A Logic-based Policy Definition Language for Network Management

Yongxin Li Ming Chen Xuping Jiang Lihua Song
Computer Department, Institute of Communication Engineering
Nanjing, 210016, P.R.China
Email: liyongxin74@263.net

Policies are increasingly used to manage large-scale distributed system. This paper proposes a logic-based policy definition language termed LPDL and defines its syntax, execution model and semantic. LPDL supports management operations that consider system's states and which need cooperating policy servers. System's states are stored in a state repository and can be manipulated by the policy server based on predefined policies when certain messages are received. LPDL has Petri Net's expressive power and Turing Machine's computing power. Administrator can use LPDL to describe system's analysis and decision functions or encapsulate these functions into physical codes flexibly based on actual need. Finally, LPDL-based network management model, its prototype and application are presented to show that LPDL can meet the requirements of network management's dynamic growth.

Keywords: Network management, Policy, Petri Net, Turing Machine, Coordination model

1. Introduction

Policies are collections of general principles specifying the desired behavior or state of a system [1]. Network management is mainly carried out by following policies about the behavior of the resources in the network. Usually, policies are coded in imperative programming language such as C/C++ or Java. They are embodied into the system's physical implementation. This makes for implementation ease and efficiency but limits what can be done with policies. For instance, it is difficult to modify, verify or analyze such policies. Policy-based network management uses declarative policy definition language to describe policies that provides some degree of management abstract. Upon this abstract, administrator can specify the desired behavior of management system by policies. Through translation and verification, policies are distributed to policy servers to interpret and execute. Administrator can add, delete or modify existing policies dynamically. Policy-based network management supports administrator to manage distributed systems in a flexible and dynamic way. It has gain increasingly used [2].

Policy definition language for network management usually adopts ECA rules. These rules indicate that when an event occurs, if some condition is true then execute the assigned action [3]. Policy defines a mapping function from event set to action set. Using this policy definition language, it makes for system's implementation ease. But execution of ECA rule is stateless. Policy server can't express system's state transitions. Nor can policy server analyze system's state to perform suitable actions. ECA rule doesn't support policy servers to communicate in order for them to cooperate efficiently. As policy definition language, ECA rules provide management abstract with very simple function. Most analysis and decision functions are yet encapsulated into physical codes.

This paper is motivated to present a new policy definition language that has stronger expressive and computing power. The management abstraction that the new policy definition language provides has stronger functions. Upon this abstract, policy server has a state repository to record system's state. When receiving message, policy server analyzes and modifies the state repository based on administrator's predefined policy. Then policy server generates a sequence of management

O. Festor and A. Pras (Eds.) : 12th International Worshop on Distributed Systems: Operations and Management DSOM'2001 Nancy France, October 15-17, 2001.

operations to send to suitable management modules to execute, and/or generates messages to transfer to other policy servers in order to enable them to work cooperatively.

We design a logic-based policy definition language (LPDL) that is based on first order logic and adopts Prolog's syntax. Administrator can use LPDL to define a set of reaction rules whose format is (<Event>, <Reaction>). Messages to a policy server can come from management modules, which are responsible for monitoring and controlling network's running, or from other policy servers. In response to these messages, the policy server generates appropriate communication events, then searches local reaction rules set, analyzes the state repository and may generate an internal event, send messages to management modules and/or other policy servers, and record state changes into state repository in order to handle with these communication events. The syntax of reaction rules encompasses two types of events, communication and internal event, and three types of actions, "data action" to manipulate state repository, "event action" to generate an internal event and "do action" to generate a message to be sent to management module or other policy server.

LPDL has expressive power of Petri Net by modeling a Petri Net with a set of reaction rules. And LPDL is Turing-equivalent by modeling a register machine as a set of reaction rules. Administrator can use LPDL to describe any analysis and decision functions in network management system. They can decide to define these functions in LPDL or imperative languages according to function's attributes and actual requirements.

The plan of the paper is as follows. In section 2, the syntax, execution model and formal semantic of LPDL are defined. In section 3, we prove that LPDL has expressive power as Petri Net and computing power as Turing Machine. In section 4, the LPDL-based network management model, its prototype implementation and application are presented. In section 5, we present introduction of related works and conclusion of this paper.

2. Definition of LPDL

2.1. Syntax of LPDL

LPDL adopts the typical syntactic conventions of logic language. The alphabet of LPDL is based on the set of the variables Γ, of the function symbols Σ and of the predicate symbols Π. We denote with σ the set of the terms built from Σ and Γ, and with γ the set of the ground terms built from Σ. E is the set of the atomic formulae built applying predicate symbols of Π to terms of σ, and E_γ is the set of ground atomic formulae built applying predicate symbols of Π to ground terms of γ. Given $t \in \gamma$ and $t' \in \sigma$, M (t, t'):= $\exists\theta$ variable substitution, θ = mgu(t, t').

Each management module and policy server is denoted by a ground term. Management module monitors and controls network's running. When a predefined condition is true, it sends a message to its assigned policy server that will generates a communication event. The format of communication

<RuleSet>	::= {<Rule>}
<Rule>	::= (<Event>, <Reaction>)
<Event>	::= <CommEvent> | <InEvent>)
<CommEvent>	::= on(<Term>, <Term>)
<InEvent>	::= on(<Term>)
<Reaction>	::= <Action>{, <Action>}
<Action>	::= <DataAction>(<Term>) | <EventAction>(<Term>) | <DoAction>(<Term>, <Term>)
<DataAction>	::= out | in | rd | no
<EventAction>	::= post
<DoAction>	::= do

Table 1 Reaction Rule Syntax of LPDL

event is on (s, t) $\in E_\gamma$ where s represents the management modules sending the message and t represents the message. Reaction rule syntax of LPDL is showed in Table 1. After generating the communication event on(s, t), policy server searches local reaction rule set, which is denoted as P, to find whether there are reaction rules to handle with the communication event. The searching process can be encapsulated by a function Z that maps from communication events to the set of reactive actions waiting to execute. $Z_P(on(s, t))=\{<Reaction>\theta\theta' \mid (on(s', t'), <Reaction>)\in P\wedge (\theta=mgu(s, s'))\wedge(\theta' =mgu(t, t')\}$. Reactive action is a sequence of predicates that is obtained by applying variable substitution to <Reaction> part of matching reaction rule.

If $Z_P(on(s, t))$ is null, policy server will not do any manipulation and wait for the next communication event. Otherwise, policy server will interpret and execute predicates sequentially of each reactive action that function Z_P returns. The predicates of <DataAction> type can retrieve and modify policy server's state repository. State repository is multi-set of ground terms. Out(t) inserts a ground term t into state repository. in(t) deletes a ground term t' from state repository given M(t, t') is true. rd(t) retrieves a ground t' from state repository given M(t, t') is true. no(t) will execute successfully if there is no ground term matching t in state repository, otherwise the execution of no(t) will fail. Predicate post(<Term>) will generate an internal event on(<Term>). Reaction rules can be defined to handle with internal events. Similarly, there is a function $Z_P(on(t))=\{<Reaction>\theta \mid (on(t'), <Reaction>)\in P\wedge (\theta=mgu(t, t'))$ to indicate some reactive actions to handle with the internal event.

Policy server maintains a message queue Q. Firstly the queue is empty. Execution of predicate do(s, t) will insert a new record (s, t) into the queue Q. When policy server finishes communication event's processing, to each record (s, t) in queue Q, it will send ground term t to management module or policy server which is denoted as s sequentially based on its order in queue Q. If a policy server receives a message from other policy server, it also generates a communication event and may trigger one or several reactive actions to execute. The message transfer between policy servers enables them to cooperate in a flexible way.

2.2. Execution model and semantic of LPDL

Policy server is a reactive system. Based on communication events and reaction rules, policy server updates system's state and generates output messages. Execution of a set R of reactive actions can be encapsulated by a function $F(T, R)=<T', Q'>$. T is the initial state of state repository, T' is the final state of state repository, Q' is final state of the message queue Q which stores messages to be sent. All modification of system's state is a single step of state transition during policy server executes the reactive actions set of a communication event. Meanwhile, policy server can't handle with any receiving messages.

Several reaction rules can be defined to handle with a communication event in order to perform several relatively independent tasks. The interleaved concurrency of several reactive actions may result system in inconsistent states. In order to avoid using complex concurrency control mechanisms, all reaction rules of a communication event can be assigned relative priorities. It is meaningless to assign relative priorities among reaction rules that handle with different communication events. Policy server will execute reactive actions sequentially based on their priorities. If reactive actions have no explicit priorities, policy server will execute them in random order. So execution of function F(T, R) is a sequent of execution of function $F(T, \{r\})$, $r\in R$. When the execution of function $F(T, \{r\})$ finishes, policy server sends messages to management modules or other policy server based on records of message queue Q and reset it empty.

During a reactive action executes, the execution of predicate post will generate an internal event which may have one or several matching reaction rules. So the internal event triggers a new set R1 of reactive actions waiting to execute. Before the execution of R1 finishes, one reactive action of R1 may also include predicate post that triggers another set R2 of reactive actions to execute. G denotes a sequence of set of reactive actions waiting to execute. G is orderly set. Every time, policy server chooses a set of reactive actions that is the smallest element of G to execute. If its execution triggers new internal event and generates new set of reactive actions, policy server adds the new set of reactive actions into G and makes it the biggest element. After executing a set of reactive actions, policy server chooses the smallest element of G to execute.

The set of reactive actions may include several reactive actions that are triggered by same one internal event. Their execution order is arbitrary. If system has explicit requirement on the execution order of two reactive action r1 and r2, administrator can define two reaction rules whose <Reaction> parts are corresponding to r1 and r2 respectively. Then two post predicates are included into reactive action to trigger two different internal events to generate reactive action set {r1} and {r2}. So {r1} and {r2} are elements of G and they have explicit execution order.

So the execution of function F(T, {r}) is a sequence of state transitions, each representing the execution of a reactive action. The execution of a reactive action is encapsulated by a function FE(r, <T, Q, G>)=<T', Q', G'>. T', Q' and G' represent new state of state repository, message queue and set of reactive actions waiting to execute respectively. The evolution of the execution of function F(t, {r}) can be described in terms of a sequence of triples <T, Q, G>. When G is empty, the execution of function F(T, {r}) finishes.

A reactive action is orderly set of predicates. The execution of function FE(r, <T, Q, G>) is process that policy server interprets and executes predicates of reactive action sequentially based on their order. The initial state is denoted as <r, T, Q, G-{r}>. The final state is <r', T', Q', G'>. The semantic of main LPDL predicates is presented in Table 2. The execution of reactive action has transaction semantic. If the r' isn't empty, it shows that the smallest predicate of the reactive action can't execute successfully. Policy server will return to initial state <T, Q, G-{R}> just as the reactive action r doesn't execute. Otherwise policy server's state is <T', Q', G'>. Then policy server continues to execute next reactive action that is in the smallest element of G'.

$<(out(t), r'), T, Q, G> \rightarrow <(r'), T \oplus \{t\}, Q, G>$	
$<(in(t), r'), T \oplus \{t'\}, Q, G> \rightarrow <(r'\theta), T, Q, G>$	where M(t, t') is true and θ=mgu(t, t)
$<(rd(t), r'), T \oplus \{t'\}, Q, G> \rightarrow <(r' \theta), T \oplus \{t'\}, Q, G>$	where M(t, t') is true and θ=mgu(t, t)
$<(no(t), r'), T, Q, G> \rightarrow <(r'), T, Q, G>$	where M(t, t') is false for every t' in T
$<(post(t), r'), T, Q, G> \rightarrow <(r'), T, Q, G \oplus \{Z_P(on(t))\}>$	
$<(do(s, t), r'), T, Q, G> \rightarrow <(r'), T, Q \oplus \{(s, t)\}, G)>$	

Table 2 semantic of main LPDL predicates

3. Expressive and Computing Power of LPDL

3.1. Expressive power of LPDL

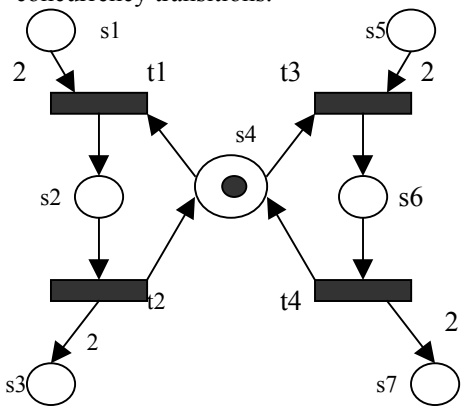
Policy server should have ability to describe state transitions of distributed system. Petri Net is one of the most used formalisms for the description and specification of concurrent and distributed system. We can show that the behavior of any Petri Net can be modeled by a set of reaction rules using LPDL.

Since the state of a Petri Net is represented by the tokens at each place, a collect of ground terms place(s, n) are stored in state repository to record the number of tokens at each place. n is integer number which is represented using a typical functional notation. So 0 represents zero, and if N denotes an integer, then s(N) denotes its successor. For instance, s(s(0)) is 2.

The behavior of a Petri Net is defined by its transitions. Whether a transition is enabled to fire is only determined by the states of its input and output places. Each transition can be modeled by a single reaction rule. At first the reaction rule retrieves current state of all input and output places of the transition t. It then checks whether the number of tokens at each input place s is bigger than the weight of the flow relation <s, t> and whether the number of tokens at each output place s' plus the weight of the flow relation <t, s'> is smaller than the capability of the place s'. Finally, it updates the number of tokens at all input and output places of transition t.

The corresponding reaction rule's event definition of a Petri Net's transitions is on(place(_)) where "_" represents any ground term. If a reactive action includes predicate post(place(_)), it will trigger internal event on(place(_)) and generate a set of reactive actions including <Reaction> parts of all

reaction rules whose event definition is on(place(_)). Policy server will execute reactive actions in the set sequentially in random order. A simple Petri Net can be represented by reaction rules by LPDL as Figure 1 illustrated. Since the execution of a reactive action has transaction semantic, policy server will cancel all system's modification of a reactive action if its execution fails and continue to execute next reactive action. Under some system's state, several transitions' fired conditions are satisfied. Policy server will select one arbitrarily to execute which represents competitive relation among concurrency transitions.



Initial state: place(s1, 0), place(s2, 0), place(s3, 0), place(s4, s(0)), place(s5, 0), place(s6, 0), place(s7, 0)

t1: (on(place(_)), (in(s1, s(s(n1))), in(s4, s(n4)), in(s2, n2), out(s1, n1), out(s4, n4), out(s2, s(n2))))

t2: (on(place(_)), (in(s2, s(n2)), in(s3, n3), in(s4, n4), out(s2, n2), out(s3, s(s(n3))), out(s4, s(n4))))

t3: (on(place(_)), (in(s5, s(s(n5))), in(s4, s(n4)), in(s6, n6), out(s5, n5), out(s4, n4), out(s6, s(n6))))

t4: (on(place(_)), (in(s6, s(n6)), in(s4, n4), in(s7, n7), out(s6, n6), out(s7, s(s(n7))), out(s4, s(n4))))

Figure 1 A simple Petri Net's LPDL description

3.2. Turing-equivalence of LPDL

Register machine is defined as a triple <R, L, S>, where R is a finite set of registers used to store arbitrarily large integer numbers, L is a set of labels used to denote statements, and S is a set of statements representing the machine's program. It has been proved that register machine is Turing-equivalent if S contains the statements inc(R_i), which increments the value of register R_i, dec(R_i), which decrements the value of register R_i provided that it is greater than zero, and jmp(R_i, label), which jumps to the statement denoted by label if the content of register R_i is zero [4, 5]. Each statement of S can be represented as the form label: statement, next_label where label identifies the statement whose execution has to be followed by that of the statement denoted by next_label. Two following forms of statements suffice to represent any recursive function [5]: 1) label: inc(R_i), next_label; 2) label: [if $R_i = 0$] jmp(R_i, new_label), [else] dec(R_i), next_label.

We can show the Turing-equivalence of the LPDL by modeling a register machine as a set of reaction rules to fully capture these two instruction schemes. For any register machine <R, L, S>:

a) Define a ground term reg(R_i) representing register R_i and store a ground term value(reg(R_i), N_i) in state repository for every register $R_i \in R$, where N_i is the integer value of register R_i represented as an s(s(…s(0)…)).

b) Define a ground term label(L_j) for every label $L_j \in L$.

c) Store a ground term statement(label(L_i), Stmt, label(L_j)) for every statement $s \in S$ in state repository, where Stmt is inc(reg(R_i)) or jmpdec(reg(R_i), label(L_j)) representing one of the two instruction schemes.

d) Define a reaction rule for each one of the possible statements to be executed:

inc : (on(exec(L)), (rd(statement(L, inc(Reg), NextL)), in(value(Reg, N)), out(value(Reg, s(N))), post(exec(NextL))))

jmp: (on(exec(L)), (rd(statement(L, jmpdec(Reg, JumpL), _)), rd(value(Reg, 0)), post(exec(JumpL))))

dec : (on(exec(L)), (rd(statement(L, jmpdec(Reg, _), NextL)), in(value(Reg, s(N))), out(value(Reg, N)), post(exec(NextL))))

The execution of predicate post(exec(StartL)) will generate a internal event on(exec(StartL)) and trigger policy server to execute the program whose first statement is denoted by StartL. Three reaction rules have same event definition that indicates that there are three concurrent reactive actions

to handle with the internal event. But only one reactive action can execute successfully. According to the transaction semantic of reactive action, the two failure reactive actions don't affect the execution of the successful reactive action. Using the process method, LPDL can perform all statements of register machine. This proves that LPDL is Turing-powerful.

4. LPDL-based network management framework

LPDL-based network management framework is illustrated as Figure 2. In a domain, one or several monitor modules can be deployed to monitor the managed devices' operation. When a predefined condition is true, monitor module will send message to policy server. Policy server will generate communication event, handle with the communication event based on local reaction rules, and generate message to control modules. Control module interprets the message from policy server and adopts appropriate management actions. Policy server may also generate messages to other policy server in order to cooperative management among several domains. Since LPDL is Turing-power and be able to perform any analysis and decision functions of management system. In order to enhance policy server's performance, some static analysis and decision functions that need much computing power can still be encapsulated by analysis & decision module which is implemented using C/C++, Java etc. Some analysis and decision functions that can be changed frequently as the system's running are defined using LPDL. During management system's running, administrator can modify policy server's reaction rules by management console in flexible and dynamic way.

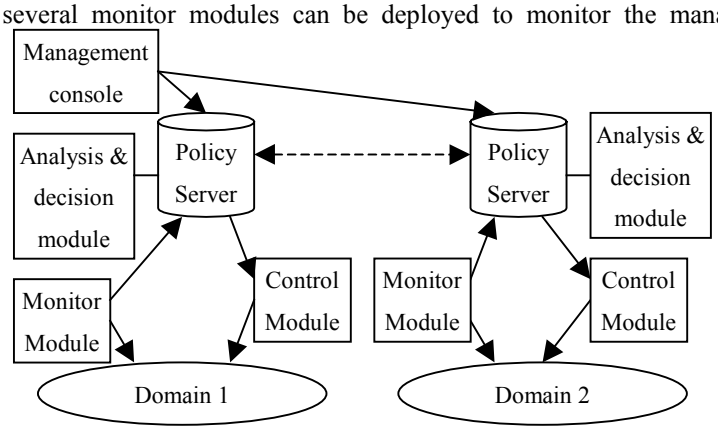


Figure 2 LPDL-based Network Management Framework

We use JDK1.2 to implement the prototype of the management framework on CORBA platform [6]. CORBA is the JavaIDL provided by JDK1.2. The objects in a domain use ORB to communication. The policy servers in different domains use IIOP protocol to communication. In order to support to add new control functions dynamically, object managers can be deployed in appropriate locations. The object manager can receive control object's Java bytecode that is created by administrator dynamically according to system's requirements. Object manager then builds the control object's instance and register it on CORBA's naming service using name that administrator defines. In order to monitor devices flexibly, we build a general monitor object GMO that can monitor three types of events: timer event, notification event and poll event. GMO provides timers to administrator to set. Once it's the time that administrator has set, GMO will generate a timer event. GMO can receive abnormal notification from other components, such as Trap message in SNMP protocol, and transform it to notification event based on event definition. GMO can also receive Boolean expressions about status of devices or components from administrator. It accesses state values of related components and calculates the expression based on these state values periodically. If the expression is true, GMO will generate a predefined poll event. When GMO generates an event, it will invoke policy server's method to transfer the event to it. Monitor objects and control objects access the managed devices via object adapter. We use the SNMP protocol stack provided by Advent Net Company to implement a SNMP adapter to enable management system to manage devices supporting SNMP.

We use the management framework to manage a network environment. The environment is comprised of two domains that are connected by two border routers. There are two communication links between the two routers. When traffic between the routers is light, system build a connection on link1 through interface1 of the routers. When the link1's utility is greater than 60%, system will build

another connection on link2 through interface2 to increase communication bandwidth between the two routers.

Firstly, a poll event Event(Router1, IF1, overload) is set on GMO1 of domain 1 to monitor the utility of router1's interface1 whose expression is $\frac{(\Delta(ifInOctets)+\Delta(ifOutOctets))\times 8}{\Delta(sysUpTime)\times ifSpeed} > 0.6$. Then administrator creates two control object R1 and R2 and invokes object manager of each domain to build the two objects' instances which provides methods to manage the two routers, for instance initialize port, create connection and close connection. Finally, administrator defines the reaction rules of policy server PS1 and PS2 of two domains.

To PS1, reaction rules are defined as followed:

```
//receive message from GMO1 that router1's interface 1 overloads
(on(GMO1, Event(Router1, IF1, overload)), (no(state(Router1, IF2, busy)), out(waitfor(Router2, IF2, ok)),
                    do(PS2, Event(Router1, IF1, overload)) ))
//receive message from PS2 that the router2 has initialize successfully
(on(PS2, state(Router2, IF2, ok)), (in(waitfor(Router2, IF2, ok)), do(R1, exec(initial, IF2)), do(R1,
                    exec(connect, IF2, Router2, IF2)), out(state(Router1, IF2, busy)) ))
```

To PS2, a reaction rule is defined as followed:

```
//receive message from PS1 that router1's interface 1 overloads
(on(TS1, Event(Router1, IF1, overload)), (no(state(Router2, IF2, busy)), do(R2, exec(initial, IF2)),
                    do(PS1, state(Router2, IF2, ok)), out(state(Router2, IF2, busy))) )
```

Administrator can add reaction rules of policy servers to regulate the two border routers to disconnect the link2 when the traffic between them is lighter than a predefined value similarly.

5. Related works and conclusion

Policy-based network management provides bridge between management goals and management actions [7], which has much research from academia and industry. The joint work of IETF and DMTF focuses on the policy's definition and storage [8, 9]. They propose to use CIM information model and LDAP directory service to describe and store policy. But they don't define the policy server's process procedure or the expressive and computing power of policy definition language. E.Lupu proposed a role-based network management model [10]. A role is a set of policies that define manager's duty and obligation. The model includes relations that are sets of policies that define communication protocols between roles to enable managers to cooperate. But the roles and relations are ECA rules whose executions are stateless. Policy's expressiveness is weak. Role and relation must be defined respectively. And the messages between roles must include the state information about the current interaction. The definitions of policies are very complex. R.Bhatia and J.Lobo use composite event and complex event to enable policy server to adopt appropriate actions based on event history [11]. It makes policy server to express and store system's state. But this mechanism's expressive power is too weak to describe state transitions of distributed system completely. Policy server hasn't analysis and decision ability either.

In order to enhance the expressive and computing power of policy definition language and support several policy servers to cooperate flexibly, we use some concepts from programming coordination model [12]. In a programming coordination model, tuple space not only provides a sharing dataspace for coordination entities, but also embodies and enforces the interaction policies among coordination entities. Many implementations of coordination models and languages use logic-based language to define the interaction policies into tuple space [13, 14]. Programming coordination model and policy-based network management are both reactive systems that react to their receiving messages based on local reaction rules.

According to the requirements of policy-based network management, we design and implement a logic-based policy definition language LPDL. In this paper, the syntax, semantic and execution model of LPDL are represented. LPDL is proved to have Petri Net's expressive power and Turning Machine's computing power. Administrator can use LPDL to describe system's analysis and decision functions or encapsulate them into physical codes flexibly according to actual need. LPDL can meet the requirement of network dynamic growth. Finally, the prototype implementation of LPDL-based network management framework is introduced. Some questions need to be studied furthermore, for instance the verification of LPDL and the performance of policy server.

6. References

[1] R.Wies, Policies in network and system management – formal definition and architecture, Journal of Network and System Management, 2(1): 63-83, 1994

[2] G.Stone, B.Lundy, G.Xie, Network policy languages: a survey and a new approach, IEEE Network, January/February, 2001.

[3] J.Widom, S.Ceri, Active Database Systems, Morgan-Kaufmann, 1995.

[4] J.Shepherdson, H.Sturgis, Computability of Recursive Functions, Journal of the ACM, 10:217-255, 1963

[5] E.Denti, A.Natali, A.Ominici, On the expressive power of a language for programming coordination media. In SAC 1998[30], pages 169-177, Track on Coordination Models, Languages and Applications.

[6] Li Yongxin, Chen Ming, etc, Study and implementation of a collaborative network management framework, accepted by IEEE Milcom2001.

[7] M Sloman, Network and Distributed System Management, Addison-Wesley Publisher Ltd, 1994

[8] J.Strassner, N.Ellesson, B.Moore, Eds., "Policy Framework Core Information Model", Internet draft draft-ieft-policy-core-schema-02.txt, Feb, 1999

[9] J.Strassner, S.Schleimer, "Policy Framework Definition Language", Internet draft draft-ietf-policy-framework-pfdl-00.txt, 17 Nov 1998.

[10] Emil Lupu, A Role-based Framework for Distributed System Management[Ph.D Thesis], University of London, 1998.7

[11] J.Lobo, R.Bhatia, S.Naqvi, A policy description language, In Proc. Of AAAI, Orlando, FL, July 1999.

[12] E.Denti, A.Natali, A.Ominici, Programmable coordination media, In Coordination Languages and Models, pages 274-288, Second International Conference COORDINATION'97, Berlin, September, 1997.

[13] A.Omicini, F.Zambonellli, Coordination of Mobile Information Agents in TuCSon, Internet Research, Vol.8, No.5, pp.400-413, 1998

[14] N.Carriero, D.Gelernter, Coordination Languages and their Significance, Communications of the ACM 35(2), 1992, pp.97-107.