Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

A Logical Reasoning Architecture for Computer Network Management

Alessandra De Paola, Luca Gatani, Giuseppe Lo Re, Alessia Pizzitola, Alfonso Urso

RT-ICAR-PA-04-02 **gennaio 2004**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

A Logical Reasoning Architecture for Computer Network Management

Alessandra De Paola[2], Luca Gatani[1,2], Giuseppe Lo Re[1], Alessia Pizzitola[2], Alfonso Urso[2]

Rapporto Tecnico N. 2: **RT-ICAR-PA-04-02**

Data: **gennaio 2004**

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sezione di Palermo, Viale delle Scienze edificio 11, 90128 Palermo.
[2] Università degli Studi di Palermo, Dipartimento di Ingegneria Informatica, Viale delle Scienze, Edificio 6, 90128 Palermo.

Contents

Abstract

Today, with increasing Internet complexity, Network Management [1, 2, 3] becomes a complex activity requiring the human intervention to create action management plans, to coordinate network assets and to face fault situations. Network Management involves several functionalities grouped together in five areas: fault, performance, configuration, accounting and security. Traditional Network Management applications are coordinated by a central management entity, as in the Simple Network Management Protocol (SNMP) [4, 5], and very often the managing entity consists of a simple interface to a human system administrator. Recently, some authors [6, 7] have proposed a knowledge-based approach to face Network Management problems. These works propose the adoption of a higher level Knowledge Representation. According to these works, our project proposes the improvement of computer network management by the adoption of artificial intelligence techniques [8].

A Logical Reasoner acts as an external managing entity capable of directing, coordinating and stimulating actions in an active management architecture [9]. Our system is based on the adoption of an active network framework, extending the distributed architecture proposed in [10] In order to capture network events, the Reasoner deploys programmable sensors on active nodes. Active networks enable network dynamic programming and allow an easy deployment of "ad hoc" solutions. The Logical Reasoner is based on the logical formalism provided by the Situation Calculus [11, 12] a logic language specifically designed for representing dynamically changing world. To implement the Reasoner, we adopted the Reactive Golog language, a specification of the Situation Calculus designed to model reactive behavior.

The remainder of this work is structured as follows. Chapter 1 introduces how the Situation Calculus, and in particular the Reactive Golog implementation, can be adopted to model dynamism of network events. Chapter 2 illustrates the Network Management architecture in which the logical Reasoner interacts with other network components; a high level description of various agents that compose the Reasoner is also presented, and section 2.1 describes the communication role of the Gateway, a fundamental component of the architecture. Chapter 3 describes how the network, in its static and dynamic aspects, is represented inside Reasoner agents. In chapter 4 are explained reasoning ways of principal Reasoner agents. Finally, in chapter 5 the Global database, used to store Reasoner inference results, is described.

Chapter 1

Situation Calculus

Our Logical Reasoning system finds its theoretical foundation in a second order logical language called Situation Calculus[11]. In the Situation Calculus representation, every domain changing is seen as result of an action. A world history is represented by a first logic term called situation, a simple sequence of actions.

Situation Calculus uses two terms to indicate situations:

S_0, denotes the initial situation;

$do(A, S)$, denotes the situation following from the situation S after the execution of the action A.

Language terms referred to world entities are objects. Relations whose truth-values may vary in different situations are called relational fluents. They are represented by means of predicate symbols which take a situation term as their last argument. The causal law between an action and the consequent change of a fluent value is expressed by an instance of the successor state axiom. For example, we can consider a simple fluent showing if a network node is on:

$node\_on(X, S)$. If this term is true, the node X is on in the situation S.

Actions that affect this fluent are:

$node\_up(X)$, denotes turning on the node X;
$node\_down(X)$, denotes turning off the node X.

The successor state axioms provide a complete description of the fluent evolution in response to primitive actions. They are needed for each predicate that may change its truth value over the time. The successor state axiom for *node_on* fluent is:

$node\_on(X, do(A, S)) :- A = node\_up(X)$.
$node\_on(X, do(A, S)) :- node\_on(X, S),\ not\ A = node\_down(X)$.

This axiom indicates that the node X becomes *on* after the execution of action $node\_up(X)$, or after the execution of every action different from the action $node\_down(X)$, if the node X is *on* in the last situation. Primitive action preconditions are rules describing when actions can be carried out given a state of the world. The preconditions are stated by fluents. For example action preconditions for actions $node\_up(X)$ and $node\_down(X)$ may be:

$Poss(node\_down(X), S) :- node\_on(X, S)$.

It indicates that the execution of the $node\_up(X)$ action is possible in a situation in which node X is not *on*.

$Poss(node\_up(X), S) :- not\ node\_on(X, S)$.

It indicates that the execution of the $node\_down(X)$ action is possible in a situation in which node X is *on*.

1.1 Reactive Golog

In order to implement the system we adopted the Reactive Golog[12] (RGolog) language as the specific reasoning environment. RGolog, is a language planned for modelling reactive systems, implemented in Prolog language. The formalization of the world in the RGolog is performed through well formed formulas of the first order logic, while the dynamism is captured through the primitive concepts of state, primitive action and fluent. We can think the state as a snapshot of the world at a determined moment. All changes to the world can be seen as the result of some primitive actions. Relations whose truth-values may vary in different situations are called relational fluents. They are represented by means of predicate symbols, which take a situation term as their last argument. Primitive actions preconditions are rules that describe when actions can be carried out given a state of the world. The preconditions are stated by fluents. The successor state axioms provide a complete description about the fluents evolution in response to primitive actions. They are needed for each predicate that may change its truth-value over the time. Procedures represent the complex actions and constitute one of the most important features of the Reactive Golog. They allow to group long sequences of primitive actions and to implement recursive formulas. Like in the imperative languages, they use formal parameters. Generally, dynamic systems are not totally isolated by the rest of the world, but they continually receive solicitations and they interact with the external world. The Reactive Golog rules allow these interactions describing how the world evolves when an external action is performed. This aspect is the so-called "reactive behaviour".

In Reactive Golog there are two types of actions: primitive actions, executed by the system, and exogenous actions, executed by the external world. Therefore, there are two kinds of interaction between the system and the external world: the system changes the world by its actions, and the external world influences the system behaviour by its exogenous actions. A primitive action has an explicit identification by the term *primitive_action*. For instance for the

action *node_up* and *node_down* we can have:

$primitive\_action(node\_up(X))$.
$primitive\_action(node\_down(X))$.

The insertion of exogenous actions into the system is realized by a Prolog rule, whose head is

$exoTransition(S, S1)$.

The situation S1, returned by this rules, follows the situation S after the execution of an exogenous action. Reactive system behaviour is implemented by a concurrent interleaving of a control procedure with a procedure for interrupt management. For convention this procedure is called **rules**. The implementation of RGolog interpreter implies that a primitive action execution triggers the introduction of an exogenous action and the execution of the **rules** procedure. The operator that returns the situation obtained from another situation after an action execution is doR operator, defined by the following rule:

$$doR(A, Rules, S, S1) : -primitive\_action(A), poss(A, S), \\ exoTransition(do(A, S), S2), doR(Rules, Rules, S2, S1).$$

The `doR` operator allows the execution also of complex actions. For example we report main complex actions used in the system implementation:

$doR(E, Rules, S, S1)$	executes of the action or the procedure E;
$doR(?(P), Rules, S, S1)$	tests the truth value of expression P;
$doR(E1 : E2, Rules, S, S1)$	executes actions E1 and E2 sequentially;
$doR(E1\#E2, Rules, S, S1)$	executes E1 or E2 indifferently;
$doR(if(P, E1, E2), Rules, S, S1)$	executes E1 if the expression P is true, else executes E2;
$doR(while(P, E), Rules, S, S1)$	executes E more times while the condition P became false.

The expression that defines a procedure is:

$proc(procedure\_name(param\_1, param\_2, .., param\_n), actions)$.

Chapter 2

System Architecture

In this work a logical reasoning system is proposed as a Knowledge based network management entity. Figure 2.1 shows the system architecture.

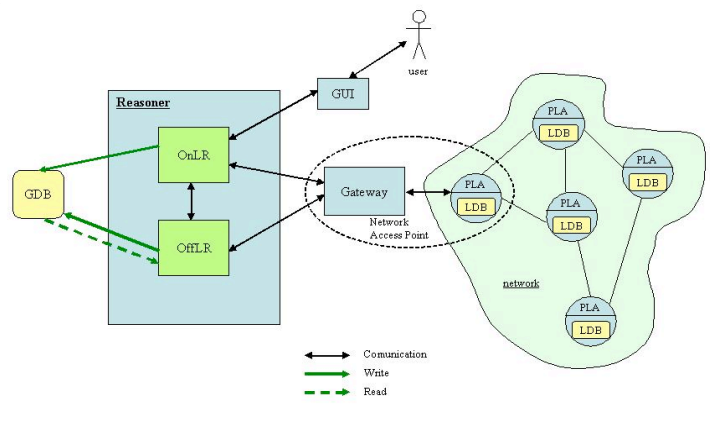

Figure 2.1: System Architecture

The whole architecture is based on the active network emerging technology. By means of active network programmability it is possible to distribute code over all the network, thus obtaining highly flexible management services that can be tuned remotely by the logical reasoning system (Reasoner). Moreover, network programmability enables the distribution of data, reducing the amount of traffic related to management activities and also the Reasoner computational load. The Reasoner sends request to the active network and receives information from it, through a network access point (Gateway). The Gateway provides a translation service to the Reasoner, offering it a transparent access to the network. This way, the Reasoner implementation is fully independent of the language adopted by the active network execution environment.

A local management agent, Programmable Local Agent (PLA), is resident on each active node. Programmability is the most important characteristic of an active node. Namely, it is possible to change dynamically the management services offered by each node, by sending opportune messages called active cap-

sules. Using these capsules, the Reasoner can distribute code over the whole network, obtaining a very dynamic and flexible management tool.

PLAs, besides answering Reasoner requests, may perform asynchronous actions whenever some events occur. This event-response mechanism is implemented by means of a particular structure of the management objects contained in the node MIB (Management Information Base). These objects are not simple variables, but complex structures containing code fragments. Because of this characteristic the Information Base is no longer called MIB, but AMIB (Active Management Information Base) to indicate active archives of information.

The capability of sending opportune messages, which can set actions that a PLA must perform when an event occurs, makes possible the introduction into the network of new behaviors or the tuning of some existing ones. Many sensors can be installed on a PLA. We can imagine a sensor as an entity that captures network events. Data obtained by sensor activity may be notified to the Reasoner to support reactive behavior, or recorded in local Data Bases (LDB) maintained on each active node. Stored information can be used by the Reasoner to analyze network history. Furthermore, each PLA provides a set of parametric sensors, i.e. sensors for which it is allowed the tuning of sampling or recording frequencies. Active network capabilities are also used to distribute code over the network in order to implement distributed management services. An example of distributed services is the service capable of loop discovering.

Fundamentally, reasoning activities can be carried out in two different ways: on-line and off-line. The two reasoning processes can be executed as independent tasks, since they are quite dissimilar and exploit different network representations. For these reasons, two separate blocks compose our logical system: the On-Line Reasoner (OnLR) and the Off-Line Reasoner (OffLR).

The two reasoning components communicate with the network in different ways because of their different goals. OnLR is responsible for reactive behavior and uses information notified by sensors in order to maintain an always-updated network representation and to gather events that can be interpreted as malfunction symptoms. It sends messages in the network to set management services and to actively interfere, modifying network behaviors. Results achieved by OnLR reasoning are also stored in a Global Data Base (GDB).

In order to perform complex "a posteriori" analysis of network functioning, the OffLR exploits information stored in both GDB and LDB. Its communication with the active network is devoted only to obtain information about past events, and its behavior can be considered static. However, also the results obtained by OffLR are stored in the GDB. As an example we provide the following scenario of Reasoner intervention. A user, typically a human supervisor, can send queries to the Reasoner through a GUI, in order to reconstruct the whole network state during a given time interval, or to examine events happened in a limited network area. OnLR manages these queries exploiting specific OffLR capabilities. A more detailed analysis reveals the modular composition both of the OnLR and of the OffLR.

OnLR is composed by three modules or agents:

OnLR_Core,

OnLR_TrafficMonitor,

QueryHandler.

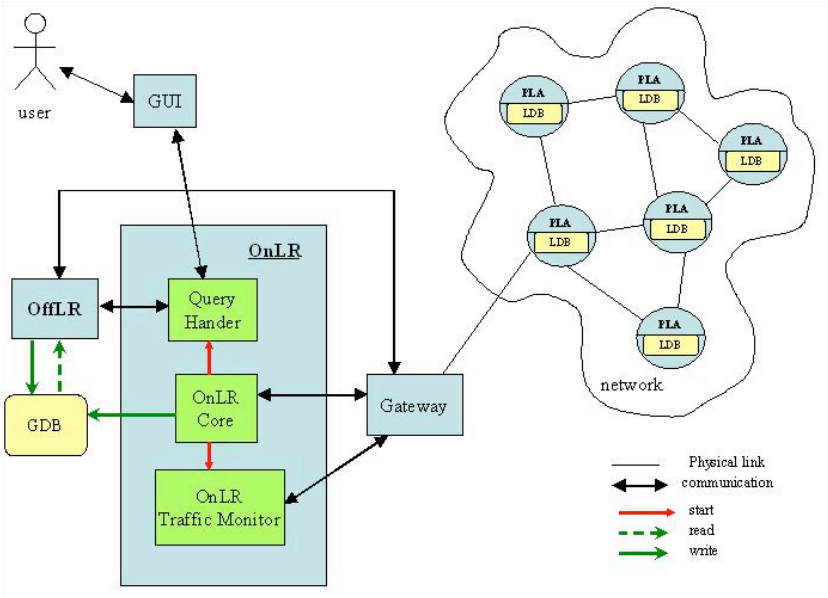

Figure 2.2: OnLR modular structure

Figure 2.2 shows the OnLR structure.
OnLR_Core agent is responsible of fault management, OnLR_Traffic-Monitor agent is responsible of traffic sensors tuning in relation to traffic condition, and QueryHandler agent answers users queries using OffLR capabilities. Since their different nature, OnLR agents were implemented by using different tools, in relation with their requirements. OnLR_Core and OnLR_Traffic-Monitor agents constitute the reactive part of the system, and are implemented using the RGolog language. Differently from previous agents, the QueryHandler, since it does not need to exhibit a reactive behavior, it is implemented in Prolog language. Furthermore, it does not communicate with the gateway but only with the GUI, in order to receive user queries, and with the OffLR, in order to answer queries.

OffLR is composed by various agents specifically designed to manage particular kinds of query. These agents have different reasoning capabilities, each of which is devoted to a specific task, or a particular functioning aspect to be reconstructed or examined. However, each agent gathers only information necessary to it, and it is specialized in performing only a kind of reasoning.
Currently, OffLR agents are:

$OffLR\_SearchOnPath$	It searches for causes of faults occurred while an end-to-end connection. It uses past inference results contained in the GDB.If necessary, it starts the OffLR_SearchLDB agent.
$OffLR\_RestoreNetStatus$	It reconstructs the network functioning history during a given time interval. It uses LDB information and, if necessary, it starts the OffLR_Disconnection agent.

$OffLR\_SearchLDB$	It searches for fault causes on a single path during a given time interval. It uses LDB information and the OffLR_Disconnection.
$OffLR\_Disconnection$	It reconstructs the node status history in order to verify network connectivity.

2.1 Gateway Communication Services

The communication between the Reasoner and the network agents is mediated by a Gateway service. Fundamentally, the Gateway offers a translation service both to the Reasoner and to network PLAs. It is a server listening to network and to Reasoner agents. In order to manage these two different kinds of communication, it adopts independent threads. For each Reasoner agent, the Gateway forks a thread connected to a correspondent one on the network side.

The communications Reasoner-Gateway and vice versa are implemented on two different channels. The Reasoner sends XML messages to the Gateway through a port on which this is listening as a server, while the Gateway sends Prolog predicates to the Reasoner through an apposite communication stream.

Last form of communication is implemented by means of libraries that enable a Java process to interface a Prolog one (on which the Reasoner agent is running). The interprocess communication libraries make the Prolog and Java processes parts of a single control thread, so, when one of this two processes is running, the other must be waiting.

In order to execute the Reasoner's commands, the Gateway operates on two sides: on the former, it operates as a translator, converting XML message originated by the Reasoner in a particular language adopted by the network execution environment; on the latter, it is responsible for injecting into the network the opportune code implementing the Reasoner requests. To this end, the Gateway maintains a list of opportune code fragments bounded to each Reasoner request, thus supplying the Reasoner with a transparent access for different network architectures. Active nodes support active packets (capsules) implemented in different languages such as PLAN, ANTS, and ASP. In our experiments, we adopted the PLAN language to implement services used by the Reasoner. However, the Gateway services allowed the Reasoner development in an independent way from the network language.

At start time, the Reasoner sends a special XML message to the port on which the Gateway is listening, giving the initial signal for the connection setup. The sending of XML messages from Prolog processes is implemented by means of C external predicates. The Gateway, at the reception of initial signal, creates an interface toward the Reasoner process returning the control to it. The Reasoner can read from the communication stream created at start time or return the control to the Gateway in order to continue its execution. This mechanism allows the introduction of exogenous actions into the OnLR_Core and the OnLR_TrafficMonitor, and that of Prolog Predicates into OffLR agents.

In the following, as an example, we show the Prolog predicate which allows the introduction of exogenous actions in the OnLR_Core.

$exoTransition(S1, S2) : -$
$\quad requestExogenousAction(E, S1),$
$\quad (E = nil, S2 = S1; notE = nil, S2 = do(E, S1)).$

$requestExogenousAction(E, S) : -$
$\quad remote\_yield(peer), read\_exdr(gw\_to\_rnlCore, E1),$
$\quad (\quad (E1 = nil; poss(E1, S)),\ E = E1;$
$\quad\quad not(E1 = nil), not\ poss(E1, S),$
$\quad\quad /* \ Action\ not\ possible \ */$
$\quad\quad requestExogenousAction(E, S)$
$\quad).$

When the Gateway receives an XML message from a Reasoners agent in order to send a request to the network, it will select an active packet and it will inject it into the network.

When the Gateway receives a message from the network, it will produce the introduction of Prolog predicates into the communication stream toward the Reasoner.

The OnLR_Core and the OnLR_TrafficMonitor agents exploit the Gateway services in order to demand the activation or tuning of services to PLAs, and to receive exogenous actions from the network.

Differently from them, the QueryHandler agent does not communicate with the Gateway, since its main task is to wait user queries and to answer them using OffLR reasoning. It is worth to notice that communication between Gateway and OnLR_Core is not limited only to XML requests to PLA, and to exogenous actions. Namely, at start time, the Reasoner asks the Gateway in order to obtain the network topology. After this request, the Gateway collects all topology information sending a special request message to every network node and waiting for their answers. It then will create a series of Prolog predicates representing the static network representation for the Reasoner that will be sent into the communication stream toward the OnLR_Core. The OnLR_Core will be responsible to store these Prolog representation in a topology.pl file, which will be used by each agent needing such static knowledge.

OffLR uses Gateway only to obtain information stored into LDB. These requests are implemented by single XML messages in which a set of parameters specifies some attributes such as the node toward which the request is addressed, the typology of information requested, or the time interval considered. The request is translated by the Gateway in an active capsule that requests the PLA to extract this information from its LDB, using all the log files that cover the whole time interval. The information retrieved is sent to the Gateway, which will translate it into Prolog predicates to insert into the apposite communication stream.

Chapter 3

Network Representation

3.1 Static Representation

OnLR and OffLR adopt a static representation of the network in order to identify the network hardware entities and their physical connections. This simple representation is achieved by means of Prolog predicates that the Gateway asserts on the basis of its knowledge of the network and sends to the OnLR at boot time. The OnLR shares this information with the OffLR. The Prolog predicates used to describe the topology and entities of the network are:

$node(X)$	denote that X is a node;
$link(X)$	denote that X is a link;
$iface(X)$	denote that X is an interface;
$iface\_node(X, Y)$	denote that X is an interface installed on node Y;
$connect(N1, N2, L, I1, I2)$	denote that interface I1, on node N1, and interface I2, on node N2, are connected by link L.

This simple and naive representation can be considered suitable for the OffLR, which does not need to know the network evolution and the occurrence of events. Similar considerations can be, also, applied to the QueryHandler agent, since it does not interoperate with network. On the other hand, OnLR agents need a dynamic network representation in order to continuously maintain an update model of the world and to reason in real-time on it.

3.2 Dynamic Representation

The status of the network can continuously change and the OnLR needs to know such changes and how to represent them. Situation Calculus represents the dynamic aspects of the world by means of a set of fluents, each of which denotes a truth value of a given feature in a particular situation. The OnLR_Core agent and the OnLR_TrafficMonitor agent may adopt different fluents in order to represent the same concept because of their different interests. For instance, the *node_status* fluent maintained by the OnLR_Core agent takes into account a complex status in order to represent both network and system actions. The monitored network can notify a node bootstrap or shutdown. The OnLR_Core

agent can require to turn on or to turn off a node in the observed network. Using this fluent, the OnLR_Core is able to understand if the shutdown of a node was abnormal (it did not send any request) or not. The primitive actions which influence the *node_status* fluent for OnLR_Core are:

$node\_up(N, Time)$	PLA action which notifies an effectively node turning on;
$node\_down(N, Time)$	PLA action which notifies an effectively node turning off;
$command\_node\_up(N)$	OnLR action to demand a node turning on from a PLA;
$command\_node\_down(N)$	OnLR action to demand a node turning off from a PLA;
$restore\_node(N)$	External action to notify that a human operator restored a node normally functioning after a fault occurred.

The *node_status* fluent of the OnLR_Core is shown in the following:

```
node_status(N, on, do(A, S)):-
   node_status(N, on, S), not A=node_down(N, Time),
       not A=command_node_down(N);
   A=node_up(N, Time).
node_status(N, off, do(A, S)):-
   node_status(N, off, S), not A=node_up(N, Time),
       not A=command_node_up(N);
   A=node_down(N, Time), node_status(N, wait_off, S);
   A=restore_node(N).
node_status(N, abn, do(A, S)):-
   node_status(N, abn, S), not A=restore_node(N);
   A=node_down(N, Time), not node_status(N, wait_off, S).
node_status(N, wait_on, do(A, S)):-
   node_status(N, wait_on, S), not A=node_up(N, Time);
   A=command_node_up(N).
node_status(N, wait_off, do(A, S)):-
   node_status(N, wait_off, S), not A=node_down(N, Time);
   A=command_node_down(N).
```

The fluent considers five possible node status values:

- on: indicates the normal node functioning;
- off: indicates that the node was normally turned off, and the possibility to turn it on;
- abn: indicates an abnormal status where the node is out of service and some external action is needed to restore normal conditions;
- wait_on: indicates node off; however the OnLR_Core has required to turn the node on, but it has not yet received any feedback;
- wait_off: indicates node on; however the OnLR_Core has required to turn the node off, but it has not yet received any feedback.

A different version of the *node_status* fluent is maintained by the OnLR_TrafficMonitor agent. This is because OnLR_TrafficMonitor is not able to modify node status, and consequently, it does not model the Reasoner actions. It only needs to know if a node is on or off, as it is shown in the following:

```
node_status(N, on, do(A, S)):-
    node_status(N, on, S), not A=node_down(N, Time);
    A=node_up(N, Time).
node_status(N, off, do(A, S)):-
    node_status(N, off, S), not A=node_up(N, Time);
    A=node_down(N, Time).
```

The OnLR_Core and the OnLR_TrafficMonitor maintain similar fluents to represents the status of some other entities (link, interface). Another fundamental feature represented by both the agents is the status of the network sensors. The following fluent, adopted for this goal, is similar for both agents:

```
sensor_status(N, Type, on, do(A, S)):-
  sensor_status(N, Type, on, S), not A=sensor_down(N, Type, Time),
    not A=command_sensor_down(N, Type);
      /*if the status was on, the command_sensor_up action
        does not cause any change*/
    A=sensor_up(N, Type, Time).

sensor_status(N, Type, off, do(A, S)):-
  sensor_status(N, Type, off, S), not A=sensor_up(N, Type, Time),
    not A=command_sensor_up(N, Type);
      /* if the status was off, the command_sensor_down action
        does not cause any change */
    A=sensor_down(N, Type, Time);
    A=node_down(N, _).

sensor_stas(N, Type, wait_on, do(A, S)):-
  sensor_status(N, Type, wait_on, S), not A=sensor_up(N, Type, Time);
  A=command_sensor_up(N, Type), not sensor_status(N, Type, on, S).
      /*command_sensor_up action sets on the status value,
        only if the status is not on*/

sensor_status(N, Type, wait_off, do(A, S)):-
  sensor_status(N, Type, wait_off, S),
    not A=sensor_down(N, Type, Time);
  A=command_sensor_down(N, Type),
    not sensor_status(N, Type, off, S).
      /* command_sensor_down action sets off the status value,
        only if the status is not off */
```

Previous fluent denotes the status of a particular sensor type installed on a given node. It is influenced by the *sensor_up*, *sensor_down*, *command_sensor_up*, and *command_sensor_down* actions, which have a similar behavior to that of the analogous actions that influence a node status. Command actions for the OnLR_TrafficMonitor are different from these, since for sensors managed by this module it is possible to specify some parametric options, such as the sampling and recording rates. The command action to turn on a sensor is:

`command_sensor_up(N,Type,SamplingRate,RecordRate)`.

Both the agents adopt a quite similar *sensor_status* fluent, alhtough it represents different concepts. For the OnLR_Core agent, the *sensor_status* corresponds to notify a working condition, while for the OnLR_TrafficMonitor, it corresponds only to store a working condition. However, OnLR_TrafficMonitor adopts a different fluent in order to explicitly represent the notification of working condition. Since the OnLR_TrafficMonitor is responsible for the tuning of traffic sensor funcionalities, it maintains a set of fluents which represent sensor parameters, such as recording rate, sampling rate and queue threshold (for queue sensor). These features are modeled by the following fluents:

```
sensor_threshold(N,Type,Thr,do(A,S)):-
        sensor_threshold(N,Type,Thr,S),not A=change_threshold(N,Type,_);
        A=change_threshold(N,Type,Thr);
        A=command_snsNotify_up(N,Type,Thr).

sensor_smplRate(N,Type,SRate,do(A,S)):-
        sensor_smplRate(N,Type,SRate,S), not A=change_smplRate(N,Type,_);
        A=change_smplRate(N,Type,SRate);
        A=command_sensor_up(N,Type,SRate,_).

sensor_recordRate(N,Type,RRate,do(A,S)):-
        sensor_recordRate(N,Type,RRate,S), not A=change_recordRate(N,Type,_);
        A=change_recordRate(N,Type,RRate);
        A=command_sensor_up(N,Type,_,RRate).
```

The actions `change_recordRate`, `change_smplRate`, `change_threshold` modify sensor parameter values. The setting of these values does not require any network feedback.

Chapter 4

Logical Reasoning

4.1 OnLR_Core inferences

The OnLR_Core managing agent observes, collects, and analyzes network events in order to detect abnormal behaviours. It performs this task by means of its reactive behaviour and its capability of interacting with the network. The interaction capability allows the OnLR_Core to exploit the services offered by the programmable networking environment, in order to perform a more in-depth analysis of its current conditions. The OnLR_Core looks for anomaly symptoms that may allow it to infer a deterioration of the network quality of services. For instance, if a network user observes a low quality in its services, perhaps its communication flow is suffering packet losses. On the basis of this information, the OnLR_Core focuses its reasoning on packet losses occurred in the network. In order to analyze packet losses, the reasoner uses a basic set of sensors which is always activated. This basic set is formed by sensors of lost packets and sensors which notify network condition variations. Sensors which reveal packet losses are named `TTL_LostPkt` and `RT_LostPkt`. The `TTL_LostPkt` sensor sends a notification whenever a packet is rejected in a network node because of its zero time-to-live. The `RT_LostPkt` sensor sends a notification whenever it is not possible routing a packet on a network node. This last condition can be caused by the absence of the corresponding entry in the routing table or by an infinite cost of the path. A Packet loss is notified asynchronously by the PLA resident on the node where the event occurs.

The `AliveNeighbor/DeadNeighbor` are two other sensors belonging to the previous set. They reveal node condition variations that they are capable of sensing, by using a mechanism based on the exchange of ICMP messages. Their actions consist, respectively, in the notification of a neighbor status variation (a neighbor previously on is become off, or vice versa). Without their sensing actions, the OnLR_Core agent could not know the status of network nodes or at least it could obtain the same information with a big overhead of message exchange with the Gateway.

The OnLR_Core agent is responsible to start and to stop other specific agents of the reasoning system. However, since all the reasoner agents are completely independent, no communication form between them is implemented.

Furthermore, the OnLR_Core maintains a fluent called *fault* in order to

identify faults which have been notified but not yet analyzed. First parameter of $fault$ fluent allows the identifictaion of the fault type and determines the number and the meaning of the other following parameters.

For instance, events considered faults are a packet loss due to a null TTL or a packet loss due to a routing table corruption. An example of the $Fault$ fluent is shown in the following:

$$
\begin{aligned}
&fault(ttl\_lostPkt(N, Src, Dest), Time, do(A, S)) :- \\
&\qquad fault(ttl\_lostPkt(N, Src, Dest), Time, S), \\
&\qquad\quad notA = fault\_down(ttl\_lostPkt(N, Src, Dest), Time); \\
&\qquad A = ttl\_lostPkt\_up(N, Src, Dest, Time).
\end{aligned}
$$

$$
\begin{aligned}
&fault(rt\_lostPkt(N, Src, Dest), Time, do(A, S)) :- \\
&\qquad fault(rt\_lostPkt(N, Src, Dest), Time, S), \\
&\qquad\quad notA = fault\_down(rt\_lostPkt(N, Src, Dest), Time); \\
&\qquad A = rt\_lostPkt\_up(N, Src, Dest, Time).
\end{aligned}
$$

The actions that influence the previous fluent are:

$ttl\_lostPkt\_up(N, Src, Dest, Time)$	PLA action that notify a lost packet for TTL at zero; N identify the node on which packet is rejected, Src and Dest are packet path parameters;
$rt\_lostPkt\_up(N, Src, Dest, Time)$	PLA action that notifies a packet loss due to routing problems; reported parameters have the same meaning of the previous action;
$fault\_down(X, Time)$	OnLR_Core action which makes false this fluent.

The first two actions are due to the PLA external notifications which are generated by the asynchronous event reporting mechanism. The fault notification contains, also, some information about its local cause. During the fault management, the OnLR_Core can also store information in the GDB, operate in order to restore network normal working conditions, or activate more deeper analyses. These analyses are capable of finding global problems that a simple local sensor cannot reveal. The logical connection between a fault and its root cause is expressed by logical rules constituting the inferential engine of the OnLR_Core.

A set of test condition in the *rules* procedure, allows the OnLR_Core to identify the typology of fault it must analyze.

As an example, `rt_lostPkt` fault is identified by the following test:

?(fault(rt_lostPkt(Node, Src, Dest), Time))

When this test succeeds the OnLR_Core must analyze different possibilities:

- If the packet destination node is off, it has not to longer investigate:

  ?(-node_status(Dest, on)):
  ?(writeln("Destination node is off")) :
  fault_down(rt_lostPkt(Node, Src, Dest), Time)

- If the destination node is on, the OnLR_Core tries to understand if it is not reachable because of a network disconnection, i.e. because a path connecting the `Src` and `Dest` nodes does not exist. In this case, the abnormality is stored into the GDB:

  ?(node_status(Dest, on)):
  ?(-minpath(Src, Dest, _, _)) :
  ?(writeln("Network disconnection")) :
  ?(sql_login(netlogDB, rnl, scalculus)) :
  ?(sql_insert('RNLInference',
  ['Type', 'Node', 'Source', 'Dest', 'Time'],
  ['disconnection', Node, Src, Dest, Time])) :
  ?(sql_logout) :
  fault_down(rt_lostPkt(Node, Src, Dest), Time)

  In some situations, it is possible to solve this abnormality turning on some backup nodes or links which can reconnect the network. It should be noticed that backup node and links in normal conditions have their status off, i.e. their status can be turned on whenever it is necessary:

  ?(minpath_backup(Src, Dest, Path1, Cost1)) :
  /*this predicate finds a connection path from Src node to Dest node containing backup node also*/
  ?(check_bac(Path1, Temp)):
  /*this predicate returns a list composed by all backup nodes contained in path Path1*/
  ?(port(P)) : resume_node(Temp, P)
  /*resume node is a procedure that sends a turning on command to each node contained in Temp list*/

- If the fault is not due to one of previous scenarios, the OnLR_Core infers a probable routing table corruption which disables the forwarding. This result is inserted into the GDB:

  ?(node_status(Dest, on)) : ?(minpath(Src, Dest, _, _)) :
  ?(writeln("Routing table corruption")) :
  ?(sql_login(netlogDB, rnl, scalculus)) : ?(sql_insert('RNLInference',
  ['Type', 'Node', 'Source', 'Dest', 'Time'],
  ['corruptedRT', Node, Src, Dest, Time])) :
  ?(sql_logout) :
  fault_down(rt_lostPkt(Node, Src, Dest), Time)

A `ttl_lostPkt` may be caused by a routing loop ob the path from *Src* to *Dest* node. In order to discover a loop, the OnLR_Core invokes a distributed monitoring service:

```
?(fault(ttl_lostPkt(Node, Src, Dest), Time)) :
?(port(P)) : ?(request_loop(Src, Dest, P)) :
fault_down(ttl_lostPkt(Node, Src, Dest), Time)
```

The distributed service which performs the loop monitoring is implemented by means of a simple agent that traverses the network from *Src* to *Dest* node. Each visited node is stored on the loop agent. If the loop agent meets a visited node, the PLA installed on this node executes a code fragment which sends a notification to the OnLR_Core. This notification, is translated by the Gateway in the following exogenous action:

`loop_up(N, Source, Dest, Time)`

The above action, together with a simmetric `loop_down` action, are captured by the fluent which is used to represent loops reported but not yet analyzed. The *Loop* fluent is shown in the following:

```
loop(N, Source, Dest, Time, do(A, S)):-
    loop(N, Source, Dest, Time, S),
      not A=loop_down(N, Source, Dest, Time);
    A=loop_up(N, Source, Dest, Time).
```

The occurrence of a Loop is analyzed by the procedure *rules* and it represents one of the events which are stored in the GDB:

```
?(loop(Node,Src,Dest,Time)) :
?(write("Loop")) :
?(sql_login(netlogDB,rnl,scalculus)) :
?(sql_insert('RNLInference', ['Type','Node','Source','Dest','Time'],
    ['loop',Node,Src,Dest,Time]))
?(sql_logout) :
loop_down(Node,Src,Dest,Time).
```

It is worth to notice that the OnLR_Core manages only the current faults and anomalies. It never looks at the past network history to understand which scenario has generated a fault, but it observes only what occurs in the network at monitoring time. Therefore results of OnLR_Core inference are recorded in a GDB table in which a single time parameter is present. This parameter

represents the time at which a PLA senses a fault or in which the OnLR_Core detects an anomaly. This GDB table is named OnLR_Inference and it is used as a buffer for results achieved by the OnLR_Core. Periodically, the OnLR_Core re-analyzes the information contained in this table, mining hidden data and upper level information. For instance, through this summarization process, it is possible to discover temporal relationships between different events, grouping together results that can refer to a single failure. Results of this data-mining process are inserted in "ad-hoc" tables, specific for each anomaly, in which two temporal parameters delimit the time interval during which the anomaly occurs.

4.2 OnLR_TrafficMonitor Inferences

The OnLR_TrafficMonitor is responsible of the traffic sensors tuning on the basis of the network congestion situation. As the OnLR_Core, this agent communicates with the Gateway agent sending XML messages and receiving RGolog exogenous actions. The OnLR_TrafficMonitor adopt a network representation which is different from that maintained by the OnLR_Core. Differently from the latter, the former agent does not look at the network behavior in order to detect failures, but it monitors how traffic flows exploit network resources, in order to tune sensor activities. Sensors will collect the data used by a special OffLR agent which will extract statistical information of network performances. For this reason, on one hand, the OnLR_TrafficMonitor does not maintain a complex *node_status* fluent similar to the one maintained by the OnLR_Core agent, while on the the other hand, OnLR_Core agent does not use fluents which represent the parametrical features of the sensors. On the same basis, also the exogenous actions are different. The OnLR_TrafficMonitor constitutes only a component of a performance management tool, currently under development. The aim of the whole subsystem will be the network traffic analysis in order to perform network traffic optimization. Traffic optimization will increase the quality of service and will minimize resource utilization.

The different phases of performance management, i.e. monitoring and control, will be distributed both on the OnLR and OffLR agents. The OnLR_TrafficMonitor manages the performance monitoring, and periodically it requires a special-purpose OffLR agent to compute the link-cost assignment by means of algorithms [13, 14], which exploit the information stored in the local data-bases (LDB). Using the results obtained in this phase, the OnLR_TrafficMonitor executes a performance control phase, where it sends the opportune actions to the network in order to apply the required adjustments. Figure 4.1 illustrates the agents and the communications involved by these activities. In this paper we describe only the OnLR_TrafficMonitor. This agent is responsible of choosing the network entity to be monitored and the monitoring parameters, such as precision level and granularity. In order to quantify the amount of resources used and how each communication flow uses network resources, the OnLR_TrafficMonitor agent adopts three kinds of sensors: Flow_sensor, InQueue_sensor and OutQueue_sensor. First sensor measures the percentage bandwidth occupied by each flow; other sensors measure the queue occupancy over the time.

A great accuracy in recording this information is very expensive, even-thought recording only statistical information with a too large granularity may cause a loss of significant information with advantage of the not relevant one.

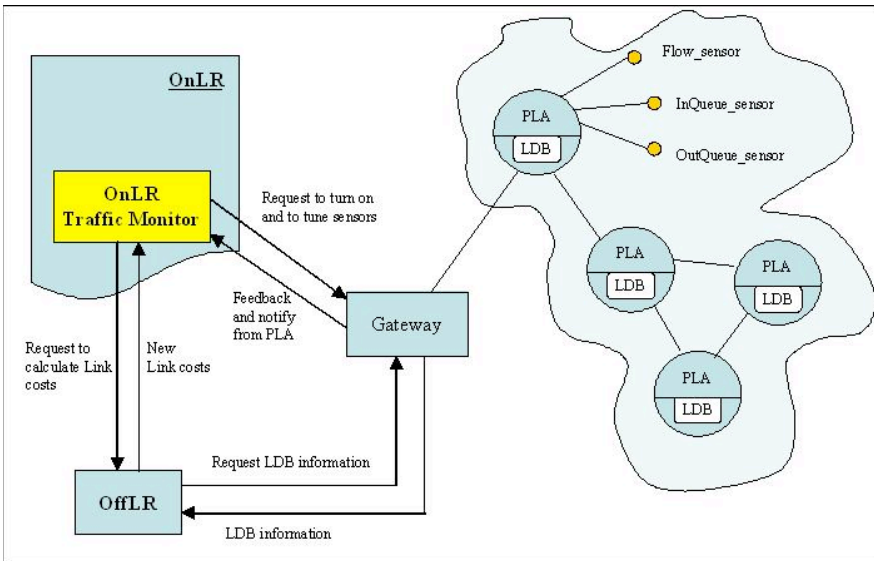

Figure 4.1: Interactions among performance management agents

The OnLR_TrafficMonitor main activity is the tuning of sensing and recording instruments in order to use great accuracy in the detection of only the meaningful information. To perform its task the agent can adjust a set of parameters of previous sensors. For instance, all these sensors offer the capability of tuning the sampling frequency (SamplingRate parameter) or the recording rate (RecordingRate parameter). This means, it is possible to tune the precision of information sampling and the granularity of information recording. Also the queue sensors offer a Threshold parameter: it represents the queue occupation level over which a notification message must be sent to the OnLR_TrafficMonitor. Queue overflow notification messages are translated by the Gateway agent in exogenous actions, such as:

A = inQoverflow_up(Node, Value, Time);
A = outQoverflow_up(Node, IFace, Value, Time).

First action refers to an abstracted single queue associated with all the incoming links, and second action refers to the link connected with a single outgoing interface. This differentiation is made necessary to attribute to congestion for outgoing link saturation more relevance than to congestion due an insufficient node computational capability. The above actions operate on $fault$ fluent which collects events notified, but not yet analyzed:

```
fault (inQoverflow (Node, Value), Time, do(A, S)):-
        A=inQoverflow_up(Node, Value, Time);
        fault(inQoverflow(Node, Value), Time, S),
          not A=fault_down(inQoverflow(Node, Value), Time).
```

```
fault(outQoverflow(Node, IFace, Value), Time, do(A, S)):-
    A=outQoverflow_up(Node, IFace, Value, Time);
    fault(outQoverflow(Node, IFace, Value), Time, S),
      not A=fault_down(outQoverflow(Node, IFace, Value), Time).
```

The `fault_down` action makes false the *fault* fluent. There are no exogenous action, which notify the link occupation by flows. This information is only stored in the LDB.

OnLR_TrafficMonitor distinguishes among different alarm levels and some corresponding different levels of information accuracy. The monitoring process starts from a low level alarm in which only the inQueue_sensors are activated with low values for all parameters. Low `SamplingRate` and `RecordingRate` values mean a low accuracy in the information recording. A low `Threshold` value means a high reactivity of the agent, given that, it receives an update whenever the low value of threshold is exceeded. When the agent receives a notification, it recalculates the alarm level and if there is a meaningful variation, it should decide if the alarm notifications are homogeneusly or locally distributed over the network. In the first case, the agent may infer that no traffic contidions are present but threshold values are too low. It, simply, performs a threshold tuning phase in order to reduce notification overhead. In the second case, if the notification distribution is locally concentrated in a small network area, it performs a more accurate spatial analysis with the aim of determining if the congestion phenomenon occurs on a single node. In the case of isolated phenomenon,the sensor tuning will occur only the on congested node, whilst in the other case, it will be applied both to the congested node and to its neighbors.

Main OnLR_TrafficMonitor procedure code is shown following:

```
( ?(fault(X,Time)) :
( ?(X=inQoverflow(Node,Value)) # ?
(X=outQoverflow(Node,Iface,Value)) ) :
   /*following procedure return new alarm level*/
   alarm_level_calcul(Node,NewLevel) :
   /*control old level*/
   ?(alarm_level(Node,OldLevel)) :
   ( ?(NewLevel¡=OldLevel) #
       /*if the old level is lower than the new level it must
         control if notification distribution is homogeneous over
         the network*/
       ?(NewLevel¿OldLevel) :
       ( /*if notification distribution is homogeneous it must
         perform a threshold tuning*/
         ?(omogeneous_distribution) : threshold_tuning
         #
         /*if there is a not homogeneous distribution it must
         perform a spatial analysis */
         ?(-omogeneous_distribution) :
         ( /*if Node congestion is an isolated malfunction
```

```
            phenomenon it must tune sensors only for this Node*/
            ?(isolated_malfunction(Node)) :
            ?(sensor_tuning([Node],OldLevel,NewLevel))
            #
            /*if also Node neighbor is involved in a congestion
            phenomenon it must tune sensor for node and for all
            its neighbor*/
            ?(-isolated_malfunction(Node)) :
            ?(findall(Neigh,ph_neighbor(Node,Neigh,_,_,_), NeighList)) :
            ?(sensor_tuning([Node—NeighList],OldLevel,NewLevel))
) ) ) )
```

A graphical view of this procedure, i.e. a state diagram representing the OnLR-_Traffic-Monitor reasoning activity, is shown in figure 4.2:

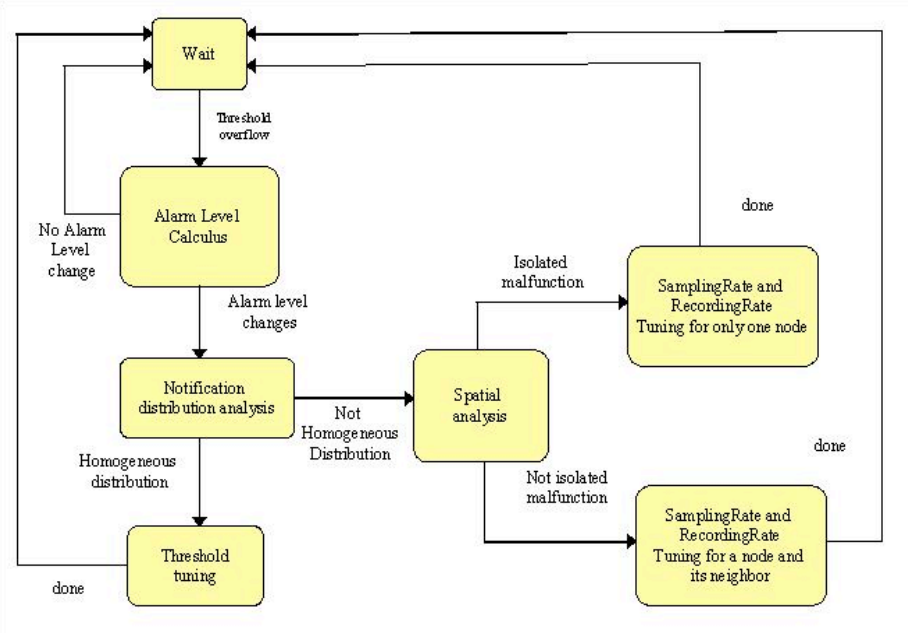

Figure 4.2: State diagram of the OnLR_TrafficMonitor reasoning

4.3 OffLR Inferences

The OffLR agent is responsible of executing an "a posteriori" network analysis, using the information stored in the LDB and in the GDB. The off-line reasoning activities are exploited by the OnLR monitoring activity and they, also, allow to answer queries about the network past history addressed by a human administrator to the Reasoner. This way, OffLR reduces the OnLR computations which involve large amount of data, thus allowing it to timely react to network

events. Off-line activity integrates the OnLR functionalities with more extensive reasoning on long time periods rather than instantaneous sensing.

However, differently from the OnLR, OffLR performs a passive monitoring and can not actively operate on the network. Its main task is retrieving, merging and mining local information in order to derive global results about the network as a whole and its functioning. Since the OffLR does not consider the network evolution, it is not implemented in RGolog, but the Prolog language is more suitable for its static representation of the network. The OffLR knowledge base does not store any information about the current network condition, but it evolves by inference about past information. OffLR reconstructs the succession of the network functioning states during a given time interval without consider which events triggered the transition from a state to another one. It does not achieve a global history reconstruction for each network entity, but produces a description of single aspects during the time, such as, for instance, the history of a node routing table, the succession of the different states of a node, etc. After this phase of data collection, the OffLR may relate different aspects on the basis of its specific reasoning procedures.

Currently, traffic performance management functionalities are being added to OffLR. With these features, OffLR will be able to perform reasoning about the congestion phenomenon using information stored in the LDBs. In turn, the OffLR to perform this task, exploits another agent capable of calculating a redistribution of link cost assignment which improves the quality of service offered by the network.

Furthermore, the OffLR is used by the network administrator to request information about the network functioning in a past time interval. The OnLR, through the QueryHandler agent, assigns each request to an opportune OffLR agent that in turn may delegate other OffLR agents. When the QueryHandler agent has to start an OffLR agent, it starts a child Eclipse process responsible of executing the agent code compilation and the starting procedure. For instance, in order to start the OffLR_SearchOnPath agent, the OffLR agent executes the following code:

```
exec([ ‘‘<ECLIPSEDIR>/bin/i386_linux/eclipse’’,‘‘-e’’,
       ‘‘compile(’init_searchOnPath.pl’), start(Src,Dest,T1,T2)’’],
      [in,out], PidCore).
```

The file “init_searchOnPath.pl” contains the instructions for compiling the RGolog interpreter and the opportune libraries, such as the mysql library necessary to communicate with the GDB. *Src* and *Dest* parameters identify the flow communication vertices, $T1$ and $T2$ are the lower and upper bound of the investigated time interval. The [in,out] pair contains references to the input and the output pipes, toward and from the child Eclipse process. Each agent that needs to communicate with the network to obtain LDB information, starts a connection with the Gateway. This communication is implemented by two different streams, as previosly explained. The first one contains XML stetements and it is directed from the OffLR to Gateway. The second one contains Prolog predicates and it is directed from the Gateway to the OffLR. The OffLR agent

reads the Prolog predicates contained in the communication stream from the Gateway agent, and it asserts them into its KB after a preprocessing analysis. OffLR sends XML messages to the Gateway only in order to obtain LDB information. In these messages it specifies the nature of information and the time interval. Currently, two types of requests are implemented: the first concerns a single communication flow, identified by a [Source, Destination] pair, and the second one concerns a global reconstruction of a past network state. For both the cases, a bounded time interval must be specified.

4.3.1 Request of information about a single path

Let us consider the following scenario. A network user wants to know the root causes of a network problem, such as, for instance, the difficulty to start a new connection, the interruption or the degradation of a connection, etc.. In order to formulate such queries to the OffLR, the user must specify two different parameters: the time interval and a pair of nodes representing the connection vertices. Figure 4.3 shows the agent interactions to answer such queries. When

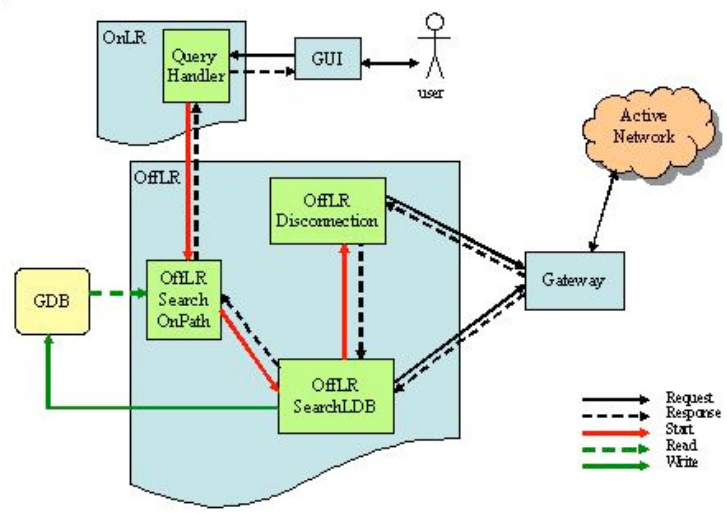

Figure 4.3: Agent involved in a SearchOnPath query

the QueryHandler agent receives such a query, it starts the OffLR_Search-On-Path to find in the GDB the anomalies which can be relevant for the monitored path during the indicated time interval.

In order to increase the efficiency and the responsiveness, the OffLR_Search-On-Path agent reuses, if possible, the results of past reasoning activities, previously inserted into the GDB. This behavior may avoid heavy computation previously done. If no information regarding the selected path and the temporal interval is found in the GDB, the information research is addressed to the involved LDBs. This research is performed by the OffLR_Search-LDB agent.

In order to verify if the GDB contains the required information, the OffLR_Search-On-Path lookups NetMonitor and PathMonitor tables. The former contains time intervals in which a global network monitoring was performed by

the OnLR_Core or by the OffLR_Restore-Net-Status. The latter contains the time intervals during which the functioning of a single path was analyzed by the OffLR_Search-LD.

The monitored time intervals, for the whole network or only for a monitored path, are obtained by OffLR_Search-On-Path agent, executing the following instructions:

```
findall(PRes,
    sql_query(['startTime', 'endTime'], 'pathMonitor', Where, '', [Pres]), PList),

      /*PList will contain all tuples returned by the following query:
        "select startTime, endTime from pathMonitor
        where pathMonitor.source=Src and pathMonitor.dest=Dest" */

findall(NRes,
    sql_query(['startTime', 'endTime'], 'netMonitor', '', '', [Result]), NList)
```

The union of the lists returned by last two predicates represents the temporal interval in which the observed path was monitored in the past by some Reasoner agent. For each of these time periods it is very probable that occurred anomalies were detected and stored in the opportune GDB table. The anomaly tables examined by the OffLR_Search-On-Path agent are RTCorruption, Loop, and Disconnection. These tables contain all the results directly obtained by the OffLR_Restore-Net-Status and the OffLR_Search-LDB, and a summarization of OnLR_Core results. Namely, while last agent performs inferences that are related to a single instant, the inferences of the other agent concern a time interval, which is coherent with `startTime` and `endTime` attributes of the three tables, above mentioned. For instance, to retrieve data from the Loop table the OffLR_Search-On-Path executes the following code:

```
findall(Result,
    sql_query(['type', 'node', 'startTime', 'endTime'], 'cause', Where, '',
          [Result]), List),

    /*List will contain all tuples obtainined from the following query:
        "select type, node, startTime, endTime from Loop
          where Loop.source=Src and Loop.dest=Dest
            and ( (Loop.startTime≤T1 and Loop.endTime≥T1)
              or (Loop.startTime≥T1 and Loop.startTime≤T2))" */
```

If the whole observation time interval was monitored, results obtained by GDB queries are returned to the QueryHandler agent and the OffLR_Search-On-Path stops its execution. If not the whole observation period was monitored, the OffLR_Search-On-Path must retrieve other information from the LDBs and perform new inferences. The OffLR_Search-On-Path starts a new process, which is responsible to compile the OffLR_SearchLDB for each time interval not monitored, and waits its termination to obtain other results. The set of information

obtained by the merging of GDB information and that retrieved by the OffLR_Search-LDB is returned to the QueryHandler agent.

The OffLR_Search-LDB is responsible to find all the possible causes of the anomalies occurred on an end-to-end connection during a given time interval, by using the information stored in the network node LDB. The agent looks for two different anomaly types, i.e. routing table corruptions and loop occurrences. This is done with the analysis of the routing tables of all the nodes along all the possible paths followed by the flow in the observation time period. Furthermore, the OffLR_Search-LDB delegates the OffLR_Disconnection agent to discover eventual network disconnections.

As above mentioned, to perform its task, the OffLR_Search-LDB analyzes the routing tables of all the nodes along the routing path from `Source` node to `Dest` node and it registers all the discovered anomalies. The OffLR_Search-LDB needs a static network representation to identify the network topology. The knowledge of the initial conditions is obtained from the file ("topology.pl") created by the OnLR_Core at bootstrap time. In order to obtain the LDB information the agent starts a connection with the Gateway and waits for the answers on an opportune communication stream. The progressive search moves along the path followed by the data of the communication flow identified by the `[Source, Dest]` pair. The search can arrest because one of the following four different reasons:

- the destination node is reached and no routing anomalies occurred along the path;
- the same node is meet two times, i.e. a loop occurs in this branch;
- it is not possible to find any routing entry toward the *Dest* node, because of a routing table corruption or an infinite cost entry;
- it is not possible to obtain information from a node because it is off.

The results obtained are compared with the OffLR_Disconnection results in order to verify if an anomaly of a routing table entry is due to a network disconnection or simply to a temporary alteration. The information about paths and periods monitored by the OffLR_Search-LDB are stored into the PathMonitor table of GDB.

The OffLR_Disconnection agent is responsible of reconstructing the on/off state transitions of all network nodes during a given temporal interval, in order to find network disconnections. This reconstruction is made by using AliveNeighbor/DeadNeighbor sensor traces. To understand when a node was on, it must analyze the sensor traces produced by its neighbors. At bootstrap time, the OffLR_Disconnection agent compiles the initial topology information and sets up a connection with the Gateway and the OffLR_SearchLDB agents. The OffLR_Disconnection task can be distinguished in two phases. In the first one it collects information from LDBs and it reconstructs the temporal behavior of the nodes. During the second phase, it uses more detailed information about paths to carry out higher level analyses.

4.3.2 Network global history reconstruction

The OffLR can be asked by a network user to reconstruct an integral and complete network history for a given time interval. The OffLR agent which performs this task can be viewed as an image of the OnLR_Core, although it does not pressnt the capability of reacting to the network events and to actively operate on the network. In order to execute its task, the OffLR agent maintains an ontological representation of the network together with some basic knowledge on the cause-effect relationship between an anomaly and its symptoms, similar to that maintained by the OnLR_Core. This way, the agent can perform the same "a posteriori" inferences of those that the OnLR_Core performs in real-time. The "Network global history reconstruction" query can be used both to verify the consistence of OnLR_Core results or to integrate them. For instance, it can be applied to a temporal interval in which the OnLR_Core was not working.

This point of view, moreover, points out the importance of the off-line reasoning capability of performing the filtering of large amount of data stored in the LDBs. These data are characterized by large sizes and low information density. Furthermore, their interpretation results difficult for a human agent since in their original form they do not result meaningful. By exploiting the OffLR filtering capalities, they can grouped in higher level concepts, thus increasing the information density. In order to manage this query, the QueryHandler agent starts the OffLR_RestoreNetStatus and waits the results to pass them the user. Figure 4.4 shows the agent behavior in order to manage this kind of query.

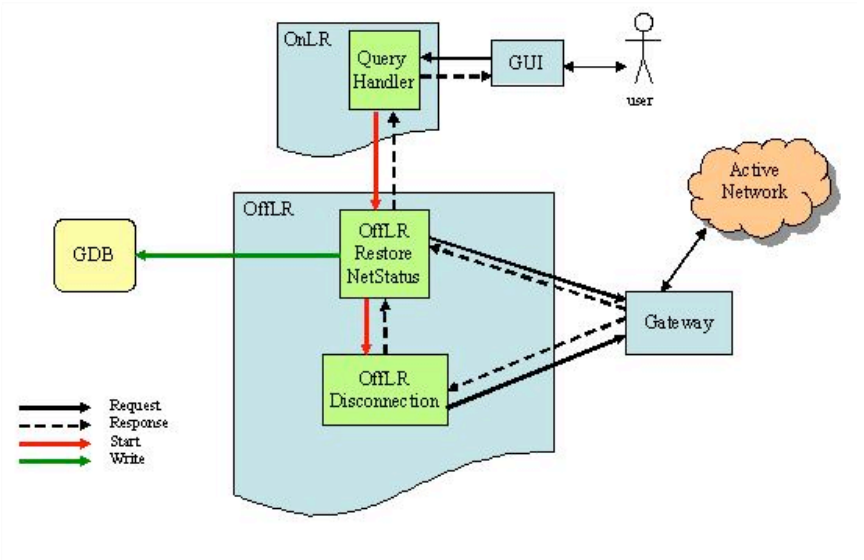

Figure 4.4: Agent Behavior for the Restore Network Status query

The OffLR_RestoreNetStatus, once retrieved data from LDBs, is capable of finding anomalies, for instance routing errors, such as loops and routing table corruptions. It, also, uses the OffLR_Disconnection reasoning to find eventual network areas isolation. When The OffLR_RestoreNetStatus obtains OffLR_Disconnection results, it merges them with its results to find all pos-

sible anomalies. Global results are inserted in the GBD and returned to the QueryHandler agent. Also an indication about the investigated time interval is stored in the GDB NetMonitor table. In order to find all anomalies, the OffLR_RestoreNetStatus agent starts from the analysis of basic symptoms such as packet losses. Its reasoning is similar to OnLR_Core reasoning. Both the modules start from a fault to find its cause. The interaction between OffLR_RestoreNetStatus and OffLR_Disconnection is strict. The former needs results of the latter to refine its reasoning about `rt_lostPkt`. The latter needs the reasoning results of the former in order to perform its analyses: for instance, only by means of these it can know which `[Source, Dest]` pair identify the path to analyze. Results obtained by both the agents are stored in GDB tables containing anomalies. The diagram of activities performed in order to reconstruct the global history of the network is shown in figure 4.5.

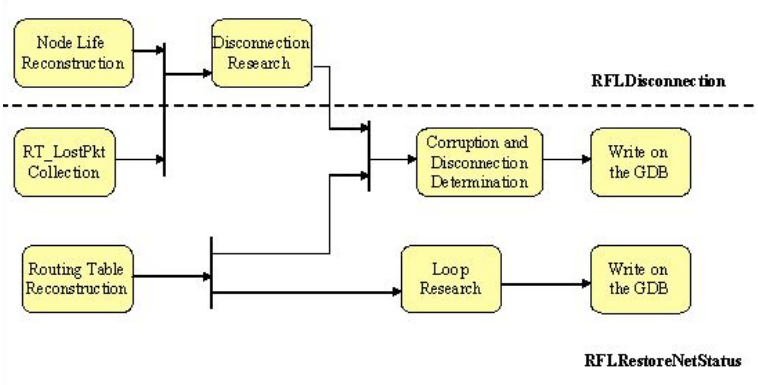

Figure 4.5: Restore Network Status Activity diagram

Chapter 5

DB

5.1 Global Database

The Global Database (GDB) stores the most relevant results inferred by the different Reasoner agents. The GDB role assumes, hence, a great importance in order to perform an "a posteriori" analysis of network problems. At the current stage of the project, the aim of a query to the GDB, is to answer user requests about problems which have involved a network area during a given temporal interval in the past. However, a wider GDB table set has been designed, necessary for the support of future reasonings. Currently, the system elements interacting with the GDB are the OnLR_Core and several OffLR agents. The first one can only add information to that contained in the GDB, whilst OffLR agents exploit this information for their own reasonings and can insert new knowledge inferred by their deductive processes. The GDB is constituted by three groups of table: the first one is formed by tables containing results of anomaly monitor processes, the second one is related to the history of network monitoring, and the third one has been designed to support the future system development on the congestion analysis.

5.1.1 Anomaly tables

Tables of this group contain records about anomalies discovered in the network by the OnLR_Core, the OffLR_SearchLDB, and the OffLR_RestoreNetStatus agents. A graphical view of this set is shown in figure 5.1.

OnLR_Inference

The OnLR_Inference table buffers the inference results produced by the OnLR-Core. Since data results are inserted in this table without any further elaboration, there are no relations among results obtained at different times but regarding the same problem. Periodically data of this table are unloaded, elaborated, and summarized by the OnLR in ordert to obtain meaningful information. The obtained information is then inserted in other tables of this group.

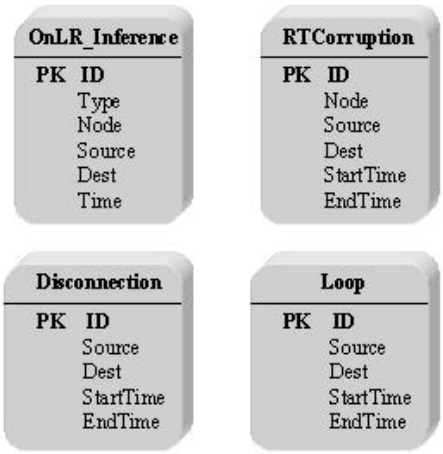

Figure 5.1: GDB network anomalies tables

OnLR_Inference Table

Attributes:

ID	Numeric field. It represents the table primary key.
TYPE	Indicates the anomaly type. Currently, it can have one of following values: { loop, corruptedRT, disconnection }.
NODE	The interpretation of this attribute depends on the Type parameter. if TYPE=loop, it represents the node that close the loop; if TYPE=corruptedRT, it represents the node that has suffered a routing table corruption; if TYPE=disconnection, it represents the node starting the Reasoner investigation in which the packet loss occurred.
SOURCE DEST	Identifies source and destination nodes on the path where the anomaly occurred. These two parameters individuate communication flows involved in the anomaly.
TIME	Indicates the time when the packet loss event, which started the Reasoner inference, occurred.

Constraints:

A Unique constraint involves all the attributes of this table except the ID one:

```
UNIQUE UN_OnLR_Inference_1 (Type, Node, Source, Dest, Time)
```

Loop

In this table are stored all the information regarding a loop on the routing. Loops are identified on the basis of the involved communication flows and of the time interval during which they occurred.

Loop

Attributes:

ID	Numeric primary key;
SOURCE DEST	Identify the source and destination nodes on the path involved by the loop. DEST is the node which the loop makes unreachable starting from SOURCE.
STARTTIME ENDTIME	Start and end instant bounding the time interval during which the loop occurs.

Constraints:

A Unique constraint involves all the attributes of this table except the ID one:

```
UNIQUE UN_Loop_1 (Source, Dest, StartTime, EndTime)
```

Disconnection

This table stores the extremes of the paths interrupted by network disconnections that make the Dest node unreachable from the Source node.

Disconnection

Attributes:

ID	Numeric primary key;
SOURCE DEST	Extremes of the path interrupted by the disconnection. Starting from the analysis of [Source, Dest] pairs it is possible to determine if the communication flow was involved in an anomaly.
STARTTIME ENDTIME	Start and end time of the period during which packets sent from the Source node were not able to reach the Dest node because of the disconnection.

Constraints:

A Unique constraint involves all the attributes of this table except the ID one:

```
UNIQUE UN_Disconnection_1 (Source, Dest, StartTime, EndTime)
```

RtCorruption

This table stores all the routing table corruptions that caused packet losses for different communication flows. For each corruption is indicated, together with the flow parameters, also the node in which the loss occurred.

RtCorruption

Attributes:

ID	Numeric primary key;
NODE	Indicates node in which the routing table corruption was detected.
SOURCE DEST	Identify source and destination nodes indicated in the lost packet header.
STARTTIME ENDTIME	Indicate the starting and ending time of the period when the routing table corruption occurred.

Constraints:

A Unique constraint involves all the attributes of this table except the ID one:

```
UNIQUE UN_RTCorruption_1 (Node, Dest, Source, StartTime, EndTime)
```

5.1.2 Monitoring history

Tables of the Monitoring History were designed in order to reconstruct the time periods during which the whole network, or a single path, have been monitored. The aim of these tables is to provide to the OffLR the information that will allow it to establish if the GDB data are sufficient to answer user queries. Figure 5.2 shows the monitoring history tables.

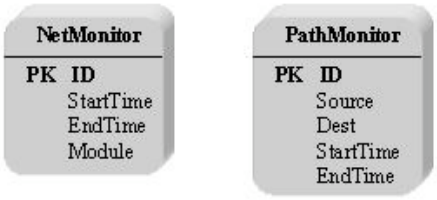

Figure 5.2: GDB monitoring history tables

NetMonitor

This table stores all time periods during which the whole network has been monitored. The monitoring can be performed by both the OnLR_Core, during its natural active monitoring task, or by the OffLR_RestoreNetStatus, when it performs the reconstruction of a past network period during which the OnLR_Core was disabled.

NetMonitor

Attributes:

ID	Numeric primary key;
STARTTIME ENDTIME	Extremes of the monitoring time interval.
MODULE	ID identifying the particular Reasoner (OnLR, OffLR) agent that has performed the monitoring activity.

Constraints:

A Unique constraint involves all the attributes of this table except the ID one:

```
UNIQUE UN_NetMonitor_1 (StartTime, EndTime, ReasonerModule)
```

PathMonitor

When the OffLR is started to answer some user queries about anomalies that involved a given path, it performs a specific analysis regarding only the particular path and not the whole network. However, the information inferred by this reasoning activities are stored in the anomaly tables. The PathMonitor table maintains the different periods when single paths have been analyzed.

PathMonitor

Attributes:

ID	Numeric primary key;
SOURCE DEST	Indicate the end nodes of the monitored path.
STARTTIME ENDTIME	Starting and ending time of the period concerning the path monitoring.

Constraints:

A Unique constraint involves all the attributes of this table except the ID one:

```
UNIQUE UN_PathMonitor_1 (Source, Dest, StartTime, EndTime)
```

5.1.3 Congestion Tables

The tables of this group were designed for future system development concerning the network congestion. Currently the information contained in these tables is not used by any agents, even-thought in the next future it will be exploited by both the OnLR and the OffLR agents. Figure 5.3 shows a graphical view of the congestion tables.

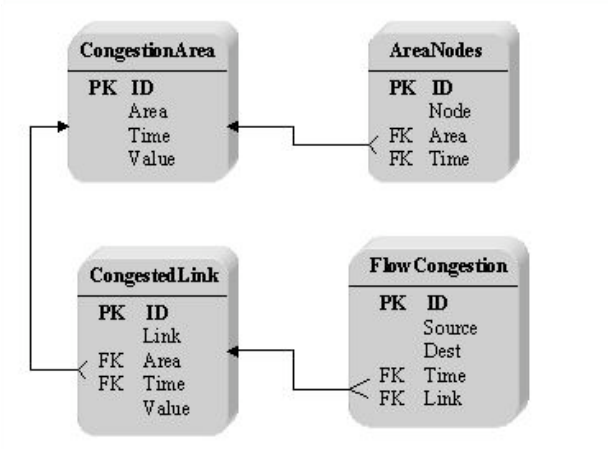

Figure 5.3: GDB congestion tables

CongestionArea

This table maintains a congestion measurement value for each monitored area. Each area will be identified by its central node. Each node belongs to a single area at each time.

CongestionArea

Attributes:

ID	Numeric primary key;
AREA	Represents the most central node of the congested area;
TIME	Represents the evaluation time;
VALUE	Congestion measurement value.

Constraints:

A Unique constraint involves all the attributes of this table except the ID one:

```
UNIQUE UN_CongestionArea_1 (Area, Time).
```

AreaNodes

This table, together with the CongestedLink one, individuates the monitored areas as the sets of nodes and connecting links. More specifically, this table allows the identification of the nodes composing an area, at a given time.

AreaNodes

Attributes:

ID	Numeric primary key;
NODE	Represents a monitored node;
TIME	Represents the evaluation time;
AREA	Represents the area where the node stays.

Constraints:

A Unique constraint involves all the attributes of this table except the ID one:

```
UNIQUE UN_AreaNodes_1 (Node, Time).
```

`[AREA,TIME]` attributes have a reference to "CongestionArea" `[AREA,TIME]` ones:

```
FOREIGN KEY (Area,Time) REFERENCES CongestionArea (Area, Time).
```

CongestedLink

Although this table defines the area concept, it does not maintain information about links composing an area, but only about links which have experienced an utilization above the alert threshold.

CongestedLink

Attributes:

ID	Numeric primary key;
LINK	Represents a monitored link;
TIME	Represents the evaluation time;
AREA	Represents the area where the node stays.
VALUE	Congestion factor calculated for the single link.

Constraints:

A Unique constraint involves all the attributes of this table except the ID one:

```
UNIQUE UN_CongestedLink_1 (Link, Time).
```

`[AREA,TIME]` attributes have a reference to the "CongestionArea" `[AREA,TIME]` ones:

```
FOREIGN KEY (Area,Time) REFERENCES CongestionArea (Area, Time).
```

FlowCongestion

This table maintains references to those flows which are most involved in a congestion situation, relate to a given area during a given temporal range.

FlowCongestion

Attributes:

ID	Numeric primary key;
SOURCE DEST	Represent source and destination nodes of the flow involved in the congestion phenomenon;
TIME	Represents the evaluation time;
LINK	Represents a link used by the flow.

Constraints:

`[AREA,TIME]` attributes have a reference to the "CongestionArea" `[AREA,TIME]` ones:

```
FOREIGN KEY (Link,Time) REFERENCES CongestedLink (Link, Time).
```

5.2 Local Databases

In order to provide support to the OffLR agent, we designed and implemented some special DBs, which are located on each node of the network and some "ad hoc" procedures to retrieve data stored on them.

The OffLR agent needs to know the network past situations, with temporal continuity.

With the aim to obtain a "continuous" historical knowledge, we introduced the concepts of *snapshot* and *log*. The *log* is a file where are stored all the changes occurred to some determined variables, and as *snapshot* we mean a complete image of all the variables taken at regular time interval, and fully registered in the *log* file. The OnLR is responsible of choosing the variables to be monitored for both the the methods. Using the above tools it is possible to trace the whole history of each variable at any time, since we can retrieve an absolute value using the previous *snapshot* and all the successive variation stored on the *log* file. For instance, this technique is particularly suitable to register the routing tables: at each *snapshot* all the routing table is stored, whilst, between two successive *snapshots* only the routing table entries which are affected by variations are registered. As above mentioned, the OffLR agent can require the data related to a specific time interval and to some particular network variables. The LDBs are organized on the basis of the time they were created, and furthermore, for each variable also the registration time is stored. Using this temporal information, it is straightforward to obtain the whole set of the values assumed by the variables in a given time period.

Bibliography

[1] L. Kerschberg, R. Baum, A. Waisanen, I. Huang, and J. Yoon, "Managing faults in telecommunications networks: A taxonomy to knowledge-based approaches," *Proc. of IEEE International Conference on Systems, Man and Cybernetics*, pp. 779 – 784, 1991.

[2] R. Kawamura and R. Stadler, "Active distributed management for IP networks," *IEEE Communications Magazine*, vol. 38, no. 4, pp. 114 – 121, Apr. 2000.

[3] S. Mazumdar and A. A. Lazar, "Objective-driven monitoring for broadband networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 3, Jun. 1996.

[4] J. F. Kurose and K. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, Third Edition. Addison-Wesley, . 2004.

[5] W. Stallings, *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2: Third Edition*, Addison Wesley, 2003.

[6] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski, "A knowledge plane for the Internet," *Proc. ACM SIGCOMM*, pp. 3 – 10, Aug. 2003.

[7] M. Wawrzoniak, L. L. Peterson, and T. Roscoe, "Sophia: an information plane for networked systems," *ACM Computer Communication Review*, vol. 34, no. 1, pp. 15 – 20, Jan. 2004.

[8] N. J. Nilson, *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann, 1998.

[9] G. Di Fatta, S. Gaglio, G. Lo Presti, G. Lo Re, and I. Selvaggio, "Distributed intelligent management of active networks," in *8th Congress of the Italian Association for Artificial Intelligence*, A. Cappelli and F. Turini, Eds., Pisa, Italy, Sep. 2003, number 2829 in LNCS, Springer Verlag.

[10] A. Barone, P. Chirco, G. Di Fatta, and G. Lo Re, "A management architecture for active networks," in *Proc. of Fourth International Workshop on Active Middleware Services*, Edinburgh, UK, Jul. 2002, pp. 41 – 48, IEEE Computer Society.

[11] J. McCarthy, "Situations, actions and causal laws," in *Semantic Information Processing*, M. Minsky, Ed., Cambridge, Massachussets, 1968, pp. 410 – 417, The MIT Press.

[12] R. Reiter, *Knowledge in action: Logical Foundations for specifying and implementing Dynamical Systems*, The MIT Press, Cambridge, Massachusetts, 2001.

[13] B. Fortz and M. Thorup, "Optimizing ospf/is-is weights in a changing world," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 4, pp. 756–767, May 2002.

[14] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving traffic demands for operational ip networks: Methodology and experience," *IEEE/ACM Trans. Networking*, vol. 9, no. 3, pp. 265–279, June 2001.