”, Edition 1, Hewlett-Packard, October 2000.
13. ITU-T, “Information Technology — Open Systems Interconnection — Systems Management: Management Domain and Management Policy Management Function”, ITU-T Recommendation X.749 (ISO/IEC JTC1/SC21 10355), 1996.
14. Jacobson, V., Nichols, K., and Poduri, K., “An Expedited Forwarding PHB”, RFC 2598, IETF, June 1999.
15. Kanada, Y., “A Representation of Network Node QoS Control Policies Using Rule-based Building Blocks”, *International Workshop on Quality of Service 2000* (*IWQoS 2000*), pp. 161–163, June 2000.
16. Kanada, Y., “Two Rule-based Building-block Architectures for Policy-based Network Control”, *2nd International Working Conference on Active Networks* (*IWAN 2000*), pp. 195–210, October 2000.
17. Kanada, Y., “Taxonomy and Description of Policy Combination Methods”, *1st International Workshop on Policies for Distributed Systems and Networks* (*Policy 2001*), Lecture Notes in Computer Science, No. 1995, pp. 171–184, Springer, January 2001.
18. Kanada, Y., “Policy Division and Fusion: Examples and A Method – or, Multiple Classifiers Considered Harmful –”, *7th IFIP/IEEE International Symposium on Integrated Network Management* (*IM 2001*), pp. 545–560, IEEE, May 2001.
19. Kanada, Y. and O’Keefe, B. J., “Diffserv Policies and Their Combinations in a Policy Server Called PolixyXpert” (Extended Abstract), *5th Asia-Pacific Network Operations and Management Symposium* (*APNOMS 2001*), p. 501, 2001, a full-paper version is available from: Kanada, Y., and O’Keefe, B. J., “Combination of Diffserv Policies in OpenView/JP1 PolicyXpert”, *Technical Report of IEICE, NS2001-246, IN2001-202*, Institute of Electronics, Information and Communication Engineers (IEICE), March 2002.
20. Kanada, Y. and Yazaki, T., “A Method of Software-Hardware Integration of Diffserv Policies for

Gigabit Routers", *16th International Workshop on Communications Quality & Reliability* (CQR 2002), pp. 12–16, 2002.

21. Kuck, D. J., Kuhn, R. H., Padua, D. H., Leasure, B., and Wolfe, M., "Dependence Graphs and Compiler Optimizations", *8th ACM Symposium on Principles of Programming Languages* (*POPL 81*), pp. 207–218, 1981.
22. Lupu, E. and Sloman, M., "Conflicts in Policy-based Distributed Systems", IEEE Trans. on Software Engineering, Vol. 25, No. 6, pp. 852–869, 1999.
23. McCarthy, D. R. and Dayal, U., "The Architecture of An Active Data Base Management System", *ACM SIGMOD International Conference on Management of Data*, pp. 215–224, 1989.
24. Moffett, J. and Sloman, M., "Policy Hierarchies for Distributed Systems Management", *IEEE* Journal on Selected Areas in Communications (Special Issue on Network Management), pp. 1404–1414, December 1993.
25. Moffett, J. D. and Sloman, M. S., "Policy Conflict Analysis in Distributed System Management", Journal of Organisational Computing, Vol. 4, No. 1, pp. 1–22, Ablex Publishing, 1994.
26. Moore, B., Ellesson, E., Strassner, J., and Westerinen, A., "Policy Framework Core Information Model — Version 1 Specification", RFC 3060, IETF, February 2001.
27. Moore, B., Durham, D., Strassner, J., Westerinen, A., and Weiss, W., "Information Model for Describing Network Device QoS Datapath Mechanisms", work in progress by IETF.
28. NGOSS Policy-Based Management Working Group, "Technical Team Work Item — Terms of Reference", TeleManagement Forum Project Charter, http://www.tmforum.org/browse.asp?-catID=1003&sNode=1003&-Exp=Y&linkID=24698&docID=1285 .
29. Nichols, K., Blake, S., Baker, F., and Black, D., "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, IETF, December 1998.
30. Shapiro, E., "Concurrent Prolog: A Progress Report", *IEEE Computer*, August 1986, pp. 44–59, 1986.
31. Snir, Y., Ramberg, Y., Strassner, J., Cohen, R., and Moore, B., "Policy QoS Information Model", work in progress by IETF.
32. Ueda, K., "Guarded Horn Clauses", *Logic Programming Conference '85*, pp. 225–2236, 1985. Also in *ICOT Technical Report*, TR-103, Institute for New Generation Computer Technology, 1985, and in New Generation Computing, Vol. 5, pp. 29–44, 1987.