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

A Network Ontology for Computer Network Management

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

RT-ICAR-PA-03-22 **dicembre 2003**

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 Network Ontology 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. 22: ***Data:***
RT-ICAR-PA-03-22 **dicembre 2003**

[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.

Abstract

This work describes an ontology for the categorization of computer networks domain. The ontology has been defined to support the development of a knowledge-based system for network management. The system is a logical reasoner capable of performing management tasks typically executed by human experts; in order to accomplish these tasks, the logical reasoner needs a detailed representation of the network domain, that is, all relevant network entities and protocols, their relationships and their functioning mechanisms. All these concepts need a standard form of representation, as past experiences in developing networking reasoning system have shown. The developed ontology can be seen as the formal and explicit specification of the conceptualization that a knowledge-based system has to use to perform automated management tasks: the concepts and relationships, the ontology defines, are to be reflected within the internal domain representation and inference mechanisms of the reasoning engine. Finally, in order to represent knowledge, as formalization language for the ontology, the Web Ontology Language (OWL) has been used.

Chapter 1

Introduction

The Ontology notion originates from Philosophy, where it refers to the metaphysical study of the nature of being and existence. In Knowledge Engineering, ontologies are used for a quite different purpose, that is, for modelling concepts and relationships on some expertise domain. In this area, ontologies can be defined as formal specifications of conceptualisations [1, 2]. A conceptualisation is an abstract, simplified view of the portion of "world that we wish to represent for some purpose; since each knowledge base or knowledge-based system is committed to some conceptualization, explicitly or implicitly, ontologies are useful instruments for knowledge sharing among AI software and for knowledge reusability purposes. Formally, ontologies define knowledge about some area of interest, providing the structural and semantic ground for computer-based processing of such knowledge. Moreover, defining a formal vocabulary and providing the represented terms with formal and univocal semantics, ontologies abstract knowledge from implementation details, increasing its reusability and standardization. Thus, human or automated agents, who commit to an ontology, can exchange information in a way that is coherent and consistent with a shared conceptual theory. All these properties represent attractive features for many AI applications, such as the construction of knowledge-based systems, which, for their nature, have to present some domain expertise. Our ontology has been defined to support the development of a knowledge-based system for network management. The system we are building is a logical reasoner capable of performing management tasks typically executed by human experts; in order to accomplish these tasks, the logical reasoner needs a detailed representation of the network domain, that is, all relevant network entities and protocols, their relationships and their functioning mechanisms. All these concepts need a standard form of representation, as past experiences in developing networking reasoning system have shown. For this reason, the ontology presented can be seen as the formal and explicit specification of the conceptualization that a knowledge-based system has to use to perform automated management tasks: the concepts and relationships, the ontology defines, are to be reflected within the internal domain representation and inference mechanisms of the reasoning engine. This way, the ontology, providing a unitary view of these concepts, increases knowledge reusability and favours coordination among human developers during the system design and development, and it also makes explicit basic assumptions about the domain. In order to represent knowledge, as more

generally as possible, we decided to choose the Web Ontology Language (OWL) [3, 4], as the formalization language for our ontology. OWL is designed for the definition of ontologies and knowledge bases. OWL has been recently standardized by the W3C Working Group in the context of Semantic Web research, where it represents a higher layer above the language stack made up of XML, RDF and RDFS. Moreover, OWL allows using available tools, such as Racer [5], for automated reasoning about ontologies. Reasoning services provided by these tools are the automated detection of inconsistent concepts and the automated classification of concepts in a "is-a" hierarchy. These services are useful during the development of wide ontologies or their merging. Since OWL represents the result of hard efforts to achieve good expressiveness retaining reasoning decidability, it presents some limitations, especially in knowledge constraining possibilities. Furthermore, because of its ties toward other Semantic Web standards (XML, RDF, RDFS), over which it is constructed, its XML/RDF syntax is quite verbose and hard to understand; this means, it is difficult to maintain and update OWL ontologies and to achieve a global view of the represented concepts, directly using this syntax. For this reason it has been very important, in our ontological effort, the presence of a powerful tool, Protégé-2000, for the graphical editing of OWL ontologies. Protégé-2000 [6] is an open-source environment for ontologies and knowledge bases. Developed at Stanford Medical Informatics, it is a large project with a wide user community. Its purpose is to make easy the knowledge formalization activity, providing suitable support for ontology definition and maintenance, and for the customisation of knowledge acquisition forms; in this way, it makes more intuitive both the ontology design and its population, performed by end-users to construct knowledge bases. Since Protégé internal knowledge model is frame-based [7], OWL support is provided by special plugins, first of all the OWL Plugin [8]. It makes possible to edit OWL ontologies using most of Protégé graphic facilities and to automatically generate the OWL source. Moreover, the ezOWL and OWLviz plugins allow to display the ontology using a graphic UML-like syntax and a graph-based representation respectively.

1.1 Ontology overview

The ontology we developed does not cover all the aspects of networking domain; it should, rather, represent the domain knowledge from the management point of view, trying to capture those aspects of networking essential for monitoring and controlling purposes. The described knowledge can then be used by network management applications to perform fault diagnosis and recovering, to analyze and evaluate performances, to plan actions aiming at improving the quality of service. In order to make possible these tasks, ontology should represent all relationships which hold among the domain concepts; it should make explicit cause-effect relationships through which distinct events taking place at different temporal moments and spatial places can be related to each other in order to represent an integrated view of network behavior; it also should capture how different network elements can influence each other to determine their global status and their dynamical evolution over time. This description of ontological knowledge provides a deep understanding of the network as a whole, presenting a high-level view of its functioning. More specifically, concepts and relations

represented in the ontology describe the main components, features and behaviors of the Internet network layer and some elements of the Internet data-link and physical layers. Besides these aspects, the ontology describes the traffic concept, the resources involved and its distribution over the network. Moreover, the ontology describes tools and services required by a managing entity to perform monitoring and controlling tasks and mechanisms through which it can use them. Furthermore, since the management system itself belongs to the representation domain, last part of the ontology should characterize the distributed framework we designed and the tools and services offered to the management applications. The Logical Reasoner (i.e. the knowledge-based system acting as a managing entity in our architecture) is represented within the ontology, in order to make explicit how it uses knowledge and how it exploits distributed supports to accomplish its tasks. The inclusion of all these concepts in the ontology, making it more complete, provides also support to the system developing activity, which constitutes one of ontology main purposes.As mentioned in the previous section, expressive Web Ontology Language (OWL) has been adopted to formalize our ontology. More precisely, our ontology is an OWL DL one, i.e. it has been defined using the OWL DL version among three OWL sublanguages. OWL DL is a restriction of OWL Full, which is the most complete (but for this reason undecidable) sublanguage. However, OWL DL is enough expressive, more than OWL Lite, which is the simplest among OWL sublanguages. Before detailing ontology description, it is worth to briefly introduce which OWL DL modeling primitives are used to represent knowledge. As each OWL ontology, this ontology is structured following a hierarchical taxonomy of classes representing the main concepts of the domain. The root of each ontology concept is the general class *Thing*, which has no special features; all other concepts descend from it. Each concept is described and specified by a set of properties attached to the correspondent class. Properties can be used to represent both the features characterizing the concept itself and the associations relating the concept to other concepts of the domain: the former are called *DatatypeProperties* while the latter are defined *ObjectProperties*. *DatatypeProperties* can take different kinds of values, such as integer, float, boolean, string, etc.. The class to which a property is attached represents the "domain" of the property, while the class, or combination of classes, over which the property can take values, is its "range". Moreover, properties can be structured in a sub-property hierarchy, can have particular features (they can be symmetrical, transitive, functional) and can be related to each other by inverse relationships. Properties attached to the most general classes are inherited by their subclasses, and have the same domain and range. Constraints (called Restrictions) are used to further specify a property within the scope of a particular class, forcing the property to take some or all values over a restricted subset of its generic range or to take a particular value. The number of distinct values a property can take is specified by cardinality restrictions, which have a class scope, too. Another feature of the language is the possibility to define classes as arbitrary Boolean combinations (union, intersection, complement, disjunction) of other classes and to declare class equivalence and property equivalence.

Using the primitives outlined above, and taking advantage of OWL expressiveness, the ontology we developed is organized in a hierarchical structure. The root classes represent the most general, and consequently, the most reusable concepts of the domain and describe the knowledge at the highest level of abstrac-

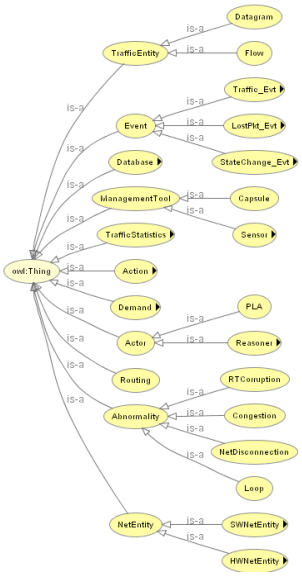


Figure 1.1: *First levels of Ontology with a "is-a" hierarchy*

tion. Exploring the hierarchy from root toward leaves, concepts lose generality to fit more suitably specific environments and architectures we considered.Figure 1.1 shows a comprehensive view of the ontology "is-a" hierarchical tree. The figure shows only first levels of the hierarchy, including all the *Thing* direct subclasses and some of their subclasses. Black arrows within a class indicate that the descending sub-tree is hidden.Network structural components are grouped in the *NetEntity* class. These components include the physical elements constituting the communication infrastructure (routers, hosts, links and interfaces) and also some software components, such as routing tables and the queues associated to node interfaces. Traffic, that is data flowing across the network and transmitted by network entities, is represented by the *TrafficEntity* class; it is viewed at different levels of abstraction and has direct references to the resources it takes up. Network elements functioning and traffic utilization of network resources are captured by status properties, whose values depend on time. Network global status is also determined by general and abstract concepts, such as *Routing* and *Demand*, which outline the set of interdependencies that contribute to define the overall network behaviour. The dynamical evolution of network status and traffic distribution is captured by the concept of *Event*, which has a precise temporal and spatial location and is related to the features it affects. *Events*, (locally caught by *Sensors*, special *ManagementTools*), can be part of the natural network dynamics or symptom of faults and abnormalities; so, they are necessary for a management application to keep up to date network representation during its evolution but also to detect faults and infer their root causes. In order to represent the generic cause-effect relationships that associate *Events* to their causes, the concept of *Abnormality* has been introduced; the representation of these associations makes a logical reasoning process able to merge

incoming events with its network representation and its high level knowledge to infer the real presence of an *Abnormality*. The remaining concepts of the ontology are mainly related to the elements of the management infrastructure: they represent how the *Reasoner*, viewed as an actor, can interact with network by means of other *Actors*, the Programmable Local Agents (PLA class). These interactions describe the mechanisms through which the *Reasoner* can retrieve data from the network, tune the monitoring system, actively modify network status and behaviour, and are represented by the concept of Action. All these actions allow *Reasoner* to diagnose *Abnormalities* and to perform statistical performance evaluations, by means of *TrafficStatistics*.

Chapter 2

Communication Infrastructure

The ontology represents all those elements of the communication infrastructure, which are necessary to define the structure and the functioning of the Internet network layer; moreover it includes some structural aspects of lower layers. The concepts represented allow the identification of network topology and the description of the main aspects of data transmission functionalities, such as routing and resource allocation for traffic management. All these concepts define the "managed objects", whose features and functioning parameters can be monitored and controlled.

All the hardware and software elements forming the communication infrastructure are modeled as *NetEntities*. The concept of *NetEntity* identifies every element which takes a role in network functioning, enabling it to carry out some tasks. The general class is specialized in two main subclasses: hardware entities (represented by the *HWNetEntity* class) and software entities (represented by the *SWNetEntity* class). The former are physical components of the network, while the latter are those non-physical elements, which explain network behavior, such, for instance, the resource allocation. The complete *NetEntity* hierarchy is shown in 2.1.
The *HWNetEntity* subclasses are:

- the *Node* class, divided into two subclasses, *Router* and *Host*;
- the *Iface* class, which represents the network boards that interface nodes to the rest of the network;
- the *Link* class, representing the physical bi-directional links which interconnect network nodes by means of their interfaces.

All these entities have a *HWStatus*, that is a **DatatypeProperty** defining their functioning status. This property can take one of three distinct values: "ON", "OFF" and "ABN". The status information "ON" indicates that the entity is active and properly functioning. The "OFF" status says that the entity, even

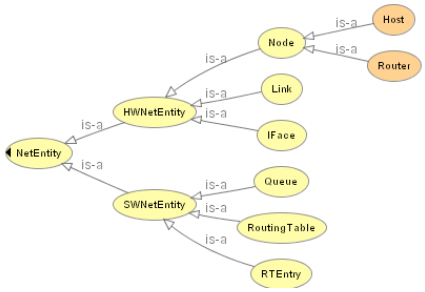
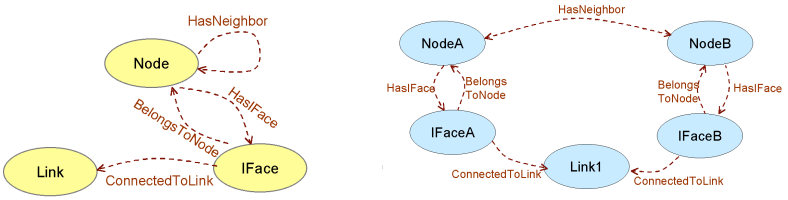


Figure 2.1: *NetEntity sub-hierarchy*

if disabled, can be yet controlled and activated if necessary. The "ABN" status indicates that the entity has gone out of order or is malfunctioning, and cannot be directly controlled. For example, if a link status is "ABN", it means that the link is down, that cannot be used to transport data and that it cannot be remotely reactivated before a direct recovery action restores its normal status

In order to identify a network topology, *HWNetEntities* are related among them by **ObjectProperties**, representing their physical associations and interconnections, as shown in figure 2.2(a). For example, in order to represent the association between an interface and the node on which it is installed, two properties are used: the *HasIFace* property and the *BelongsToNode* property. The former is declared in the Node class and has multiple values on the class *IFace*; the latter, vice versa, is declared in the *IFace* class and can take a single value on the *Node* class. These two properties are stated to be each the inverse of the other one, so they create a bi-directional reference between the two classes. Similarly *IFaces* are related to *Links* through the *ConnectedToLink* property, which allows to identify how the nodes they belong to are interconnected by means of physical links. The *ConnectedToLink* property can generally take multiple values. Although data are moved from an interface to another one by a single link, our model allows also the representation of backup links. Backup links can be opportunely activated to restore connectivity when a disconnection



(a) a (b) b

Figure 2.2: a. Object Properties for network topology definition. - b. Association among HWNetEntity instances defining the connection between two nodes.

occurs; the multiple cardinality of the *ConnectedTolink* property allows that the substitution of a "normal" link with its backup one un-affects the logical connection between two nodes. This way, namely, only the *ConnectedToLink* value changes, whereas the two nodes are reconnected using the same interfaces (and IP addresses. To constraint these concepts more explicitly, a boolean property, *Backup*, is attached to the *Links* class in order to represent each link type. The logical connection between two neighbor nodes has been explicitly represented, using the property which has been attached to the *Node* class; *HasNeighbor* is a symmetric property (i.e. if A is a neighbor of B, it is also true the inverse) and it relates members of the *Node* class to other members of the same class. The intrinsic limitations of OWL expressiveness do not allow to relate the *HasNeighbor* property to other topology information, thus limiting the knowledge representation. Figure 2.2(b) shows an example of *HWNetEntity* instances related among them by the above mentioned properties, and defining the connection between two nodes. A more deeply look at *HWNetEntities* reveals that each class owns several properties that define more precisely their nature and functioning. For instance *IFaces* are uniquely identified by their *Address* and have references to the queue associated to them, where packets waiting for processing or transmission are stored. *Links* are modeled in a very detailed way, because of their importance in monitoring resource allocation and traffic distribution and in improving network performance. They have properties (such as *Technology*, *Bandwidth* etc...), describing their physical features, and a *Cost*, which represents the routing information about their cost shared at a given moment by the whole network. Other important properties represent link usage, relating links with the traffic flows they are transmitting. We will describe in more details these last properties in the following sections, when traffic concepts will be clearer and traffic distribution representation will be exhaustively examined. Network nodes are catalogued as *Hosts* and *Routers* (figure 2.1), differing for the number of interfaces they have: the first are all those nodes that have more than one *Iface* (i.e. where the *HasIFace* property has a minimum cardinality of two) while *Hosts* have only one.All *Nodes*, *Routers* or *Hosts*, perform routing functions and need routing information to accomplish them. This is most important functioning aspect from a management point of view and it has been formalized by the *HasRoutingTable* property, which links each node with its own *RoutingTable* (figure 2.3).

The *RoutingTable* class constitutes a subclass of the more general *SWNetEntity* class (figure 2.1). The concept of *SWNetEntity* models the software

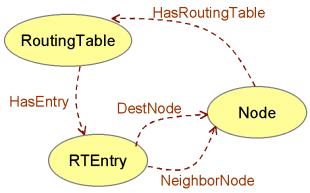


Figure 2.3: ObjectProperties relating the Node, RoutingTable and RTEntry classes.

components that play a central role in network functioning and that are relevant for monitoring purposes. The *SWNetEntity* class is further divided in three subclasses: besides the *RoutingTable* class, there are the *RTEntry* class and the *Queue class.*The first two classes, *RoutingTable* and *RTEntry*, are both used to describe the routing information present on a node. *RoutingTables* are defined as aggregations of instances of the *RTEntry* class, through the *HasEntry* **ObjectProperty**. Members of the *HasEntry* class represent routing table rows, and contain the information needed to forward packets toward a single destination; they have explicit references to the destination *Node* they concern (*DestNode* property) and to the neighbor *Node* (*NeighborNode* property) to which packets are forwarded to reach the destination. This specific representation choice, using a class to represent a single routing table entry, has made possible to express the ontological connection existing between routing information and the network physical nodes it refers to (see figure 2.3). The cost of the routing path toward the destination is also represented, by means of the *Cost* **DatatypeProperty**. In order to provide knowledge about the consistency and correctness of routing information of a node, a status DatatypeProperty, *RTStatus*, has been attached to the *RoutingTable* class. This property can take one of two values, "Normal" and "Corrupted", where the second one represents some entry lack or an abnormal state. Abnormal entries can present bit corruptions or infinite cost, and both cases make impossible routing for related destinations. The third *SWNetEntity* subclass is the *Queue* class (figure 2.1). This concept is used to represent the buffers where packets waiting for processing or transmission are temporarily stored. To model separately the two cases, the *QueueType* property has been introduced, with one of the two values "IN" and "OUT". *Queues* are associated to *IFaces* by the *AssociatedToIFace* property, while in the *IFace* class there are two distinct functional properties, *HasInQueue* and *HasOutQueue*, meaning that each *IFace* has exactly one *Queue* of "IN" type and exactly one *Queue* of "OUT" type. Queues, together with links, represent network resources whose occupation is an important parameter in measuring traffic load and network performances. Namely, high average levels of in-queues occupancy can be symptom of node computational resource lack with respect to the actual traffic load, while out-queue high levels occupancy can indicate link overload. The different meaning of in-queue and out-queue overflows is the reason for their accurate modeling. Because of their importance, they present properties defining their total size and their usage percentage, and explicit references (as we will see in the following section) to the traffic data they are used by.

Chapter 3

Traffic data and network traffic distribution

Traffic is an important component of network representation, because the main functionality the network offers to its users is to move data from a location to another one. Network management deals with the observation and evaluation of functioning parameters to understand whether and how well network meets efficiency and robustness requirements carrying out the above task. Therefore, it is impossible to build a network ontology whitout traffic concepts.
The root class of all traffic concepts is the *TrafficEntity* class, representing all data flowing through the whole network. Although data could be considered as "software" elements, they differ from the concepts represented by the *SWNetEntity* class: namely, while *SWNetEntities* are part of the network communication infrastructure that actively contribute to its functioning and participate in traffic management, *TrafficEntities* represent the traffic crossing the network and exploiting its infrastructure, its resources and its ability to move data. The general concept of *TrafficEntity* (see figure 3.1) is specialized into two subclasses, *Datagram* and *Flow*, representing data at different levels of abstraction. The first one identifies a single IP Datagram, which is the atomic traffic unit managed by the Internet network layer, and it has properties representing its typical features like size, source and destination parameters (*Length*, *SourceNode* and *DestNode* properties). The Flow class defines traffic at higher level, aggregating datagrams flowing along the same routing path and thus sharing the same source and destination parameters. The Flow class owns the *HasDatagram* property, it takes multiple values over the *Datagram* class and it defines the relationship with the set of units composing the flow. Moreover, the Flow class has the *SourceNode*

Figure 3.1: TrafficEntity hierarchy

and *DestNode* properties, but, due to OWL limitations, the constraint between their values and the correspondent values in the Datagrams composing the flow must remain implicit. In order to locate the Flow concept into the temporal dimension, two datatype properties, *StartTime* and *EndTime*, have been attached to the class defining the time interval during which the flow takes place. A flow is intended as a continuous data stream, although it is composed of discrete elements (i.e. datagrams); two datagrams sharing the same routing path can be considered as part of the same flow if they occur at two sufficiently close time instants. It should be noticed that the Flow concept modelled here is only a relaxed version of the typical "network connection", identified by five parameters: $\langle SourceNode, DestNode, SourcePort, DestPort, TransportProtocol \rangle$; this choice has been motivated by the monitoring and controlling tasks the ontology is defined for and fits well the representation of the three lower layers of the Internet only. Roughly speaking, this view of a flow captures only those parameters related to the routing function and those features sufficient to understand traffic distribution. In order to explain how network resources are allocated for data transmission, specific properties have been introduced explicitly binding traffic data to the resources, that is queues and links, they exploit. For instance, flows have bi-directional references to the links they flow through by means of the *OnLink* property in the *Flow* class and the *UsedByFlow* in *Link* class. The same way, *Queues* refer to the *Datagrams* they contain at a given instant through the *HasDatagram* property. Because of the importance of resource usage measurement, additional properties, such as *UsedBandwidth* and *FlowBandwidth* in the *Link* class or *UsedSpace* in the *Queue* class, precisely quantify data load. The *UsedBandwidth* property defines the overall used bandwidth percentage of a link, while the *FlowBandwidth* property defines the bandwidth used by a single flow in bit per second. It is implicit the association between those flows referred in the *UsedByFlow* property and the correspondent usage bandwidth expressed in the *FlowBandwidth* property; the binding between the sum of these flow bandwidths and the total *UsedBandwidth* is implicit too. Figure 3.2 shows the associations and properties used to represent traffic distribution.

The classes and properties described above define the structural aspects of network functioning and traffic. Nevertheless, some interdependencies cannot be captured through a detailed and low level description of single elements and need a more general view of the global network status and traffic distribution. For example, the links holding a flow are simply determined by the routing tables of the intermediate nodes on the path from source to destination, and the flow rate depends on the data amount the source tries to transmit; Furthermore, a

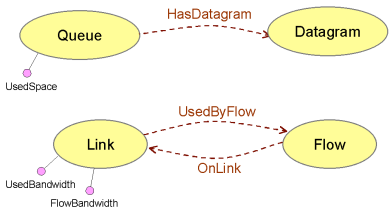


Figure 3.2: Association relating TrafficEntity subclasses to the used resources

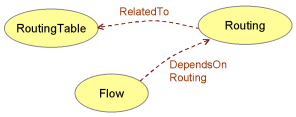


Figure 3.3: Associations between *Routing*, *RoutingTable* and *Flow* classes

flow depends only on these routing tables or is only determined by its source demand. Namely, the flow rate (and the used links bandwidth) depends also on the presence and the rate of other flows, sharing the same links and the computational resources at the same nodes. To express this kind of knowledge, two more general classes have been introduced, formalizing the abstract concepts of *Routing* and *Demand* respectively. Although these classes have not a direct binding to any network element, they are useful in describing network load and performances. Both classes are direct *Thing* subclasses (see figure 1.1).

The *Routing* concept is used to represent how the routing in the whole network, determines resource allocation and usage, thus affecting performances. This concept has two main relationships with other ontology classes: the *RelatedTo* property, which links the *Routing* class to the *RoutingTable* one, and the *DependsOnRouting* property, relating the class *Flow* to the class *Routing* (see figure 3.3). The first property shows that node routing tables represent concretely the global routing activity, while the second one represents the dependency between the resources employed by a flow along its path and the routing activity performed by all network nodes, which forces other flows to share the same path.

The *Demand* concept represents the hypothetical traffic load that would be transmitted over the network without resource limitations. The *Demand* class owns only two properties: *StartTime* and *EndTime*, representing the considered time interval. The *Demand* class is furtherly specialized into two subclasses: *DemandMatrix* and *DMEntry* (see figure 3.4), in which each source demand is quantified. The *DemandMatrix* class is an aggregation of *DMEntries*, each one representing the data amount (*Value* property) each source wanted to transmit toward all the possible destinations (*SourceNode* and *DestNode* properties). *Demand* is obviously an abstract concept, but its representation is useful to make explicit that it determines real traffic load. This relationship is expressed

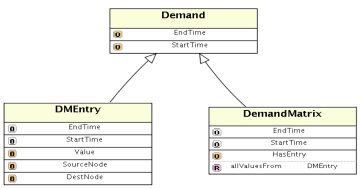


Figure 3.4: *Demand* class and its subclasses

Figure 3.5: *ObjectProperty* relating the *Flow* and *Demand* classes

by the *DependsOnDemand* property, linking members of the *Flow* class to the *Demand* class (see figure 3.5).

Chapter 4

Events and abnormalities

As already said, *NetEntities* define network structural describing their statu; *TrafficEntities* describe network traffic and its distribution over *Links* and *Queues*, *Routing* and *Demand* outline the internal mechanisms that generate a particular global status of the network. The external representation of network dynamic evolution from a state to the following one can be done introducing the concept of *Event*. The *Event* class represents a general concept, used to identify whatever can happen in the network; it has the only *Time* and *OnEntity* properties, indicating the instant and location at which it takes place. This general concept is then specified into more concrete classes which model different kind of event. *Event* hierarchy is displayed in figure 4.1.

The Event class has three main subclasses, which group together events having a similar nature and affecting network state in a similar way: *LostPkt_Evt*, *StateChange_Evt* and *Traffic_Evt*. *LostPkt_Evts* are used to indicate that a datagram has been discarded by a router, and they are subdivided according to the local reason that has caused the loss: *RT_LostPkts* indicate datagram losses because of forwarding impossibility during the execution of the routing function, while *TTL_LostPkts* indicate datagram discarding because of TimeToLive expiry. *StateChange_Evts* indicate functioning status changes on hardware or software entities (such as the going up or down of a *Link* or a *Rout-*

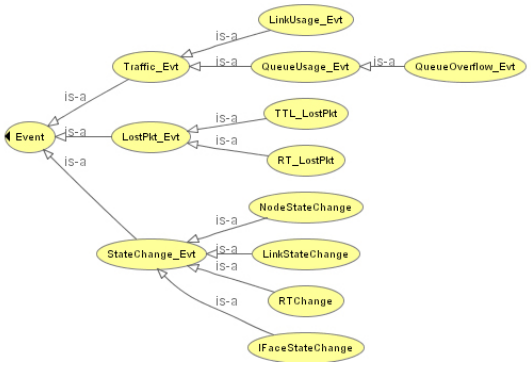


Figure 4.1: Event hierarchy

ingTable updating) and are further subdivided into *NodeStateChange*, *LinkStateChange*, *IFaceStateChange* and *RTChange*. *Traffic_Evts* describe the dynamic flowing of traffic data through network resources and they are distinguished in *LinkUsage_Evt* and *QueueUsage_Evt*. This last event is further specialized by a more specific event, the *QueueOverflow_Evt*.
Events in network changing play a role that is expressed by properties relating them to entities where they take place and to functioning aspects they affect. The *Event* class owns the *OnEntity* property, relating it to the *NetEntity* where the event takes place. Event subclasses have restrictions about the entity type in the *OnEntity* range. For example, within the *StateChange_Evt* class, the value of the *OnEntity* property identifies the hardware or software entity whose state has changed; in its *LinkStateChange* subclass, the same property is restricted to have values only over the *Link* class. The three *StateChange_Evt* subclasses, *LinkStateChange*, *IFaceStateChange* and *NodeStateChange* classes, also have the *ChangeType* property which can have values "ON" and "OFF", indicating whether the pointed *HWEntity* is now active or not. Instead, the *RTStateChange* class has the *ChangedEntry* property which takes values over the *RTEntry* class, and explicitly indicates the entries that have been substituted, deleted or added. Events descending from the *Traffic_Evt* present the same properties used before to indicate resource usage, this mean that they update this property values accordingly to actual traffic distribution. Therefore, the *LinkUsage_Evt* has the *UsedByFlow* and *FlowBandwidth* properties, notifying the change of the link bandwidth used by a single flow (this way, we also made explicit the binding between a flow and the correspondent bandwidth it takes on a specific link, a relation that cannot be expressed in the *Link* class). In a similar way, a *QueueUsage_Evt* has the *UsedSpace* property, that is also owned by the *Queue* class. As an example, figure 4.2 shows how the relationship between a *QueueUsage_Evt* and the features it affects is represented within the ontology.

Events embody the knowledge necessary for a management knowledge-based system to keep up to date its network representation and to undertake diagnosis tasks. They can be part of normal network dynamics or can be considered as fault symptoms. In this second case, the root cause of such symptoms is formalized, within the ontology, in the *Abnormality* class and is related to events

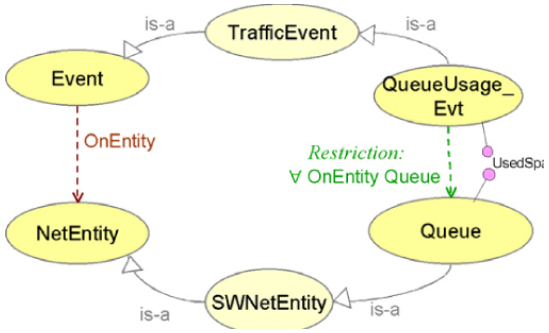


Figure 4.2: Association between *Event* and *Entity* classes

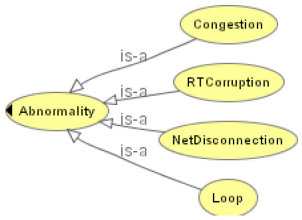


Figure 4.3: Abnormality hierarchy

by general cause-effect relationships. An *Abnormality* is linked to the set of events it causes by means of the *Causes* property, inverse of the *CausedBy* property in the *Event* class. *Events* and *Abnormalities* have a very different ontological nature: while Events are instantaneous manifestations of a change or happening, have a precise temporal and spatial location and can be locally caught by "ad hoc" sensors, *Abnormalities* represent a more global kind of information and define a misfunctioning state, which can involve many network elements placed at different locations, can persist over time and can reveal itself in a wide range of external symptoms. For this reasons *Abnormalities* and *Events* have no inheritance relationships in their ontological representation and they are direct subclasses of the generic *Thing* class. The multiple symptom manifestation is reflected by the multiple cardinality of the *Causes* property.

The abnormalities represented in our ontology are those detected by the Logical Reasoner; therefore, as shown in figure 4.3, there are four *Abnormality* subclasses: *Loop*, *Disconnection*, *RTCorruption* and *Congestion*. Each of these classes has suitable properties defining what are entities involved in the abnormal situation and, what is the *Abnormality* location (e.g. source and destination parameters of cyclic or disconnected paths, nodes and links involved in a congestion situation etc...). Moreover, each *Abnormality* is related explicitly to its possible symptoms by restrictions on the *Causes* property. Restrictions have been placed on the *CausedBy* property of the *Event* class, too. For instance, through restrictions the *Causes* property in the *Disconnection* class is forced to take values only over the *RT_LostPkt* class, while this kind of event can in turn be caused by both Disconnection and *RTCorruption* abnormalities. All the restrictions expressing the cause-effect relationships between *Event* and *Abnormality* subclasses are shown in figure 4.4. It is worth to notice that, in our ontological representation, *Events* are not classified on the basis of their normal or abnormal nature, but with respect to the network change they produce. This form of representation derives from the active interpretation performed by a reasoning process that identifies the abnormal nature of an *Event* and detects its *Abnormality* root cause. Namely, while *Events* can be locally detected, Abnormalities need an analysis process to be detected, combining different events with more general knowledge and with global status information. For example, a *Disconnection* abnormality, which commonly causes *RT_LostPkt* events on the involved routers (where specific sensors are installed), can't be locally identified; it is necessary a comprehensive view of network status and a high

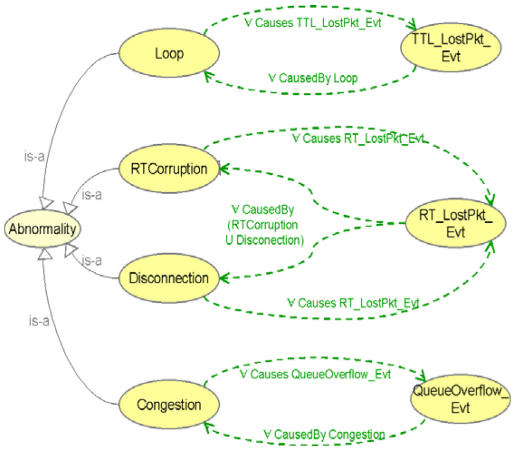


Figure 4.4: Restrictions on the *Causes/CausedBy* properties relating *Abnormality* and *Event* subclasses

level knowledge about *RT_LostPkt* possible causes to determine its presence. Since *Events* are very important in the execution of management tasks, they represent a central concept of this ontology and occur as reference in a wide set of properties, describing how they are locally caught and stored, notified to the central management application and used to undertake fault diagnosis and statistical performance evaluation. All these properties will be full detailed in the remainder of the paper.

Chapter 5

Actions and tools for the execution of management tasks

The ontology previously described covers the conceptual definition of the network as a self-contained system; it features its structure, functioning and dynamic evolution, capturing the high level knowledge that is necessary to understand the different factors, interdependencies and cause-effect relationships which contribute to determine network behavior. This account of knowledge makes a management application able to perform high-level management tasks. Even though the "operational knowledge", which such an application uses to solve problems and undertake specific strategies, is not represented (according to ontology purposes), our ontology models the most representative management infrastructure elements and the most representative interactions that enable management. Therefore ontology doesn't represent, for instance, the logical path the reasoner uses to classify events and detect abnormalities, nor the criteria it uses to evaluate performances or the reasoning and computations it performs to decide on suitable interventions. The ontology describes the underlying infrastructure and knowledge necessary to accomplish these tasks, making explicit the relationship between monitoring and controlling tools and the network functioning parameters they respectively observe and affect, thus providing the Logical Reasoner with a high level view of how to use them. However, the first part of the ontology remains the most general and, therefore, the most reusable one.
Next section briefly introduces the system architecture and outlines how its main elements are represented within the ontology. Then, the following paragraphs full detail ontological representation of the interactions between these elements, grouping them according to phase they are involved in and the management tasks they support.

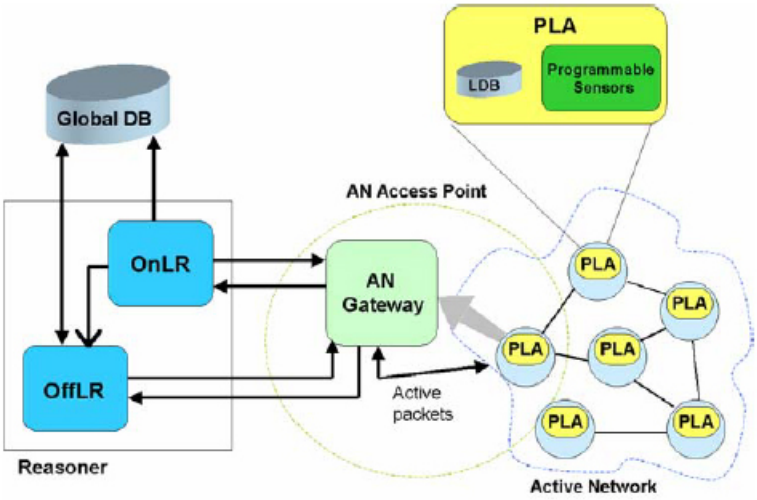


Figure 5.1: Architecture overview

5.1 Architecture overview and architectural element representation

The management system we are developing is based on the presence of a logical inference engine, the Reasoner, and distributed agents, the programmable Local Agents. The first one acts as a centralized management application while the second one provide the Reasoner with support for the execution of high-level management tasks.

The architectural framework is based on the Active Network paradigm. Reasoner and PLAs represent the end points of management communication. Communication is made possible by a gateway service provided by one of network nodes. The Reasoner issues command and receives information from ANGateway having the task of receveing and sending active packets from the network; active packets can contain code to be executed on network nodes. Figure 5.1 shows the network management architecture and its main elements, while figure 5.2 shows the main classes used by the ontology to represent them.
PLAs are provided with a set of sensors, which are able to monitor specific aspects of the node, that is, specific functioning parameters. Thanks to network programmability provided by Active Network, sensors can be dynamically deployed by Reasoner across the whole network; moreover they can be opportunely tuned by Reasoner exploiting PLAs services. Sensors are represented, within the ontology, by the *Sensor* class, subclass of the *ManagementTool* class (see figure 5.3), which groups together some instruments that the managing entities, (Reasoner and PLAs) can use to perform management tasks. Beside the *Sensor* class, another *ManagementTool* specialization is the *Capsule* class, which represents active packets containing some code to be executed on network nodes. Sensor ability to capture local events is expressed by the *CatchesEvent* property, relating the *Sensor* class to the *Event* one. The Sensor class is further subdivided into more specific classes, each one representing a particular

Figure 5.2: The main classes used to represent the management framework

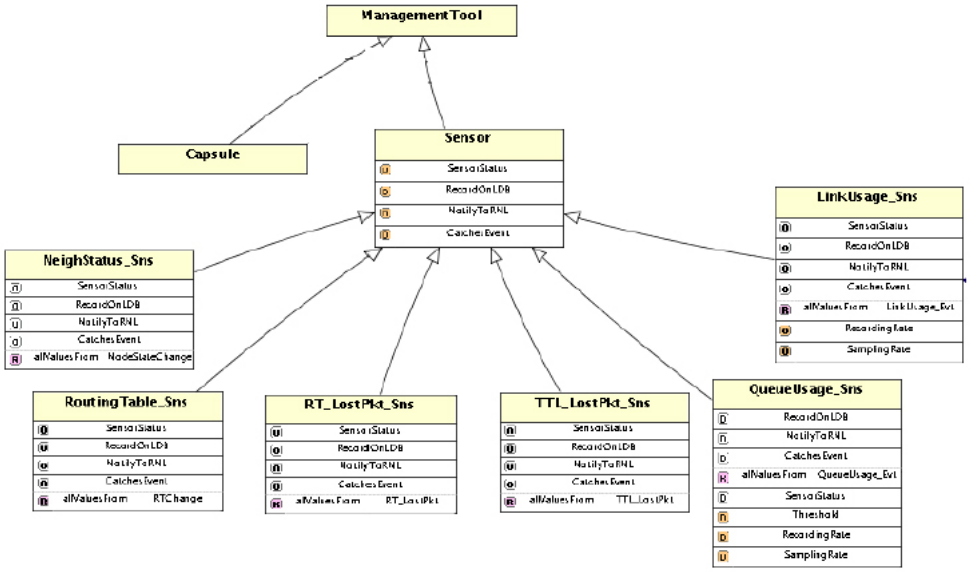


Figure 5.3: *ManagementTool* hierarchy

kind of sensor, which catches a particular kind of event. Furthermore, some *Sensors* have additional properties, which represent monitoring parameters the Reasoner can tune. Sensor tuning consists in the variation of these parameters, such as the sampling frequency at which observations take place or the conditions under which events are raised. For example, the *QueueUsage_Sns* class have the *SamplingRate* property, which represents the sampling frequency at which queues are observed; instead, the *Threshold* parameter represents the alarm value above which a *QueueOverflow_Evt* has to be raised. The Reasoner performs two kinds of reasoning processes: reactive on line reasoning and off-line analyses about past. Since these two management activities have been developed as independent tasks, two different reasoning agents can be singled out. The OnLineReasoner (OnLR) performs dynamic reasoning: it collects real-time information about the network, reasons about it to infer abnormalities and, on the basis of its inferences, opportunely reacts to tune the monitoring system and to undertake fault recovery interventions. Complex statistical analysis and performance evaluation are more difficult tasks, involving knowledge about network past functioning and requiring complex reasoning; they are executed by the OffLineReasoner (OffLR), which can in turn suggest suitable actions aiming at improving the quality of service. OffLR suggestions are then transformed into concrete interventions by the OnLR; namely, it is the only agent that effectively performs control tasks and issues commands to be locally executed by the distributed agents.
The distributed agents, then, have two accomplish two main tasks: they have to manage monitoring information, making it available for the Reasoner, and to execute issued commands, allowing remote control of managed devices.
PLAs and Reasoner are represented as *Actor* subclasses (see figure 5.2). The *Actor* concept has been introduced in order to represent the active role played by managing agents. *Actors* are described through their properties and, through the actions they perform, which make clear actors interactions and the mechanisms through which they carry out their tasks.
PLA class is shown in figure 5.4. It has some properties identifying PLA location, status and managed elements. It has the *OnNode* property, which specifies the node on which the PLA is running and takes values in the *Node* class. The *PLAStatus* DatatypeProperty defines PLA status, expressing whether the PLA is now active or not. Since PLAs are responsible for sensor concrete tuning and for observations notification and storing, they have references to the installed sensors and to the installed LDB; these associations are represented by the *HasSensor* and *HasLDB* properties, which have value respectively in the *Sensor* and *LDB* classes. The *SupportedSensorType* property represents all kinds of sensors that can be installed on the considered PLA. PLAs representation, together with the actions they perform, lets one understand the information flow from the network to the Reasoner for monitoring information retrieval and, vice versa, the command flow from the Reasoner to the network for control purposes.
Figure 5.5 shows the *Reasoner* class; it has the property *InfersAbnormality* describing its reasoning capability and, expressing one of its features that cannot be seen through its actions.
The *Reasoner* class is further specialized into two subclasses, the *OnLineReasoner* (*OnLR*) and *OffLineReasoner* (*OffLR*), which represent the two different agents into which the Reasoner is decomposed. The role distinction between the two agents is made clear by the different kinds of actions they execute, as

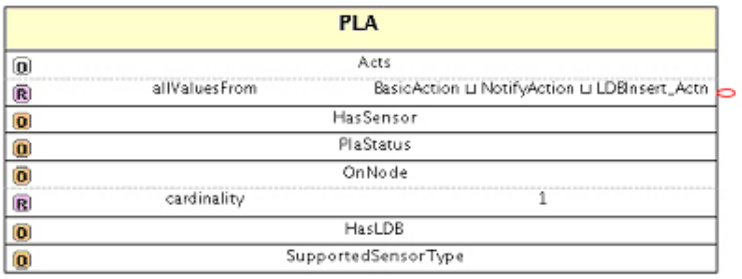


Figure 5.4: *PLA* class

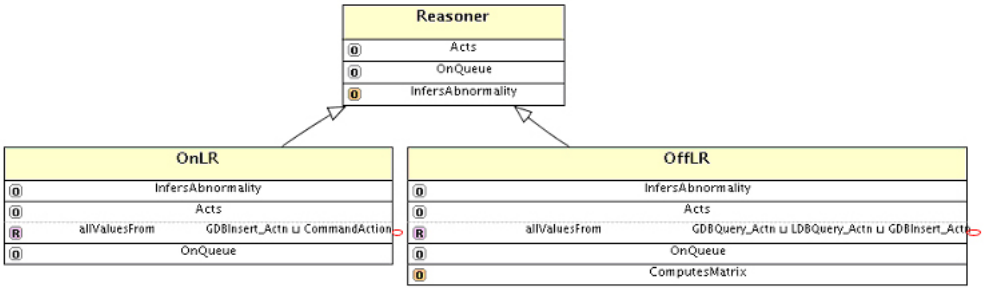


Figure 5.5: *Reasoner* class and its subclasses

shown by opportune restrictions on their *Acts* property. Another distinction between the *OnLR* and the *OffLR* classes is represented by the *ComputesMatrix* property, only present in the *OffLR* class. This property ranges over the *TrafficMatrix* class, which is a subclass of the more general *TrafficStatistics* one representing statistical information about traffic load and traffic distribution and are drawn out by means of complex computations that the *OffLR* is able to perform. The OffLR retrieves the information it need from suitable information bases. Sensor observations are locally stored into special databases, called LocalDatabases, installed on network nodes and managed by PLA. LocalDatabases can be remotely queried by the OffLR and they are not the only information source for the OffLR. Nemely, there is also a centralized database, called GlobalDatabase (GDB), where both the OnLR and the OffLR store the results of their inferential process. GDB information is retrieved only by the OffLR, to support its reasoning about past. The two database are represented, in the ontology, by the *LocalDatabase* and *GlobalDatabase* classes, both direct subclasses of the *Database* class (see figure 5.6). The *Database* class has no special properties, while each of its subclasses has a single property which establishes the correspondence between the stored information and the network aspects it concerns.

In particular, the *LDB* class has the *CaughtEvent* property, while the *GDB* has the *InferredAbnormality* property. *LDBs*, in fact, store rough data about all monitored functioning parameters, that is, they store all detected events.

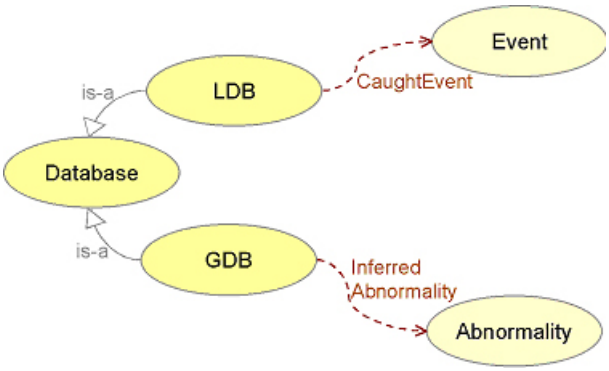


Figure 5.6: *Database* hierarchy and the ObjectProperties of related subclasses

The *GDB*, instead, stores higher level information about past, so its information content is more dense and meaningful. The *InferredAbnormality* property, relating the *GDB* class to the stored *Abnormalities*, expresses the above concept. The two *CaughtEvent* and *InferredAbnormality* properties, making clear the ontological relationship that holds among stored information and network functioning, provide this information with univocal semantics. To understand what is actors' active role, it is necessary to examine their actions. The *Action* class and the Actor one are related to each other by two inverse properties: the *Acts* property in the *Actor* class, which ranges over the Action class, and the *ActedBy* property in the *Action* class (see figure 5.7). *Actions* are classified on the basis of their nature and of the effects they are able to produce in network behavior; therefore, each actor can execute different kinds of actions. To express explicitly the role of each actor, restrictions are attached to the *Acts* and *ActedBy* within the *Actor* and *Action* subclasses, respectively. These restrictions identify what actions each actor performs and, on the contrary, what actors can execute each action. Figure 5.8 shows all these restrictions, giving a first overview of *Actors* role and, displays all of *Action* direct subclasses.
Since *Actions* are used to describe the mechanisms through which management takes place, they will be described in detail according to the management tasks they are involved in.

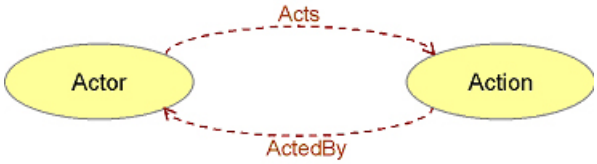


Figure 5.7: Relationships binding *Actor* and Action classes

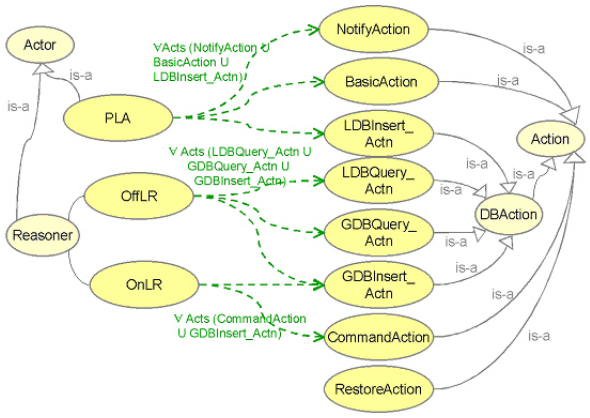


Figure 5.8: Restrictions on the *Acts* property expressing what kind of *Action* each *Actor* performs

5.2 Information gathering

In order to explain the mechanisms of monitoring information collection and their storage in the Reasoner, the ontology describes the events capturing mechanisms and those *Actions* executed by *PLAs* to store information locally or to provide the OnLR with real-time data; it also describes those *Actions* the OffLR uses to retrieve information from databases about past network functioning, and those that both Reasoner agents use to store their inferential results for further off-line reasoning. The first step toward information gathering is represented by the sensor monitoring activity. Each *Sensor* instance is linked to those events it captures through the *CatchesEvent* property. This property, whose generic range is the *Event* class, has suitable restrictions in each *Sensor* subclass, expressing what sensors capture what events: for instance, sensors of *TTL_LostPkt_Sns* type are related to instances of the *TTL_LostPKT_Evt* class and, similarly, the *RoutingTable_Sns* class is related to the *RTStateChange* one (see figure 5.9).

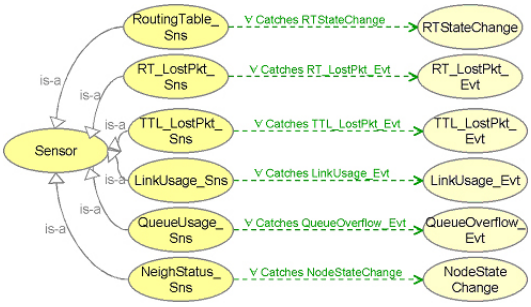


Figure 5.9: Restrictions on the *CatchesEvent* property, expressing what kind of *Event* each *Sensor* captures

The representation of the ontological association between *Sensors* and *Events* is necessary for the OnLineReasoner in order to opportunely deploy sensors over the network and to tune them according to the current network state; only if the Reasoner knows what sensors are able to collect the information it needs, it can activate and adjust them to capture the right information at the right moment and place.
Sensors and sensor observations are locally managed by the Programmable Local Agents running on network nodes. PLAs notify events to the Reasoner throughhe actions that are represented by the *NotifyAction* class, direct subclass of the *Action* class. The *NotifyAction* class owns three main properties, *Notifies*, *CaughtEvent* and *SensorID*, and its *ActedBy* property is restricted to take values in the *PLA* class. The *CaughtEvent* and the *SensorID* properties are used to identify the notified event and the sensor which has caught it. The *Notifies* property relates the action to the actor who is notified. Since the OnLR is the only agent which need real-time information about network dynamic evolution to support its reactive behavior, the *Notifies* property has values only in the *OnLR* class. Figure 5.10 shows how events are caught and notified in real-time.
To store sensor observations into the LocalDatabases, PLAs perform the *LDBInsert_Actn*; it is a special kind of *DBAction*, which is one of the *Action* direct subclasses. The *LDBInsert_Actn* class (as the *NotifyAction* one) has the *CaughtEvent* and *SensorID* properties; these properties relate it to the *Event* that has to be stored and to the *Sensor* which has caught it (see figure 5.11).

The *NotifyAction* and the *LDBInsert_Actn* represent the two main services concerning sensor observation treatment. Since the OnLR can enable or disable notification and recording services as necessary, the above actions are performed only when required; for this reason two flags have been attached to the *Sensor* class; moreover the *NotifyAction* and *LDBInsert_Actn* has been bound to their values. The two flags are *NotifyToRNL* and *RecordOnLDB*, and indicate whether the notification and recording services, respectively, are enabled. Within the *NotifyAction* and the *LDBInsert_Actn* classes, the *SensorID* property is constrained to take values only over those instances of the *Sensor* class having the value "true" in their *NotifyToRNL* and *RecordOnLDB* flags respectively. Moreover, in the *Sensor* class, the *SensorStatus* property indicates

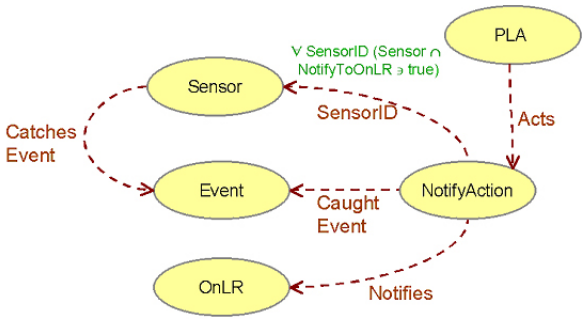


Figure 5.10: Relationships expressing how sensor observations are notified to the OnLR

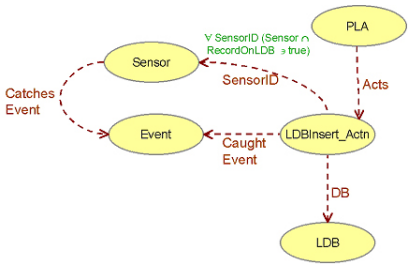


Figure 5.11: Properties expressing how sensor observations are stored into LocalDatabases

whether a *Sensor* is "ON" or "OFF", where an "OFF" status implies that both the two mentioned flags are "false".
Since the OffLR is devoted to off-line complex computations, it does not "react" to network evolution, nor it takes any measure directly. So, the only actions the OffLR perform aim at information retrieval and storing. Off-line information retrieval is formalized by the *LDBQuery_Actn* and *GDBQuery_Actn* classes (see figure 5.12). To collect rough data directly from *LocalDatabases*, the OffLR performs the *LDBQuery_Actn*. Through this action, it can retrieve all the events that have taken place during a specific time interval. The time interval is specified by the *StartTime* and *EndTime* properties, while the collected events are pointed by the *CaughtEvent* property. The *LDBQuery_Actn* also inherits from the *Database* class the *DB* property, which identifies the queried *Database*. The *GDBQuery_Actn* class, very similar to the *LDBQuery_Actn* one, represents information retrieval from the *GDB*. Reflecting the type of information *GDB* stores, it differs from the *LDBQuery_Actn* class only for the *InferredAbnormality* property, which substitutes the *CaughtEvent* property. It takes values over the *Abnormality* class and identifies those *Abnormalities*, eventually of a specific kind, which have involved the network during the specified time interval.

The actions that permit the storage of information about detected abnormalities in the GlobalDatabase are represented as instances of the *GDBInsert_Actn*

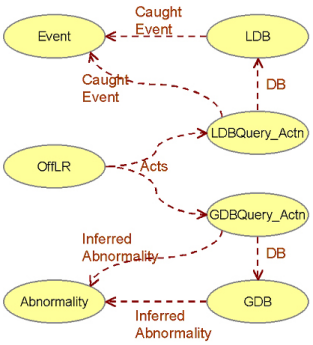


Figure 5.12: Actions performed by OffLR to retrieve information and related associations

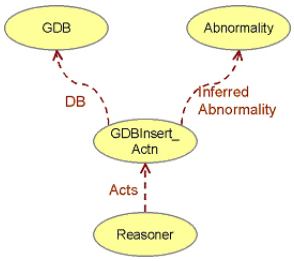


Figure 5.13: The Action used to store information into the GlobalDatabase and its related associations

class, also subclass of the *DBAction* one. As shown in figure 5.13, this action is executed by both OnLR and OffLR to allow further off-line reasoning. The *GDBInsert_Actn* class is very similar to the *LDBInsert_Actn* one, instead of the *CaughtEvent* property it owns the *InferredAbnormality* one.

5.3 Network control and monitoring

In this section we are going to analyse the control mechanisms through which network control can be performed by the OnLR. We will also describe the *RestoreAction* class, which represents control actions executed by external actors. Since the OnLR is devoted to reactive management, the event notification makes it able to keep up to date its network representation, following network dynamic evolution. Moreover it can perform on-line inferences by merging incoming events with its network status representation.
The OnLR can then use the results of its on-line reasoning to "react" to network status. This reactive behavior shows the OnLR in two functionalities: fault recovery and monitoring tuning. Moreover, since the OnLR can interact with the network to affect its behavior, it undertakes actions on the basis of OffLR suggestions aiming at performance improving. To accomplish control tasks and to opportunely focus monitoring resources, the OnLR needs to know the remote services it can exploit and the commands it can issue; in particular, it has to know sensor features and to represent how its actions affect network behavior. For this reason, the ontology includes classes and properties to represent this knowledge. In order to represent explicitly OnLR ability to influence network behavior, the *CommandAction* class has been introduced. *CommandActions* are in opposition to the *BasicAction* class, which represents those actions performed by PLAs on the basis of OnLR commands. The ontological difference between the two classes is that *BasicActions* can directly affect network behavior, while *CommandActions* cannot do it. Nevertheless, *CommandActions* representation is necessary to express OnLR ability to initiate a sequence of steps that will finally cause network status changes. Namely, each *CommandAction* is related to the correspondent *BasicAction* to be executed, by means of the *RequestedAction* property. To explicitly represent effects produced by *BasicActions*, they have the *Causes* property relating them to the *Events* they cause. The *BasicAction* class is intended to represent the internal mechanism that forces a change, while *Event* instances represent their external manifestation. As it should be noticed,

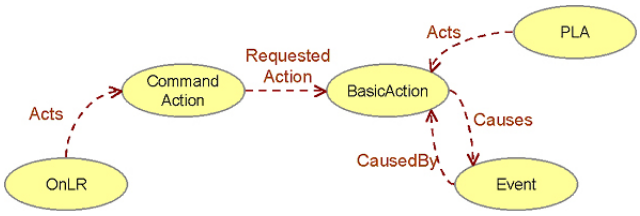


Figure 5.14: Set of properties which represent how OnLR commands are executed and affect network behavior

this is the same cause-effect relationship between *Events* and *Abnormalities*; this is the reason why the same pair of properties, *Cause/CausedBy*, is used in both cases. *Events* caused by *BasicAction* instances can be detected by sensors as other events. The representation of the class chain between *OnLR*, *CommandAction*, *BasicAction*, *Event* (see figure 5.14) provides OnLR commands with clear semantics, since it expresses explicitly what is their ontological association with the network functioning changes they determine.

CommandAction and *BasicAction* classes are both direct subclasses of the *Action* class and they have the same hierarchical structure, creating, then, a one-to-one correspondence between them. This correspondence is clarified by the *RequestedAction* property, which, as mentioned above, links each *CommandAction* to the correspondent *BasicAction* to be executed. Restrictions are placed on the *RequestedAction* class within each *CommandAction* subclass to link each command only to the correspondent *BasicAction*. Figure 5.15 shows *CommandAction* hierarchy.

The *CommandAction* class also has the *ToPLA* property, which has values in the *PLA* class and identifies the PLA whose support is requested for command execution. The *CommandAction* class is specified (as *BasicAction* class) into three subclasses: *ChangeLinkCost_Cmd*, *SwitchLink_Cmd* and *TuneSensor_Cmd*, whose correspondent classes in the *BasicAction* subtree are respectively *ChangeLinkCost_Actn*, *SwitchLink_Actn* and *TuneSensor_Actn*. While the first two classes represent network control actions, the *TuneSensor_Cmd* groups together all those actions through which the OnLR can adjust sensors in order to capture the most meaningful information at the right moments and places.

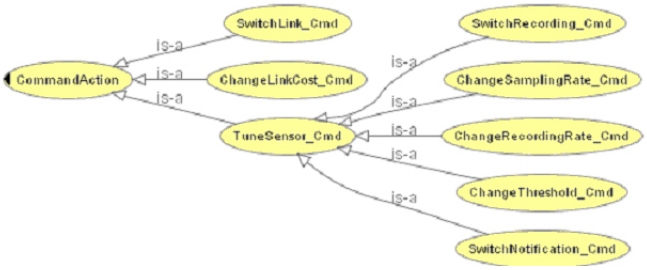


Figure 5.15: *CommandAction* hierarchy

Parameters necessary to specify each command are not included in the considered *CommandAction* subclass, but only in the related *BasicAction*. Namely, since the *CommandAction* is already related to the *BasicAction* by means of its *RequestedAction* property, further parameter specification would be redundant. For example, the link to be switched is specified within the *SwitchLink_Actn* class (through the *OnLink* property) and not in the *SwitchLink_Cmd* class.
The *SwitchLink_Cmd* is an example of a reactive control action, which causes instances of the *LinkStateChange_Evt*. The *ChangeType* property attached to the *SwitchLink_Actn* specifies if the link has to be activated or disabled. Switching on a link represents a fault recovery action. Namely, it is issued by the OnLR when a disconnection occurs in order to restore connectivity; it takes advantage from backup link presence (modelled within the ontology as *HWNetEntities* description explains). The *ChangeLinkCost_Cmd* is an example of those actions, undertaken on OffLR suggestion, aiming at improving network performances. They need complex off-line statistical analyses for network performance evaluation and are the result of suitable algorithms for OSPF link weight optimisation. On the basis of this computation, the OnLR effectively orders the link cost redistribution, issuing the *ChangeLinkCost_Cmd*. The requested *ChangeLinkCost_Actn* presents the *OnLink* and *Cost* properties. The former relates the action to the link it concerns. The latter is a datatype property that identifies the new link weight. It is interesting to notice that the *Causes* property of this *BasicAction* is restricted to take values in the *RTStateChange* class. This association makes clear the ontological relationship between the action and the functioning aspects it affects; together with the *Routing* concept, it represents what knowledge the Reasoner need to capture how traffic distribution is related to the routing activity and, so, how traffic can be redistributed to exploit better available resources and meet user needs.
The *TuneSensor_Cmd* class is subdivided into five subclasses, each modifying a monitoring parameter of programmable sensors. Its correspondent *TuneSensor_Actn* class has the *SensorID* property, which identifies what sensor has to be tuned. Each *TuneSensor_Actn* subclass has suitable restrictions on its *SensorID* property, necessary when the action affects monitoring parameters that only some sensors have. For example, the *ChangeThreshold_Actn* concerns only the *QueueUsage_Sns* class, because only *QueueUsage* sensors have a threshold parameter. Two *TuneSensor_Actn* subclasses, *SwitchRecording_Actn* and *SwitchNotification_Actn* concern the enabling of the two main services associated to sensor observations described above, in the context of information gathering. These actions affect the two flags, *NotifyToRNL* and *RecordOnLDB*, in the *Sensor* instance pointed by the *SensorID* property. The other three *TuneSensor_Actn* subclasses, *ChangeRecordingRate*, *ChangeSamplingRate* and *ChangeThreshold*, are used to change sensor monitoring parameters. The considered parameter is specified by the same properties that describe it in the *Sensor* class. For example, the new threshold value for a *QueueUsage_Sns* instance is specified by the *Threshold* property in the *ChangeThreshold_Actn* class, which is the same property as in the *QueueUsage_Sns* class. Knowledge about the sensor monitoring parameters and the mechanisms to change them can be used by the OnLR not only to gather the necessary information, but also to make the right information available for OffLR statistical computations. For example, the traffic sensors have many parameters (SamplingRate, RecordingRate, Threshold) which the OnLR tunes according to the actual traffic load in or-

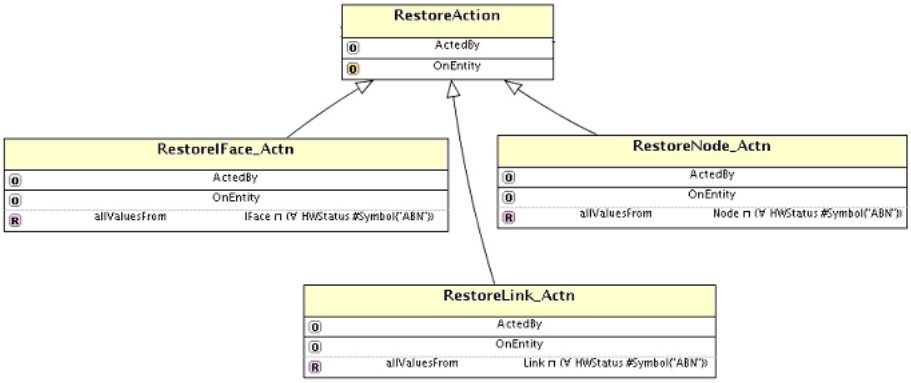


Figure 5.16: *RestoreAction* class hierarchy

der to focus attention and intensify monitoring in those areas where congestion situations have been inferred. So, the OffLR will successively have much more information about the most meaningful periods and areas for performance evaluation purposes. A special kind of control actions, which are not acted by the actors represented within the ontology, is the *RestoreAction* class. This class represents those actions performed by external actors (i.e. human operators) in order to recover the proper functioning of some entities. *RestoreActions* are necessary when an entity is gone out of order and cannot be remotely controlled because it needs a direct intervention. This kind of intervention is typically necessary only for hardware entities, so the *RestoreAction* class is subdivided into three subclasses, each concerning a *HWNetEntity* subclass: *RestoreNode_Actn*, *RestoreIFace_Actn*, *RestoreLink_Actn*. The *OnEntity* ObjectProperty relates the *RestoreAction* class to the *HWNetEntity* one and it is further specified by suitable restrictions within *RestoreAction* subclasses (see figure 5.16). As previously said, an abnormal state in *HWNetEntity* functioning is represented by an "ABN" value in its *HWStatus* property. So, the *OnEntity* property can take values only over those *HWNetEntity* instances whose *HWStatus* is actually "ABN". The *RestoreAction* can change this state into "OFF", meaning that entity can be controlled again and reactivated as necessary. Then, the representation of *RestoreActions* is necessary for the Reasoner to understand the mechanism through which external interventions recover abnormal states, and the Reasoner can update properly entity status.

5.4 Inferential processes: fault diagnosis and statistical analyses

The Reasoner ability to reason about network events, merging them with its global status representation and with its high level knowledge, is one of its features that cannot be represented through its actions. To represent explicitly this aspect of management, two properties have been introduced: *InfersAbnormality* and *ComputesMatrix*. The first one represents the fault diagnosis

activity, which is performed by both Reasoner agents, then it is attached to the *Reasoner* class. Instead, the *ComputesMatrix* property is present only in the *OffLR* class, because it refers to statistical complex computations performed only by this agent. As we have outlined in previously sections, the *InfersAbnormality* property is useful to represent active interpretation of events performed by the Reasoner, which effectively establishes the connection between event instances and abnormalities at their origin. The logical process which guide the Reasoner toward abnormalities diagnosis is supported by knowledge about network topology, network status at the considered time instants, general cause-effect relationships between events and abnormalities and also representation of the actions performed by PLAs executing it commands. Knowledge about the general cause-effect relationships between *Events* and *Abnormalities* (expressed by means of the *Causes* and *CausedBy* properties as seen above) is used by the Reasoner to know where and how to look for fault root causes; given some alarms related to possible abnormal events, it uses this account of knowledge to gather the right information, and to analyse the right network areas and the right network functioning aspects in order to infer the presence of a specific abnormality. In this mechanism, particular events (for example, packet losses and queue overflows) are "alarms" that initiate further reasoning and analysis in the Reasoner inferential process. It is worth to notice that the cause-effect relationships relating ontology *Event* and *Abnormality* classes are intended to be the high level knowledge guiding the information retrieval and the logical process toward fault detection; instead, the binding between a specific Event instance and its specific *Abnormality* cause represents the result of this inference process; only at the end of this process, events will be classified. Thanks to active interpretation, two instances of the same kind of *Event* can be considered both "normal" or "abnormal" according to the actual network status. The above consideration explains also other representation choices, such as the classification of events according to their nature and the explicit formalization of action effects on network functioning. During reactive management, knowledge about action effects is necessary to make the OnLR able to interpret incoming events in the right way. Looking at the classes already described, we can see as the representation of the class chain between *OnLR*, *CommandAction*, *BasicAction*, *Event* makes the OnLR able to recognize what events are caused by its commands. The ability to discern unforeseen events from those generated by control commands makes possible to interpret the same event in different ways. For example, when the OnLR orders to switch off a *HWNetEntity* (as with the *SwitchLink_Cmd* class), it knows that the consequent *StateChange_Evt* raised by sensors is caused by its command, then, it knows that it isn't abnormal, but a foreseen result of its actions. This way, the entity status in OnLR representation will not become "ABN", but will take the "OFF" value. It is worth to notice that sensor observations are not "intelligent": sensors can only detect the activity of an entity; so the raised *StateChange_Evts* can have only two values in their *ChangeType* property: "ON" and "OFF". On the contrary, entity status in Reasoner's representation can have three values: "ON", "OFF", and "ABN": between event notification and status updating there is Reasoner's interpretation ability. At the end of some inferential steps an event will be classified as "normal" or "abnormal". If it is part of normal network dynamics, its *CausedBy* property will not have any value; if it is caused by issued commands, its *CausedBy* property will have as value the *BasicAction* at its origin; if it is

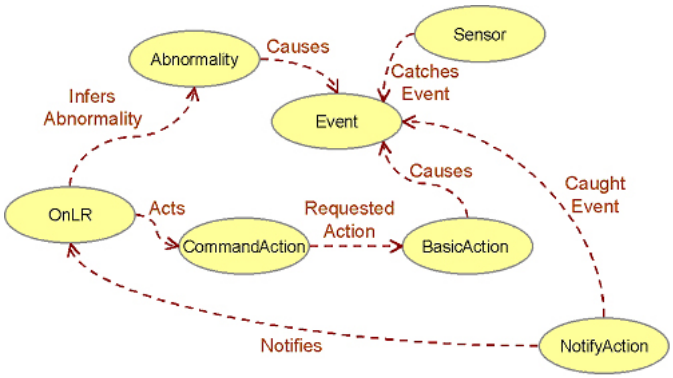


Figure 5.17: Set of properties describing how reactive management takes place

caused by a fault situation, its *CausedBy* property will be bound to the inferred *Abnormality*. Figure 5.17 shows the main properties which allow to understand how OnLR receives real-time monitoring information, how it infers the presence of some abnormalities and reacts to network state performing an active events interpretation.
Off-line reasoning, besides fault diagnosis purposes, aims at statistical evaluation of network performances. In order to represent this ability, the *ComputesMatrix* property relates the OffLR class to the *TrafficMatrix* class, descending from the *TrafficStatistics* class that represents statistical information about traffic load and traffic distribution. To represent the time interval they concern, the *StartTime* and *EndTime* datatype properties are attached to the class. The *TrafficStatistics* class is divided into two subclasses: *TrafficMatrix* and *TMEntry* (see figure 5.18).

As the DemandMatrix class, the TrafficMatrix class is represented has an aggregation of TMEntry instances, through the HasEntry property. The TMEntry class represents the traffic information (Value property) concerning only a pair of source and destination nodes, referenced by the SourceNode and DestNode

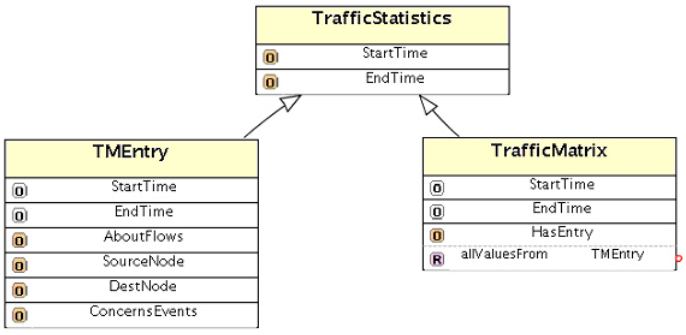


Figure 5.18: *TrafficStatistics* hierarchy

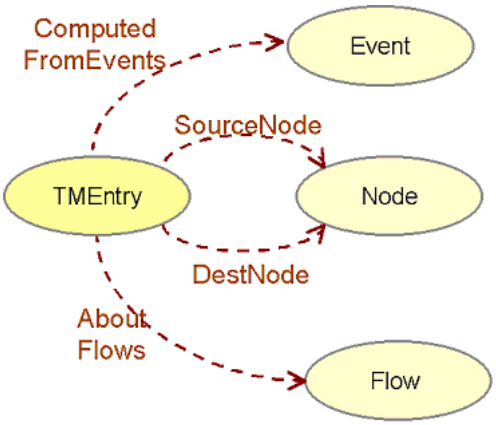


Figure 5.19: ObjectProperties relating each *TMEntry* instance to the events, flows and nodes it concerns hierarchy

properties (see figure 5.19). These two properties take values over the Node class and show the ontological relation between the statistical information and the nodes it refers to. Since statistical information is drawn out from a large amount of rough data about past network functioning, the ComputedFromEvents property makes clear the tie between original data and high-level statistical results. This property has values in the *LinkUsage_Evt* subclass of Event, because this kind of event represents the basic input for traffic statistics computation. The AboutFlows property expressly relates the statistical information in a TMEntry to the traffic which has flowed between the two source and destination nodes. This property makes clear that, even if statistical computation produces only a single result, it summarizes traffic information about (generally) different data flows. During the whole considered time interval, in fact, various independent flows could have taken place and, for this reason, the AboutFlows property has a multiple cardinality. The constraints between event occurrence time and the statistic time interval are implicit, as also are those relating the SourceNode and DestNode values in the TMEntry class to the same properties values in the referenced flows (AboutFlows property).
Statistical information represents information about network functioning at a higher level of abstraction; it is used by the OffLR to evaluate medium performances, to detect resource deficiency and to suggest suitable interventions. For example, statistical traffic information can be used to redistribute link costs, forcing routing to exploit available resources better. Figure 5.20 shows the main properties used to describe off-line information retrieval and analyses.

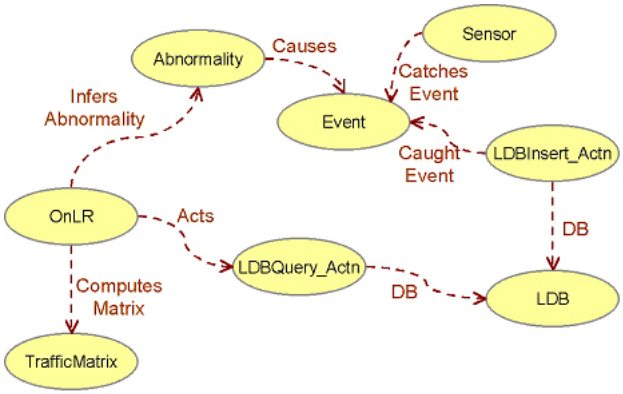


Figure 5.20: Set of properties describing how off-line analyses take are performed

Bibliography

[1] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199 – 220, 1993.

[2] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing," *International Journal of Human-Computer Studies*, vol. 43, pp. 907–928, 1995.

[3] G. Antoniou and F. van Harmelen, "Web Ontology Language: OWL," in *The Handbook on Ontologies in Information Systems*, S. Staab and R. Studer, Eds. 2003, Springer Verlag.

[4] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, "OWL Web Ontology Language guide," in *W3C Recommendation,*, Feb 2004, http://www.w3.org/TR/owl-guide/.

[5] V. Haarslev and R. Mø"ller, "Racer: An OWL Reasoning Agent for the Semantic Web," in *International Workshop on Applications, Products and Services of Web-based Support Systems,*, Al-Shaer and G. Pacifici, Eds., Canada, Oct 2003, pp. 91–95, in conjunction with the 2003 IEEE/WIC International Conference on Web Intelligence.

[6] J. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, and S. W. Tu, "The evolution of Protégé: An environment for knowledge-based systems development," *International Journal of Human Computer Studies*, vol. 58, no. 1, pp. 89–123, 2003.

[7] N. F. Noy, R. W. Fergerson, and M. A. Musen, "The knowledge model of Protégé-2000: Combining interoperability and flexibility," in *Proc. 2nd Int'l Conf. on Knowledge Engineering and Knowledge Management*, Juan-les-Pins, France, 2000.

[8] H. Knublauch, M. A. Musen, and A. L. Rector, "Editing description logic ontologies with the Protégé OWL plugin," in *International Workshop on Description Logics - DL2004*, i, Ed., Canada, 2004.