A Semantic-oriented Framework for System Diagnosis

Manuela Popescu
University of Besançon
France
manuela.popescu@univ-fcomte.fr

Pascal Lorenz
University of Haute Alsace
France
lorenz@ieee.org

Jean Marc Nicod
University of Besançon
France
jean-marc.nicod@lifc.univ-fcomte.fr

Abstract - **In the field of system and network diagnosis, there is a variety of modeling and inference methods reported in literature. However, very few are focusing on the validation and knowledge transfer in case of similar symptoms. Many researchers targeted the use of a generic diagnosis framework, semantic-oriented solutions, and temporal aspects related to diagnosis validation. Event correlation and action triggering are essential for an accurate diagnosis decision. However, no industry-wide solution was considered so far. In this article, we are proposing an adaptive framework for diagnosis validation and transfer of information from successful outcomes for future use and optimization of the diagnostic activity. It is shown that this mechanism allows a post-validation of successful diagnosis actions, optimizing the diagnosis process and increasing its accuracy. We are presenting a series of ontology-driven mechanisms for system diagnosis. Mainly, we introduce event ontology and concepts related to semantic tag clouds and show how to manage the activities to build an ontology-based diagnosis. Additionally, we consider temporal aspects related to diagnosis validation. Event correlation and action triggering are essential for an accurate diagnosis decision. There are several time-related challenges referring to event timestamps, timely event correlations, and timely corrective actions, in both absolute time (precise moment), or relative time (between events, actions, and events and actions). We consider a series of temporal operators defining the event relative temporal position that allows a more fine grain interpretation of the system behavior. A combination of proposed mechanisms is used to complete the main functions of a diagnosis engine.**

Keywords - system diagnosis, diagnosis validation, knowledge transfer; ontology-based diagnosis; semantic tag clouds; progressive ontology; temporal features.

I. INTRODUCTION

System and network diagnosis is vital if network infrastructures are to function efficiently and maintain reliable delivery of service to customers. There are various management and control mechanisms acting on a system to monitor security, performance, optimization and so on. In case of faulty or unexpected behavior, diagnosis is usually the main activity triggered by the symptoms of the system under supervision.

Figure 1 depicts a very high level view of the diagnosis process. Collectors have been developed to assemble systems health data parameters. These parameters can fall within normal value ranges (accepted, desired, expected, etc.), or they can reflect an unhealthy situation (value not recognized, out of range, denied by policy, inconsistent or out of context). A formalized set of this data constitutes the symptoms of the network under consideration. System problems can thus be identified and the most suitable solution for healing is computed and implemented. A validation step is then necessary to assess the success of the repair. Validation is also performed to prevent illness, following small parameters variation from accepted range, or after a calm period.

We can identify two loops of the diagnosis process: (a) one loop deals with measuring the system parameters (system state, events, i.e., pre-conditions) and taking the most suitable actions; this is referred to as the *diagnosis loop* and (b) a second loop deals with validating that the corrective actions were indeed successful; this is referred to as the *validation loop*. The validation loop has two main goals: (a) to establish the new state of the system, i.e., post-conditions and (b) to gather knowledge on how to solve future similar situations, in case the actions taken were considered successful. In general, there is little or no cross-interaction between these two loops.

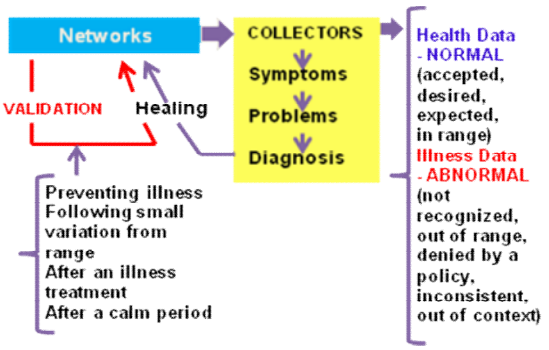

Figure 1. Diagnosis Theory [17, 25]

In addition, we introduce the concept of *Quality of Diagnosis (QoD)* into the validation loop to help make accurate decisions, based on past successful actions applied to similar situations.

However, there are too many actions in a system generating an unmanageably large number of events. It is estimated that a maximum of 8% of these events are considered by diagnosis systems; the rest of them are being discarded. Additionally, there is a diversity of technology domains driven by special diagnosis mechanisms. Mapping those mechanisms in a heterogeneous environment and correlating the diagnosis actions overwhelm the operators in NOCs (Network Operation Centers), leading to unsatisfactory solutions.

Towards an automated diagnosis, we introduce a progressive ontology (leading to progressive diagnosis) and therefore "progressive validation" of successful actions. This is intended to solve potential conflicts of the post-conditions of the actions already validated as "successful" and to evaluate the accuracy of the diagnosis actions (preciseness versus permanent damage). The basis is the definition of event ontology, followed by several concepts identifying the cause of a symptom and potential diagnosis actions.

The complexity of networks and distributed systems gives rise to management challenges when unexpected situations occur. There is an overwhelming number of feedback events coming from the system in the form of status reports towards the monitoring and management applications and human operators. Actually, very few of these events, less than 10%, can be considered for potential status understanding and remedy. Given the numbers, it is inevitable that many relevant events are dropped. The remedy actions can come too late (and sometimes be useless). There are numerous management applications in commercial use. However, the variety of the systems to be managed, their complexity, and the fact that most of the successful decisions are rarely recorded, rise serious challenges in the ability to accurately handle unexpected situations.

Some of the multiple causes leading to the current state are (i) lack of successful validation of corrective actions, (ii) heterogeneity of the events to be handled, and (iii) incomplete correlation and time synchronization between status reports, decision processing and corrective actions.

A step towards automated diagnosis was introduced in [23], where an event ontology and a progressive diagnosis ontology were proposed. Event dependencies captured by ontology and specific event relations were formalized. Probable cause and recommended actions were associated with events. Additionally, an augmented specification for actions was proposed to help the validation loop. Both proposals had as a target the reuse of knowledge for problem fixing, identification of recommended diagnosis actions, and validation of successful actions.

The third identified challenge is time-related; this refers to event timestamps, timely event correlations, and timely corrective actions, in both absolute time (precise moment), or relative time (between events, actions, and events and actions). This aspect is more difficult, as many events issued at different timestamps might be processed for event compression/aggregation. The correct adoption of temporal aspects can solve potential conflicts among the post-conditions of the actions already validated as "successful" and helps evaluate the accuracy of the diagnosis actions (preciseness versus permanent damage).

In this paper, we introduce a generic framework, propose an event ontology, highlight the relevance of temporal aspects, identify the challenging issues, and propose a new timestamp approach. We consider a series of temporal operators defining event relative temporal position that allows a more fine grain interpretation of the system behavior. A combination of proposed mechanisms is used to complete the main functions of a diagnosis engine.

II. RELATED WORKS

There is a variety of approaches dealing with the collection and actions-taking loop of the diagnosis process. However, very few solutions are focusing on the validation and knowledge transfer in case of similar symptoms.

2.1 Diagnosis Approaches

There are many modeling and inference methods of diagnosis, deriving from different areas of computer

science, including artificial intelligence, graph theory, neural networks, information theory and automata theory [1]. The most widely used diagnosis techniques are expert or knowledge-based systems [2] (rule-, model- and case-based systems, decision trees and neural networks). *Rule-based* techniques provide a powerful tool for eliminating the least likely hypotheses in small systems [3]. However, deep knowledge regarding the relationships among system entities was included [5, 6, 7] to address shortcomings related to the inability to learn from experience, inability to deal with new problems and difficulty in updating the system knowledge [4]. Model traversing techniques [8, 9] use formal representation of a system with clearly defined relationships among network entities. Model traversing techniques reported in literature use object-oriented representation of the system [10]. They are usually *event-driven* and naturally enable the design of distributed fault localization algorithms. Graph-theoretic techniques employ a Fault Propagation Model (FPM) [9], which is a graphical representation of all faults and symptoms occurring in the system and commonly take the form of causality or dependency graphs. Some graph-theoretic techniques include divide and conquer algorithm [11], context-free grammar [12], codebook technique [13], belief-network approach, and bipartite causality graphs [14].

Despite a vast research effort, there are open problems regarding diagnosis in complex systems, such as multi-layer fault diagnosis, distributed diagnosis, temporal correlation fault diagnosis in mobile ad hoc networks and root-cause analysis in a service-oriented environment.

2.2 Validation and Knowledge Transfer

Two challenging post diagnosis and repair actions are (1) validation and (2) knowledge acquisition and transfer.

For validation, an audit must be performed on the system status to assess the success of the repair actions. Knowledge acquisition and transfer regarding successful repair actions require a formal representation of the system, specification of the symptoms and the actions taken in particular cases, as well as a methodology to have this information available to facilitate later diagnosis of similar situations.

The main prior knowledge needed for diagnosis is the set of system failures and the relationship between the observed symptoms and the failures. Previous knowledge may be explicit, such as a table lookup, or may be deducted from some domain knowledge. This represents deep, model-based knowledge. Alternatively, or in addition to this knowledge, information may come from past experience with the system. This type of knowledge is known as shallow, compiled, evidential or process history-based knowledge [15].

To understand the system behavior expressed by issued events, we need to see how to instruct the diagnosis engine to trigger appropriate actions. A primary condition is to understand the information carried out by an event. Despite more than 3 decades of efforts, no common event syntax and semantic were achieved. Most of the existing solutions target a special area and it is rarely normalized; therefore, diagnosing heterogeneous systems is a challenge.

Log files represent the most used information source. Parsing the log files is the only way to extract the event information. Some log files have minimal event information, whereas others contain a variety of unstructured information. Since the amount of log file event is very large, the event processing should handle the file index of events already processed, especially whether the system is rebooted. Several log files may contain event information, e.g., Syslog log files, system console log files, application message files, etc. While log files are the most difficult source of events, Syslog files have a certain degree of structure. However, all log files require more normalization to help parsing and automate event processing.

Syslog messages (associated with Unix environments) are tailored as a free-form text message together with some defined fields, such as severity, facility, timestamp, etc. [17].

The structure of *SNMP* (Simple Network Management Protocols) *messages* is based on SNMP MIB (Management Information Base) Model, which is quite complex to be handled &&3]. However, this structure facilitates the normalization of these messages into a common format. An SNMP agent can asynchronously send SNMP messages to *a trap* receiver (SNMPv2). The mechanism called *informs* is provided by SNMPv2c that adds a reply message to a trap, as an acknowledgement. However, notification is not ensured to reach the destination. Since both SNMPv2c and SNMPv3 can deliver SNMP notifications via traps or informs method, the sending agent can select the mechanism to be used. This raises coordination challenge concerning message structure and information contained in it. A management process can poll with a given frequency various state parameters and built complex event structure based on the collected information. There is no validation of a successful action after a diagnosis action is performed.

The *RMON* (Remote Monitoring) mechanism defines a generalized threshold-based alarm, which generates in turn an RMON event that eventually causes a SNMP notification [20].

Other devices can use a Syslog to trap converter, as a unified mechanism to send traps or informs. However, most of details contained in the message text must be parsed for further processing, as for Syslog events.

Events can be generated at any level; the network management system itself can generate events using any of the mechanisms identified above.

Event formats were proposed for specialized domains, such as intrusion detection reports and vulnerability reports, but no event dependency or instruction on how to manage them were proposed. A proposal to direct event management was proposed in [25], where the event itself carries the processing instructions. The instructions were derived from the event source point of view, with no correlation with other events.

Most of the current attempts failed because of the complexity of target systems, leading to huge event models, practically describing unmanageable situations.

It appears that ontology per domain is more suitable; the initial condition is to have an event ontology. Definitively, inter-domains ontology matching is a challenge; however, it allows tackling the problem step-by-step.

2.3 Temporal Aspects

Temporal features are related to several generic aspects concerning (i) inaccurate (wrong, un-synchronized, or missing) clocks, (ii) loss of events, and (iii) hierarchical event processing at layers exposing different clocks. These are somehow related to event propagation skew but also to different syntactic and semantic implementation decisions of the timestamps (including time zones). One approach in dealing with real-time measurements of propagation skew uses a statistical evaluation to update the timer values [27].

Some diagnostic constraints might be temporal. In [23], temporal constraints are used for event tags to define the event ontology and to detect the relative temporal constraints. Walzer *et al.* [28] use specific operators for time-intervals with quantitative constraints in rule-based systems to trigger certain actions. In the following sections, we present the main approaches used to specify temporal aspects on events and actions.

2.3.1 Temporal aspects for events

Timestamps are usually carried by the events themselves; basic events possess special timestamp fields that are instantiated when an event instance occurs. Timestamps are storing time in the native format of the platform in which the event processing runs. There are two standard ways to represent the time: (i) using the universal time, or (ii) using time zones. Since one still needs to preserve the zone indication for a device for hourly performance reports, the representation in the universal time is only for the computational point of view. Another standard way to represent the time is the UNIX-format time as a four-byte integer that represents the seconds elapsed since January 1, 1970. For the same reasons, the time zone of the source device should be stored.

An event might have multiple timestamps; the source timestamp (not always present), the logging host timestamp, the console timestamp, and the processing timestamp. Temporal correlation and event aggregation should consider all these timestamps.

Event processing and correlation need a time-based logic to express the relative position of start / end /duration of the events [24]. While attempts were identified for classifying the relative position of the events, no particular commercial solutions are known where a full range of temporal situations is used.

2.3.2 Temporal Aspects for Actions

An enhanced action model was proposed in [23]. One temporal aspect is related to the triggering condition (guard). Others temporal aspects are related to the temporal dependencies between actions, *i.e.*, some action must start at a given period after one action was triggered or was deemed successfully finished.

A diagnosis-oriented augmented action definition was introduced in [23], as follows.

action::= <<guard><ID><post-conditions>
<mode><conflicting>,

where

ID::= READ | *WRITE* | *DELETE* | *CHANGE* | *, etc.*

mode ::= <potential | *recommended* | *successful <context>>,*

with
potential: any diagnosis action that is designated as being related to a potential domain

recommended: any potential action that is perceived as solving a given problem, eventually based on a diagnoses history
successful: when post-conditions were validated as true

context: <d:D, c:C>

d:D is d instance of Domain
c:C is c instance of Cloud

Also in [23], we associated the notion of "conflicting" with a given *action*, which designates the actions a potential action is in conflict with, in a given domain:

conflicting ::= <a_1, a_2,… a_k | a_i:A>

A <guard> is acting as pre-conditions and igniter (initial timestamp), and the <post-conditions> are expected to be true (after the action is considered successfully performed). In general, actions are applied following a simple rule:

IF <pre-conditions>

THEN <action> WITH <post-conditions>

Post-conditions are assumed to hold. A composition of actions, a plan, is a set of related actions and it is used to specify dependencies between actions. This is schematically represented in Figure 2. The model can be summarized as follows, where a plan is introduced as a temporal combination of atomic actions (see ID above) [29].

policy::= IF <pre-cond> THEN {<> 1<action> 1<plan>}

[ELSE {<> 1<action> 1<plan>} <action> 1<plan>}] <post-cond>]

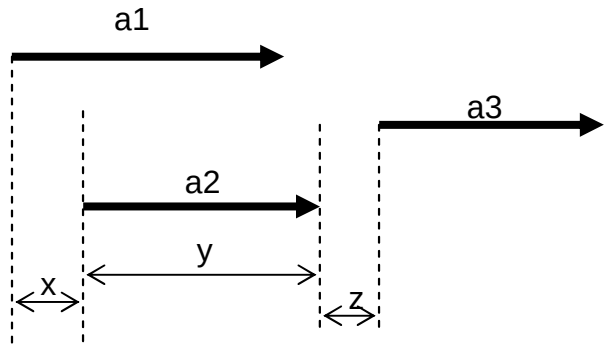

Figure 2. A plan ─ actions: a_1, a_2, a_3; time durations: x, y, z

Based on the analysis of the state of the art, we conclude that there is a need for a unified timestamps approach and a set of operators that must be used in synchronism to express the dependency between events, between actions, or between events and actions [25, 26].

In this article, we propose a representation of temporal features allowing various semantics used to correlate the events and the actions.

III. A SITUATION-BASED DIAGNOSIS SYSTEM

Let us first introduce the basic concepts used to formalize the system used here for validation, knowledge acquisition and transfer.

(1) *Symptoms* are external manifestations of failures. They can be observed as alarms, which alert of a potential failure. The alarms can originate from management agents via management protocol messages (SNMP traps, CMIP EVENT-REPORT etc.) from management systems monitoring the network status (*ping*) system log-files or character stream sent by external equipment [1].
(2) We introduce the concept of *situation* as representing the symptoms and the failure state of the system in consideration.
(3) A *problem* represents the failure state of the system and possible causes. This concept allows us to deal with events which might not be directly observable. Many types of events might not be directly observable due to (i) their intrinsically unobservable nature, (ii) local connective mechanisms built into a management system that destroy evidence of fault occurrence or (iii) lack of management functionality needed to provide evidence that a fault occurred. The state of the system implicitly represents these kind of un-observable events.
(4) A *context* represents a subset of states (including system topology, dependencies, configuration, etc.), services and their users, at a given time.
(5) We also introduce the use of *Quality of Diagnosis (QoD)* into the validation loop mentioned in Figure 1. This concept will drive more accurate decisions, based on past successful actions applied to similar situations.

We classify the states of a system in 3 types, associated with symptoms and probable causes, as shown in Figure 3.

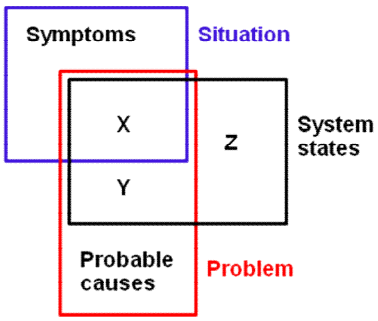

Figure 3. Basic concepts relationship

The system states of type X are states producing observable symptoms that indicate failures, the system states of type Y are states producing non-observable events of failure and the system states of type Z are behaviorally expected states that are not associated with failures.

3.1 Approach on Diagnosis Loop

As shown in on Figure 1, three related concepts are needed for a successful repair: (i) symptoms, as defined above, (ii) problems, as a set of potential causes based on the situations and (iii) diagnosis actions (most suitable) in a given situation. These concepts are applied in successive steps.

Let S, P, D, E, and A be the set of Symptoms, Problems, Diagnosis, Events, and Actions respectively. The diagnosis process can be summarized by (1).

$$(E)S \rightarrow P \rightarrow D(A) \quad (1)$$

Let $\mathbf{s_i}$, $\mathbf{p_i}$, $\mathbf{d_i}$, $\mathbf{e_i}$ and $\mathbf{a_i}$ represent a given instance of a symptom, problem, diagnosis, event and action respectively.

We introduce three types of symptoms, based on the completeness of the information coming from the system.

(1) *Reactive* - A set of events may reflect a set of problems that can be repaired by a set of diagnosis actions. In particular, the set of events may be received is a certain time window. In the case of reactive symptoms, most of the events occur spontaneously, i.e., SNMP traps, informs [16].

(i) Time-agnostic diagnosis:

$$\mathbf{[e_1, e_2, e_3 \ldots e_n] \rightarrow \{p_i\} \rightarrow \{d_i\}}$$

(ii) Time-oriented diagnosis (temporal context)

$$\mathbf{[e_1, e_2, e_3 \ldots e_n]t_1 \rightarrow \{p_i\}t_1 \rightarrow \{d_i\}t_1}$$

(2) *Proactive* - A set of events may be missing just one extra event before being able to infer a set of problems associated with the system. Depending on the nature of the event still to come, a different set of problems can be inferred. For proactive symptoms, new information can be solicited, for example, using polling mechanisms via specialized queries. In SNMP, GET state or the value of a parameter, is a solution. The decision of triggering such queries belongs to the diagnosis engine that might identify a potential symptom.

$$\mathbf{[e_1, e_2, e_3 \ldots e_{n-1}] + [e_n] \rightarrow \{p_i\}}$$
$$\mathbf{[e_1, e_2, e_3 \ldots e_{n-1}] + [e'_n] \rightarrow \{p'_i\}}$$

It is important to notice that the nature of the expected event might lead to different classes of problems.

(3) *Pre-emptive* - When a symptom is not complete, a threshold might be set on an expected set of events. This threshold depends on the type of events (Boolean, Integer, etc.). When the expected events are crossing the threshold (1) will take place. A simplified representation is shown below.

$$\mathbf{[e_1, e_2, e_3 \ldots e_{n-1}] + [threshold\ on\ \{e_i\}] \rightarrow \{p_i\},}$$

where "threshold" is used in a general sense, e.g., belonging to a class of events, occurring in a temporal vicinity (ε) of $\mathbf{e_{n-1}}$, or at least of delay of (δ) from $\mathbf{e_{n-1}}$.

All three types of symptoms described above are context-independent. When the context is taken into consideration, the relationship (1) becomes:

$$(E)S \rightarrow [P, C] \rightarrow D\mathbf{(A)} \quad (2)$$

where **C** is the set of possible contexts.

Most of the time, systems experience different problems in different contexts, leading to different diagnosis, i.e.,

$$\mathbf{[e_1, e_2, e_3 \ldots e_n]t_1 + [context_1] \rightarrow \{p_i\} \rightarrow \{d_i\}}$$
$$\mathbf{[e_1, e_2, e_3 \ldots e_n]t_2 + [context_2] \rightarrow \{p'_i\} \rightarrow \{d'_i\}}$$

The main manifestation of the diagnosis is the set of actions and their results (post-conditions of the respective actions). Generally, it is assumed that the post-conditions are true. However, potentially conflicting repair actions might leave the system in a questionable (or unknown) state, even when the post-conditions of each action hold.

In the next section, we propose and analyze a validation loop based on QoD.

3.2 Approach on Validation Loop

In general, diagnosis consists of a set of potential actions intended to fix a situation in a given context. Only some of them might be successful for a given problem in a given context.

The goal of the validation loop is to determine the successful diagnosis actions in the given situation, and to transfer this knowledge to the diagnosis engine for use in future similar situations.

We are introducing the QoD into the validation loop, as shown in Figure 4.

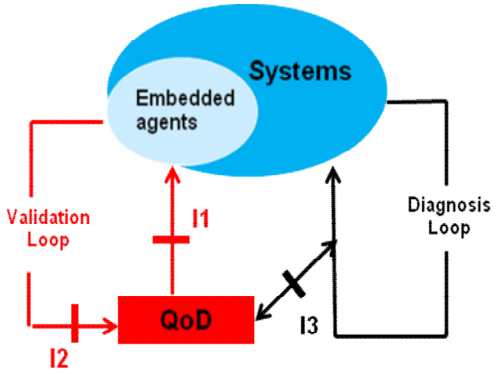

Figure 4. QoD

The QoD module is a dedicated validation engine, which interacts through three specialized interfaces with the system and the diagnosis loop. In particular, it receives diagnosis feedback from specialized system agents via the Interface I2 and asks for additional information (i.e., for audits) via the Interface I1. QoD communicates the diagnosis results to the Diagnosis Loop via the Interface I3. I3 can also be used by the diagnosis loop to ask the QoD for auditing information related to a given situation.

A diagnosis action has a set or pre-conditions and guarantees a set of post-conditions. QoD deals with validation (or evaluation) of post-conditions. Therefore, the QoD engine acts only during-diagnosis or post-diagnosis.

<pre-conditions>
<action-id> (3)
<post-conditions>

Based on the framework presented here, in the following section we propose the QoD mechanism.

3.3 QoD Mechanism

We identify two behavioral modes of the QoD engine: (i) listening mode and (ii) audit mode. In the listening mode, the QoD is waiting for input from the embedded agents or from the diagnosis loop. The audit mode can be triggered (i) by the diagnosis loop following the application of a set of repair actions to resolve a situation (ii) for a given time period or (iii) can be scheduled periodically/frequently.

At the end of the audit process, QoD returns to the diagnosis loop the subset of successful actions from all possible actions taken to repair a certain situation, in a certain context.

[S, C, {successful actions}] (4)
→ Diagnosis Loop

By request from the diagnosis loop, QoD can monitor: (a) a given action, (b) a set of given actions, or (c) all potential actions. The diagnosis loop provides the set of actions to be monitored, QoD being solely a validation engine.

The diagnosing actions are taken as part of the diagnosis loop. The successful actions, determined by the validation loop, will be the ones that make the system pass from a state of type X to a state of type Z.

Definition of a successful action

if
$\{state_i\} \rightarrow \{action_i\} \rightarrow \{state_j\}$
where
$\{state_i\} \in X \wedge$
$\{statej\} \in Z$
then
$\{action_i\}$ is successful

Note: An action can be successful in one context and failure in another context.

The validation loop will transfer the information regarding successful actions to the diagnosis loop for optimizing the diagnosis loop activity. The format of the information returned to the diagnosis loop can have 2 forms:

(1) The successful action, composed of pre-conditions, id and post-conditions as well as the symptom, problem and context.

[<pre-cond><id><post-cond> | <S,P,C>]

(2) The successful action, composed of pre-conditions, id and post-conditions as well as

a pointer. The pointer indicates the list of <S,P,C> in which the action in consideration was successful. The validation engine is responsible for keeping and updating this list.

[<pre-cond><id><post-cond> | <pointer>]
where
<pointer> = {…<S,P,C>$_i$, <S,P,C>$_j$…}

We present below an algorithm summarizing the QoD mechanism.

Algorithm

input:
System states, X, Y, Z
S, P, C
a_i
pre-condition a_i = state$_i$
post-condition a_i = state$_j$
update type [one action, pointer]
if state$_i \in X \ \wedge \ state_j \in Z$
then
forward <a_i> <S,P,C>
or
update <a_i><pointer>
where
<pointer> =
{…<S,P,C>$_i$, <S,P,C>$_j$…}

In summary, the QoD engines takes as input the possible system states and their type (X , Y or Z), the symptom, problem, context, the action to validate, the initial system state (pre-condition), the final system state (post-condition) and the update type. If the initial system state is of type X and the final system state is of type Z, then (1) forward to the diagnosis loop the action and the set <S,P,C> or (2) update the pointer, where the pointer represents the list of all possible <S,P,C> combinations known so far in which the action under consideration is successful.

To tackle the problem, we propose an ontology for diagnosis composed of (i) an event ontology, (ii) tag and semantic diagnosis tag clouds, and diagnosis clouds matching.

3.4 Event Ontology

To address the issues described in the previous sections, we propose here an event ontology.

The three main concepts of the ontology are: (1) *Event*, (2) *Domain* and (3) *Event Manager*.

An *Event* is a software message that indicates something of importance has happened. A *Domain* is defined as a technological area, such as VPN or VoIP, a sub-network or specific management area, such as fault, security. An *Event Manager* provides real-time information for immediate use and logs events for summary reporting used to analyze network performance.

The main relations between these concepts are depicted in Figure 5. The Events are owned by the Event Manager. Also, an Event refers to a Domain.

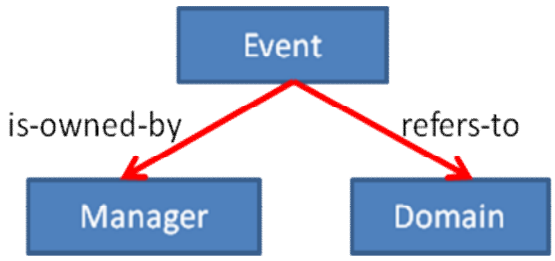

Figure 5. Event Ontology Main Concepts

In addition to the main relationships between an Event, Manager and Domain, an Event is defined by explicitly-declared event ID, source, version, timestamp, priority, and free-text (see Figure 2). An event can be stored at an URL, Library-ID, Repository or Log Server. These locations can be explicitly-declared or found as a result of a search. An Event can be referred to by another Event or can refer to another Event. Also, an Event can be similar to another Event. The ontology identifies the relationship between events (example “is-similar-to<event>”,”refers-to <domain>” etc.) and the management properties of a given event (e.g., “can-be-accessed-by <application>”, “can-be-modified-by <application/human>” etc.).

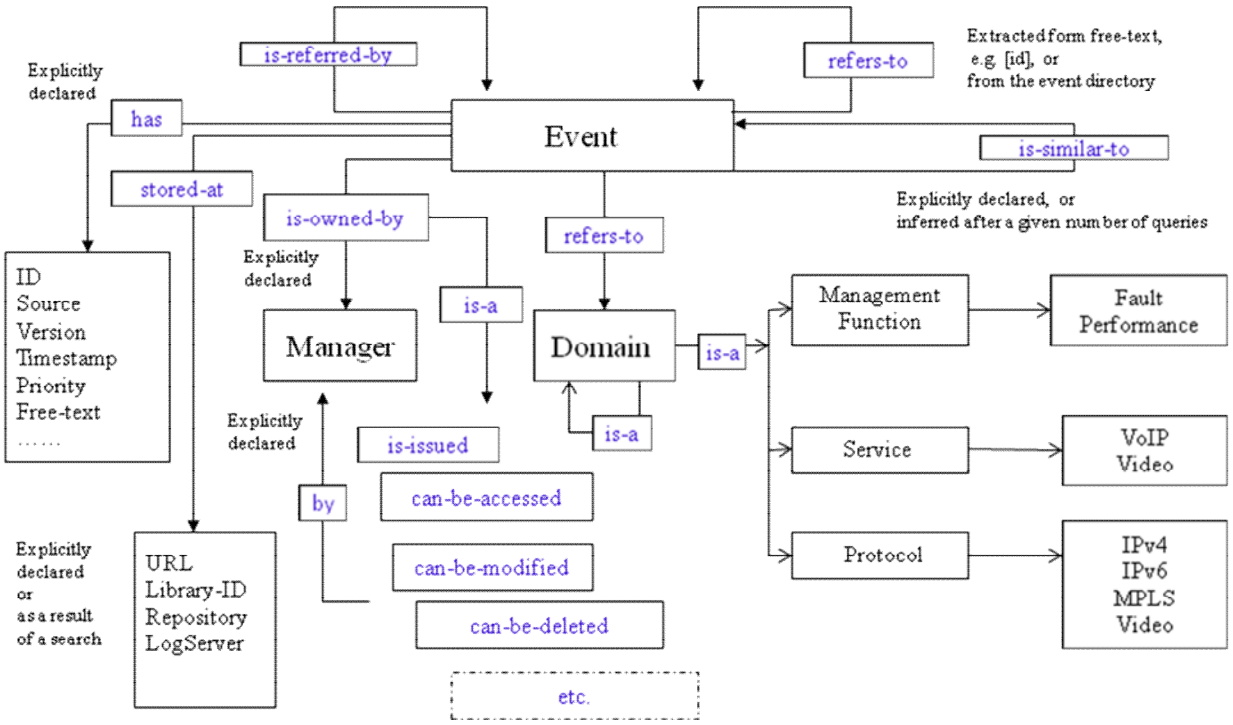

Figure 6. Details of the Event Ontology

In Figure 6, an Event depends on a Manager by <is-owned-by> relation that can be refined in a few inheriting relations such as <can-be-accessed-by>, <can-be-modified-by>, or <can-be-deleted-by>. As an example, the last relation defines what managers have rights to delete a particular event. If we combine with the proposal from [25], the manager is also instructed how to proceed.

Expanding the ontology from Figure 5, we illustrate in Figure 7 the main concepts and the core relations. Apart the dependency relations, we also consider the aggregation relation, where events are aggregated during the processing. The main relation <is-owned-by> is specialized in a few relations, as managers can be specialized too.

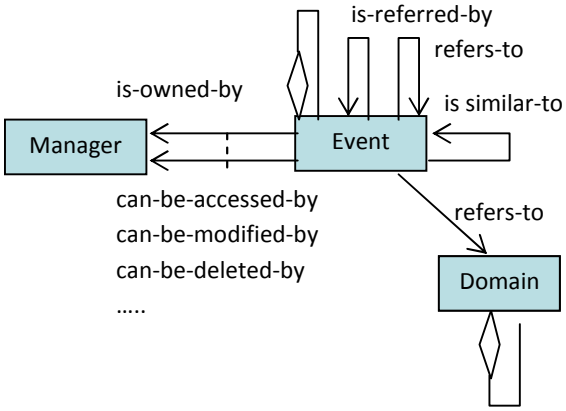

Figure 7. Adding relations to the main concepts (UML notation)

A domain instance can be one of the specialties that have been illustrated in Figure 6.

3.5 Semantic Tag Clouds

The first challenge in deriving semantic clouds is the complexity, when considering all the expansions of the diagnosis domains. As an example, let us consider the following tag subdivisions targeting fault diagnosis (List 1). We have a coarse grain tag embedded hierarchy, while a fine grain tag list can be progressively developed. As a note, each time a tag expansion is made, a validation is required.

- TAGS
  - IPv6, IPTV
    - fault in IPv6
      - fault in IPTV
        - security IPv6
        - security sensors
        - IPTV with IPv6
        - IPTV management
- Refined /TAG list/
  - latent fault, authentication, M5, IPSec, link-Syslog, CLI-Z

List 1. Example of Tag List

Two concepts are considered in defining (identifying) tags, as expressed in Figure 8.

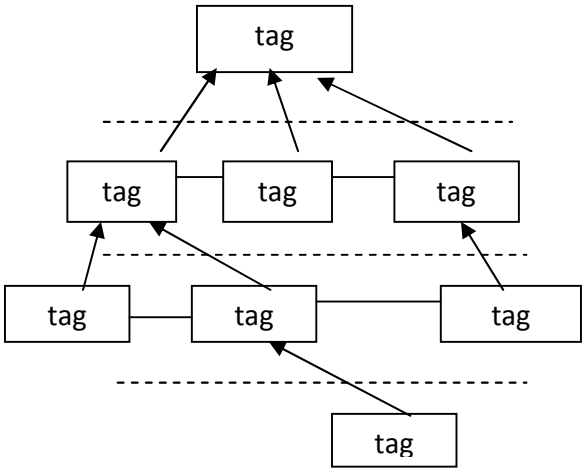

Figure 8. Hierarchical tags and their relations

As suggested by List 1, tags have a *hierarchy* within each domain; this defines a special dependency relation between tags. Also, at each layer, tags may embed different associations (e.g., *port-85* and *SNMP traps*).

During the diagnosis activities, tags are relevant, but without the tag relationships, no much progress can be achieved. One step ahead is to form tag clouds; a tag cloud refers to a tag list focusing on a particular domain (see Figure 9).

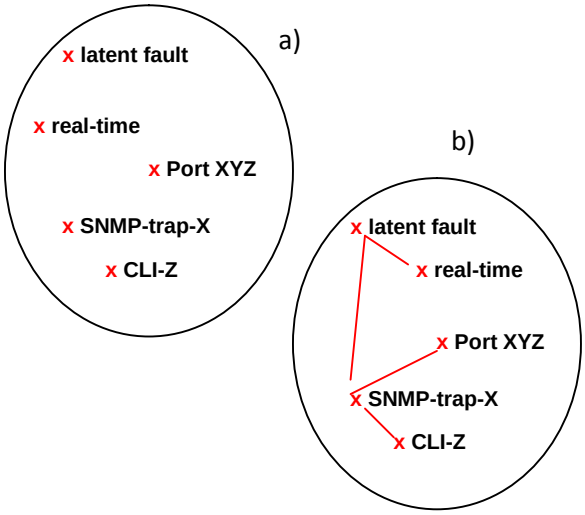

Figure 9. Tag Clouds and Semantic Tag Clouds

As an example, we localize in Figure 9 (a) a tag cloud concerning real-time fault diagnosis. We identify a series of relationships between a given CLI (Command Line Interface), a SNMP-trap that refers to a given port, on one side, and between latent fault and real-time tags, on the other side (Figure 9 (b)).

3.6 Mapping Semantic Clouds in a Context

Let us assume that there are two semantic clouds, as shown in Figure 10. Semantic cloud#1 defines the tags and their relationships for a fault related to a power supply issue, while Semantic cloud #2 relates to a potentially real-time and latent fault.

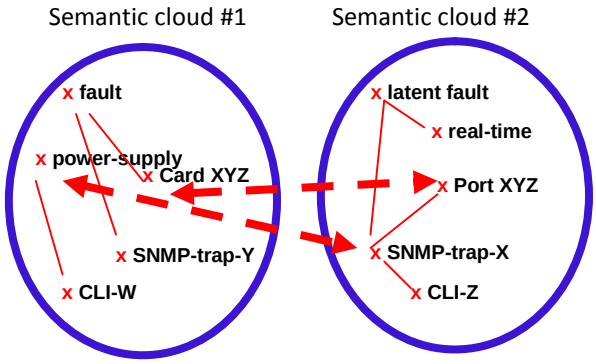

Figure 10. Cross-semantics of Semantic Tag Clouds

An appropriate diagnosis is triggered by linking the port ID and the card ID, then identifying the SNMP trap defined for power-supply behaviors. With these cross-semantic connections, one can identify the potential CLI leading to the situation, or a particular fault on power supply.

As mentioned before, the number of events produced by a system exceeds by far the capacity of processing them. An accurate selection of those events 'of interest' reduces the burden of monitoring and diagnosing and leads to a more adapted solution.

On the other side, the event mediation is less process consuming as both the diagnosis engine and the validation engine [17] refer to the same event ontology.

3.7 Progressive Ontology

A diagnosis engine is validating successful diagnosis actions based on a given set of semantic tag clouds. As expected, since the technology evolves, new devices and applications are developed. We adopt a series of heuristics on validating new semantic tags and new semantic tag clouds.

3.7.1 New Semantic Tags

For the purpose of ontology control, the following heuristic was experimented:

START
New tag TAG
1. Temporary accept a 'tag'
2. Set a threshold of its use for a time window
3. If the diagnosis engine uses it in that time window, then insert it as *permanent* in the 'tag cloud'
4. Then consider tag relations based on the diagnosis-context (situation)
5. If the diagnosis engine do not use the new tag in the given time window, add it to the *list* of *potential tags* to be considered in the future, but avoid definitive insertion
END

Heuristic #1. Inserting a New Semantic Tag

The tag inventory process keeps a few records concerning a specific tag, e,g., the cloud it belongs to, if it is on a potential tag list, or whether it was declined for a list of clouds.

3.7.2 Semantic Tag Cloud State

As the technology evolves, building new tag clouds and maintaining the created and validated ones are current cloud management activities. To accurately communicate to a diagnosis engine what clouds can be used, we introduced the cloud state. Cloud state belongs to *{progressive, in_test, validated, obsolete}*.

The state *progressive* denotes a cloud that it is in a building phase; it can be updated, but not used in a diagnosis process. *In_test* represents a cloud that achieved a certain degree of completeness; in this state, different diagnosis are tested and mapped against the cloud semantic relations to validate them. The state *validated* is declared by the diagnosis engines; it allows a semantic tag cloud to be used in the diagnosis decisions. *Obsolete* is declared when none of the components of a cloud is in use any more.

3.7.3 Inter-clouds Relations

The main achievements with ontology are building small models and link them via relations. Therefore, building inter-clouds relations are crucial for diagnosis. The procedure is captured by the following heuristic:

START
1. Identify the clouds potentially related
2. Identify the intra-cloud relationships
3. Validate the cloud status
4. Build inter-clouds relationships and validate them
5. Identify for each semantic tag cloud:
   a. "one-hop" related clouds
   b. type of relationship
      /start/cloud1<->end/cloud2/
END

Heuristic #2. Steps for building inter-cloud relations

3.7.4 Specific Aspects in Defining Tags

Tags are context-driven; they are defined without a global view, by different teams. Building inter- and intra-cloud relations is conditioned by a full understanding of the semantic of a tag and its relations with other tags. The following aspects are considered: (i) *temporary tags*, (ii) *synonymous tags*, and (iii) *ambiguous tags*. Some of the tags are originally embedded into a given cloud; until at least one relation with the existing tags is identified, it is labeled as 'temporary'. Synonymous tags are more difficult to be identified; tag definition is usually human-driven, if there is no formal semantic behind tag definitions. Ambiguous tags can be identified inter- and intra-clouds. Disambiguation is achieved by either duplicating a given tag in two tags with dissimilar identifications, or by reducing the semantic of a considered tag.

3.7.5 Diagnosis Actions Associated with Tag Clouds

In the proposed progressive ontology, each semantic tag cloud has attached a list of suggested potentially successful diagnosis actions.

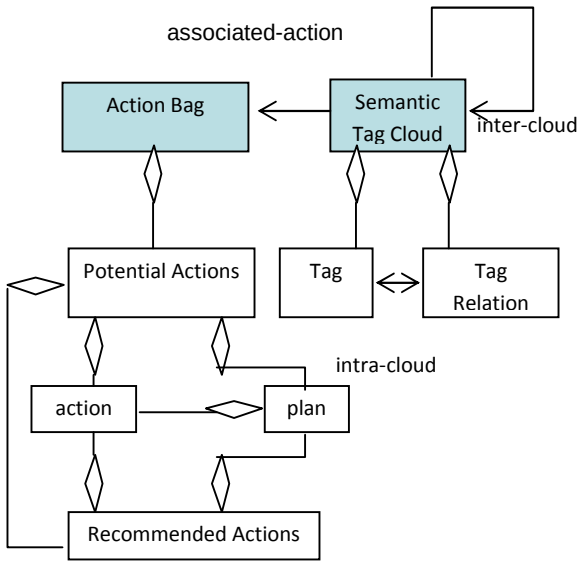

Figure 11. Semantic tag clouds and associate actions (UML notation)

We depict in Figure 11 the diagnosis counter-part of event ontology modeling, i.e., the required actions. There are a number of diagnosis *potential* actions associated with a

semantic cloud. A subset of them is *recommended*, based on the successful diagnosis history. In triggering a diagnosis decision, we consider two potential solutions: an *action*, or a *plan*. A plan is a predefined combination of some actions, implying parallelism, serialization, and temporal relations between actions.

There are some aspects when associating many tag semantic clouds and the actions belonging to them. Within a semantic tag cloud, some actions are defined with special *guards* (for validating the triggering), or explicitly specify *conflicting* actions (in given conditions). When two or more semantic tag clouds are mapped some conflicts might disappear, while *emerging* conflicts may occur, too. To deal with all these aspect a better formalism is needed to express structures, relationships, actions, and constraints during the diagnosis activity (and its validation). The following section deals with some of these aspects.

3.8 Timestamps

This section describes aspects related to timestamps, event correlation with temporal operators and gives an example of use of temporal operators.

In a hierarchical model, an event model should allow multiple timestamps, depending on the event hosting and processing. In an XML-like specification, we introduce for the device (source), host (server), and processing application (management application or console), the timestamp and the time zone a source, host or processing application belongs to.

TABLE I: Timestamp specification

<time>
<device_time> device_time></device_time>
<device_zone> device_time_zone</device_zone>
<server_time> server_time</server_time>
<server_zone> server_time_zone</server_zone>
< processor_time> event_processor_time</ processor_time>
<processor_zone> processor_time_zone</processor_zone>
</time>

The timestamp of the event is best set by the event producer (*device_time*). The timestamp representing the moment of event registration on the server, *server_time* is of relevance for correlation. Finally, the timestamp of the entity performing correlation or event processing is relevant for synchronization among multiple such event processing systems.

Any of these three entities can belong to different time zones that should be considered when temporal priorities count.

The values of these parameters are set by various entities. Some protocols provide the capability to supply the time in the occurred event, or the time when the event producer sent the event. With the Network Time Protocol (NTP) the time from event producers will be the most accurate. Alternatively, the time registered by the event processing system might be considered.

We advocate the following representation, similar to Syslog protocol, *e.g., device_time: Jan 1 14:22:45* represents the local time on the device at the time the message is signed. For devices with no clocks, *device_time: Jan 1 00:00:00* should be the representation.

3.8.1 Event Correlation with Temporal Operators

Temporal relations are used to build time-dependent event correlations between events. For instance, we may correlate the alarms that happened within the same 10-minutes period, which means the correlation window is 10 minutes. We abstract an event and consider only the temporal aspects.

Let ***e1*** and ***e2*** be two events defined on a time interval:

$T_1 = [t_1, t_1']$
$T_2 = [t_2, t_2']$
and e_1 within T_1
e_2 within T_2

two events occurring within the time intervals T_1 and T_2, respectively.

The following temporal relations R(t) or R are identified:

R(t):: = {**after**(t), **follows**(t), **before**(t), **precedes(**t)}

R ::= {**during**, **starts**, **finishes**, **coincides**, **overlaps**}

The following deductions hold:

after: e_2 ***after(t)*** $e_1 \Leftrightarrow t_2 > t_1 + t$

follows: e_2 ***follows(t)*** $e_1 \Leftrightarrow t_2 \geq t_1' + t$

before: e_2 ***before(t)*** $e_1 \Leftrightarrow t_1' \geq t_2' + t$

precedes: e_2 ***precedes(t)*** $e_1 \Leftrightarrow t_1 \geq t_2' + t$

during: e_2 ***during*** $e_1 \Leftrightarrow t_2 \geq t_1$ *and* $t_1' \geq t_2''$

starts: e_1 ***starts*** $e_2 \Leftrightarrow t_1 = t_2$

finishes: e_1 ***finishes*** $e_2 \Leftrightarrow t_1' = t_2'$

coincides : e_2 ***coincides*** *with* $e_1 \Leftrightarrow t_2 = t_1$ *and* $t_1' = t_2'$

overlaps: e_1 ***overlaps(ε)*** $e_2 \Leftrightarrow t_2' \geq t_1' \pm \varepsilon > t_2 \geq t_1 \pm \varepsilon$
where ε is the accepted threshold for measurement variation.

With respect to the algebraic properties of the temporal relations,

- all are transitive, except **overlaps**,
- **starts**, **finishes**, **conincides** are also symmetric relations.

3.8.2 Example of Using Temporal Operators

In [22], time-oriented diagnosis was defined as

$$[e_1, e_2, e_3 \ldots e_n]t_1 \rightarrow \{p_i\}t_1 \rightarrow \{d_i\}t_1,$$

where

p_i, d_i, and e_i represent a given instance of a problem, diagnosis, and event, respectively.

As an example, let us consider the instantiation:

$$[e_1, e_2, e_3] \mid e_2 \textbf{\textit{follows}}(x)\ e_1\ \&\ e_2 \textbf{\textit{overlaps}}(\varepsilon)\ e_3\} \rightarrow p_{123} \rightarrow d_{123}$$

where x is the time duration between e_1 and e_2.

As a note,

$$[e_1, e_2, e_3] \mid e_2 \textbf{\textit{precedes}}(x)\ e_1\ \&\ e_2 \textbf{\textit{overlaps}}(\varepsilon)\ e_3\} \rightarrow p'_{123} \rightarrow d'_{123}$$

represents a different problem and therefore, a different diagnosis.

In the case that the above specification designates a given diagnosis and it is determined that e_1 did not follow e_2 after time x, a diagnosis engine issues an anomaly (no concrete diagnosis is derived).

An event has a series of event attributes, which we represent as:

$e = (f_1, f_2, f_3 \ldots, f_n)$
where f: $(value:V)$,
where V is the type of the attribute

Examples of event attributes that we consider are:

f_1: *ID*
f_2: *source*
f_3: *timestamp*
f_4: *timezone*
f_5: *English text defining the potential cause*
etc.

$e.f_3$ represents the value of attribute f_3 in event e.

The operators on relative event position (**follows**, **overlaps**, etc.) are related to the attributes f_3 and f_4.

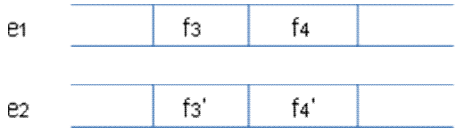

Figure 12.Timestamp and timezone event fields

In this example, $e_1.f_4$ and $e_2.f_4$' are known, since they represent the timezones of the sources of the two events. Only $e_1.f_3$ and $e_2.f_3$' need to be set by the local clocks. Let us assume that:

clk_1 sets $e_1.f_3$ and clk_2 sets $e_2.f_3$',
where *clk* is the local clock of the event source.

$|clk_1 - clk_2| \leq \varepsilon_{12}$,
where ε_{12} is the clock skew between the two local clocks for two domains represented by two semantic clouds [23].

e_2 ***follows***(x) e_1 is computed as follows:
$(e_1.f_3 + \varepsilon_{12}) + x < e_2.f_3$ (for the same time zone)

For different time zones, this becomes:
$[(e_1.f_3 + \varepsilon_{12}) \blacksquare Abs(e_1.f_4)] + x < (e_2.f_3) \blacksquare Abs(e_2.f_4)$,
where $\blacksquare$ $Abs(e.f_4)$ represents the operator for normalizing the time between timezones.

Following the same logic, e_2 ***overlaps***(ε) e_3 for different time zones is computed as follows:
$|(e_2.f_3) \blacksquare Abs(e_2.f_4) - (e_3.f_3) \blacksquare Abs(e_3.f_4)| < \varepsilon_{23}$
where
|x| is the absolute value of x
and
ε_{23} represents an acceptable error.

These event-based computations are performed each time a diagnosis is triggered and validated.

In the next section we will use this example in the diagnosis scenario.

3.9 Using Temporal Features for Diagnosis

This section presents a formal specification of the ontology-based diagnosis, considering temporal relations. Let us assume that the diagnosis engine and the Quality of Diagnosis (QoD) engine introduced in [22] have to trigger the following operations: INTERPRET, APPLY, VALIDATE and MARK.

- Diagnosis engine: INTERPRET events from the system.
- Diagnosis engine: APPLY the diagnosis actions.
- Quality of Diagnosis engine:
  VALIDATE the diagnosis actions.
  and
  MARK successful actions.

The APPLY, VALIDATE and MARK functions were shown in [23]. We reconsider the example with INTERPRET functionality as well.

As discussed in [23], there is a semantic tag hierarchy within each domain, with special dependency relations between semantic tags. Within a domain, semantic tags and their relations form a semantic tag cloud; a domain might have multiple semantic tag clouds associated with it. Let us assume that a system is represented by two semantic tag clouds (Figure 13). Semantic cloud #1 defines the tags and their relationships for a fault related to a power supply while Semantic cloud #2 relates to a potentially real-time and latent fault.

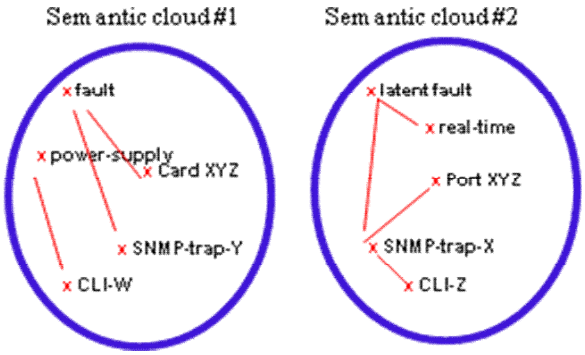

Figure 13. Two Semantic Tag Clouds [23]

When some event patterns occur and diagnosis actions must be triggered (and validated), the Diagnosis Engine interprets the events from the system and applies the diagnosis actions. Next, the Quality of Diagnosis engine validates the actions and marks the successful actions.

The following algorithm is used by the engines to perform the required actions for a given occurrence of combinations of events. A particular series of events occurs as shown in the INTERPRET part of the following algorithm (we use the '.' Notation, i.e., *a.b* means the property '*b*' of the instance '*a*'). When the conditions (2) and (3) explained in Section III hold, the necessary condition to enter the rest of the algorithm is met.

START

INTERPRET
IF {[e_1, e_2, e_3] | e_2 precedes(x) e_1 & e_2 overlaps e_3}
CLOCK = t_0

Legend (for details, see [18]):
r_C: R_C | r_C ::= <c_1: C, c_2: C>, cloud to cloud relation
r_T: R_T | r_t ::= <t_1: T, t_2: T>, tag to tag relation
r_{CA}: R_{CA} | r_{CA} ::= <c :C, {a_i : A | p_i: P}>, cloud to action relation
r_{dto}: R_{Dto} | r_{dto} ::= <e:E, d:D>, event to domain relation.

AND e_1 belongs to $cloud_1$
AND e_2 belongs to $cloud_2$
AND e_3 belongs to $cloud_2$
AND $x < t_0$
THEN
ERROR
ELSE
ASSUME
e_2 precedes(x) e_1 & e_2 overlaps e_3 = = TRUE
AND
IF there is exist r_c < $cloud_1$, $cloud_2$>
AND $cloud_1$.state = active
AND $cloud_2$.state = active
AND
IF there is r_{dto} <e_1, $domain_1$>
AND tag_1 belongs to $domain_1$
AND tag_1 belongs to $cloud_1$
AND tag_2 belongs to $domain_2$
AND there is r_T <tag_1, tag_2>
AND there is r_{CA1} <$cloud_1$, {$action_1$}>
AND there is r_{CA2} <$cloud_2$, {$action_1$}>
WITH
$action_1$ = {a_1, a_3, a_6}
AND
$action_2$ = {a_1, a_5, a_7}
THEN

APPLY {{a_1, a_3, a_5, a_6, a_7} – {
a_1.conflicting ∪
a_3.conflicting ∪
a_5.conflicting ∪
a_6.conflicting ∪
a_7.conflicting}
VALIDATE
a_1.post-conditions = TRUE
a_3.post-conditions = TRUE
a_5.post-conditions = TRUE
a_6.post-conditions = TRUE
a_7.post-conditions = TRUE
MARK
a_1.mode = successful
a_3.mode = successful
a_5.mode = successful
a_6.mode = successful
a_7.mode = successful
END

IV. CASE STUDY

The concepts proposed in this paper are implementable, at-large, but considerations must be taken when designing new systems to follow the formal approach. In this section, we highlight the main challenges of a full implementation of the proposed diagnosis solutions. While fixing legacy systems

remains a complex problem, avoiding the same errors for newly designed systems seems be to the most appropriate approach. We illustrate partial solutions for different concepts introduced via a case study simulation. The case study is conducted in connection with Cisco DFM (Device Fault Manager).

4.1 Challenges in Implementing the New Approach

Two main complementary contributions were proposed in this paper, namely: (i) a new mechanism for validating diagnosing actions, while promoting the reuse of diagnosis actions in similar situations, and (ii) a new approach on events and actions. While the usefullenss of each new concept and mechanism was highlighted when they were introduced, developping a large scale system using these concepts requires substantial effort and time.

There are several "de facto" constraints in developping a totally new diagnosis approach. In the following sections, we consider the most important challenges, based on our experience in validating our approach.

4.1.1 Customized Event Model used by Different Systems

One difficulty in validating our approach is related to the customized model used by different system/network devices/software to construct and send an event. To complicate the task further, even subsequent releases of the same entity might expose different event models. For this reason, any previous diagnosis algorithms and diagnostic systems become obsolete. Since designing and implementing a new event is relatively easy, there are thounsands of useless events (not easily captured by any diagnosis algorithm or system) and, consequenlty, thousands of "exception handling" cases; the latter are usually sent to a human operator in NOCs (Network Operating Centers).

Even in cases when a standard is available, the recommendations are too generic . For example, in ITU recommendation X.745 [81], the parameter status can be "mandatory", "optional", "conditional" or "not applicable". Aditionally, an 'implementstion status" ("implemented" or "not-implemented") needs to be considered (Figure 14).

<parameter>
<parameter status>:: = *mandatory* | *optional* | *conditional* | *not applicable or out of scope*
<implementation status> ::= *implemented* | *not implemented*
<parameter/>

Figure 14. Recommendations for Event Parameters

In this case, even if an event model is enforced, the value of *parameter status* and *implementation status* are not helpful, since, for the same event definition, some features might not be mandatory or implemented.

In other cases, the implementaiton of the events is very customized and leaves little opportunity for syntax harmonization, yet alone consideration for the semantic aspects. For example, Figure 15 presents a Syslog event issued by Cisco's GSR (Gigabit Switch Router) [30].

%EE48-3-ALPHAERR: [chars] error: cpu int [hex] mask [hex]\n addr [hex] data hi [hex] data lo [hex] parity [hex]\n fs [hex],pf [hex], prep [hex] (pc [hex]), pc1 [hex]\n plu int [hex] int en [hex] err en [hex] err [hex] ecc [hex]\n tlu int [hex] err [hex] ecc [hex], mip [hex] (pc [hex]), pc2 [hex]\n pst [hex], cam [hex], nf [hex], pop [hex] (pc [hex]), ssram adr [hex] err [hex]\n pipe ctl [hex] avl [hex] end [hex] fatal [hex], gather [hex]\n xmb read [hex] rclw1 [hex] rclw2 [hex] rclr [hex]\n tailw1 [hex] tailw2 [hex] tail [hex]\n sts1 [hex] sts2 [hex]\n pcr regs: fs [hex] prep [hex] plu [hex] pc1 [hex] tlu [hex] dummy [hex] mip [hex] pc2 [hex] mtch [hex] post [hex] pop [hex] gthr [hex]

Figure 15. Example of a Syslog Event issued by GSR (Cisco)

We see that there are many different types of formats for the events. Automating extraction of useful information from event text message is close to impossible. To extract a useful piece of information, or *token*, a diagnosis tool has to know the format apriori and write "*rules*" to parse messages into tokens, when the messages are generated from various sources. Currently, Syslog messages are generated by numerous sources: various IOS protocol code modules, device driver code module, etc. Updating all these sources implies changing tens of millions of lines of code, a task close to impossible.

We note here that SNMP traps have a standard format, using ASN1 (Abstract Syntax Notation One) [30]. An example of an SNMP trap notification is shown in Figure 16.

4.1.2 Events are Issued in Isolation

The second challenging aspect consists in the fact that change reports (events) are issued in isolation, with an implicit semantic on their potentially related events. This is due to the fact that the events are designed by the entity designers; they do not have a full picture of where the entity will be used, its interactions, etc.

Most of the diagnosis decisions are human-held heuristics. In many cases, intuition plays a major role. Some diagnosis decisions are experience-based, and

very rarely based on rigourous knowledge processing. As a drawback, accuracy and correctness migth lead to a late diagnosis.

To illustate the difficulty in dealing with legacy systems for a correct diagnosis, we consider two examples.

First, let us consider the example of a SNMP trap notification as shown in Figure 16. SNMP MIBs define these kind of messages in a standard way by NOTIFICATION-TYPE. The varbind will indicate neighbor = xxx.yy.zzz.mm; status = Down. Despite the more formalized strauctures of SNMP traps, a tool cannot syntactically "tokenize" the message following the same approach, unless the message is somehow recognized. This means that a tool user or developer has to know the message format, write a "rule" to recognize the format, then, when the message arrives with the prefix "%BGP-5-ADJCHANGE", that relevant "rule" can be triggered to tokenize the text content.

```
Nov 11 11:52:40.735: %BGP-5-ADJCHANGE: neighbor xxx.yy.zzz.mm Down - Peer closed the session
```

Figure 16. Example of a SNMP Trap Notification

Writing such rules is a cumbersome activity, especially since a rule needs to be written for each event. Additioanlly, having millions of rules is not scalable, raises conflicts, and diagnosis decisions come too late.

```
VSEC-6-VLANACCESSLOGNP: vlan [dec] (port [dec]/[dec]) denied ip protocol = [dec] [int] -> [int], [dec] packet(s)
```

Figure 17. A Report Syslog Event

Next, let us consider the example presented in Figure 17. To understand the explanation of the Syslog message shown here, a special rule must be written. The rule should capture that this message indicates that an IP packet from the identified VLAN and physical port that matches the VACL log criteria was detected. The first [dec] is the VLAN number, the second [dec]/[dec] is the module/port number, the third [dec] is the L4 protocol type, and the fourth [dec] is the number of packets received during the last logging interval. The first [int] is the source IP address, and the second [int] is the destination IP address.

As a conclusion, the huge volume of events issued by a system, the large variation in both synatax and semantic, gives little chance to have a general diagnosis approach that covers all the cases, for all system entities. While fixing legacy systems remains a complex problem, avoiding the same errors for newly designed systems seems be to the most appropriate approach.

4.2 Validation of the Paper Proposal

We can summarize that the paper presents a step-by-step system diagnosis approach by proposing a way to define events (with a well defined syntax and semantic), and their links with potentially related events. Instead of a "unique format", we proposed an ontology-based approach, per small domains.

While writing rules has proven to be a tedious activity, if not almost impossible, discovering event patterns, associating a potential list of recommended diagnosis actions for each pattern, and keeping track of succesfull actions (via diagnosis validation engine) seems to be a feasable approach.

4.2.1 Industrial Connection of the Solution – Cisco Device Fault Manager (DFM)

We recall that the status on the network elements is captured via notifications, polling, etc., and a behavioral deviation may be identified. The simulations were performed in connection with Cisco DFM (Device Fault Manager) [30], which will be briefly presented below.

Cisco's Device Fault Manager (DFM) is a specialized software offering real-time assistance for system diagnosis. The main activities it performs are:
(i) Monitors and displays the operational network health data;
(ii) Analyzes the events triggered in the network and determines when a possible fault has occurred, and
(iii) Sends notifications to pre-determined users through Graphical User Interface display or through a series of configurable notifications.

Figure 18 [30] shows the tasks performed by DFM and the relations with other components. The software constantly gathers information from system elements, analyzes and prioritizes the data, and sends the appropriate notifications.

DFM can send three types of notifications:
(i) SNMP Trap Notifications – DFM processes the traps and events it receives, and generates its own traps with the format defined in CISCO-EPM-NOTIFICATION-MIB [30].
(ii) E-mail Notification - DFM sends e-mail messages with the information about the event and the alert that caused it.
(iii) Syslog Notification - DFM generates Syslog messages that can be forwarded to Syslog daemons on remote systems.

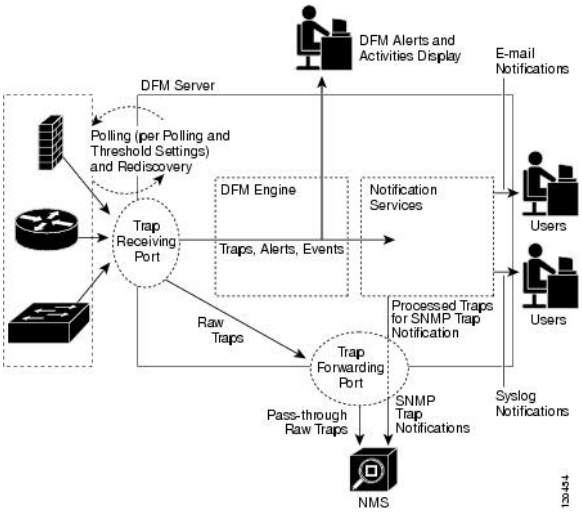

Figure 18. DFM Architecture [30]

DFM uses the concept of *subscriptions* to allow the specification of the elements needed for its activity. SNMP Trap Notification subscriptions, or E-mail subscriptions, have the following elements in common:
(i) Devices, representing the devices of interest;
(ii) Alert Severity and Status;
(iii) Event Severity and Status (the names of the events can also be customized);
(iv) Recipients, representing one or more hosts to receive the SNMP traps or uses to received the e-mails, and
(v) Name, needed to uniquely identify the subscription.

DFM analyzes the events as they arrive, or polls the elements regularly. When an event or alert occurs that matches a subscription, a notification is sent. An event trigger can be either normal polling, a threshold that was exceeded, or a trap that was received.

A category of traps are generated by system elements, but are not processed by DFM; these re referred to as *pass-through traps.* They appear in the Alerts and Activities Display of the software, if they were generated by a device managed by DFM. The software can be configured to forward pass-through traps from managed or unmanaged devices to other elements. If DFM does not know which device generated the trap, it ignores the trap.

DFM perform system elements polling via two mechanisms:
(i) Using a high-performance, asynchronous ICMP poller with two threads.
(ii) Using the SNMP poller, which has ten synchronous threads running in parallel.

The two polling mechanisms are coordinated for optimal results.

4.2.2 Event Definition

In the previous sections, we presented a way to define an event; we applied this approach for a limited number of events, as a proof of concept for the proposal. The progressive approach we adopted for the validation was to develop the event features, represented in fields, in a context, as shown in Figure 19.

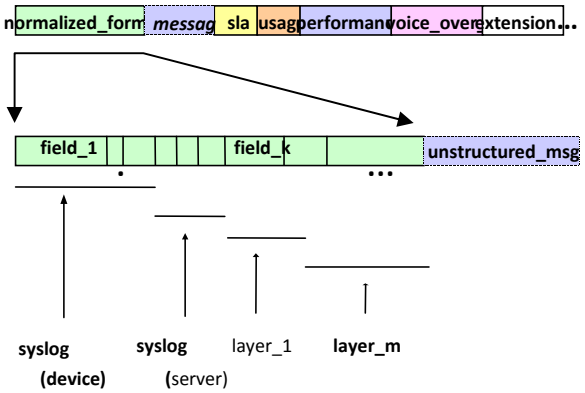

Figure 19. A Progressive, Context-oriented, Event Definition

We started with a basic set of fields (features), with a precise syntax and semantic. Then, we considered extended features in a domain. This helped control the diagnosis rules in a particlar context. For example, there are specific rules in SLA domain, such as SLA violations, SLA penalties, etc. In our simulation, the *SLA domain* and *performance domain* were considered. This approach was facilitated by the fact that, for each domain, there are well defined and accepted concepts, i.e., semantic tags. For each such domain, we built a semantic tag cloud. Therefore, bridging between a semantic tag cloud belonging to the *performance domain* and another one belonging to the *SLA domain* was easier.

4.2.3 Symptoms-Problems-Diagnosis Rules

Based on the trio S-P-D (symptoms-problems-diagnosis), a diagnosis process was triggered, performed by the "diagnosis engine" and based on the framework presented in Figure 20.

In Figure 20, the "management console", the "management system (mgmt)", and the "device fault manager" are existing components. Since we were not focusing on gathering network events and status, a DFM-like input was used for the simulation. As the current network entities do not issue events in the required form to apply our strategy, we rather used transformation processes to get the information as needed.

The selected events were normalized first ("1" in Figure 20) following the general process pictured in Figure 19. The QoD engine ("2" in Figure 27) took as input the status of monitored parameters (counters, CPU, memory, ports, etc.), while ("3" in Figure 27) processed the coming events to identify patterns, and, hence, correlate them, and sent the recommended actions associated with a given pattern to the mgmt (DFM) (actually, to the operator console). It also stored the situation (set of the watched states) when a set of the recommended actions are issued. In a real environment, it also stores the status of the system after one or more selected actions are declared "successful". This allows the finding of a more rapid solution when the same situation occurs.

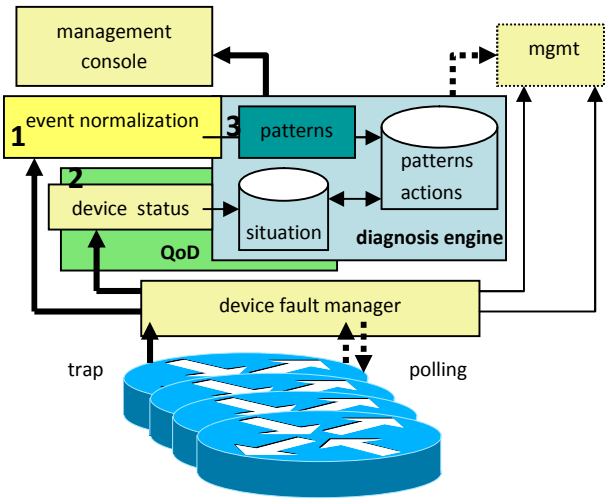

Figure 20. Simulation Architecture Considering DFM features

It is expected that a validation takes place, either preventively (for preventive diagnosis), following a small variation in the normal behavior, after a diagnosis, or a stable period.

Implementation of the diagnosis decisions is based on the same symptom-problem-diagnostic trio paradigm. Based on symptoms sent by NEs (Network Elements) to the monitoring and diagnosis applications, or polled by these applications themselves, a set of problems (or only one single problem) become candidates for leading to a diagnostic. Upon the confidence degree of the diagnosis engine, more information may be polled from the NEs to complete, make more accurate, or re-validate a status with the final goal of triggering the most suitable action on the managed system, or providing the most accurate diagnosis (best-practices, recommended actions, reports) to the network operator or managing systems.

S-P-D may be executed on different engines, as they may be distributed; in our case, we used one single diagnosis engine.

Recursive receiving/polling actions may be performed at any stage of processing, within S, P, and D processes.

4.2.4 Events Processing

The first step in the simulation process was to normalize the input events, as shown in Figure 21. The main achievement of the normalization was that the same property represented by the field value of an event, will always be on the same position in the normalized event, regardless of its initial position in the non-normalized input event. Obviously, there are empty fields in the normalized event for use when new event types arrive, with new fields/features. Following an ontology model previously described, the normalized events also included pointers to other event types that an event has relations to.

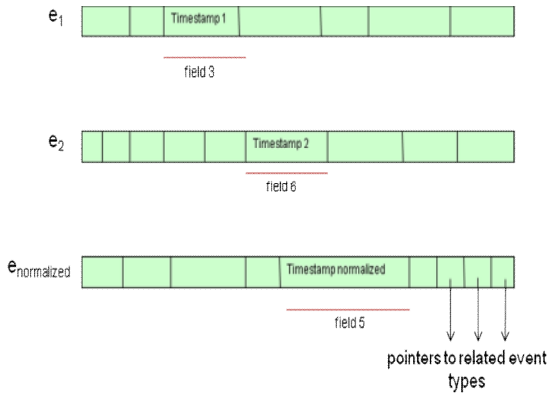

Figure 21. Event Normalization

Post-event normalization, several operations were needed to process the events in terms of their number and referring to the problems they report. The events falling within a specified time window were considered. We recall that a group of events that are correlated form what we refer to as *a pattern*. We used the following *operators* to identify the patterns in our simulation:

(i) Compression [a, a,...,a] ➔ *a*

Compression is done when *identical alarms* are produced by the same network element and they have the same message contents except timestamps. Compression is the task of reducing multiple occurrences of identical alarms into a single representative of the alarm.

(ii) Filtering [a , p(a)< H] ➔ *Ø*

Filtering is the most widely used operation to reduce the number of alarms presented to the operator. If some parameters *p(a)* of alarm *a*, e.g., priority, type, location of the network element, timestamp, severity, etc., do

not fall into the set of predefined legitimate values *H*, then alarm *a* is simply discarded or sent into a log file.

(iii) Suppression [a , C] ➔ *Ø*

Suppression is a context-sensitive process in which an event *a* is temporarily inhibited depending on the dynamic operational context *C* of the network management process. Temporary suppression of multiple alarms and control of the order of their exhibition is a basis for dynamic focus monitoring of the network management process.

(iv) Count [n a]* ➔ *b*

Count results from counting the number of repeated arrivals of identical alarms. When a pre-defined threshold is met, the specified number of alarms are substituted by a new alarm.

(v) Escalation [n a, p(a)]* ➔ *a, p'(a), p'>p*

Escalation assigns a higher value to some parameter *p'(a)* of alarm *a*, usually the severity, depending on the operational context, for example, the number of occurrences of the event.

(vi) Generalization [a, a⊂b] ➔ *b*

Alarm *a* is replaced by its super class *b*. Alarm generalization has a potentially high utility for network management. It allows one to deviate from a low-level perspective of network events and view the situations from a higher level.

(vii) Specialization [a, a⊃b] ➔ *b*

Specialization is the opposite procedure of alarm generalization. It substitutes an alarm with a more specific subclass of this alarm.

(viii) Temporal relation [a T b] ➔ *c*

Temporal relation T between *a* and *b* allows them to be correlated depending on the order and time of their arrival.

(ix) Clustering [a, b, ..T,∧,∨,¬,] ➔ *c*

Clustering allows the creation of complex correlation patterns using logical operators ∧*(and)*, ∨(*or)*, and ¬(*not)* over component terms. The terms in the pattern could be primary network alarms, or previously defined correlations (following the event ontology).

These operators were applied to the normalized events to identify several patterns in our time window. Each event contains information on the source of the event. The "state" of all event sources in a pattern was gathered by polling the sources themselves. These states acted as "pre-conditions" to the diagnosis process. An example of a pattern format is presented below:

{(e_1, $source_1$, $status_1$), (e_2, $source_2$, $status_2$)} ::= $pattern_1$

The sources of the events in a pattern are part of a semantic tag cloud, which is associated with a set of recommended actions. For each pattern, if all the sources belong to the same semantic tag cloud, the recommended actions associated with that cloud are forwarded to the management console. If the sources in a pattern belong to multiple semantic clouds, a union of the recommended actions of all these semantic clouds is forwarded to the management console.

4.2.5 Actions Triggering

When the diagnosis P-D rule identifies specific actions to be triggered, different rules may be in place with respect to triggering rights, such as:

(i) Only the network operator may be allowed to perform the suggested actions. In this case, the diagnosis must be human-understandable, accurate, unambiguous, and feasible. This is mainly related to the event format and specification of the diagnosis actions. DFM uses appropriate Tcl_Tk (Tool Command Language Toolkit) scripts to determine if a message needs to be sent to a human operator of to other management applications for processing.

(ii) The diagnosis engine itself is automatically triggering the diagnosis actions. This was not yet experimented.

(iii) Other management applications are in charge of the diagnosis actions and their results.

In one of these manners, the recommended actions are applied and the diagnosis engine is notified to poll the new states of the event sources determine the "post-conditions". If the post-conditions hold, the action is marked as "successful" for use in future similar situations.

4.2.6 Points to Consider for Implementation

There are several points to consider when implementing the system proposed in this paper. We briefly mention some of them in this section.

(i) Console information is crucial in detecting severe crashes allowing a key message to describe the probable cause. In some situations, the device is not longer able to send the message to some server, or any other device. Commonly, Cisco routers are configured to send console events as Syslog messages. However, even in the case of crashes, the messages printer just before can help diagnose the nature of the problem.

(ii) Cisco IOS devices (IOS 11.2 and above) can be configured to send Syslog messages to a Syslog message-trap converter. Several SNMP notifications have been pre-configured, while others are enabled. Certain are mandatory (coldStart, warmStart, authenticationFailure, linkDown, linkUp, egpNeighborLoss). Since authenticationFailure and LinkUp/LinkDown may trigger a lot of events, one can apply some selective thresholds. Alternatively, a polling procedure may be started on those parameters causing these events. Equally, link state notification should be

enabled only for some objects, devices, and especially for LinkDown.

(iii) For those entities that require event configuration, one can use an approach where the device checks the trigger point and generates the events according to the threshold type (no need to collect and filter data). The risk of this method is that events can be lost for many reasons other than non-crossing threshold points. Threshold on devices are called *agent-based threshold*. SNMP MIB and RMON MIB allow setting thresholds on devices. Alternatively, data can be collected at a management station and analyzed against the appropriate threshold. This kind of setting is called *network management system-based triggering*.

V. CONCLUSION AND FUTURE WORK

The concepts proposed in this paper are implementable, at-large, but considerations must be taken when designing new systems for follow the formal approach. In this section, we highlighted the main challenges of a full implementation of the proposed diagnosis solutions. While fixing legacy systems remains a complex problem, avoiding the same errors for newly designed systems seems be to the most appropriate approach. We illustrated partial solutions for different concepts introduced via a case study simulation.

As a result, the successfully marked actions can be re-used as recommended actions when similar event patterns occur. When an event pattern inventory exists, a similar algorithm is associated with each pattern. In this case, the Diagnosis Engine behavior is a combination of all these algorithms.

We presented an adaptive framework for diagnosis validation and transfer of information from successful outcomes for future use and optimization of the diagnostic activity. It was shown that the current diagnosis techniques needs validation of successful actions and a process for knowledge transfer. We introduced the QoD, an engine that validates the successful actions and transfer knowledge for use in similar situations. This new approach pave the way to adaptive diagnosis, where only those previously classified as “successful actions” are deemed to be applied in similar situations.

Towards automation, a few open issues are in our plans for future exploration and solutions. Criteria for symptom similarly must de defined and experimented in different contexts. Metrics for measuring accuracy of the QoD evaluation should be derived per technology domain and domain context. We intend to explore a progressive ontology (leading to progressive diagnosis) and therefore “progressive validation” of successful actions. This is intended to solve potential conflicts of the post-conditions of the actions already validated as “successful” and to evaluate the accuracy of the diagnosis actions (preciseness versus permanent damage). A specific target is to propose a flow engine, as an instantiation of the QoD that identifies context-based temporal patterns of successful or failure diagnosis actions.

VI. REFERENCES

[1] M. Steinder and A. S. Sethi, A survey of fault localization techniques in computer networks, Science of Computer programming 53 (2004) 165-194

[2] A. Patel, G. McDermott, and C. Mulvihill, Integrating network management and artificial intelligence, in: B.Meandzija, J.Westcott (Eds.), Integrated Network Management I, North-Holland, Amsterdam, 1989, pp. 647–660

[3] I. Katzela, Fault diagnosis in telecommunications networks, Ph.D. Thesis, School of Arts and Sciences, Columbia University, New York, 1996

[4] L. Lewis, A case-based reasoning approach to the resolution of faults in communications networks, in: H.G. Hegering, Y. Yemini (Eds.), Integrated Network Management III, North-Holland, Amsterdam, 1993, pp. 671–681

[5] M. Frontini, J. Griffin, and S. Towers, A knowledge-based system for fault localization in wide area network, in: I. Krishnan, W. Zimmer (Eds.), Integrated Network Management II, North-Holland, Amsterdam, 1991, pp. 519–530

[6] J. Goldman, P. Hong, C. Jeromnion, G. Louit, J. Min, and P. Sen, Integrated fault management in interconnected networks, in: B. Meandzija, J. Westcott (Eds.), Integrated Network Management I, North-Holland, Amsterdam, 1989, pp. 333–344

[7] C. Joseph, J. Kindrick, K. Muralidhar, and T. Toth-Fejel, MAP fault management expert system, in: B.Meandzija, J.Westcott (Eds.), Integrated Network Management I, North-Holland, Amsterdam, 1989, pp. 627–636

[8] B. Gruschke, Integrated event management: Event correlation using dependency graphs, in: A.S. Sethi (Ed.), Ninth Internet. Workshop on Distributed Systems: Operations and Management, University of Delaware, Newark, DE, October 1998, pp. 130–141

[9] S. Kätker and K. Geihs, A generic model for fault isolation in integrated management systems, Journal of Network and Systems Management 5 (2) (1997) 109–130

[10] K. Houck, S. Calo, and A. Finkel, Towards a practical alarm correlation system, in: A.S. Sethi, F. Faure-Vincent, Y. Raynaud (Eds.), Integrated Network Management IV, Chapman and Hall, London, 1995, pp. 226–237

[11] I. Katzela and M. Schwartz, Schemes for fault identification in communication networks, IEEE/ACM Transactions on Networking 3 (6) (1995) 733–764

[12] A.T. Bouloutas, S. Calo, and A. Finkel, Alarm correlation and fault identification in communication networks, IEEE Transactions on Communications 42 (2–4) (1994) 523–533

[13] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo, A coding approach to event correlation, in: A.S. Sethi, F. Faure-Vincent, Y. Raynaud (Eds.), Integrated Network Management IV, Chapman and Hall, London, 1995, pp. 266–277

[14] M. Steinder and A. S. Sethi, End-to-end service failure diagnosis using belief networks, in: R. Stadler, M. Ulema (Eds.), Proc. Network Operation and Management Symposium, Florence, Italy, April 2002, pp. 375–390

[15] V. Venkatasubramanian, R. Rengaswamy, K. Yin, and S. N. Kavuri, "A review of process fault detection and diagnosis, Part I: Quantitative model-based methods," Computer and Chemical Engineering, vol.27, pp.293–311, 2003.

[16] W. Stallings, SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards, Addison-Wesley Publishing Company, 1993, ISBN 0-201-63331-0

[17] M. Popescu, P. Lorenz, and J.M. Nicod, An adaptive Framework for Diagnosis Validation, The Proceedings of The Third International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP) 2009. IEEE Press, pp. x-y

[18] Cisco IOS Network Management Configuration Guide, 2008 http://www.cisco.com/en/US/docs/ios/netmgmt/configuration/guide/nm_esm_syslog_ps6441_TSD_Products_Configuration_Guide_Chapter.html (last accessed December 2009)

[19] W. Stallings, SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards, Addison-Wesley Publishing Company, 1993, ISBN 0-201-63331-0

[20] Network Management System: Best practices White Paper, 2008 http://www.cisco.com/en/US/tech/tk869/tk769/technologies_white_paper09186a00800aea9c.shtml (last accessed December 2009)

[21] Augmented Bachus-Naur form http://web.mit.edu/macdev/mit/doc/www/devdoc/Augmented%20BNF.html

[22] M. Popescu, P. Lorenz, and J.M. Nicod, An Adaptive Framework for Diagnosis Validation, The Proceedings of The Third International Conference on Advanced Engineering Computing and Applications in Sciences, ADVCOMP 2009, Sliema, Malta, pp. 123-129, IEEE Press

[23] M. Popescu, P. Lorenz, M. Gilg, and J.M. Nicod, Event Management Ontology: Mechanisms and Semantic-driven Ontology, The Proceedigns of The Sixth International confernces on Networking and Servcies, ICNS 2010, Cancun, Mexico, pp. 129 - 136, IEEE Press

[24] W. Stallings, SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards, Addison-Wesley Publishing Company, 1993, ISBN 0-201-63331-0

[25] M. Popescu et al., US Patent 7275017, Method and apparatus for generating diagnoses of network problems

[26] L. Lamport, TLA: Temporal logic of Actions, http://research.microsoft.com/enus/um/people/lamport/tla/tla.html (last accessed August 12, 2010)

[27] R. Griffith, J.L. Helelrstein, G. Kaiser, and Y. Diao, Dynamic Adaptation of Temporal Event Correlation for QoS Management in Distributed Systems, 2006 www.cs.columbia.edu/techreports/cucs-055-05.pdf (last accessed August 2, 2010)

[28] K. Walzer, T. Breddin, and M. Groch, Relative temporal constraints in the RETE algorithm for complex event detection, Proceedigns of the Second International Conference on Distributed Event-based Systems, 2008, pp. 147-155

[29] M. Popescu, Temporal-oriented policy-driven network management, Master Thesis, Mcgill University, Canada 2000, p. 140

[30] M. Popescu, Semantic Mechanisms for Cross-Domain System Diagnosis Mécanismes Sémantiques pour le Diagnostic des Systèmes, PhD Thesis, University of Besançon, France, 2010