Intelligent Fault Diagnosis in Computer Networks

Xin Hu

Kongens Lyngby 2007
IMM-THESIS-2007-49

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

As the computer networks become larger and more complicated, fault diagnosis becomes a difficult task for network operators. Typically, one fault in the communication system always produces large amount of alarm information, which is called alarm burst. Because of the large volume of information, manually identifying the root cause is time-consuming and error-prone. Therefore, automated fault diagnosis in computer networks is an open research problem.

The aim of this thesis is to develop a software system for Motorola Denmark, which assists network operators to diagnose fault in an intelligent, highly accurate and efficient way.

In this thesis, we shall analyze the current fault diagnosis techniques. Then we shall propose a generic framework for constructing fault diagnosis systems used in computer networks. Finally, we shall design and implement such a system specifically for Motorola Denmark.

Keywords: Fault localization, Fault Diagnosis, Event Correlation, Rule-Based Reasoning, Model-Based Reasoning

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the degree in Master of Science in Computer Systems Engineering.

The project, titled "Intelligent Fault Diagnosis in Computer Networks", has been carried out by Mr. Xin Hu during the period between October 1st , 2006 and May 31st , 2007.

This thesis was supervised by Mr. Jørgen Fischer Nilsson, Professor within the Department of Informatics and Mathematical Modelling at the Technical University of Denmark, and was collaboration with Motorola, Denmark.

The thesis consists of a summary report and a prototype system SECTOR which can automatically diagnose the faults in a computer network.

Lyngby, May 2007

Xin Hu

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Jørgen Fischer Nilsson, for helping me throughout the whole period, for sharing his brilliant ideas which always resulted in very interesting discussions.

Secondly, I would also like to thank Mr. Søren Sørensen from Motorola, Denmark. He really gave me great help and valuable first-hand information regarding Dimetra system.

Thirdly, I will show my great acknowledgment to my dearest parents for their love, support and encouragement every second.

Finally, thanks to all my friends in Denmark and in China who care about and support me all the time!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Background

Today's computer networks, for instance telecommunication networks, are becoming much larger and more complex. One single fault occurred in one network component may cause considerably high volume of alarms to be reported to network operators, which is called *alarm burst*. Alarm burst may be a result of (1) fault re-occurrence, (2)multiple invocations of a service provided by a faulty component, (3) generating multiple alarms by a device for a single fault, (4) detection of and issuing a notification about the same network fault by many devices simultaneously, and (5)error propagation to other network devices causing them to fail and, as a result, generate additional alarms [1]. Thus, it is a challenge for network operators to quickly and correctly identify the root cause by analyzing those large amount of alarms.

Dimetra [20] is a radio networking system provided by Motorola. The fault diagnosis in **Dimetra** is currently handled manually. Operation staffs browse alarms which are delivered to **FullVision (FV)** [20, 21] from all kinds of physical devices or logical objects, and then analyze those alarms to find possible problems causing alarms. This manual process is not able to scale well when the system gets larger and more complicated. Furthermore, customers can not tolerate such an ad hoc, error-prone and labor-intensive approach. Last but not the least, it will considerably increase the cost if customers buy fault diagnosis solution from the third parties. Therefore, developing an intelligent fault diagnosis solution is a critical requirement for Motorola, Denmark.

1.2 Project Goals

This project was set up based upon the requirements mentioned above. The main goal of this project is to develop a simple and flexible prototype system for Motorola Denmark, which automates the process of fault diagnosis in **Dimetra** system in an intelligent, highly accurate and efficient fashion. If possible, a generic framework and model for constructing such a system workable for multi-domain networking systems will be proposed.

1.3 Project Scope

Due to the limitation of time and the complexity of **Dimetra** system, a basic and simplified **Dimetra** system was investigated, which consists of core elements and supports voice only operation. Hence, the developed system only handles fault diagnosis in such a basic **Dimetra** system at the moment.

On the other hand, as a prototype system, the evaluation was carried out in a simulated environment rather than the real field. Furthermore, the evaluation was only based on several fault scenarios (test cases), it is therefore not thorough yet. Last but not the least, the developed system has no Graphic User Interface and may contain some bugs since it is not a production system.

However, with more development, the author believes the developed system could be used as a real world application in the future.

1.4 Main Work

In this thesis, a generic framework for fault diagnosis, which is based on alarm (event) correlation technology, was proposed. It mainly follows the principles of model-based reasoning but also combines idea from the rule-based reasoning. With this framework, developers can model all kinds of networking system, identify and model diagnostic knowledge, and finally build a fault diagnosis system.

This framework was implemented by a system named **SECTOR** - **S**imple **E**vent **C**orrela**TOR**. SECTOR relies on the alarm (event) correlation technology. The evaluation shows that SECTOR can identify the right faults from alarm flood with acceptable latency.

1.5 Structure of the Report

Eight chapters and four appendixes are included in this report.

Chapter 1 gives a brief introduction to the project including the background, goals, scope and achievements.

Chapter 2 introduces the theory in the domain of fault diagnosis. Relevant concepts are explained in this chapter. Furthermore, it describes several techniques which can be applied in the fault diagnosis, as well as examines their advantages and disadvantages.

Chapter 3 introduces and describes the Dimetra system. Basic components are described with particular emphasis on the their functionalities as well as the dependencies between them. Furthermore, alarms reported by those components are analyzed. Finally, a fault propagation model is represented for a sample Dimetra.

Chapter 4 presents the proposed framework, on which a fault diagnosis system can be built based. This framework utilizes the idea of event correlation and combines the rule-based and the model-based solutions. It is designed to be as generic as possible in order to be used in other domains.

Chapter 5 concentrates on the design of a fault diagnosis system-SECTOR, which is based on the proposed framework. The functionalities of the SECTOR system are defined in this chapter. Furthermore, it describes the whole system architecture together with the communication between the different parties in the system.

Chapter 6 describes the implementation of the SECTOR system using Java language. A description of all system modules, as well as the the class diagrams have also been provided. In addition, important implementation details are given.

Chapter 7 demonstrates how the SECTOR system has been tested and evaluated. The test strategies used in the test are described. The major test cases and their results are provided. At the end of this chapter, the performance evaluation based on the results is given.

Chapter 8 is the conclusion of this thesis. It concludes this project by analyzing the achieved goals, and the limitations which identify the possible future work.

Appendix A presents the class diagrams of some important classes.

Appendix B lists the source code of all important classes.

Appendix C introduces the XML description files, including the one for model description and the one for the event specification.

Appendix D introduces the test cases of the project as well as the results.

Chapter 2

Fault Diagnosis

Fault diagnosis, informally speaking, is a process of finding faults according to the observed symptoms. Fault diagnosis referred in this thesis is the one in the context of networking systems. Currently, fault diagnosis in computer networks remains an open research problem [2]. It is because there is not one single solution that can address all issues.

This chapter introduces the theory of fault diagnosis by illustrating related concepts and techniques, and tries to give readers a basic understanding of the fundamental ideas behind fault diagnosis. This chapter is mainly based on a survey in [2] by following its way to describe the theory of fault diagnosis.

2.1 Concepts of Fault Diagnosis

Some basic concepts are introduced first.

Event, as an exceptional condition occurring in the operation of hardware or software of a managed network, is considered as a central concept in the context of fault diagnosis [2]. The hardware or software associated with an event is named as *managed object*. Events can be classified as *primi-*

tive or *composite* events [3, 4]. *Primitive events*, pre-defined in a system, are usually directly generated in managed objects. *Composite events* are conceptual events which are constructed from primitive events or low-level composite events.

Faults (also referred to as *problems*) are network events that are causes for malfunctioning [2, 5]. Thus, *faults* can cause other events. A class of faults which are not themselves caused by other events are named *root causes*. Faults may propagate across the entire network. It is because that many network objects are dependent on each other, and a fault in one object always causes faults in its depending objects. *Fault propagation* is one cause of alarm burst.

Symptoms are defined as external manifestations of failures [2]. A symptom is observed as an *alarm*, a notification of the occurrence of a specific event [5]. *Event* and *Alarm* are two interchangeable notions in some papers.

Fault diagnosis is a process of finding out the original cause for the received symptoms (alarms) [5]. It usually involves three steps [2]:

- *Fault detection*, an on-line process which indicates that some network objects are malfunctioning according to the alarms reported by those objects.
- *Fault localization* (also referred to as *fault isolation*, *alarm/event correlation* and *root cause analysis*), a process that proposes possible hypotheses of faults by analyzing the observed alarms.
- *Testing*, a process that isolates the actual fault from a number of possible hypotheses of faults.

This thesis concentrates on the second step of fault diagnosis since it is the most essential step.

Alarm/Event correlation, is a technique that conceptually interprets multiple alarms/events so that those having the same root cause are grouped [2, 4, 6]. After correlation, the number of alarms (event notifications) is reduced but the semantic contents are increased. Thus, *Alarm/Event correlation*, as the most popular fault localization technique, dramatically helps network operators find root cause from high volume of information. The most important correlation types are listed as follows [4, 5, 7]:

- *Compression*: Reduction of alarms which are the notification of multiple occurrence of one event into a single alarm.
- *Counting*: Substituting a new alarm to a specified number of alarms associated with a recurring event.

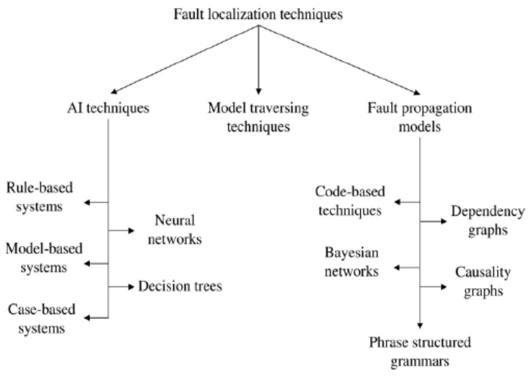


Figure 2.1: Classification of fault localization techniques [2]

- *Causal Relationship*: Correlating alarms when the events behind them have causal-effect relationship.
- *Temporal Relationship*: Correlating alarms according to the order or the time at which alarms are generated. It is because that alarms caused by the same fault are likely to be observed in certain order or within a short time after the fault occurrence. Note that the temporal relationship between alarms may not exactly reflect the one between events. Because some alarms will be generated earlier than those with lower priority but whose corresponding events occurred earlier.

There are numerous fault localization techniques. A classification of the existing solutions is presented in Fig. 2.1 [2]. These solutions include artificial intelligence (AI) techniques, model traversing techniques and graph-theoretic techniques (fault propagation models). Some interesting techniques will be described in the following sections.

2.2 Graph-theoretic techniques

Graph-theoretic techniques are based on a fault propagation model (FPM), which is a graphical model describing which symptoms may be observed when a specific fault occurs [2, 8]. FPM models all faults, symptoms, and the causal relationships between them. Hence, fault localization algorithms can identify the most possible faults by analyzing the FRM. A FRM can be represented as either a causality graph or a dependency graph.

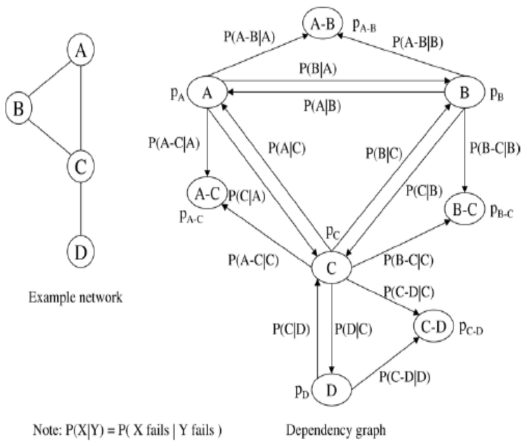


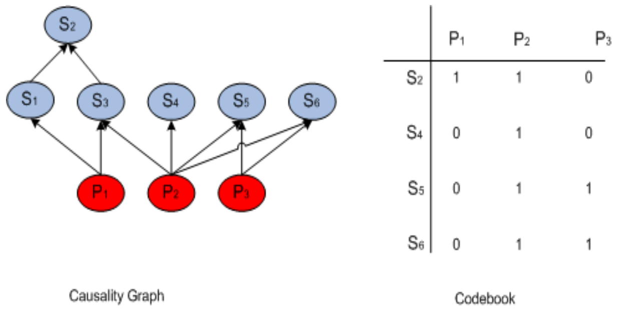
Figure 2.2: Simple network and a corresponding dependency graph [2]

As a directed acyclic graph, a *causality graph* $G_c(E, C)$ maps events into its nodes E, and maps cause-effect relationships between events into edges C. An edge $(e_i, e_j) \in C$, which is denoted as $e_i- > e_j$, shows that event e_i causes event e_j [2, 4]. Moreover, a probability can be associated with an edge $(e_i\ , e_j)$ to indicate how possible event e_j could occur provided that event e_i has occurred.

A *dependency graph* is a directed graph $G_d = (O, D)$, whose nodes O correspond to a finite, non-empty set of objects in a system; and whose edges D represent dependency relationships between objects. A directed edge $(o_i, o_j) \in D$ denotes a dependency that o_i will get affected if its dependent object o_j is faulty [2]. The uncertainty about dependencies can be modeled by assigning a conditional probability to the edges D. [9]. Fig. 2.2 [2] shows an example network and its dependency graph.

It is quite often to use a dependency graph as a system model due to the mapping of network objects. On the other hand, causality graphs are more used with fault localization algorithms to identify faults since they provide a more detailed view of faults and events in a system [2].

In the following sub-sections, two graph-theoretic techniques will be presented.



	P1	P2	P3
S2	1	1	0
S4	0	1	0
S5	0	1	1
S6	0	1	1

Figure 2.3: Codebook derived from an example causality graph

2.2.1 Codebook technique

Codebook technique learns idea from the coding technique and proceeds in two phases: *codebook generation* and *decoding* [10, 11].

A *codebook*, a matrix of codes identifying individual problem events, is firstly constructed based on a causality graph. A *code* is a vector $(s_0, s_1, ...s_n)$. Each s_i corresponds to a symptom event S_i. In the deterministic context, s_i takes value 0 or 1. When s_i equals 1, the symptom event S_i must occur as the consequence of the problem event identified by that code. In the indeterministic context, it is natural to assign s_i a value from 0 up to 1. The bigger the value of s_i is, the more possible that event S_i can be caused by the problem event identified by that code. A sample *codebook* derived from a sample causality graph is presented in figure 2.3. Note that not all symptoms are used to generate this *codebook*. It is because that some symptoms do not contribute further information indicating problems except the one which has already been provided by other symptoms. Therefore, the elimination of those symptoms bring higher efficiency but without loss of information. E.g. symptom S_1 is eliminated in presence of symptom S_2, even though S_1 is the effect of problem P_1 as well.

Once the *codebook* is created, the process of finding problems can be considered as a process of decoding of observed symptoms to a set of problems. Because of the existence of spurious or lost symptoms in the real world, only problems whose codes optimally match the observed symptoms are selected as the result of fault diagnosis.

Distinction between problems is measured in terms of Hamming distance[1] be-

[1]In information theory, the Hamming distance between two strings of equal length is the number of positions for which the corresponding symbols are different. E.g. the Hamming distance between 10**11**101 and 10**01**001 is 2.

tween their codes. [11] defines that the *radius* of a *codebook* is half the minimal Hamming distance among codes. When the radius is 0.5, each code can distinguish problem from one another but the decoding is not resilient to noise. A conclusion is given in [11]: "Generally, we can correct observation errors in $k-1$ symptoms and detect k errors as long as k is less than or equal to the radius of the codebook."

Codebook technique is very efficient because the codebook is generated only once at development time and decoding process is very fast by utilizing minimal distance decoder at run time. The computational complexity is bounded by $(k+1)log(p)$, where k is the number of errors that the decoding phase may correct, and p is the number of problems [10]. However, the accuracy of the codebook technique is unpredictable when more than one problem occur with overlapping sets of symptoms [2]. In addition, codebook has to be regenerated whenever system configuration changes. As a result, this technique is not suitable for frequently changed environments unless the codebook can be automatically generated according to current system configuration [11].

2.2.2 Context-free grammar

Context-free grammars (CFGs) [43] is a natural candidate to represent a hierarchically organized communication network [12]. In this model, the indivisible network components can be represented as *terminals*, and compound network components correspond to *variables*, which are built from the already defined variables or terminals according to some *production rules* . An example network is given in Fig. 2.4 to show how CFGs is used to model a communication network. In this network, the basic units are four terminal points: A, B, C, D and three channels: $channel-AB, channel-BC, channel-CD$. The network can be represented by the following production rule:

```
NETWORK -> LINK-AB . LINK-BC . LINK-CD
```

Each link can be further represented by productions:

```
LINK-AB -> NODE-A . CHANNEL-AB . NODE-B
LINK-BC -> NODE-B . CHANNEL-BC . NODE-C
LINK-CD -> NODE-C . CHANNEL-CD . NODE-D
```

In some cases, CFGs can more effectively model complicated dependent relationships than the dependence graph. Consider the case where a channel consists

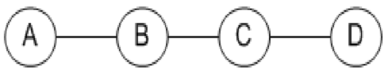


Figure 2.4: A sample network

of two subchannels. The channel is operational if any of the subchannels is operational. This is difficult to model using a dependence graph but it is easy to model using a CFGs [12]. Because a CFGs is able to encode semantics, e.g., the operation of one system is dependent on the operation of its subsystems which are dependent on the operation of basic devices and components.

[12] proposed two fault identification algorithms based on CFGs. Both algorithms try to find the best explanation. The first one chooses a minimum set of faults that explains all observed alarms. If there are more than one such a set, the one with least information cost is chosen. The information cost for one fault is defined as the negative of the logarithm of the probability of that fault. On the other hand, the second algorithm finds faults that explain parts of observed alarms with the minimal information cost in order to handle the case of lost or unreliable alarms which is not considered in the first algorithm.

Both algorithms rely heavily on a-priori information which is either guessed or can be experimentally gained. Furthermore, they are rather complex and should be considered as a guideline for designing fault localization algorithms [2]. Thus, Fault diagnosis based on CFGs may be far away from a practical solution until a more effective algorithm is proposed. However, CFGs provides a general model to represent the network and algorithms applied with this model can solve the fault identification problem in the presence of multiple faults, and lost and spurious alarms.

2.3 AI techniques

Systems implemented in AI techniques are referred as expert systems. Various solutions are derived from the field of AI. They are rule-, model-, and case-based reasoning tools as well as decision trees, and neural networks. All these solutions are examined in the following subsections.

2.3.1 Rule-based Approach

Rule-based approach is significantly used in many commercial fault diagnosis products. In rule-based systems, the diagnostic knowledge of a human expert is modeled as rules, which are saved in a knowledge-base. Formally, rules are expressed in form of production rules, *e.g. if A then B*, where *A* is called *antecedent* and *B* is called *consequent*. Antecedent is usually the assertion on the frequency and the source of an alarm as well as the values of its properties [13]. In some cases, temporal relationships among several events are also tested [3]. Consequent is usually the action executed when a rule is fired (the corresponding *antecedent* is *true*), e.g. alert the occurrence of a fault or suppress low-priority alarms.

Once rules are defined, the fault localization process is driven by an inference engine, the central controlling component in a rule-based system. The inference engine usually uses a forward-chaining inferencing mechanism, which executes in a sequence of rule-firing cycles to reach a conclusion explaining the situation e.g. observed alarms.

A main goal of research on rule-based fault localization systems is the design of the rule-definition language. Two rule-based diagnostic systems: **ACE** and **JECTOR**, are given as examples.

ACE [13] defines a domain specific language to specify correlation, which matches a group of alarms stemming from a common fault. Rule conditions (antecedents) are expressed in terms of alarm type, arrival time, frequency as well as the number of alarm occurrences. Conditions are classified into: *recognition condition*, *collection condition* and *cancellation condition*. The recognition and cancellation conditions are used to recognize and cancel alarms respectively, which are crucial to problem identification and resolution. Collection condition, on the other hand, is able to compress alarms and reduce distraction. Each rule is characterized by one or more recognition conditions and possibly a collection and/or cancellation condition too. Actions in ACE can range from simple clearing of alarms to network problem correction. The designers of ACE believe that such a rule language representation can better lends itself to solving the problem.

In JECTOR [3], correlation rules are represented as composite event definitions which can precisely express complex timing constraints among correlated event instances. Alarms generated by the managed network devices are defined as primitive events. A composite event is composed of primitive and other composite events, which are correlated due to the causal relationship or temporal relationship between them. These relationships with other constraints are specified in the condition part of a composite event definition. A composite event can be asserted when its condition part has been verified. Thus, the result of correlation can be viewed as occurrences of the corresponding composite events.

Rule-based approach is widely used because human experts' knowledge can be intuitively defined as rules. Furthermore, it does not require profound understanding of the underlying system, which eases developers from domain learning. However, rule-based approach has the following downsides:

- The procedure of knowledge acquisition, which is based upon interviews with human experts, is always time-consuming, expensive and error-prone. However, some approaches can automatically derive correlation rules based on the statistical data, e.g. [14].
- It is unable to learn from experience, therefore the rule-based systems are subject to repeating the same errors.
- It is difficult to maintain because rules frequently contain hard-coded network configuration information.
- It is unable to deal with unseen problems [40].
- It is difficult to update system knowledge [40].

2.3.2 Model-based Approach

In contrast with the traditional rule-based approaches, model-based approaches rely on some sorts of deep knowledge beside the surface knowledge (rules). This deep knowledge is known as system model, which may describe system structures (e.g. network elements and the topology) and its behaviors (e.g. the process of alarm propagation and correlation) [6].

The system model usually uses an object-oriented paradigm [6, 11, 16, 17] to represent network elements as well as the relationship between them. Netmate model [16, 17] is a generic network element class hierarchy, which may be a good basis for modelling other specific network systems. Netmate models some generic network element classes, their attributes and relationships. A *class* is a template for a set of real network elements. All network elements that are instances of one class share the properties defined in that class. Netmate classes are organized along an inheritance hierarchy. Each subclass inherits properties from its superclass. Therefore, inheritance allows system components to be treated generically regardless of their specific details when they are not relevant. Fig. 2.5 [16] shows Netmate's network class hierarchy. *Network Object*,

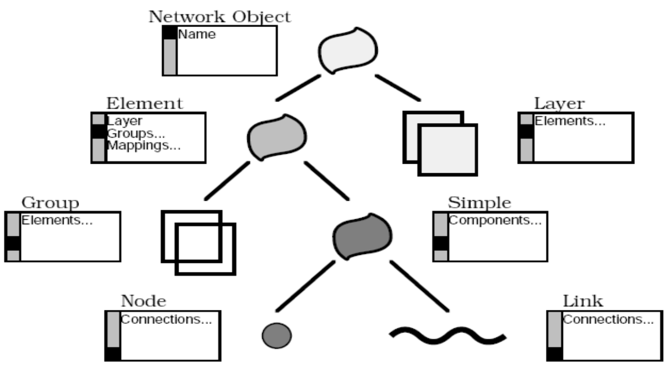


Figure 2.5: Network model class hierarchy [16]

the root of Netmate hierarchy, has two subtypes *Element* and *Layer*. Instances of *Element* are in *Layer* instances, and may be members of *Group* instances. The attribute *Mappings* of one *Element* instance keeps track of its functional counterparts in another layer. Instances of *Node* and *Link* can be considered as *Simple* instances, and additionally be components of other *Simple* instances, or connected to other *Simple* instances. Netmate hierarchy can be reusable across applications by simply adding specific classes into the hierarchy.

IMPACT [6] is a platform for alarm correlation, adopting model-based approach. The proposed model contains a structural component and a behavioral component (Fig. 2.6 drawn according to the figure in [6]). The structural component contains a network configuration model, describing actual NEs (network elements) as well as the relationships among them; and a network element class hierarchy, describing the NE types in an object-oriented way. The behavioral component, by its turn, includes a message class hierarchy, a correlation class hierarchy and several correlation rules. The message class hierarchy describes the alarms generated by NEs and supports alarm generalization. Correlation class along with rules are used to describe the network state based on interpretation of network events. As shown in Fig. 2.6, NE classes, message classes, correlation classes and rules are related by producer/consumer dependencies. Such dependencies are illustrated as: NEs produce messages, messages produce correlation, and rules consume all the above. These dependencies along with other constraints could guarantee the consistency, correctness and completeness of the knowledge base.

Due to the use of deep knowledge, model-based approaches are able to address some issues in rule-based systems. The diagnostic knowledge (rule) is now easy to maintain since its condition part associates system model instead of

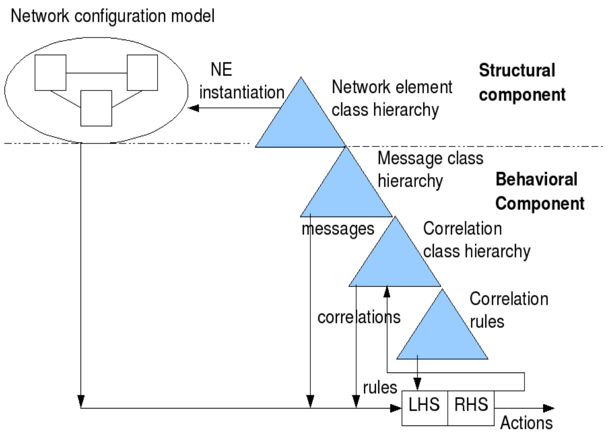


Figure 2.6: Model of IMPACT [6]

hard-coded network configuration. The condition part asserts current network configuration by utilizing predicates referring to the system model. Predicates test the current relationships among system components. Additionally, knowledge in model-based systems can be organized in an expandable, upgradedable and modular fashion by taking the advantage of object-oriented paradigm. Moreover, model-based systems have the potential to solve novel problems [2]. Although model-based approaches are superior to rule-based approaches, they have problems about obtaining models and keeping the models up-to-date.

2.3.3 Case-based Approach

Contrary to rule-based and model-based systems, case-based systems can learn from past cases to propose solutions for new problems [40]. Here, the knowledge is in terms of *cases* not *rules* or *models*. Besides their ability to learn case-based systems are not subject to changes in network configuration [2]. However, it is a complicated and domain-dependent process to adapt an old case to a new situation. [40] proposes a technique named *parameterized adaption* to address this issue. Additionally, the case-based approach may be not used in real-time alarm correlation due to the time inefficiency [42].

2.3.4 Neural Network Approach

A neural network consists of interconnected nodes called neurons to model the neural network in the human brain. They have the ability of learning and therefore can be used to model complex relationships between inputs (observations) and outputs (causes). They are claimed to be robust against noise or inconsistencies in the input data. However, the neural network based systems require long training periods and their behavior outside their area of training is difficult to predict [13].

2.3.5 Decision Tree Approach

A decision tree models an expert's decisions and their possible consequences and can be used to guide a process of diagnosis to reach the root cause. Expert knowledge can be simply and expressively resented by using decision trees [2]. Moreover they have crucial advantage of yielding human-interpretable results, which is important for network operators [44]. However, their applicability are limited due to the dependence on specific applications and the poor accuracy in the presence of noise [2, 45]. A decision tree is usually constructed from data by using the machine learning technique [44].

2.4 Model traversing techniques

Model traversing techniques model network objects especially the relationships among them. Starting from the object that reported an alarm, the fault identification process is able to locate faulty network elements by exploring these relationships [2]. Thus, they are natural candidates when relationships between objects are graph-like. Model traversing techniques are resilient to frequent network configuration changes [8]. However, they have a disadvantage that they can not model the situations in which failure of an object may depend on a logical combination of other object failures [1].

2.5 Summary

This chapter described some basic concepts in the fault diagnosis. Furthermore, various techniques are presented as well as their advantages and disadvantages.

Alarm/Event correlation is considered to be the most popular idea behind most of fault localization techniques due to its power of establishing relationships between alarms/events.

The techniques presented in this chapter cover a large part of research. However, there is not a single technique which is the best, in terms of precision, complexity, performance and adaptation to changes, to solve the generic problems in fault diagnosis. Consequently, some researchers try to combine different techniques to devise a better solution [8, 18].

In general, rule-based approaches can be used for a simple system which is rarely changed. Model-based systems present an additional system model in relation to rules, which make they superior to the pure rule-based systems but does not make them more attractive due to the difficulty of obtaining and update the model. Although case-based systems are less sensitive to changes in network, they are not suitable for handling real-time alarm correlation. In addition to their own problems, neural networks and decision trees both rely on a long training period and may not work outside the area of training.

Codebook technique is interesting due to its performance and robustness. However, it is required a way to handle the changes of networks. Moreover, it may not work when more than one fault occur with overlapping sets of symptoms.

Context-free grammar is attractive for its ability to model hierarchically system. However, all available algorithms applicable to model constructed by context-free grammar are too complicated to be used in real systems.

Although model traversing techniques are resilient to frequent network configuration changes, they can not model the situations in which failure of an object may depend on a logical combination of other object failures.

After introducing the fundamentals of fault diagnosis, the next chapter aims at describing the Dimetra, which is the subject network in this thesis.

CHAPTER 3

Analysis of Dimetra

A good understanding of domain is critical before starting to find solution. This chapter introduces a basic and simplified Dimetra system and presents the whole system diagram. Fundamental components as well as the dependencies between them are described. Moreover, alarms of those components are analyzed in order to identify the faults associated with those components. Finally, a fault propagation model for a sample system is presented according to the dependencies in that system and the alarm analysis for its components. This chapter is primarily based on [20, 21, 22].

3.1 System Introduction

Dimetra [20] is the abbreviation for **DI**gital **M**otorola **E**nhanced **T**runked **RA**dio. Motorola Dimetra system is a sophisticated range of digital radio equipments that deliver the full benefits of the TETRA standard [1]. It is designed to meet the needs of the users of both Private Mobile Radio networks and Pub-

[1]TETRA is a specialist Professional Mobile Radio and two-way transceiver (colloquially known as a walkie talkie), the use of which is restricted to use by government agencies, and specifically emergency services, such as police forces, fire departments, ambulance services and the military. More information can be found at [19]

lic Access Mobile Radio systems. The voice service that Dimetra offers allows people to call each other within the same organization.

A Dimetra system can be organized in three levels. From the top down, they are *system-*, *zone-*, and *site-level*. In the system-level, a Dimetra system consists of one or multiple zones. Each zone comprises of multiple BTS sites[2], and a master site[3] as a central control point for all intra-BTS sites. In the site-level, a BTS site and a master site further contain their specific lower-level components. Refer to the project scope introduced in section 1.3, only a basic and simplified Dimetra is interesting to this project. More specifically, a basic and simplified system could be the one consists of one single zone and only support voice operation. The following sections will describe fundamental components in such a basic and simplified system, including mobile station, radio channels, BTS site and master site, as well as some important low-level components inside BTS site or master site.

3.2 Mobile Station (MS)

The mobile station is a two-way voice communications device which provides users the ability to make and receive calls. A mobile station is always registered with one BTS site in order to communicate with other mobile stations. Mobile stations communicate with BTS sites on some control channels, while a traffic channel is used for communications between mobile stations. Figure 3.1 [20] shows a sample mobile station in real life.

3.3 Radio Channels

There are two kinds of channels existing in Dimetra system. They are the *control channel* and the *traffic channel*.

3.3.1 Control Channel (CC)

The control channel is for mobile stations to send call requests to and receive traffic channel assignments from BTS sites. A mobile station always tunes to

[2] It will be introduced in section 3.4
[3] It will be introduced in section 3.5

Figure 3.1: MTH500 Mobile Station [20]

the control channel except when it is assigned to a call on a traffic channel. When a call is completed, the mobile stations involved in the call switch back to the active control channel.

3.3.2 Traffic Channel (TCH)

Opposed to the control channel, the traffic channel is used to transfer voice traffic between mobile stations. It is considered as the resource to make a call and managed by BTS site.

3.4 BTS Site

BTS is the acronym for **B**ase **T**ransceiver **S**ystem. It is a remote segment within the Dimetra IP system responsible for call processing and mobility services within a local geographical area. BTS has three subtypes: **EBTS**, **MBTS** and **MTS**. For instance, EBTS, an important type of BTS, stands for **E**nhanced **BTS**.

In a multiple site Dimetra, a group of BTS sites are connected to a particular

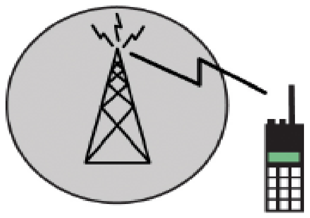


Figure 3.2: A BTS site with a mobile station [20]

master site via individual site links. Equipments at such master site, mainly the zone controller, coordinates the operation of those BTS sites so they can cooperate with each other to work in a wide area trunking mode. When BTS sites are in such mode, communication can be established between not only mobile stations registered with the same site, but also those registered with different BTS sites. Under certain conditions, e.g. zone controller is broken or site link is down, a BTS site can operate independently in site trunking mode, which means only services to mobile stations registered with that site are provided. Thus, mobile stations registered with that site can not communicate with those registered with other sites. Figure 3.2 [20] shows an example of BTS site.

A BTS site consists of one or more base radios, a site controller, etc. The next two subsections briefly describe base radio and site controller.

3.4.1 Base Radio (BR)

The base radio serves as a radio transmitter and receiver in a BTS site. Thus, base radios provide the control channel as well as the traffic channels to the BTS site containing them. A base radio is controlled by a site controller.

3.4.2 Site Controller (SC)

The site controller is an important component in BTS site. It controls resources within a BTS site, including assigning traffic channels to mobile stations and managing base radios.

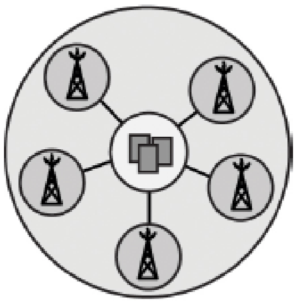


Figure 3.3: A zone consisting of a master site and five BTS sites [20]

3.5 Master Site

It is the central control point for the operation of a multiple site system (Zone). It is the site within a radio system that performs control, call processing, and network management functions. A master site connects to and manages multiple BTS sites, which forms a zone. Figure 3.3 [20] shows a sample zone.

Equipments at master site coordinate call processing, assignment of system wide area resources, and distribution of audio to all BTS sites in the system. It is at this site that the zone controller and the network management system are located. The following two sub-sections describe the core components at the master site.

3.5.1 Zone Controller (ZC)

Zone controller directs and controls most of the components in a zone, including coordinating the operation of the individual BTS sites; and is responsible for zone-level resource (radio channels) allocation.

3.5.2 Network Management System - FullVision Server

Network management system is composed of tools, commonly known as **FCAPS**, for fault, configuration, accounting performance and security management. The fault management function is the most interesting part since it is directly related to fault diagnosis.

FullVision server is the tool for monitoring system health and managing faults. Network operators can use it to monitor the status of components in the system, such as zone controllers, or BTS sites. As the primary troubleshooting tool, FullVision server allows network operators to view alarm information reported by network devices. More details about the use of FullVision can be found in [21, 22].

3.6 Site Link

Site Link is a wide area network (WAN) communication link that connects a Dimetra master site to a remote BTS site. Site links must be operational to support the control and audio traffic between the remote BTS sites and the master site.

3.7 System Diagram

The components described in this chapter do not cover all components in a Dimetra system. However, they are necessary and enough to give readers an idea how a basic and simplified Dimetra system can be constructed from those components. Such a Dimetra system can simply contains one single zone, which in turn consists of one master site and multiple BTS sites. BTS sites connect to the master site via individual site links. BTS sites and master sites are further composed of their own low-level components.

The system diagram in figure 3.4 shows a sample basic Dimetra which consists of one master site and two BTS sites. Components in low level are shown as well as those in high level. As described in sub-section 3.4.1 and 3.4.2, a site controller controls base radios. Accordingly this diagram uses a dashed line to represent the control path between the site controller and the base radio.

3.8 Alarm Analysis

Refer to section 2.1, alarms are notifications of the occurrences of events, e.g faults. An alarm displayed in FullVision provides valuable information, e.g. current state of the source object and a meaningful message, to indicate the problem behind that alarm. The format and content of one alarm log follows

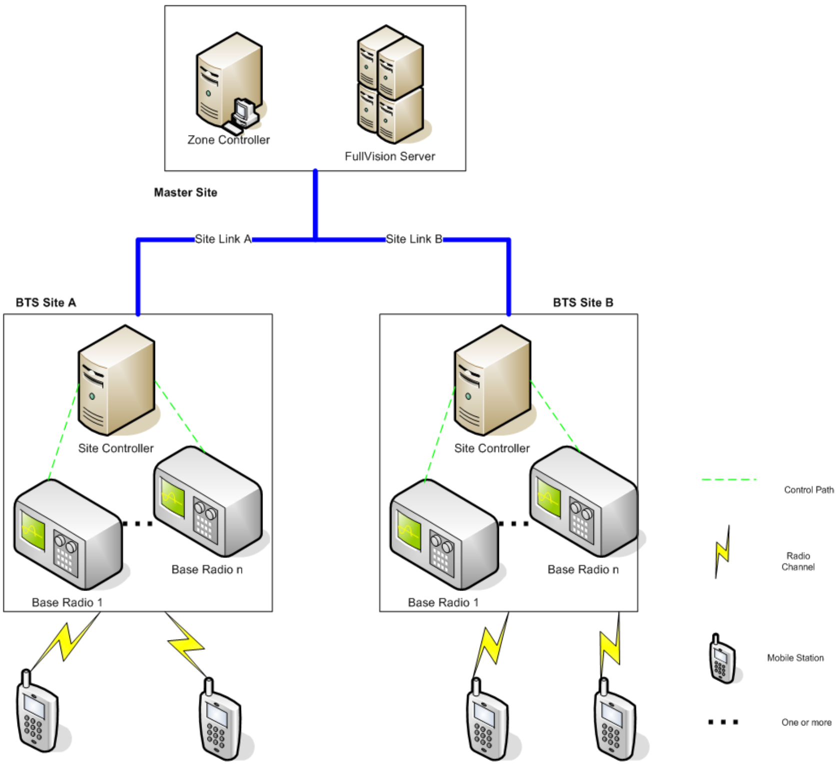


Figure 3.4: System Diagram for a basic Dimetra

```
132a7b76-9590-71db-0ba2-0a0ce90a0000, 1167213196, 62,
EbtsBaseRadio_1.1:zone11, 0, EbtsBaseRadio_1.1:zone11: ....
(3) DISABLED       (3004) LOCKED  Wed Dec 27 10:55:07.210 CET 2006,
5, 1.3.6.1.4.1.11.2.17.1.0.58916872, 864, SNMPv1-event,
.1.3.6.1.4.1.11.2.17.1.0.58916872, 10.12.233.10, 0, OV_Message,
8175, 0.0.0.0, IP, 2006-12-27 10:53:16, 6
```

Figure 3.5: Sample alarm log

some pre-defined mechanisms. Hence, it is necessary to understand Dimetra-specific alarms prior to using them during the process of diagnosis.

Each Dimetra alarm can be viewed as a 19-*tuple*, $a = (attr_1, attr_2, ...attr_{19})$. Every $attr_i(0 < i < 20,\ i\ is\ interger)$ corresponds to a property. The most important properties are *nodename* and *message*, which show the source object of this alarm and the indication of possible cause separately. Details about other properties can be found in the chapter 2 of [21]. Each alarm log is comma separated. A sample alarm log is given in Fig 3.5, where the fourth and sixth fields correspond to *nodename* and *message* properties respectively. These two properties tell that this alarm was reported by a base radio *EbtsBaseRadio_1.1:zone11* which was disabled due to a lock operation.

The value of the *message* property for a specific object is generated based on a template, which is comprised of the general information as well as the specific information. The specific information is, e.g., the name of the source object, while the general one is the information regarding the state and cause for a class of Dimetra objects. Chapter 4 of [21] describes the templates for alarm messages[4] associated with Dimetra objects. By analyzing those alarm message templates, mappings between alarms and faults can be built and possible faults associated with each object can also be identified. Furthermore, a fault propagation model can be constructed based on the alarm analysis.

According to [21], an alarm message template for a particular class of objects can be viewed as a 4-*tuple* (*State Number* , *State Text, Cause Number*, *Cause Text*), when the specific information is not taken into consideration. Moreover, templates associated with the same class of objects can be identified only by a pair of *(State Number, Cause Number).* Thus, for the sake of simplicity, such a pair is used to represent an alarm message template when it is only distinguished with other templates that associated with the same class of objects. For instance, if only templates for base radio are considered:

[4]The alarm message refers to the message property of an alarm.

```
(3, 3004) is equivalent to "(3) DISABLED      (3004) LOCKED"
```

where state number is 3, cause number is 3004, state text is *DISABLED* and cause text is *LOCKED*.

The following sub-sections interpret alarm message templates associated with EBTS base radio, EBTS site[5], EBTS (ZC)[6], zone controller and ZC site control path[7]. These interpertations reval that an object can report alarms due to some internal or external problems. Internal problems are considered as faults which originate within this object, while external problems occur in other objects and cause this object to report certain alarms. Note that this alarm analysis is primarily based on the description in the chapter 4 of [21]. **Therefore, it may be not completely applicable to a real Dimetra system due to possible customized configuration.**

3.8.1 Alarms of Base Radio

Table 3.1 lists the problems and their corresponding alarm message templates associated with base radio. There are four internal problems i_1, i_2, i_3 and i_4 and two external problems e_1 and e_2.

3.8.2 Alarms of EBTS

By analyzing alarms of EBTS, the author found that EBTS does not have any internal problems which originate from EBTS and all alarms reported by EBTS only indicate the problems of other objects. This can be explained by the the fact that EBTS is considered as a logical container object and thereby does not have any possible internal errors. It also illustrates how fault propagates along related components. For instance, if there is any fault in base radio, which provides radio channels to EBTS site, EBTS will get affected and report alarm messages look like $(31, 31002)$ or $(31, 31003)$ or $(31, 31004)$, or any two or three of these alarm messages.

Table 3.2 lists alarm message templates of EBTS in the *state/cause* column as well as the corresponding problems.

[5]Refer to section 3.4, EBTS site is a sub-type of BTS site

[6]ZC's view of the EBTS Site, a logic object

[7]A part of site link

Problem	**State/Cause**
i_1. Base Radio is not responding	(1,1022)
i_2. A Base Radio failure occurred	(3,3005) (3,3007) (3,3008)
i_3. Encryption subsystem has been failed	(3,3021) (13,13021)
i4. Base Radio 1 has been failed Base Radio 2 has been failed . . . Base Radio 8 has been failed	(7,7014) (8,8015) . . . (14,1423)
e_1. The states of all other EBTS components are abnormal	(3,3004)
e_2. The Base Radio's control link to Site Controller has been failed	(3,3006)

Table 3.1: Alarms analysis of EBTS Base Radio

Problem	**State/Cause**
e_1. Base Radio(s) has been failed	(31,31002), (31,31003), (31,31004)
e_2. The voice link to the EBTS has been failed	(31,31003)
e_3. Link between this site and the master site is down	(51,51003), (51,51005), (61,61005)

Table 3.2: Alarms analysis of EBTS

Problem	State/Cause
e_1. EBTS site is not wide trunking due to no voice channel	(101,101004)
e_2. EBTS site is not wide trunking due to no control channel	(101,101005)
e_3. EBTS site is not wide trunking because site control path is down	(101,101006)

Table 3.3: Alarms analysis of EBTS Site (ZC)

Problem	State/Cause
Switch has been failed	(3,3002), (5,5002)
Ethernet card has been failed	(3,3004), (5,5004)
Hard disk has been failed	(3,3006)
Power supply has been failed	(3,3007)
Zone is mis-configured	(5,5008)

Table 3.4: Alarms analysis of ZC

3.8.3 EBTS Site (ZC)

It is a logic object, which shows the zone controller's view of EBTS site. It is considered as the manager of EBTS site, which monitoring the state of EBTS site.

Table 3.3 shows the analysis of alarm messages of EBTS site (ZC).

3.8.4 Alarms of Zone Controller

As EBTS, zone controller does not have any internal problems because it is considered as a logical container. All alarms reported by Zone Controller can be used to find problems of other components.

Table 3.4 shows the analysis of alarm messages of zone controller.

Problem	State/Cause
Connection is down	(1,1003)
The preferred link is down	(3,3006)

Table 3.5: Alarms analysis of ZC Site Control Path

3.8.5 Alarms of ZC Site Control Path

ZC site control path is the control path from zone controller to EBTS site. It can be viewed as a part of site link.

Table 3.5 lists alarms of this object and the problems which cause those alarms.

3.9 Fault Propagation Model

An important point drawn after the analysis of alarms is: faults can propagate along related objects. A fault propagation model can be used to illustrate this point. This model can be built based on the alarm analysis and dependencies between objects described in previous sections. As noted in section 3.8, alarm analysis may be not fully reflected things in a real Dimetra system. Hence, it is possible that the corresponding fault propagation model is not completely precise. However, this model could be refined with the help of domain experts. This section will give a sample Dimetra system as well as its fault propagation model.

For the sake of simplicity, this sample system contains one base radio, one EBTS site, one EBTS site (ZC), one zone controller and one control path between EBTS and zone controller. A dependency graph depicted in figure 3.6 is used to represent the fault propagation model according to the introduction in section 2.2. Table 3.6 interprets the meaning of each dependency edge in Fig. 3.6.

Edge	Meaning
Base Radio to EBTS	When base radio is faulty, EBTS will get affected and report message like (31,31002), (31,31003), (31,31004)
EBTS to EBTS (ZC)	When EBTS is faulty, EBTS (ZC) can detect its abnormal state. Alarms (101,101004), (101,101005), (101,101006) may be reproted according to the actual state of EBTS
EBTS to ZC Site Control Path	When EBTS is disabled, ZC Site Control Path is broken so alarms (1,1003) or (3,3006) will be reported
ZC Site Control Path to EBTS	When Site Control Path is down, EBTS can not work in wide area trunking mode. Thus, alarms (51,51005)or (61,61005) will be observed
Zone Controller to EBTS	When Zone Controller is disabled, EBTS can not work in wide area trunking mode since the control path is down. As a result, alarms (51,51003), (51,51005), (61,61005) may be observed
Zone Controller to ZC Site Control Path	When Zone Controller is disabled, ZC Site Control Path is down as a result. Hence, alarms (1,1003) or (3,3006) may be reported

Table 3.6: Interprets of Dependency Graph in figure 3.6

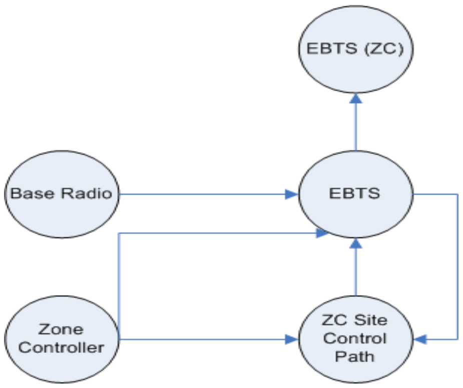


Figure 3.6: Dependency Graph of a sample system described in section 3.9. See the interpretation in Table 3.6

3.10 Summary

This chapter introduced a basic Dimetra system. Fundamental components are described and a system diagram is presents to show how those components cooperate to form a basic Dimetra system. The alarm analysis is very useful and important. It tells the way to read the informant information contained in an alarm. Moreover, it reveals the faults could occur in the Dimetra system and contributes to build the fault propagation model.

Chapter 4

A Framework for Fault Diagnosis in Dimetra

Recall in section 2.1, the event correlation is introduced as the most popular technique used in fault diagnosis. This chapter proposes a framework which is based on the idea of event correlation and combines the rule-based and the model-based approaches. Although this framework is proposed for constructing a fault diagnosis system for Motorola's Dimetra system, it is generic enough for other networking systems.

The former part of this chapter reviews some related solutions and gives a short comparison among those solutions. This comparison is the basis for choosing reasonable solutions that can be used in this thesis. Next, the proposed framework is presented with its three components. Finally, some final considerations are given in the section of summary.

4.1 Review of Related Solutions

Various solutions for fault diagnosis have been described in chapter 2. But no one is the best to solve generic problems in fault diagnosis refer to the comparison in section 2.5.

Codebook solution is very interesting in terms of running time. However, the precision is not predictable when more than one problem occur with overlapping sets of symptoms. Furthermore, the codebook is not independent on actual network configuration.

Context-free grammar solution can represented system model in a structured way. Moreover, the fault localization algorithms that it applies are not subject to lost and spurious alarms. However, these algorithms are too complicated to be used in the real application.

Diagnostic knowledge is naturally represented as rules. But a pure rule-based system has many disadvantages since it relies only on surface knowledge. Model-based solution can address some of issues in rule-based solution due to the use of a system model.

Case-based solution is resilient to system changes and has the ability to learn. However, it is unable to be used in the real-time alarm correlation. In addition to its own limitation, some practical things make it impossible to be a candidate solution. Recall that this solution relies on a "CaseBase" which can not be easily accessed by the author due to some confidential reasons. On the other hand, there is another team already working on this solution in Motorola. It is not reasonable to choose the same solution.

Other solutions such as decision trees or neural networks are not considered because they all require a large amount of training data which are difficult to be generated.

Model traversing techniques are not thoroughly researched in this thesis due to its limitation to model all failure situations as well as the limited time of this thesis.

In all, the combination of the rule-based and model-based solutions may be the best option for this thesis since the researched Dimetra system is quite small and simple.

4.2 The Proposed Framework

The framework proposed in this thesis is based on the idea of event correlation. It adapts from a similar framework proposed in [6] and utilizes the concept [3] of using composite events in the event correlation. This framework combines both the model-based solution and the rule-based solution. Although this frame-

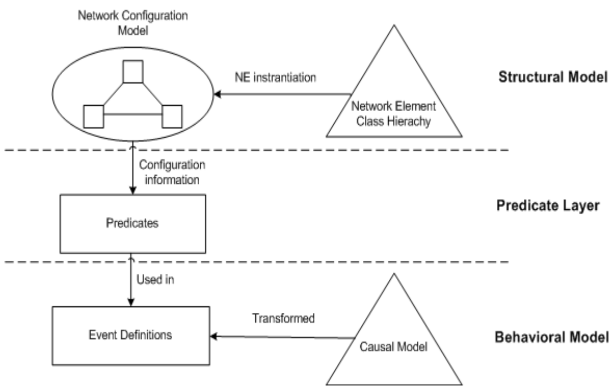


Figure 4.1: The Proposed Framework

work is proposed for Dimetra system, it is generic enough to be used for other networking systems.

This framework as shown in Fig. 4.1 contains three components: *structural* and *behavioral* models plus a *predicate layer*.

The structural model describes the managed network. It contains two parts: the *network element class hierarchy* and the *network configuration model*. The network element class hierarchy organizes classes of actual network elements in an object-oriented fashion. The network configuration model stores the information about a specific network, including the relationships (*management*, *containment* and *connectivity*) between network elements. Network elements in the network configuration model are instances of classes in the network element class hierarchy. Hence, a network configuration model is considered to be instantiated from a network element class hierarchy.

In opposition to the structural model, the behavioral model describes the dynamics of event correlation. It includes a *causal model* and a number of *event definitions*. The causal model represented as a causality graph models a set of fault propagation scenarios by associating events occurring in the system. According to the causal model, developers can identify a list of events and create their definitions which are used during the process of event correlation.

The predicate layer provides a number of *predicates* that associate the behavioral model with the structural model. Predicates are used in event definitions to retrieve configuration information from the structural model.

The following sub-sections will describe these three components in more details.

4.2.1 Network Element Class Hierarchy

This network element class hierarchy is based on variations around the *Netmate model* described in [11, 16, 17]. It uses an object-oriented paradigm to represent classes. Classes in this hierarchy describe network element types, such as links, servers, internetworking devices, etc. A class defines properties that owned by all network elements which are instances of that class. For instance, every NE (network element)[1] has a name, which can be defined as a property in its corresponding class. Moreover, a class can define a set of common properties whose values are shared by all instances of that class. Those properties, like *class variable*[2] in object-oriented paradigm, are named *class properties*. This class hierarchy emphasizes *relationship* properties, which represent the *containment*, *management* and *connectivity* dependencies between NEs. Note that relationships can be one-to-one, one-to-many, or many-to-many, and each relationship has an inverse.

Classes are organized into an inheritance hierarchy. It allows subclasses to inherit property definitions from their superclasses. In addition, inheritance brings more flexibility since different NEs that have a common superclass can be treated generically when their specific details can be ignored.

This class hierarchy is depicted in figure 4.2. The root of this hierarchy is the most generic class *Element*. It has the *name* property, whose value represents the name of a particular NE. There are two classes: *Manager* and *ManagedObject* in the next level. The dashed line between them represents a management dependency. That is, instances of the *Manager* class manage instances of the *ManagedObject* class, and vice versa, there is a *managedBy* relationship from instances of *ManagedObject* class to instances of *Manager* class. A management dependency can be recorded by two properties *ManagedObject.managers* and *Manager.managedObjects*. The first one is used for a *ManagedObject* instance **MO** to keep track of all *Manager* instances which are managing **MO**. The other one is used for a *Manager* instance **M** to keep track of all *ManagedObject* instances which are managed by **M** at the moment. Similar to the management dependency, the containment dependency[3] can be recorded by properties *ManagedObject.components* and *ManagedObject.containers*, which store component elements for a container element, and container elements for a component element, respectively. Class *ManagedObject* can be further divided into class *Link* and class *Node*. A connectivity dependency is represented as a dashed line

[1]For the sake of simplicity, NE(s) is used to replace network element(s) when there is no misleading.

[2]It has a value that is shared by all instances of a class

[3]Containment dependency, as well as connectivity dependency are defined for managed object only.

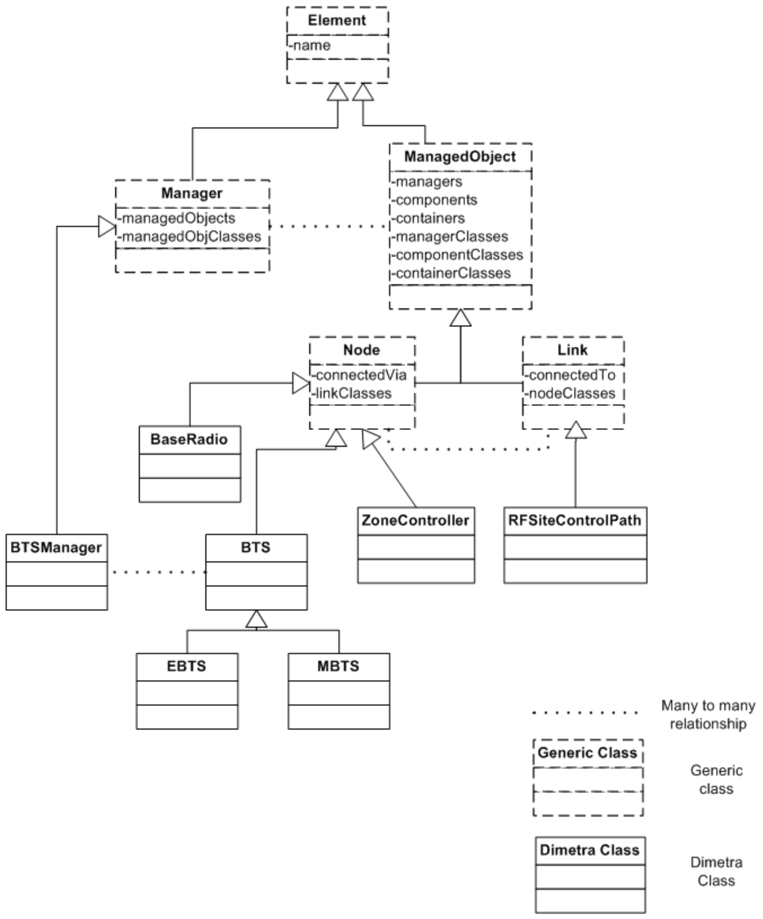


Figure 4.2: Network Element Class Hierarchy (the description of Dimetra classes is shown in table 4.1)

Class	Description	Constraint
RFSiteControlPath	Control path between BTS and Zone Controller	Connects to *BTS* and *ZoneController*
ZoneController	Zone Controller device	Connected via *RFSiteControlPath*
BTS	Base Transceiver System	Contains *BaseRadio*; connected via *RFSiteControlPath*; managed by *BTSManager*
EBTS	Enhanced Base Transceiver System	Inherit constraints from *BTS*
MBTS	Mini Base Transceiver System	Inherit constraints from *BTS*
BaseRadio	Base Radio device	Contained in *BTS*
BTSManager	A logical element reports the status of BTS from zone controller's point of view (see sec. 3.8.3)	Manages *BTS*

Table 4.1: Description of Dimetra classes

between these two classes. As management dependency, connectivity information such as a particular node is connected via a particular link can be separately stored in properties *Node.connectedVia* and *Link.connectedTo*. Classes introduced so far are generic enough to be re-used in other network systems in addition to Dimetra, so called *generic classes*.

The rest of classes are specific to Dimetra system, which are called *Dimetra classes*. Dimetra classes inherit property definitions from generic classes and possess their specific values. Some inherited properties: *managedObjClasses*, *managerClasses*, *containerClasses*, *componentClasses*, *linkClasses* and *nodeClasses* are class properties. They are used as relationship constraints for Dimetra classes, which help developers construct a consistent and complete structural model. For example, instances of *BTS* class represent BTS sites. According to the fact that a BTS site consists of several base radios (see section 3.4), the property *componentClasses* in the *BTS* class shall contain at least one value, class *BaseRadio*. The value of this property is shared by all *BTS* instances to guarantee each instance contains the right NEs as components. Table 4.1 gives the description of each Dimetra class as well as its constraits. Instead of having a single class *BTS* to represent all kinds of BTS sites, two classes *EBTS* and *MBTS* are added as its subclasses. It is because more specific classes are required in some cases.

This class hierarchy is scalable. New classes may be easily added into by inheriting existing classes. Furthermore, it can be easily used to model NE classes for other domains. The only modification is to remove the Dimetra classes and add new specific classes which shall inherit the generic classes already defined in the class hierarchy.

4.2.2 Network Configuration Model

The network configuration model describes a particular network system. It stores network configuration information, which is information about all NEs in the system as well as the relationship between them. NEs in this model are all instances of classes defined in the network element class hierarchy. Furthermore, the constraints of each class are enforced during the construction of this model. For instance, developers can not build a model which specifies a base radio to contain one particular EBTS site. The network configuration model is viewed to be instantiated from the network element class hierarchy

It is very easy and intuitive to represent the configuration model as a graph, whose nodes correspond to NEs and edges correspond to relationships. However, this sort of model is only understanded by human beings. In order to make a machine-understandable model, some sort of modelling languages can be used. In this thesis, XML[4] is selected to describe the model because:

- It is a standard language to describe machine-readable information.
- As a general-purpose language, the information described by XML can be shared by different systems.
- It has a standard way and lots of utility to parse information from an XML file.
- With XML, the author saved time from designing a new modelling language and the algorithm for compiling it, which is beyond the scope of this project.

A sample system is given to demonstrate how to construct a configuration model based on XML. For simplicity, this system contains elements: a zone controller (**ZC**), a EBTS site (**ebts01**) and its component: a base radio (**ebts01_br01**),

[4]eXtended Markup Language, a general-purpose markup language [23].

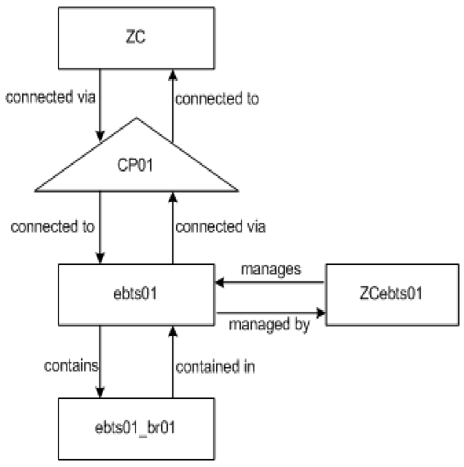


Figure 4.3: Graphic model of the sample system

the site control path (**CP01**) between **ZC** and **ebts01**, and a logical element[5] (**ZCebts01**) monitoring **ebts01**. The graphic model for such system is depicted in Fig. 4.3. The corresponding XML representation is given in Fig. 4.4.

The first section of this XML document is called *NE declaration section*, in which a list of NEs are listed as: `<Element name = '...' class = '...'/>`. The `name` attribute specifies the name of a NE and the `class` attribute specifies the class of that NE. All classes in that XML document refer to the classes defined in Fig. 4.2. The following sections specify the management, containment and connectivity dependencies respectively. The values of the `name` attribute in these three sections all refer to the names defined in the first section. Take the *containment dependency section* in Fig. 4.4 as an example. Only one `<Container>` tag is specified, which means there is only one containment dependency. It shows that there is an element named *ebts01*, which is an instance of *EBTS* defined before. This element contains one component which is specified in a `<Component>` tag. This tag tells that this component is called *ebts01_br01*, which is an instance of *BaseRadio* defined before. Detailed description of each element tag or attribute tag can be found in appendix C, the section C.1.

By constructing an XML document as above, a configuration model can be parsed and saved into a knowledge base.

[5]It is the EBTS (ZC) object described in section 3.8.3

```
<?xml version="1.0" encoding="UTF-8"?>
<NetworkConfig>
<!-- NE declaration section -->
  <!-- specify a base radio -->
  <Element name='ebts01_br01' class='BaseRadio'/>

  <!-- specify a EBTS instance-->
  <Element name='ebts01' class='EBTS'/>

  <!-- specify a zone controller instance-->
  <Element name='ZC' class='ZoneController'/>

  <!--  specify a EBTS (ZC) instance-->
  <Element name='ZCebts01' class='BTSManager'/>

  <!--  specify a site control path instance-->
  <Element name='CP' class='RFSiteControlPath'/>

<!-- Containment dependency section-->
  <Containment >
    <Container name='ebts01'>
       <!-- 'ebts01' contains 'ebts01_br01'-->
       <Component name='ebts01_br01'/>
    </Container>
  </Containment>

<!-- management dependency section-->
  <Management>
    <!-- 'ZCebts01' is a manger of 'ebts01'-->
    <Manager name='ZCebts01'>
      <Managed name='ebts01'/>
    </Manager>
  </Management>

<!-- Connectivity dependency section-->
  <Connectivity>
    <Link name= 'CP'>
    <!-- 'CP' has two end-points -->
      <Point name='ebts01'/>
      <Point name='ZC'/>
    </Link>
  </Connectivity>

</NetworkConfig>
```

Figure 4.4: XML-formatted model of the sample system

4.2.3 Predicate Layer

The predicate layer works as a bridge between the structural model and the behavioral model. By using predicates, events defined in the behavioral model can associate with current network configuration without hard-coding any information. It helps developers define a more generic and flexible behavioral model. Currently, the functionality of predicates can be grouped into the following categories:

1. Determining the type of one network element. E.g., *isBaseRadio(name:String): boolean* returns *true* if the element identified by *name* parameter, is a base radio; otherwise it returns *false*.
2. Evaluating the management relationship between two network elements. E.g. *isManagedBy(a:String,b:String):boolean* evaluates if a is managed by b or not.
3. Evaluating the containment relationship between two network elements. E.g. *isContainedIn(a:String,b:String):boolean* evaluates if a is contained in b or not.
4. Evaluating the connectivity relationship between two network elements. E.g. *isConnectedTo(a:String,b:String):boolean* evaluates if a is connected to b or not.

4.2.4 Causal Model

Causal model is the primary component in the behavioral model since it is the basis for building event definitions. With the help of the causal model, developers can easily identify events and specify them.

Causal model takes the form of causality graph. Refer to section 2.2, a causality graph's nodes correspond to events and its edges describe cause-effect relationships between events. The nodes corresponding to observable events (alarms) can be easily identified as primitive events. On the other hand, developers can also identify composite events from the causal model. In this model, events that have a common root cause are connected as a *path* or *tree*. Thus, those events can be correlated and generalized into a high-level event by utilizing the concept of composite event. For example, there is an edge (e_i, e_j) in the causal model, which represents a very simple fault propagation scenario. Since event e_i and event e_j are causally related, a conceptual composite event e can be identified to correlate events e_i and e_j.

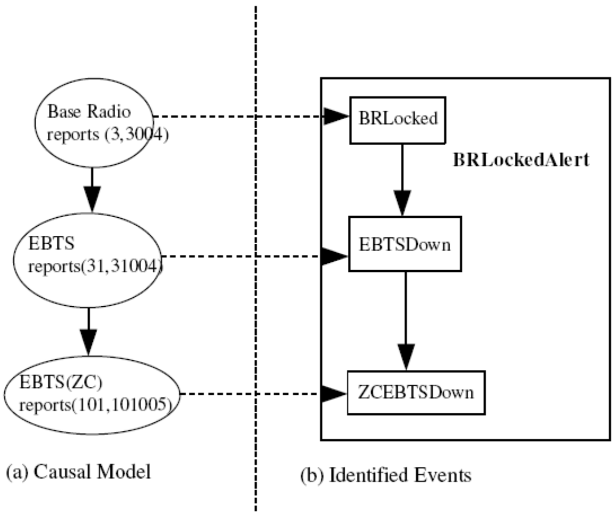


Figure 4.5: Causal Model and Identified Events

In addition to help events identification, rough event definitions for composite events can be derived from the causal model as well. Consider an edge (e_i, e_j) in a causal model. It is high likely that the cause event e_i occurred before the occurrence of the effect event e_j. This derived temporal relationship is very important since a temporal relationship can reveal important diagnostic information about event relationships [3]. Hence, a higher level event e correlating events e_i and e_j can be specified based on that temporal relationship as: event e is considered to occur whenever event e_i is followed by event e_j. Refer to section 2.1, the real temporal relationship may not as expected as the one derived from the causal relationship. Therefore, the derived rough event definition should be reviewed by experienced domain experts and refined during the evaluation phase.

Besides composite events, primitive events can be defined according to the properties of alarms, e.g. the *nodename* and *message* properties.

An example is given to illustrate the concepts described so far. Figure 4.5(a) depicts a sample causal model, which describes a fault propagation scenario existing in the sample system introduced in section 4.2.2. This scenario is described as "When the base radio (*ebts01_br01*) is locked, alarm (3,3004)[6] will be reported. The EBTS (*ebts01*) will then get affected and send alarm (31,31004)[7] due to the containment relationship between these two elements. Consequently,

[6]See section 3.8.1 about alarm (3,3004)
[7]See section 3.8.2 about alarm (31,31004)

the manager of that EBTS (*ZCebts01*) detects the error state of *ebts01* and sends alarm (101,101005)[8]." All nodes in Fig. 4.5(a) correspond to observable events, and therefore three primitive events: **BRLocked**, **EBTSDown** and **ZCEBTSDown** are identified according to nodes, which capture alarms (3,3004), (31,31004) and (101,101005) respectively. Furthermore, since these three primitive events are causally related, a composite event **BRLockedAlert** correlating these three primitive events can be identified. Identified events are shown in Fig.4.5(b). The arrows in Fig.4.5(b) represent the possible temporal relationships between these primitive events by mapping arrows in Fig.4.5(a) representing causal relationships.

4.2.5 Event Definitions

After identifying all events in Fig. 4.5(b), an event specification language is required to specify those events. Moreover, an event-detect engine is required to detect occurrences of events based on their definitions. It would be a huge workload to design such a language and an event-detect engine. Due to the limited time, it is reasonable to use **Esper** [24], an open-source event stream processing and event correlation engine, which enables applications process large volumes of incoming messages or events in real-time. Esper allows developers to use a SQL[9]-style event query language - EQL as well as a pattern language to specify events. Besides, it offers an engine for detecting events.

The rest of this section will illustrate how to use EQL and Esper pattern to specify events in Fig. 4.5(b). More details about EQL and Esper pattern, e.g the syntax and built-in operators are not introduced this report but can be found in [24].

As a primitive event, **BRLocked** which indicates alarm (3,3004) has been reported from the base radio *ebts01_br01*, can be captured according to the *nodename* property and the *message* property from alarm streams. Hence, it is specified as below:

```
insert into BRLocked
select * from AlarmLog
where message like '%(3)%DISABLED%(3004)%LOCKED%'
      and nodename='ebts01_br01'
```

[8]See section 3.8.3 about alarm (101,101005)

[9]Structured Query Language, a language to create, retrieve, update and delete data from database systems [25].

In the above event definition, **AlarmLog** is the alias of the underlying alarm stream. This event definition is read as: Only alarms, which are reported by *ebts01_br01* and have the alarm message matching the common template for the alarm message (3,3004), are considered as occurrences of the **BRLocked** event. If readers are familiar with SQL, it is easier to understand this definition: select alarms (3,3004) from an alarm stream, and insert them into an event stream **BRLocked** which is a collection of BRLocked instances.

In order to make **BRLocked** event more general, not only associated with base radio *ebts01_br01*, predicate *isBaseRadio(name: String)* can be used. This predicate will check current configuration model to evaluate that if one alarm is reported from a base radio or not. Hence, a more general **BRLocked** event is defined as:

```
insert into BRLocked
select * from AlarmLog
where message like '%(3)%DISABLED%(3004)%LOCKED%'
and isBaseRadio(nodename)
```

Similarly, event **EBTSDown** indicating alarm (31,31004) reported by EBTS and **ZCEBTSDown** indicating alarm(101,101005) reported by EBTS (ZC) are defined respectively as:

```
insert into EBTSDown
select * from AlarmLog
where isEBTS(nodename) and
message like '%(31)%NO TRUNKING%(31004)%NO CONTROL CHANNEL%'

insert into ZCEBTSDown
select * from AlarmLog
where isBTSManager(nodename) and
message like '%(101)%NOT WIDE TRUNKING%(101005)%NO CONTROL CHANNEL%'
```

Note that *isBTSManager(name:String)* and *isEBTS(name:String)* are two predicates to evaluate if alarms are reported by EBTS(ZC) component or EBTS site, respectively.

Events specified so far are all primitive events. A more complex event, composite event, can be specified based on these three primitive events to capture a base-radio-locked scenario. According to Fig. 4.5(b), we can see that **BRLockedAlert** event is considered to occur whenever event **BRLocked** is followed by event **EBTSDown**, which is then followed by event **ZCEBTS-Down** in turn. However, this expected temporal order is not preserved in the real alarm trace. Due to its high priority, alarm (101,101005) corresponding to event **ZCEBTSDown** is actually received earlier than alarm (31,31004) corresponding to event **EBTSDown**. In order to capture that scenario, a composite event **BRLockedAlert** is defined as:

```
insert into BRLockedAlert
select A.nodename, A.message, A.event_time from
pattern [every (A=BRLocked -> B=ZCEBTSDown -> C=EBTSDown
         where timer:within(30 sec) )]
where isContainedIn(A.nodename,C.nodename)
 and isManagedBy(C.nodename, B.nodename)
```

Predicate *isContainedIn(a:String,b:String)* determines if base radio a is contained in EBTS b or not. Similarly, isManagedBy(a:String,b:String) determines if EBTS a is managed by EBTS(ZC) b or not.

The code segment *A=BRLocked − > B=ZCEBTSDown − > C=EBTSDown where timer:within(30 sec)* is critical. It refers three primitive events defined above and assigns alias for individual events for the sake of simplicity. EQL operator "− >" represents a *followed-by* relationship between operants, and thus be used to specify the temporal relationship among these three events. Additionally, these three events are correlated only if they occurred within 30 seconds since a base radio was locked according to some diagnostic experience. Hence, the timing condition (30 sec) limits event detector only to match any 3 primitive events that happen 30 seconds within each other. As a result, wrong correlation of independent alarms is eliminated.

4.3 Summary

This chapter presented a framework which is used to construct a fault diagnosis system for Motorola's Dimetra system. However, this framework is generic enough to be used in other network systems. It is because that the ideas including the use of predicates and composite events as well as the derivation of event definitions from a causal model are universal for all domains. In addition, the only Dimetra specific associated with this framework (the Dimetra classes in the network element class hierarchy) can be easily replaced since this hierarchy is constructed in an object-oriented way.

This framework combines the rule-based and the model-based solutions. Thus, the author believes that systems implementing this framework are superior to pure rule-based systems. The next chapter will start to design a system implementing this framework.

Chapter 5

Design of the SECTOR system

This chapter designs a **S**imple **E**vent **C**orrela**TOR** (SECTOR) system, which is based on the framework proposed in the previous chapter. Moreover, this chapter describes the whole system architecture of SECTOR in great details, as well as the communication between the different components in the system.

5.1 System Overview

The SECTOR system implementing the framework presented in the previous chapter is specifically developed for Motorola, Denmark to handle the fault diagnosis in their Dimetra system. SECTOR is implemented in Java language [26]. More details regarding to its implementation is introduced in the next chapter. This chapter will focus on its architecture.

Recall the proposed framework from the previous chapter, SECTOR should have the following major functionalities:

1. Providing an environment for developing network element class hierarchy.

2. Constructing the network configuration model
3. Providing an environment for developing predicate layer.
4. Providing an environment for developing event definitions.
5. Performing event correlation.

The components in SECTOR could be divided into two major parts: the *development environment* and the *run-time environment*. The development environment consists of components which perform functionalities 1, 2, 3 and 4 listed above. However, the event correlation is performed by components from the run-time environment.

5.2 System Architecture

The architecture of the SECTOR system is illustrated in figure 5.1. This figure presents the components in the system as well as how they interact with one another. All components are described as follows:

- **Network Element Class Editor**: It is used for developers to build network element class hierarchy. Thus, this editor is a development environment component. In order to provide a developer-friendly environment, this editor is designed to support modelling class hierarchy by utilizing UML[1]. Moreover, it is able to automatically generate source code for network element classes from a UML-based class hierarchy. Due to the limited time, this editor is not developed in this thesis. Currently, *Poseidon* [28], a UML modeling tool is used. Any UML modeling tool such as *IBM Rational Rose* [29] can be used as well.
- **Network Config Editor**: It is used for users to input the network configuration model in the form of XML. This editor belongs to the development environment. This editor can be any text editor. The output of this component is considered as a plain text version of the configuration model.
- **Molder**: This component parses configuration information from a XML document, and uses network element classes to construct a binary version of configuration model which can be stored in a knowledge base. It is a development environment component as well.

[1]Unified Modeling Language [27]

- **Configuration Base**: It stores a configuration model and allows predicates to query the configuration information that it stores. The stored configuration model is created during development time, and required or referenced during run time. Currently, all configuration models are stored in the memory. In the future, they can be stored in database systems.

- **Predicate Editor**: As a part of development environment, it is used to define predicates. The predicates in SECTOR are implemented as Java classes. Currently, *Eclipse* [30] is used as the predicate editor.

- **Predicate Base**: It is a set of Java classes stores in the file system. Predicates are created at development time. At run-time, each predicate will refer to a configuration model to test the existence of a relationship among network elements.

- **Event Spec Editor**: It is used for developers to input the event specifications and is a part of the development environment. Event specifications created at development time specify the events, which will be monitored by event correlation engine at runtime. Event definitions defined in EQL and Esper pattern are included in the event specification. Some specifications may refer to predicates. Event specifications are written in XML , more detail is given in appendix C.2. Thus, this editor can be any text editor.

- **Event Spec Base**: This component parses event specifications from a XML file and stores them in memory. In the future, database system can be used to store event specifications.

- **Event Registration Server**: This component registers event specifications defined in *Event Spec Base* to event correlation engine at run-time.

- **Event Adaptor**: It gets alarms from different alarm sources (real network or alarm log) and standardizes alarms into the format that event correlation engine can understand. All standardized alarms are sent to event correlation engine.

- **Event Correlation Engine**: This component performs the event correlation. The result of event correlation is the occurrences of composite events. Thus, it monitors the standardized alarms sent by Event Adaptor in order to capture the registered events. The detected event can be sent to relevant *Event Subscriber* or itself for further detecting of composite events. During the monitoring, this engine may invoke predicates to assert the occurrence of one event. Esper is used to work as event correlation engine in this project.

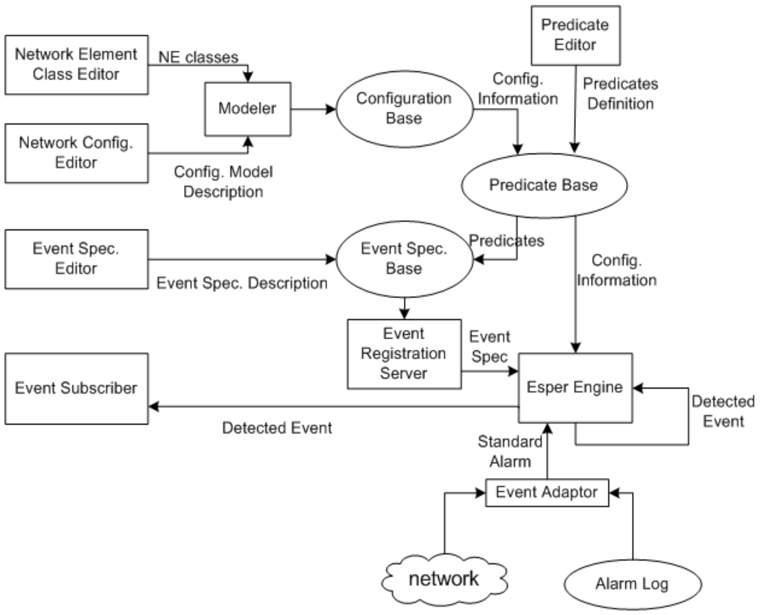


Figure 5.1: The Architecture of SECTOR

- **Event Subscriber**: It could be any component that executes when its subscribed event has occurred. For instance, *Event Display* which subscribes all detected events can show network operator the result of correlation. Furthermore, event subscriber could be another fault diagnosis system. In that case, SECTOR may work as event (alarm) filters to concise information which can be further used by other fault diagnosis system.

5.3 Summary

This chapter presented a design of a Simple Event CorrelaTOR (SECTOR) system. SECTOR system implements the framework described in the previous chapter. The whole system architecture of the SECTOR system was presented, and each component has been thoroughly analyzed and described.

Chapter 6

Implementation

The architecture of SECTOR system has been elaborately introduced in Chapter 5. This chapter will go into details with the implementation of SECTOR system, such as the use of design patterns, the design of class diagrams, etc. Furthermore, some interesting technical issues in the development of the system will be described. The implementation of SECTOR system uses the Java [26] language and relies on several third-party components.

6.1 Modular design

Due to the complexity of the whole SECTOR system, it is a good strategy to subdivide the system in smaller parts (modules) that are easily implemented. Moreover, one module can be easily replaced or updated without influencing the running of SECTOR .

The implementation consists of implementing seven different modules (Fig. 6.1):

- *network element classes module*: This module consists of the editor to specify the network element class hierarchy in UML as well as the Java classes generated from that UML class hierarchy. As introduced in previous chapter, *Poseidon* is used as such editor.

- *modeler module*: This module implements the model construction functionality. *Eclipse* is used as the *Network Config. Editor* since it can edit XML document.
- *predicate module*: It implements the predicate layer. Predicates are implemented as Java classes, and therefore *Eclipse* is used as the *Predicate Editor*.
- *event registration module*: The main functionality of this module is to register the specified event into the event correlation engine. *Eclipse* is used as the *Event Spec. Editor*
- *event adaptation module*: The main functionality of this module is to translate alarms into a standard format and then send them to the event correlation engine.
- *event correlation module*: This module contains an engine to perform event correlation. Esper is used to as the event correlation engine. Hence, this module is not discussed in this chapter.
- *event subscription module* This module is considered as the client application of SECTOR. It consists of applications subscribing the events monitored by event correlation module.

Each module is composed of Java classes. The following sections will show how each module is implemented by Java classes.

6.2 Design Patterns

Design patterns are used during the implementation since they are general repeatable solutions to commonly occurring problems in software design [31]. Basically, design patterns are well tested and in a high abstract level. The use of design patterns can help design a reliable, flexible and extendable SECTOR system. On the other hand, much time can be saved by taking advantage the existing design patterns.

The following subsections describe two design patterns mainly used in the implementation.

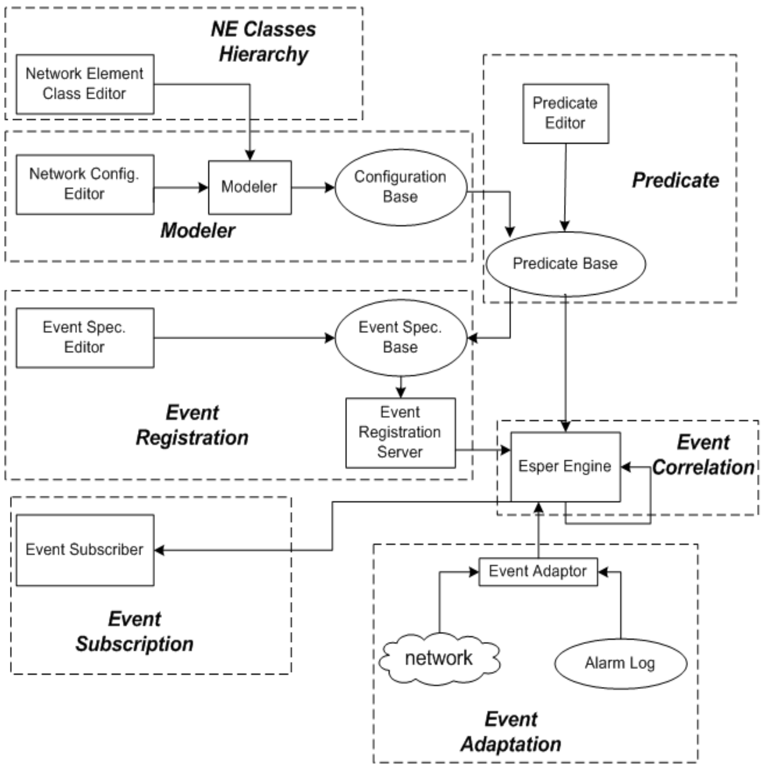


Figure 6.1: SECTOR system - Main modules

6.2.1 Strategy Pattern

This pattern is widely used in the classes design. The idea of **Strategy Pattern** [32] is to encapsulate the concept that varies to an interface, not an implementation. This pattern lets the algorithm vary independently from clients that use it. By using the **Strategy Pattern**, SECTOR can be built as a loosely coupled interchangeable parts. That loose coupling makes SECTOR much more extensible, maintainable, and reusable.

Java interface[1] is used to implement this pattern. Developers can create various implementations for one interface and the actual implementation which clients are accessing is transparent to clients. Thus, developers can freely change the underlying implementation in the future.

6.2.2 Observer Pattern

This pattern [33] (sometimes known as **Publish/Subscribe pattern**) defines a one-to-many dependency between objects so that when one object changes state, all of its dependents (observers) are notified and updated automatically. The **Observer Pattern** help loose coupling of observed objects and observers. So observers can keep their states synchronized without necessarily needing direct knowledge of their subjects (observed objects), facilitating system more reusable.

In SECTOR, `EventSubscriber` interface implements this pattern. Hence, every observer (called subscriber in SECTOR), which subscribes to a particular event should implement that interface. Once such event is detected, all its subscribers will get informed and actions defined in those subscribers will be executed.

6.3 Package Overview

In Java, all classes and interfaces can be grouped in packages. Each package [35] can be viewed as a module in which classes and interfaces are organized according to their functionality, usability as well as category they should belong to. The whole SECTOR consists of several Java packages. The overview of all packages is shown in Fig. 6.2. The core packages in SECTOR are:

[1] Java interface, an abstract type in Java [34]

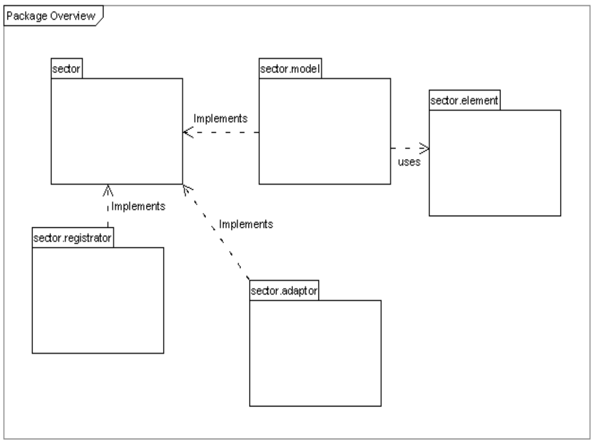


Figure 6.2: Packages Overview

- `sector` – The root and fundamental package of SECTOR. It defines different interfaces which are implemented by different classes performing the major functionalities of SECTOR. Additionally, a class implementing the predicate module is defined in this package (see section 6.7). This package also provides an entry point class which integrates all modules and provides the full functionality offered by SECTOR. A helper class which provides the common utility for all classes is defined in this package as well.
- `sector.element` – This package contains classes generated according to a network element classes hierarchy defined in UML.
- `sector.model` – This package contains classes implementing the functionality of constructing the configuration model.
- `sector.registrator` – This package contains classes implementing the functionality of registerring event specification into the Esper Engine.
- `sector.adaptor` – This package contains classes implementing the functionality of abstracting the actual alarms source and sending alarm to Esper Engine.

Classes are described in the following sections.

6.4 SECTOR Fundamental

As the fundamental package of SECTOR, package `sector` defines a list of interfaces whose implementations provide the main functionalities of SECTOR. The class diagram of this package is shown in Fig. 6.3. Note that only important *fields* and *methods* are shown for each class or interface. Detailed diagrams for classes or interfaces in `sector` package are shown in appendix A.1. The classes and interfaces in this package are described in more details below.

6.4.1 Model (interface)

`Model` interface (diagram in the appendix A, section A.1.1) defines a list of methods to manipulate a network configuration model. By invoking those methods, the network configuration information can be added into or remove from or retrieved from a model. Furthermore, it defines a list of methods which evaluate predicates regarding current information. The implementation of this interface represents a network configuration model which provides current network configuration information from its underlying knowledge base. The introduction of this interface is based on the **Strategy Pattern**. Thus, the concrete model can store its network configuration information in memory or a database system, which is transparent to client components.

6.4.2 Modeler (interface)

`Modeler` interface defines a single method *getModel()* to build a model from a model description file, e.g. an XML document. The introduction of this interface follows the **Strategy Pattern**. Thus, the algorithm to build model from a model description file is pluggable. The diagram of this interface refers to Appendix A.1.2.

6.4.3 EventSpec (interface)

`EventSpec` interface defines a list of methods to create and manipulate an event specification. Refer to chapter 4, an event specification specifies events that required to be monitored. Besides it also specifies subscriber components which will be informed when the events, to which they subscribe, are detected. The implementation of this interface represents a concrete event specification which could be parsed from an XML document or a description file in other format. The introduction of this interface is based on the **Strategy Pattern**. Thus, the implementation of building an event specification is pluggable. The diagram of this interface refers to Appendix A.1.3.

6.4.4 EventRegistrator (interface)

`EventRegistrator` interface defines a single method *register()* to register an event specification into Esper Engine. This method accepts an implementation of interface `EventSpec` as the only input parameter. The introduction of this interface is based on the **Strategy Pattern**. Thus, the actual algorithm of the registration of an event specification is pluggable. The diagram of this interface refers to Appendix A.1.4.

6.4.5 EventSubscriber (interface)

It is a wrapper of `UpdateListener` interface defined in Esper component. This interface simply extends `UpdateListener`, and therefore derives the only method *update()*, which will be invoked when new events are available. Client applications of SECTOR shall implement this interface. This design follows **Observer Pattern**. The diagram of this interface refers to Appendix A.1.5.

6.4.6 EventAdpator (interface)

The most important method defined in `EventAdaptor` interface is *start()*, which gets alarms from its associated alarm source and sends them to Esper Engine. Additionally, it exposes several methods revealing meta-information about the alarms sent to Esper Engine. Due to the meta-information, Esper Engine can know the alias of the coming alarm stream as well as the format of those alarms. For any specific alarm source, there shall be a corresponding implementation of `EventAdaptor` interface responsible for standardizing alarms into a format that Esper engine can understand. The introduction of this interface is based on the **Strategy Pattern**. Thus, the actual implementation is pluggable and make SECTOR more reusable for other network systems. The diagram of this interface refers to Appendix A.1.6.

6.4.7 Predicater (class)

`Predicater` class represents the **Predicate Layer** defined in the framework of SECTOR. It provides a list of *static*[2] methods which provide the predicates on the configuration information of current network. More information is given in section 6.7, which introduces the implementation of *predicate module*.

6.4.8 Sector (class)

`Sector` class is the main entry point of SECTOR system. It integrates all functionalities provided by SECTOR, which are building a model(*buildModel()*), setting up the predicate layer(*setUpPredicater()*), registering event specification(*registerEvent()*), reading alarms from alarm source and detecting events(both in *start()*). It wraps all subcomponents which are the underlying providers for those functionalities. The source code is in Appendix B, section B.1.1.

6.5 Implementation of Network Element Class Hierarchy

As mentioned in section 6.3, package `sector.element` defines all network element classes which compose the network element class hierarchy. A simplified class diagram for this package is shown in appendix A.2. Important classes in this package are described below.

6.5.1 Element (class)

This class is the root of the Network Element Class Hierarchy. It represents the most generic NE. It has one field *name*, which represents the identification of a particular NE. This class defined a method *getName()* to return the value of *name*.

[2]keyword in JAVA to create fields and methods that belong to the class [39].

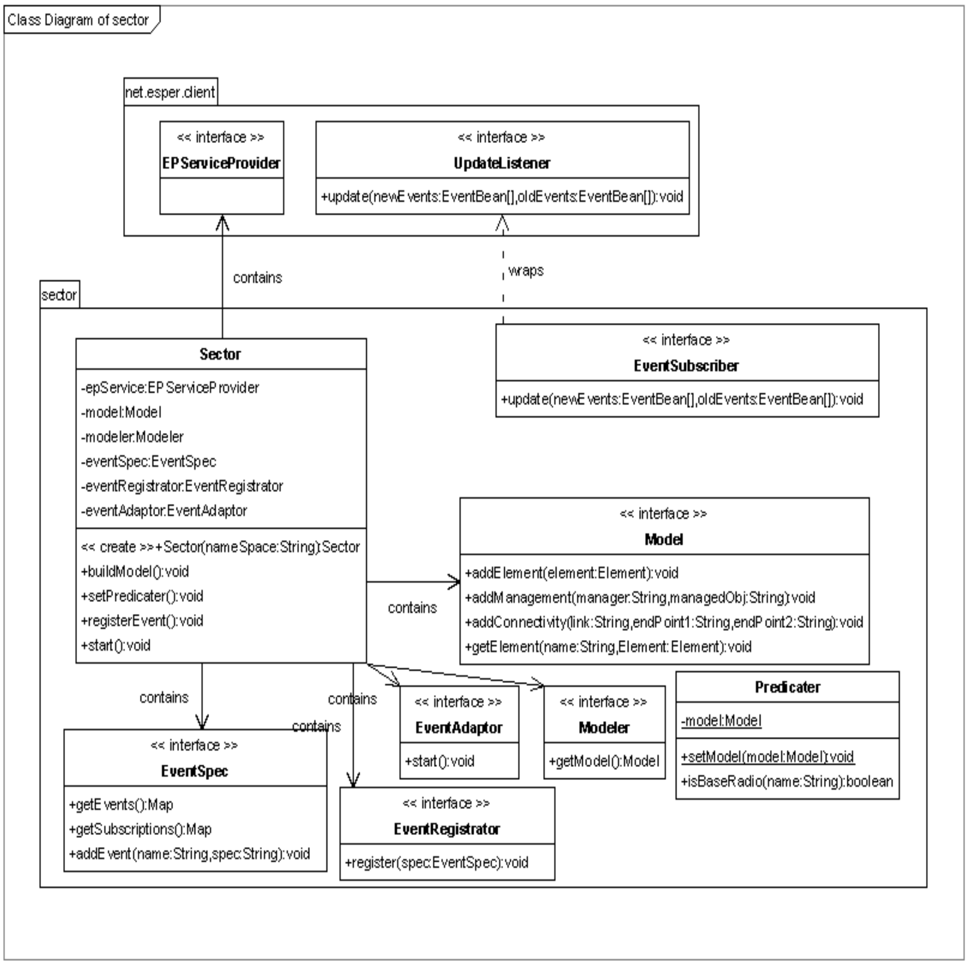

Class Diagram of sector
net.esper.client
<< interface >>
EPServiceProvider
<< interface >>
UpdateListener
+update(newEvents:EventBean[],oldEvents:EventBean[]):void
contains
wraps
sector
Sector
-epService:EPServiceProvider
-model:Model
-modeler:Modeler
-eventSpec:EventSpec
-eventRegistrator:EventRegistrator
-eventAdaptor:EventAdaptor
<< create >>+Sector(nameSpace:String):Sector
+buildModel():void
+setPredicater():void
+registerEvent():void
+start():void
<< interface >>
EventSubscriber
+update(newEvents:EventBean[],oldEvents:EventBean[]):void
<< interface >>
Model
+addElement(element:Element):void
+addManagement(manager:String,managedObj:String):void
+addConnectivity(link:String,endPoint1:String,endPoint2:String):void
+getElement(name:String,Element:Element):void
contains
contains
contains
contains
<< interface >>
EventAdaptor
+start():void
<< interface >>
Modeler
+getModel():Model
Predicater
-model:Model
+setModel(model:Model):void
+isBaseRadio(name:String):boolean
<< interface >>
EventSpec
+getEvents():Map
+getSubscriptions():Map
+addEvent(name:String,spec:String):void
<< interface >>
EventRegistrator
+register(spec:EventSpec):void

Figure 6.3: Class Diagram of package sector and its dependent classes

6.5.2 Manager (class)

The `Manager` class extends the `Element` class to represent a logic element, which manages or monitors other elements in the network. This class is defined as abstract class so that it can be a super-class of more specific classes(e.g. `ZConBTS` class) which represent concrete manager elements in Dimetra system.

This class has a field *managedObjs* declared as `java.util.Map` to keep track of all NEs which are managed by an instance of that class. `Manager` class implements methods to manipulate the set of currently managed NEs. It also implements a method *isManaging()* to determine if it is managing a specified NE or not. Each specific manager element may have the constraint about the elements it can manage. Thus, *getManagedObjectClasses()* is defined as abstract method, which returns the types of NEs that one actual manager can manage.

6.5.3 ManagedObject (class)

`ManagedObject` class extends the `Element` class to represent the managed NEs. This class is defined as abstract class so that it can be a super-class of more specific classes(e.g. `Node` class).

This class has three fields `managers`, `containers` and `components` which store all NEs managing, containing and contained in an instance of `ManagedObject` class, respectively. This class implements methods to add, remove or retrieve NEs which are stored in its three fields. It also implements methods to evaluate the management or containment relationship between its instances and other elements. Each specific `ManagedObject` element may have the constraint about its management or containment relationship with other elements. Thus, `ManagedObject` class has three unimplemented methods which return the types of elements that can manage, contain or be contained in the `ManagedObject` element.

6.5.4 Node (class)

`Node` class represents NEs which can be connected via links. It extends the `ManagedObject` class so that instances of `Node` class may be managed by some manager elements. This class is defined as abstract class. Classes (e.g. `BaseRadio` class) which extend `Node` class are specific to Dimetra domain.

`Node` class has a field *connViaLinks* which stores all link elements connected to the instance of this class. This class implements methods to manipulate or retrieve those link elements. It also implements methods to evaluate the connectivity relationship between its instances and other elements. Each specific `Node` element may have the constraint about its management and containment with other elements. Thus, `Node` class does not implement the abstract methods derived from `ManagedObject` class. Furthermore, it has another unimplemented method (*getConnViaLinkClasses()*) returning the types of link elements via which the `Node` element can be connected.

6.5.5 Link (class)

`Link` class represents a network link which is connected to a list of nodes. It extends the `ManagedObject` class so that instances of `Link` class may be managed by some manager elements. This class is defined as abstract class. Classes (e.g. `RFSiteControlPath` class) which extend `Link` class are specific to Dimetra domain.

This class has a field *endPoints* which stores two `Node` elements as endpoints. `Link` class implements a method to retrieve `Node` elements which are connected via the instance of `Link` class. `Link` class also implements method to evaluate the connectivity relationship between its instances and other elements. Each specific `Link` element may have the constraint about its management or containment relationship with other elements. Thus, `Link` class does not implement abstract methods derived from `ManagedObject` class.

6.5.6 Dimetra classes

In addition to the classes introduced so far, the rest of classes in the hierarchy are called Dimetra classes which represent NEs in Dimetra system and therefore are non-abstract classes. These classes implement the abstract methods defined in their super-classes.

As introduced in table 4.1, each Dimetra class has constraints regarding management, containment or connectivity relationships between its instances and other NEs. There are two ways to implement these constraints. One way is to use *static field*[3]. For example, `BaseRadio` class representing the base radio has a static field *containerClasses* to keep all types of NEs that can contain base

[3]The value of a static field is shared by any instances of one class

radios. According to the description in table 4.1, this field at least contains a value: `sector.element.BTS.class`. The other way is to use *constructor*[4]. This way is only used for sub-classes of `Link` class. A link element is considered to exist when its two endpoints are existing, therefore constructors of `Link` class and its sub-classes all take two endpoints as input parameters. The types of endpoints are specified in the constructors, which guarantees that a link element is connected to the allowed NEs. For instance, constructors of `RFSiteControlPath` class, which represents the link between a BTS site and a zone controller, take two input parameters: an instance of `BTS` class and an instance of `ZoneController`.

6.6 Implementation of Model Construction

Refer to section 6.1, a modeler module is responsible for building a model from one configuration file. The constructed model is saved in some sort of knowledge base. Interfaces regarding this functionality have already been defined in package `sector`. SECTOR provides their default implementation in package `sector.model`. A simplified class diagram of this package along with related interfaces is shown in Fig. 6.4. Detailed diagrams and source code can be found in appendix A and appendix B respectively. The implementation details are described in the following sub-sections.

6.6.1 Model Description File

A model description file describes the network configuration information and is able to be processed by computers. Refer to the section 4.2.2, an XML document can be used as the model description file. Note that the XML used in SECTOR implementation is slightly different from the one defined the in proposed framework. Link elements are not specified in NE-declaration section but implicitly specified in the definition of connectivity relationships. This modification can prevent developers from declaring a link element but forgetting to specify its two endpoints later in a connectivity specification. Detailed introduction regarding the DTD[5] and tags is given in appendix C, the section C.1.

[4] A special method to create an instance of a class

[5] Document Type Definition, which defines the legal building blocks of an XML document

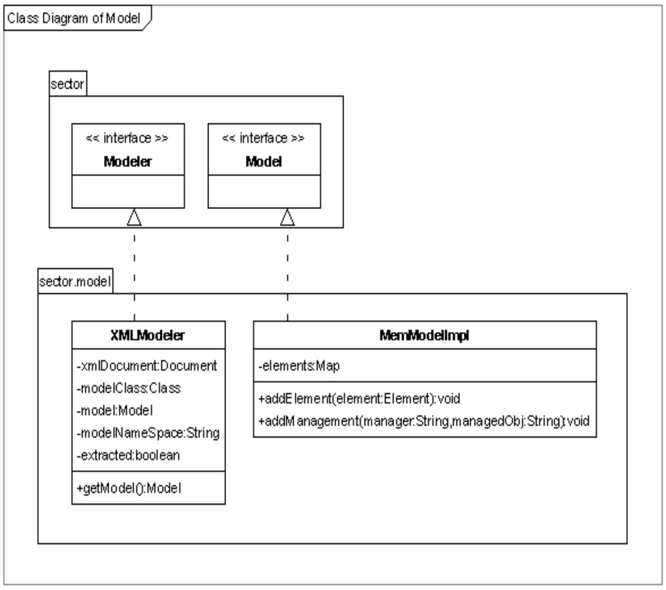


Figure 6.4: Class Diagram of package sector.model and dependent classes

6.6.2 A default Model Implementation

The default model in SECTOR keeps all network configuration information in memory. It is represented by `MemModelImpl` class, which implements the `Model` interface. All model information is stored in its field *elements* defined as `java.util.Map`. This field is a collection of $(key, value)$ pairs, where the *key* is the name of one network element and the *value* corresponds to that network element. Methods regarding editing or accessing a configuration model are implemented by manipulating this field. Methods for editing a model will be invoked by modeler classes to build a model. Furthermore, it implements methods to evaluate predicates regarding configuration information. Predicate Layer is build on the top of those methods.

6.6.3 A default Modeler

SECTOR offers a default modeler which builds the network configuration model from an XML document. `XMLModeler` class represents the default modeler by implementing the `Modeler` interface. This class has a field *modelClass*, which determines the type of models can be generated by a modeler. Hence, a modeler can generate a model relying on either memory or database. The value of this field is set according to a parameter in a configuration file.

Note that `XMLModeler` class focuses on extracting network configuration information from the XML document. The actual work for adding the configuration information into a model is done by invoking methods offered by underlying `Model` implementation.

6.7 Implementation of Predicate Layer

The **Predicate Layer** is implemented by the *predicate module*, which only consists of one class: `Predicater`. The implementation of this module does not follow the way that an interface is separated from its implementations. It is because that the predicate module is very simple. It relies on the associated model to provide predicates (see section 6.6.2). Moreover, a predicater class must declare its predicates as static methods so that they can be used in the event definitions understandable for Esper. However, interfaces are not able to declare static methods in Java. Source code of `Predicater` class refers to Appendix B, section B.1.2.

6.8 Implementation of Event Registration

Refer to section 6.1, the event registration is performed by *event registration module*. It consists of interfaces: `EventSpec` and `EventRegistrator` and their default implementation classes defined in package `sector.registrator`. A simplified class diagram of those classes and interfaces is shown in Fig. 6.5. Important details are illustrated below.

6.8.1 Event Specification File

The event specification is specified in a file. As the model description file, an XML document is used to describe the event specification. There are two sections in such an XML description file. All events are listed with their names and corresponding definitions in the first section. The order of listed events is important. It follows the rule that an event e should be listed after all events referenced in e's definition. The next section specifies the event subscriptions so Esper engine can know which subscribers it should inform when their subscribed

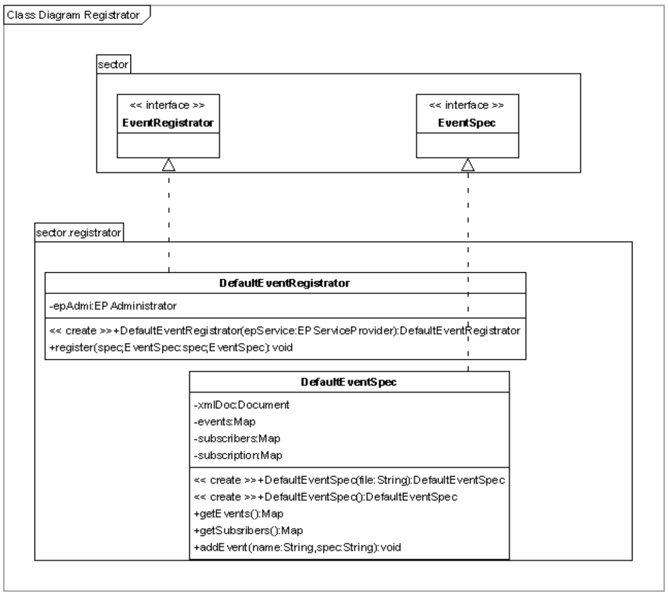

Class Diagram Registrator
sector
<< interface >>
EventRegistrator
<< interface >>
EventSpec
sector.registrator
DefaultEventRegistrator
-epAdmi:EP Administrator
<< create >>+DefaultEventRegistrator(epService:EP ServiceProvider):DefaultEventRegistrator
+register(spec;EventSpec:spec;EventSpec): void
DefaultEventSpec
-xmlDoc:Document
-events:Map
-subscribers:Map
-subscription:Map
<< create >>+DefaultEventSpec(file: String):DefaultEventSpec
<< create >>+DefaultEventSpec():DefaultEventSpec
+getEvents():Map
+getSubsribers():Map
+addEvent(name:String,spec:String):void

Figure 6.5: Class Diagram of package sector.registrator and dependent classes

events occur. Refer to appendix C.2 to see more details regarding the XML document.

6.8.2 A default Event Spec. Base

`DefaultEventSpec` class is the default implementation of `EventSpec` interface and works as the *event spec. base*. It can parse an event specification from a XML file. Furthermore, it offers APIs to allow developers to create an event specification programmatically. The source code refers to Appendix B, section B.3.1.

6.8.3 A default Event Registrator

`DefaultEventRegistrator` class is the default implementation of `EventRegistrator` interface. It contains one field *epAdmin*, which is the administrative interface to the Esper engine. Through that interface, event specifications defined in EQL patterns and EQL statements can be registered into Esper Engine. The source

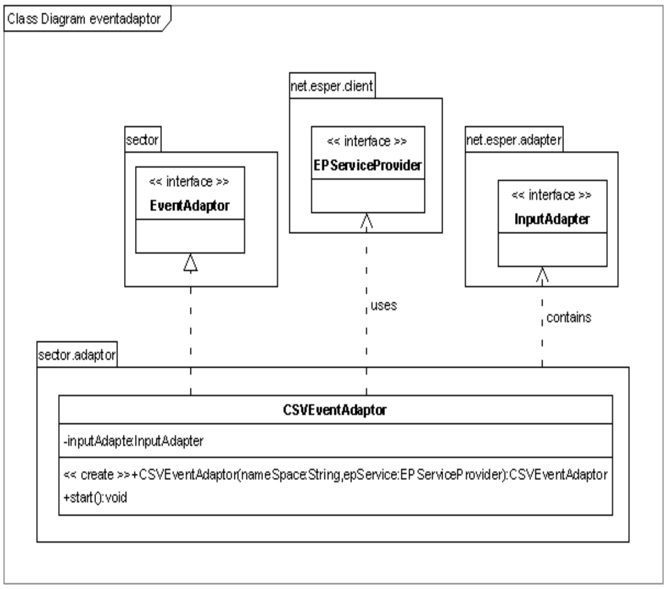


Figure 6.6: Class Diagram of sector.adaptor and dependent classes

code refers to Appendix B, section B.3.2.

6.9 Implementation of Event Adaptor

The only component in the `event adaptation module` is the event adaptor. This adaptor is represented by `CSVEventAdaptor` class, the default implementation of `EventAdaptor` interface. It reads alarms from a CSV file. This class is a wrapper of Esper InputAdapter-classes because it simply calls methods provided by Esper InputAdapter-classes to implement its functionality. Additionally, it implements several methods to show the metadata about the alarms sent to Esper Engine, such as the properties of alarm. A simplified class diagram of those classes and interfaces is shown in Fig. 6.6. The source code for `CSVEventAdaptor` class can be found in Appendix B, section B.4.1.

6.10 Implementation of Event Subscription

As introduced in section 6.4.5, every event subscriber shall implement `EventSubscriber` interface. The action invoked when the subscribed event occurs are defined in *update()* method. SECTOR defines two simple event subscribers, which print the meaningful notices to *stdout* as soon as their subscribed events are detected. Refer to Appendix B, section **??** to view the source code of these two subscriber classes.

6.11 Summary

This chapter presented an implementation of the SECTOR system. Before the implementation, the seven modules of the system were presented, each consisting of one or more components. Moreover, the design pattern used for class design are introduced. Furthermore, the implemented system was described in details with classes which are organized in individual packages.

Chapter 7

Testing and Evaluation

In general, software testing and evaluation make sure that the system runs as expected without failures. Two test strategies are used in this project. One is to test individual classes independently, which is known as *Unit Testing*. The other is to test the whole system after it has been constructed, which is known as *Integration Testing*. This chapter presents how these two test strategies are carried out. More focus will be on the integration testing. The last part of this chapter will discuss the performance of SECTOR system in terms of correlation speed and correctness.

7.1 Unit Testing

This type of testing is carried out by performing a complete functional and structural tests for every class. Different sets of test cases are devised to test various methods in different situations. It is also called *white box testing* [36].

JUnit [37], an open source unit testing framework for the Java, is adopted for achieving the automation of unit testing. Various test classes which consist of different test cases are implemented for important classes in SECTOR. These classes are organized in package `sector.test`. The source code of test classes are shown in appendix B.5.

Many test cases have been created, but only a small part of them (the most important and most interesting ones) will be presented in the following subsections, with particular emphasis on testing the model construction and event registration modules.

7.1.1 Testing on Model Construction

The unit testing for the model construction focuses on two main classes: `XMLModeler` and `MemModelImpl`. Because `XMLModeler` class counts on methods of `MemModelImpl` class to construct a configuration model, only `MemModelImplTest` class, the test class for `MemModelImpl` class, is presented here.

`MemModelImplTest` class defines lists of test cases (wrapped by methods) to test the behavior of a `MemModelImpl` instance. The most important test cases are:

- *Add a legal NE into a model*: A network element is considered to be legal if its name is not duplicated with names of NEs already in the model and this NE complies with its constraints.
- *Add an illegal NE into a model*: Two methods are defined as test cases. `testAddBaseRadioWithSameNames` tries to add two base radio elements which have the same name (ID) into a model. `testAddBadRFSiteControlPath` tries to add one illegal `RFSiteControlPath` element into the model. This element is illegal because none of its endpoint elements are pre-existed.
- *Add a legal relationship into a model*: A legal relationship refers to the management, containment, connectivity relationship that comply with pre-defined constraints.
- *Add an illegal relationship into a model*: Three kinds of methods are defined to add some illegal management, containment, connectivity relationships. For instance, method `testAddBadContainment` tries to add one bad containment relationship, in which a base radio contains a BTS site.

The Fig. 7.1 is the screenshot after running `MemModelImplTest`. It shows that there are 21 test cases defined in that test class and all of them can pass.

7.1.2 Testing on Event Registration

The unit testing for the event registration is carried out on classes `DefaultEventSpec` and `DefaultEventRegistrator`. However, only the test on `DefaultEventSpec`

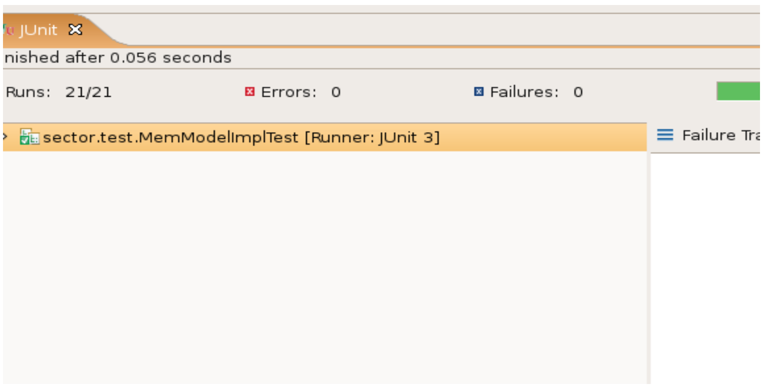

JUnit
nished after 0.056 seconds
Runs: 21/21 Errors: 0 Failures: 0
sector.test.MemModelImplTest [Runner: JUnit 3]
Failure Tra

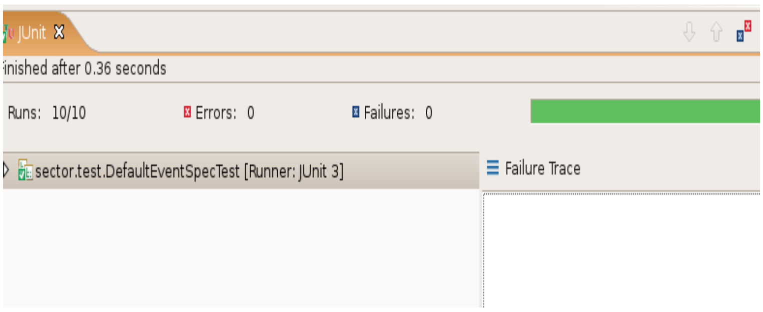
Figure 7.1: The screenshot after running `MemModelImplTest`

class is introduced here due to its complexity.

`DefaultEventSpecTest` is the test class for `DefaultEventSpec` class. The most important test cases defined in that class are:

- *Add a legal event into an event spec. base*: An event is considered to be legal if its name is not duplicated with names of events already in the event specification.
- *Add an illegal event into an event spec. base*: Method *testBadAddEvent* is defined to add an event whose name is the same as the name of another event already in event spec. base.
- *Add a legal subscription into an event spec. base*: A legal subscription means that a subscriber subscribes an event existing in the event specification.
- *Add an illegal subscription into an event spec. base*: In method *testBadAddSubscription*, a subscriber tries to subscribe an event which has not been defined.

Fig. 7.2 is the screenshot after running `DefaultEventSpecTest`. It shows that there are 10 test cases defined in that test class and none of them fails.



```
inished after 0.36 seconds
Runs: 10/10    Errors: 0    Failures: 0
sector.test.DefaultEventSpecTest [Runner: JUnit 3]
```

Figure 7.2: The screen shot after running `DefaultEventSpecTest`

7.2 Integration Testing

After all modules are tested independently, this type of testing is carried out to verify the behavior of the SECTOR system as a whole.

The integration testing is also based on test cases and follows the mechanism of black box testing [38]. There is one fault scenario per each test case. The following subsections will analyze the behavior of SECTOR in each fault scenario.

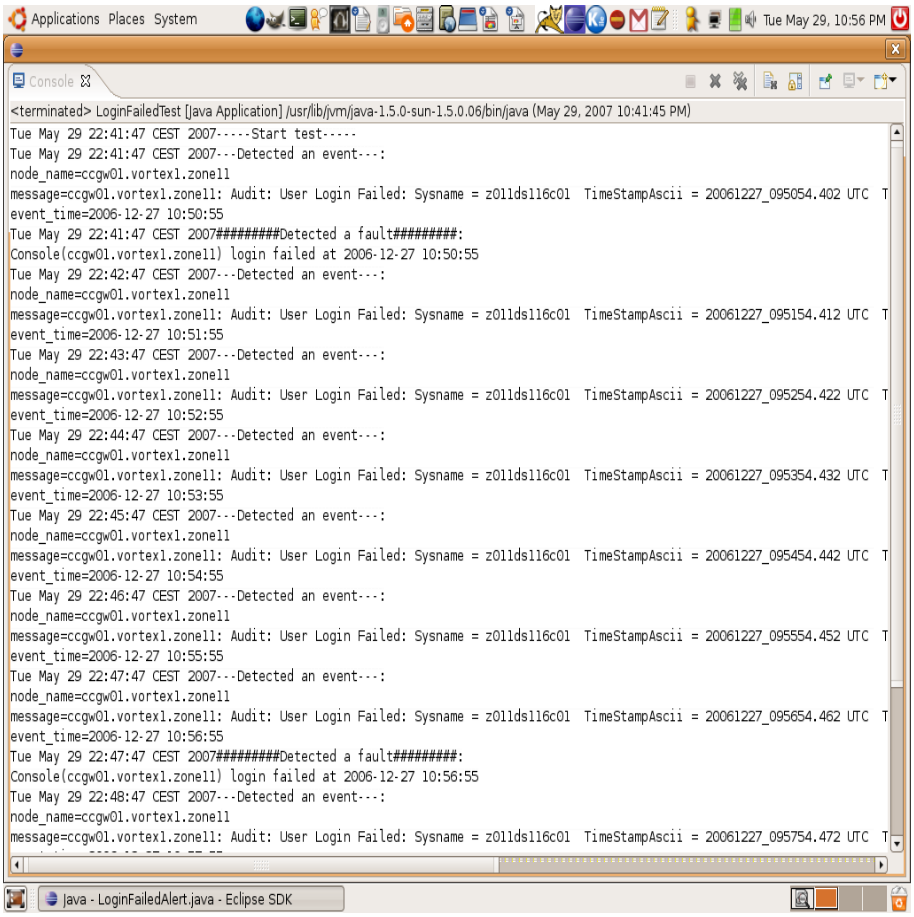
7.2.1 Console Login Failed

This scenario demonstrates how SECTOR can be used to compress or suppress the repetitive non-important events (alarms) into a single event.

Due to some reasons (mis-configuration), a "*console login failed*" alarm is repetitively reported every one minute though it is not a critical alarm. Hence, it could generate lots of redundant information for a certain time. Obviously it is desirable to suppress these alarms into one event message. A primitive event *LoginFailed* is specified according to these alarms. One composite event *SuppressedLoginFailed* is specified to correlate *LoginFailed* occurred in last 5 minutes[1]. This event is reported to operators only if a new *LoginFailed* event is detected and no *SuppressedLoginFailed* event was reported within previous 5 minutes. Refer to appendix D.1 to see more details, e.g. event definitions.

Package `sector.test.integration.suppression` contains the classes which test the behavior of SECTOR in this scenario. The source code regarding this

[1]This time could be adjusted according to specific policy of network management


Applications Places System
Tue May 29, 10:56 PM
Console
Java - LoginFailedAlert.java - Eclipse SDK

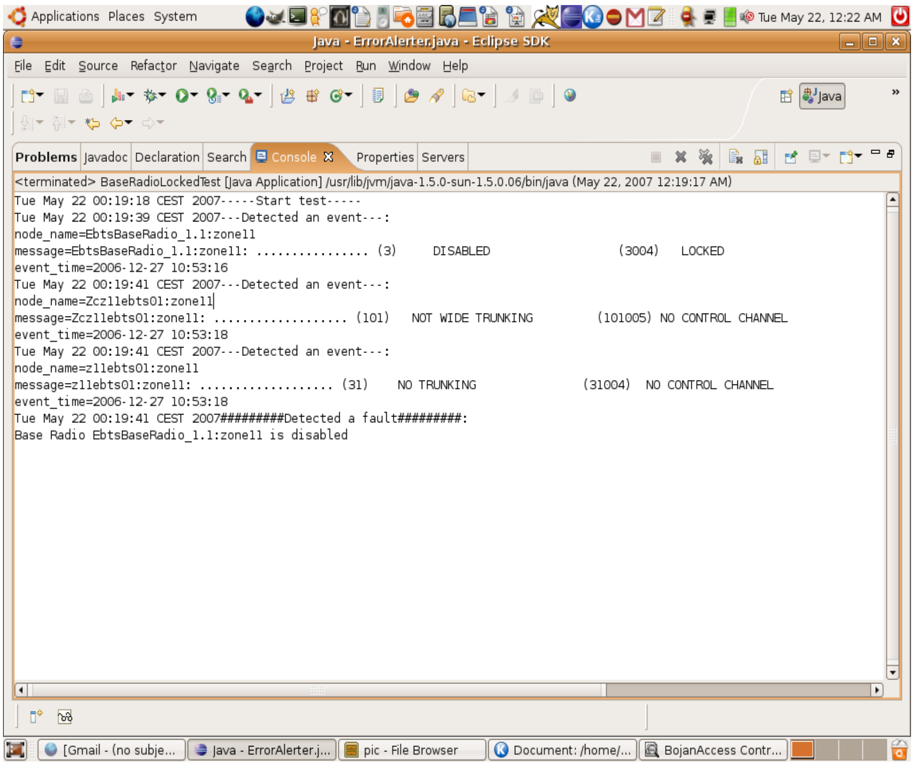
```
<terminated> LoginFailedTest [Java Application] /usr/lib/jvm/java-1.5.0-sun-1.5.0.06/bin/java (May 29, 2007 10:41:45 PM)
Tue May 29 22:41:47 CEST 2007-----Start test-----
Tue May 29 22:41:47 CEST 2007---Detected an event---:
node_name=ccgw01.vortex1.zone11
message=ccgw01.vortex1.zone11: Audit: User Login Failed: Sysname = z011ds116c01  TimeStampAscii = 20061227_095054.402 UTC  T
event_time=2006-12-27 10:50:55
Tue May 29 22:41:47 CEST 2007#########Detected a fault#########:
Console(ccgw01.vortex1.zone11) login failed at 2006-12-27 10:50:55
Tue May 29 22:42:47 CEST 2007---Detected an event---:
node_name=ccgw01.vortex1.zone11
message=ccgw01.vortex1.zone11: Audit: User Login Failed: Sysname = z011ds116c01  TimeStampAscii = 20061227_095154.412 UTC  T
event_time=2006-12-27 10:51:55
Tue May 29 22:43:47 CEST 2007---Detected an event---:
node_name=ccgw01.vortex1.zone11
message=ccgw01.vortex1.zone11: Audit: User Login Failed: Sysname = z011ds116c01  TimeStampAscii = 20061227_095254.422 UTC  T
event_time=2006-12-27 10:52:55
Tue May 29 22:44:47 CEST 2007---Detected an event---:
node_name=ccgw01.vortex1.zone11
message=ccgw01.vortex1.zone11: Audit: User Login Failed: Sysname = z011ds116c01  TimeStampAscii = 20061227_095354.432 UTC  T
event_time=2006-12-27 10:53:55
Tue May 29 22:45:47 CEST 2007---Detected an event---:
node_name=ccgw01.vortex1.zone11
message=ccgw01.vortex1.zone11: Audit: User Login Failed: Sysname = z011ds116c01  TimeStampAscii = 20061227_095454.442 UTC  T
event_time=2006-12-27 10:54:55
Tue May 29 22:46:47 CEST 2007---Detected an event---:
node_name=ccgw01.vortex1.zone11
message=ccgw01.vortex1.zone11: Audit: User Login Failed: Sysname = z011ds116c01  TimeStampAscii = 20061227_095554.452 UTC  T
event_time=2006-12-27 10:55:55
Tue May 29 22:47:47 CEST 2007---Detected an event---:
node_name=ccgw01.vortex1.zone11
message=ccgw01.vortex1.zone11: Audit: User Login Failed: Sysname = z011ds116c01  TimeStampAscii = 20061227_095654.462 UTC  T
event_time=2006-12-27 10:56:55
Tue May 29 22:47:47 CEST 2007#########Detected a fault#########:
Console(ccgw01.vortex1.zone11) login failed at 2006-12-27 10:56:55
Tue May 29 22:48:47 CEST 2007---Detected an event---:
node_name=ccgw01.vortex1.zone11
message=ccgw01.vortex1.zone11: Audit: User Login Failed: Sysname = z011ds116c01  TimeStampAscii = 20061227_095754.472 UTC  T
```

Figure 7.3: The screenshot after testing the "console loging failed"

package is given in appendix B.6. The result of testing is shown in Fig. 7.3. It clearly shows that first *SuppressedLoginFailed* event[2] is reported immediately after the occurrence of the first *LoginFailed* event[3]. The next *SuppressedLoginFailed* event is reported 6 minutes later though 6 *LoginFailed* event have been detected during that time.

[2]The tag `#########Detected a fault#########` indicates the occurrence of a *SuppressedLoginFailed* event.

[3]The tag `---Detected an event---` indicates the occurrense of a *LoginFailed* event


Applications Places System
Tue May 22, 12:22 AM
Java - ErrorAlerter.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Problems Javadoc Declaration Search Console Properties Servers
[Gmail - (no subje... Java - ErrorAlerter.j... pic - File Browser Document: /home/... BojanAccess Contr...

```
<terminated> BaseRadioLockedTest [Java Application] /usr/lib/jvm/java-1.5.0-sun-1.5.0.06/bin/java (May 22, 2007 12:19:17 AM)
Tue May 22 00:19:18 CEST 2007-----Start test-----
Tue May 22 00:19:39 CEST 2007---Detected an event---:
node_name=EbtsBaseRadio_1.1:zone11
message=EbtsBaseRadio_1.1:zone11: ................ (3)     DISABLED                    (3004)   LOCKED
event_time=2006-12-27 10:53:16
Tue May 22 00:19:41 CEST 2007---Detected an event---:
node_name=Zcz11ebts01:zone11
message=Zcz11ebts01:zone11: .................. (101)   NOT WIDE TRUNKING         (101005) NO CONTROL CHANNEL
event_time=2006-12-27 10:53:18
Tue May 22 00:19:41 CEST 2007---Detected an event---:
node_name=z11ebts01:zone11
message=z11ebts01:zone11: .................. (31)    NO TRUNKING               (31004)  NO CONTROL CHANNEL
event_time=2006-12-27 10:53:18
Tue May 22 00:19:41 CEST 2007#########Detected a fault#########:
Base Radio EbtsBaseRadio_1.1:zone11 is disabled
```

Figure 7.4: The screenshot after testing the "base radio is locked"

7.2.2 Base Radio is Locked

This test case is to test the behavior of SECTOR in "base radio locked" scenario which has already been described in section 4.2.4. Recall that section three alarms will be reported when a base radio is locked. Correspondingly, three primitive events have been defined to capture these alarms in section 4.2.5. Moreover, a composite event correlating these three primitive events has been defined as well which leads to the root cause: a base radio is locked. More details regarding this test can be found in appendix D.2.

Package `sector.test.integration.br` contains the related test classes. The source code regarding this package is given in appendix B.7. The result after running `BaseRadioLockedTest` class (the entry point of this test suite) is shown in 7.4. It shows that the root cause is reported after the occurrences of three primitive events.

7.2.3 EBTS is Disabled

A EBTS site is disabled by the operator to produce this scenario. There are two control paths between this EBTS site and a zone controller. One is the primary path, the other is the backup path.

According to an elaborated alarm analysis and the fault propagation model depicted in section 3.9, it can be derived that site control paths connected to that EBTS site and the *EBTS (ZC)* managing that EBTS site will all get affected if that EBTS site is disabled. Two primitive events *SitePathDown* and *ZCEBTSPathDown* are defined to capture those alarms. In addition to these two events, more events can be defined in order to make a more precise diagnosis. It is possible that these two events occurred when a site path is down but the EBTS site is still functional. Based on the domain knowledge, when a path is down, a backup path will be activated, which can be captured by a primitive event *ActiveBackupSitePath*. If this EBTS site is indeed disabled, an alarm will be reported to indicate that the backup path is down as well, which is an instance of event *SitePathDown*. Hence, a composite event *BothSitePathDown* is defined to capture the scenario (both site paths are down). Moreover, a primitive event *EBTSUnreacherable* is defined to capture the alarm generated by **SNMP**[4], which further indicates that this EBTS site is disabled. In all, the root cause of this scenario can be diagnosed by a composite event *EBTSDownAlert*, which correlates events *BothSitePathDown*, *ZCEBTSPathDown* and *EBTSUnreacherable*. More information regarding the test of this scenario is described in appendix D.3.

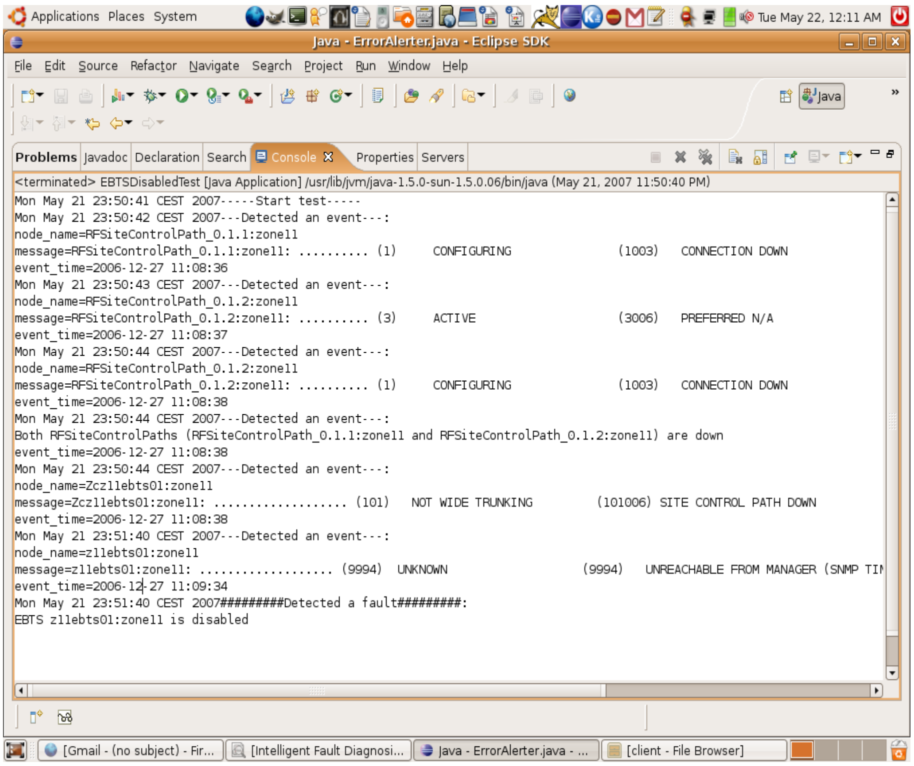
Package `sector.test.integration.ebts` contains the classes which test the behavior of SECTOR in the "EBTS site disabled" scenario. The source code regarding this package is given in appendix B.8. The result of testing is shown in 7.5. It shows that the root fault is alert after detecting 5 primitive events and 1 composite event (*BothSitePathDown*).

7.3 Performance Evaluation

This section gives the performance evaluation based on the test results shown previously. The performance of SECTOR is evaluated in terms of precision and latency.

All tests demonstrate that SECTOR can successfully diagnose the root cause

[4]Simple Network Management Protocol [41]


Applications Places System
Tue May 22, 12:11 AM
Java - ErrorAlerter.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Problems Javadoc Declaration Search Console Properties Servers
[Gmail - (no subject) - Fir...
[Intelligent Fault Diagnosi...
Java - ErrorAlerter.java - ...
[client - File Browser]

```
<terminated> EBTSDisabledTest [Java Application] /usr/lib/jvm/java-1.5.0-sun-1.5.0.06/bin/java (May 21, 2007 11:50:40 PM)
Mon May 21 23:50:41 CEST 2007-----Start test-----
Mon May 21 23:50:42 CEST 2007---Detected an event---:
node_name=RFSiteControlPath_0.1.1:zone11
message=RFSiteControlPath_0.1.1:zone11: .......... (1)     CONFIGURING               (1003)   CONNECTION DOWN
event_time=2006-12-27 11:08:36
Mon May 21 23:50:43 CEST 2007---Detected an event---:
node_name=RFSiteControlPath_0.1.2:zone11
message=RFSiteControlPath_0.1.2:zone11: .......... (3)     ACTIVE                    (3006)   PREFERRED N/A
event_time=2006-12-27 11:08:37
Mon May 21 23:50:44 CEST 2007---Detected an event---:
node_name=RFSiteControlPath_0.1.2:zone11
message=RFSiteControlPath_0.1.2:zone11: .......... (1)     CONFIGURING               (1003)   CONNECTION DOWN
event_time=2006-12-27 11:08:38
Mon May 21 23:50:44 CEST 2007---Detected an event---:
Both RFSiteControlPaths (RFSiteControlPath_0.1.1:zone11 and RFSiteControlPath_0.1.2:zone11) are down
event_time=2006-12-27 11:08:38
Mon May 21 23:50:44 CEST 2007---Detected an event---:
node_name=Zcz11ebts01:zone11
message=Zcz11ebts01:zone11: .................. (101)   NOT WIDE TRUNKING        (101006) SITE CONTROL PATH DOWN
event_time=2006-12-27 11:08:38
Mon May 21 23:51:40 CEST 2007---Detected an event---:
node_name=z11ebts01:zone11
message=z11ebts01:zone11: .................. (9994)  UNKNOWN                  (9994)   UNREACHABLE FROM MANAGER (SNMP TI
event_time=2006-12-27 11:09:34
Mon May 21 23:51:40 CEST 2007#########Detected a fault#########:
EBTS z11ebts01:zone11 is disabled
```

Figure 7.5: The screenshot after testing the "EBTS site disabled"

for each scenario based on the specified events. The number of *false positive* is zero because no additional cause is incorrectly diagnose as the source of error. Although the test results are impressive, it is still far away from the fully demonstration. Firstly, more test cases are required. Secondly, all event definitions are tested in one single system. It is reasonable to test those event definitions in more systems. Thirdly, the tests are carried out without considering the case of lost or spurious alarms. It is quite interesting to see the precision when those situation are taken in consideration. Last not the least, tests should be carried out in the field when SECTOR is tested as a production system.

Refer to screenshots shown in Fig 7.3 to Fig. 7.5, each composite event is detected as soon as the last primitive event, which is correlated by that composite event, is detected. There is almost no latency. It is expected. Because Esper is used as the event-detect engine and it is declared to detect complex patterns among events in real-time. Although it shows that SECTOR can correlate events with low latency, more tests with higher event occurrence rates should be carried out to evaluate the performance of SECTOR.

7.4 Summary

The former part of this chapter described how to test SECTOR system with two kinds of test strategies: *unit testing* and *integration testing*. Important test classes are described to introduce the unit testing. Several fault scenarios are devised in the integration testing. The results shown that SECTOR works well in those fault scenarios.

The second part of the chapter focused on the performance evaluation. It concluded that the evaluation of SECTOR is impressive but more tests are required.

Chapter 8

Conclusion

This chapter concludes the thesis work. The first part discusses how this thesis achieved the goals defined in section 1.2 and summarizes the main contribution of the project. The latter part of this chapter discusses some limitations and identifies possible future work

8.1 Achieved Goals

The primary goal of this thesis is to develop a prototype system that can automatically diagnose faults in a basic Dimetra system. The SECTOR system implemented in this thesis, as demonstrated in test cases, can work as a fault diagnosis system. It provides a rich set of features including an environment for developing event definitions and models, as well as an engine for correlating events in real time. Furthermore, the SECTOR system is designed to be extensible attribute to the use of strategy pattern. Hence, Motorola can extend SECTOR to add more features or enhance existing modules. Instead of running as a standalone fault diagnosis system, the SECTOR system can co-operate with other fault diagnosis systems. In that case, SECTOR filters out redundant alarms and passes a concise alarm stream to some higher level fault diagnosis systems.

In addition to the SECTOR system, another important contribution of this thesis is to propose a generic framework for constructing fault diagnosis systems. The network element class hierarchy in this framework is defined in an object-oriented way, which makes this class hierarchy more reusable. Moreover, ideas including the use of predicates and composite events as well as the derivation of event definitions from a causal model can all guide to construct fault diagnosis system for other networks.

8.2 Future Work

During the designing and implementing, some issues are not addressed or ignored due to the limitation of time. These issues introduce several limitations to the work done in this thesis.

The first limitation is there is not a mechanism to automatically generated event definitions from a causal model. Currently, all events are specified in EQL and Esper pattern. It is only developer-friendly but not domain expert-friendly. Hence, it could be an interesting subject to develop a method for automatically conversion between domain knowledge and event definitions.

The fault diagnosis in SECTOR system is deterministic at the moment. It is a big downside because alarms can be lost or spurious in a real system. Thus, to support the indeterministic diagnosis is a primary requirement in the future.

Additionally, more GUIs should be developed. Moreover, instead of using so many third party tools, more own components are required.

Appendix A

Class Diagram

A.1 Class Diagrams for the `sector` Package

Note that only the diagrams for the important classes or interfaces are depicted.

A.1.1 `sector.Model` interface

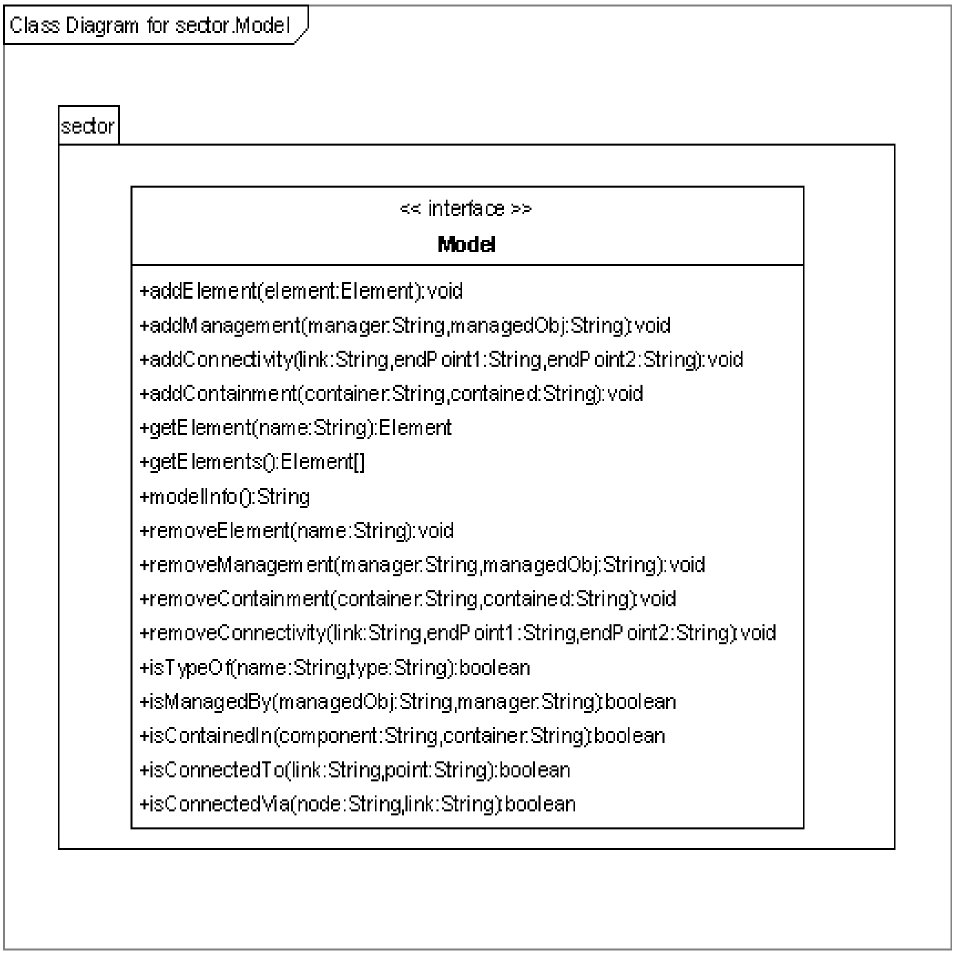

Class Diagram for sector.Model
sector
<< interface >>
Model
+addElement(element:Element):void
+addManagement(manager:String,managedObj:String):void
+addConnectivity(link:String,endPoint1:String,endPoint2:String):void
+addContainment(container:String,contained:String):void
+getElement(name:String):Element
+getElements():Element[]
+modelInfo():String
+removeElement(name:String):void
+removeManagement(manager:String,managedObj:String):void
+removeContainment(container:String,contained:String):void
+removeConnectivity(link:String,endPoint1:String,endPoint2:String):void
+isTypeOf(name:String,type:String):boolean
+isManagedBy(managedObj:String,manager:String):boolean
+isContainedIn(component:String,container:String):boolean
+isConnectedTo(link:String,point:String):boolean
+isConnectedVia(node:String,link:String):boolean

Figure A.1: Diagram for `sector.Model` interface

A.1.2 `sector.Modeler` interface

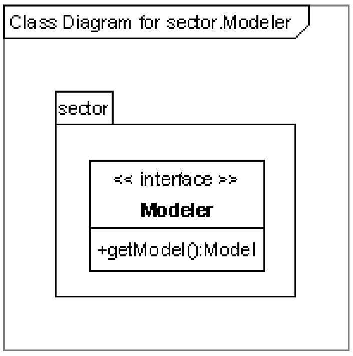


Figure A.2: Diagram for `sector.Modeler` interface

A.1.3 sector.EventSpec interface

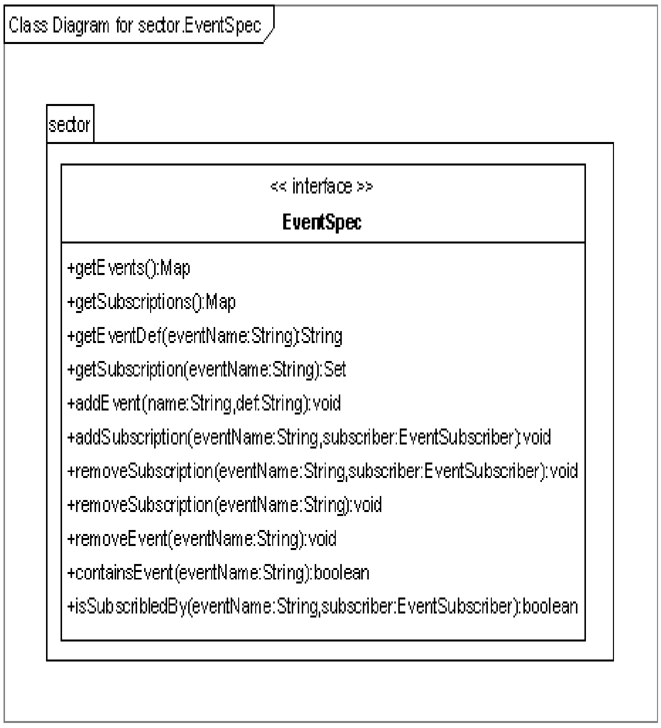

Class Diagram for sector.EventSpec
sector
<< interface >>
EventSpec
+getEvents():Map
+getSubscriptions():Map
+getEventDef(eventName:String):String
+getSubscription(eventName:String):Set
+addEvent(name:String,def:String):void
+addSubscription(eventName:String,subscriber:EventSubscriber):void
+removeSubscription(eventName:String,subscriber:EventSubscriber):void
+removeSubscription(eventName:String):void
+removeEvent(eventName:String):void
+containsEvent(eventName:String):boolean
+isSubscribledBy(eventName:String,subscriber:EventSubscriber):boolean

Figure A.3: Diagram for `sector.EventSpec` interface

A.1.4 sector.EventRegistrator interface

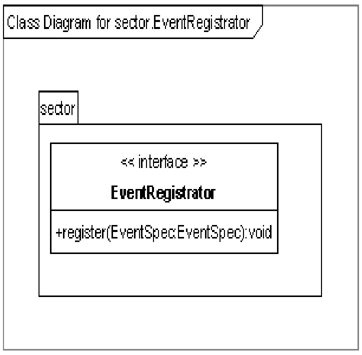

Class Diagram for sector.EventRegistrator
sector
<< interface >>
EventRegistrator
+register(EventSpec:EventSpec):void

Figure A.4: Diagram for `sector.EventRegistrator` interface

A.1.5 `sector.EventSubscriber` interface

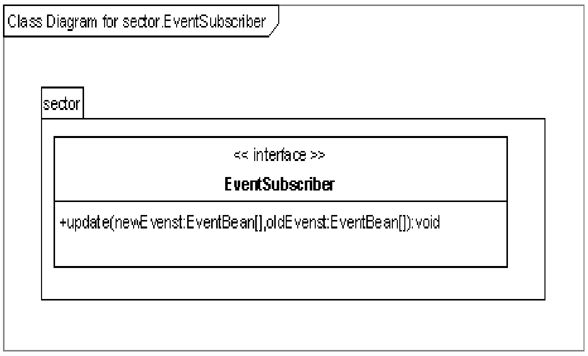

Class Diagram for sector.EventSubscriber
sector
<< interface >>
EventSubscriber
+update(newEvenst:EventBean[],oldEvenst:EventBean[]):void

Figure A.5: Diagram for `sector.EventSubscriber` interface

A.1.6 sector.EventAdpator interface

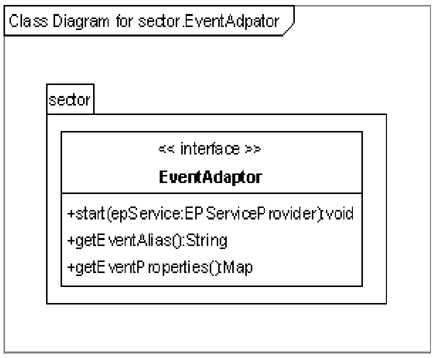

Class Diagram for sector.EventAdpator
sector
<< interface >>
EventAdaptor
+start(epService:EPServiceProvider):void
+getEventAlias():String
+getEventProperties():Map

Figure A.6: Diagram for `sector.EventAdpator` interface

A.2 Class diagrams for the network element class hierarchy

Note that only important fields or methods (e.g. the abtract method) are shown in diagrams.

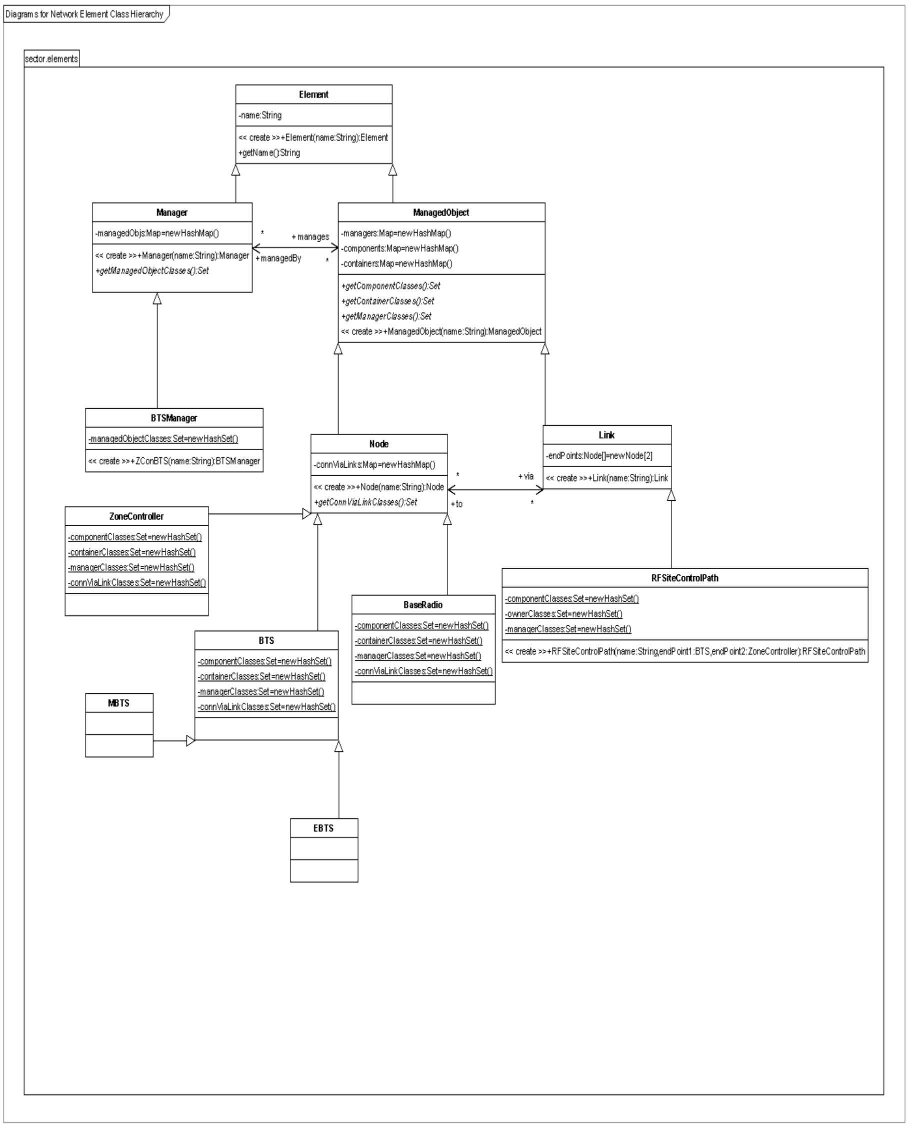

Diagrams for Network Element Class Hierarchy
sector.elements
Element
-name:String
<< create >>+Element(name:String):Element
+getName():String
Manager
-managedObjs:Map=newHashMap()
<< create >>+Manager(name:String):Manager
+getManagedObjectClasses():Set
*
+ manages
+managedBy
*
ManagedObject
-managers:Map=newHashMap()
-components:Map=newHashMap()
-containers:Map=newHashMap()
+getComponentClasses():Set
+getContainerClasses():Set
+getManagerClasses():Set
<< create >>+ManagedObject(name:String):ManagedObject
BTSManager
-managedObjectClasses:Set=newHashSet()
<< create >>+ZConBTS(name:String):BTSManager
Node
-connViaLinks:Map=newHashMap()
<< create >>+Node(name:String):Node
+getConnViaLinkClasses():Set
*
+ via
+to
*
Link
-endPoints:Node[]=newNode[2]
<< create >>+Link(name:String):Link
ZoneController
-componentClasses:Set=newHashSet()
-containerClasses:Set=newHashSet()
-managerClasses:Set=newHashSet()
-connViaLinkClasses:Set=newHashSet()
RFSiteControlPath
-componentClasses:Set=newHashSet()
-ownerClasses:Set=newHashSet()
-managerClasses:Set=newHashSet()
<< create >>+RFSiteControlPath(name:String,endPoint1:BTS,endPoint2:ZoneController):RFSiteControlPath
BaseRadio
-componentClasses:Set=newHashSet()
-containerClasses:Set=newHashSet()
-managerClasses:Set=newHashSet()
-connViaLinkClasses:Set=newHashSet()
BTS
-componentClasses:Set=newHashSet()
-containerClasses:Set=newHashSet()
-managerClasses:Set=newHashSet()
-connViaLinkClasses:Set=newHashSet()
MBTS
EBTS

Figure A.7: Simplified Class diagram for the Network Element Class Hierarchy

Appendix B

Source Code

Note that only important classes or interfaces are listed here.

B.1 Package `sector` - SECTOR Fundmental

B.1.1 Sector.java

```
package sector;

import java.lang.reflect.Constructor;

import net.esper.client.Configuration;
import net.esper.client.EPServiceProvider;
import net.esper.client.EPServiceProviderManager;
import sector.test.TestHelper;

import com.topcoder.util.config.ConfigManager;
import com.topcoder.util.config.Property;


/**
```

```
 * <p>Sector class is the main class of this component. It
     provides the
 * frameworkan implementation of EventAdaptor interface. It
     gets alarms from a CSV file. It is
 * a wrapper of Esper InputAdapter classes. Thus, this class
      simply calles methods provided by Esper
 * InputAdapter classes to offer funcationality defined in
     its interface EventAdaptor.
 * </p>
 */
public class Sector {
        private static final String MODELER = "Modeler";
        private static final String NAMESPACE = "NameSpace";
        private static final String CLASS = "Class";
        private static final String EVENTSPEC = "
            EventSpecification";
        private static final String FILEPATH = "FilePath";
        private static final String EVENTADAPTOR = "
            EventAdaptor";
        private static final String EVENTREGISTRATOR = "
            EventRegistrator";
        private static final String ESPERSERVICE = "
            EsperService";
        private static final String AUTOIMPORT = "AutoImport
            ";

        private Modeler modeler;
        private Model model;
        private EventSpec eventSpec;
        private EventAdaptor eventAdatpor;
        private EPServiceProvider epService;
        private EventRegistrator eventReg;

        public Sector(String namespace) throws
            SectorCreationException{
                Helper.checkString(namespace, "namespace");

                initModeler(namespace);
                initEventSpec(namespace);


                intiEventAdaptor(namespace);
                initEsperService(namespace);
                initEventRegistrator(namespace);

        }

        public void buildModel() throws
            ModelDescriptionException   {
                model = modeler.getModel();
        }

        public void setUpPredicater() {
                Predicater.setModel(model);

        }

        public void registerEvent() throws
            EventRegistrationException{
                this.eventReg.register(eventSpec);
        }

        public void start() throws
            EventAdatporRunTimeException{
                this.eventAdatpor.start(epService);
        }

        private void initEsperService(String namespace)
            throws SectorCreationException {
                ConfigManager cm = ConfigManager.getInstance
                    ();
                Property serviceProp;
                Configuration configuration;
                try {
                        serviceProp = cm.getPropertyObject(
                            namespace, ESPERSERVICE);
                        String[] imports = serviceProp.
                            getValues(AUTOIMPORT);
                        configuration = new Configuration();
                        for(String anImport : imports){
                                configuration.addImport(
                                    anImport);
                        }
                } catch (Exception e) {
                        throw new SectorCreationException("
                            Can not initiate Esper Service
                            due to "+e.getMessage(),e);
                }

            configuration.addEventTypeAlias(eventAdatpor.
                getEventAlias(), eventAdatpor.
                getEventProperties());
            epService = EPServiceProviderManager.
                getDefaultProvider(configuration);

        }

        private void initEventRegistrator(String namespace)
            throws SectorCreationException {
                ConfigManager cm = ConfigManager.getInstance
                    ();
                Property eventRegistratorProp;
                try {
                        eventRegistratorProp = cm.
                            getPropertyObject(namespace,
                            EVENTREGISTRATOR);
                        String eventRegistratorClassNM =
                            eventRegistratorProp.getValue(
                            CLASS);

                        Class eventRegClass = Class.forName(
                            eventRegistratorClassNM);
                        Constructor con;
                        //System.out.println("class!!!!"+
                            eventRegistratorClassNM);
                        con = eventRegClass.getConstructor(
                            new Class[]{EPServiceProvider.
                            class});
                        this.eventReg = (EventRegistrator)
                            con.newInstance(new Object[]{
                            epService});
                } catch (Exception e) {
                        throw new SectorCreationException("
                            Can not initiate Event
                            Registrator due to "+e.getMessage
                            (),e);
                }

        }

        private void intiEventAdaptor(String namespace)
            throws SectorCreationException {
                ConfigManager cm = ConfigManager.getInstance
                    ();
                Property eventAdatporProp;
                try {
                        eventAdatporProp = cm.
                            getPropertyObject(namespace,
                            EVENTADAPTOR);
                        String eventAdaptorClassNM =
                            eventAdatporProp.getValue(CLASS);
                String ns = eventAdatporProp.
                   getValue(NAMESPACE);

                Class eventAdatporClass = Class.
                   forName(eventAdaptorClassNM);
                Constructor con;

                con = eventAdatporClass.
                   getConstructor(new Class[]{String
                   .class});
                this.eventAdatpor = (EventAdaptor)
                   con.newInstance(new Object[]{ns})
                   ;
            } catch (Exception e) {
                throw new SectorCreationException("
                   Can not initiate Event Adaptor
                   due to "+e.getMessage(),e);
            }

        }

        private void initEventSpec(String namespace) throws
           SectorCreationException {
            ConfigManager cm = ConfigManager.getInstance
               ();
            Property eventSpecProp;
            try {
                eventSpecProp = cm.getPropertyObject
                   (namespace, EVENTSPEC);
                String eventSpecClassNM =
                   eventSpecProp.getValue(CLASS);
                String file = eventSpecProp.getValue
                   (FILEPATH);

                Class eventSpecClass = Class.forName
                   (eventSpecClassNM);
                Constructor con;
                //System.out.println("file :"+file+"
                    class "+eventSpecClassNM);
                con = eventSpecClass.getConstructor(
                   new Class[]{String.class});
                this.eventSpec = (EventSpec) con.
                   newInstance(new Object[]{file});
            } catch (Exception e) {
                throw new SectorCreationException("
                   Can not initiate Event
                   Specification due to "+e.
```

```
                    getMessage(),e);
        }



}

private void initModeler(String namespace) throws
    SectorCreationException {
        ConfigManager cm = ConfigManager.getInstance
            ();

        try {
                Property modelerProp = cm.
                    getPropertyObject(namespace,
                    MODELER);

                String modelerClassNM = modelerProp.
                    getValue(CLASS);
                String modelerNS = modelerProp.
                    getValue(NAMESPACE);

                Class modelerClass = Class.forName(
                    modelerClassNM);
                Constructor con;

                //since namespace is not set, try to
                     invoke the constructor without
                //any parameter to instantiate that
                    Modeler implementation
                if(modelerNS==null||modelerNS.trim()
                    .length()==0){
                        con = modelerClass.
                            getConstructor(new Class
                            [0]);
                        this.modeler = (Modeler) con
                            .newInstance(new Object
                            [0]);
                }

                //try to invoke the constructor
                    taking one string parameter to
                //instantiate that Modeler
                    implementation
                else{
                        con = modelerClass.
                            getConstructor(new Class
                            [] {String.class});
                        this.modeler = (Modeler) con
                            .newInstance(new Object
                            []{modelerNS});
                }
        } catch (Exception e) {
                throw new SectorCreationException("
                    Can not initiate Modeler due to "
                    +e.getMessage(),e);
        }

}


public static void main(String[] args){
        try {
                //System.out.println("class: "+Long.
                    class.getName()+long.class.
                    getName());

                TestHelper.loadMultipleXMLConfig("
                    sector.Sector", "config.xml");
                TestHelper.loadMultipleXMLConfig("
                    sector.model.XMLModeler", "config
                    .xml");
                TestHelper.loadMultipleXMLConfig("
                    sector.adaptor.CSVEventAdaptor",
                    "config.xml");

                Sector sector = new Sector("sector.
                    Sector");
                sector.buildModel();
                System.out.println("finish the model
                    ");
                sector.setUpPredicater();
                sector.registerEvent();
                sector.start();
        } catch (SectorCreationException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
}

}
```

B.1.2 Predicater.java

```java
package sector;

import sector.element.BTS;
import sector.element.BaseRadio;
import sector.element.EBTS;
import sector.element.MBTS;
import sector.element.RFSiteControlPath;
import sector.element.BTSManager;
import sector.element.ZoneController;
/**
 * <p>Predicater class represents the Predicate Layer in the
     framwork of SECTOR.
 * It provides a list of static methods which evaluate the
     configurtion information
 * of current network.
 *
 * It contains a member variable which associates with the
     current network configuration model. </p>
 */
public class Predicater {
        private static Model model ;
        public static void setModel(Model newModel){
                Helper.checkNull(newModel,"newModel");
                model = newModel;
        }


/**
 * The first group of predicates determine the type of a
     particular network element,
 * whose name is specified by parameter name
 */
        public static boolean isBaseRadio(String name){
                if(model == null){
                        return false;
                }
                return model.isTypeOf(name,BaseRadio.class.
                    getName());
        }

        public static boolean isEBTS(String name){
                if(model == null){
                        return false;
                }
                return model.isTypeOf(name,EBTS.class.
                    getName());
        }

        public static boolean isZoneController(String name){
                if(model == null){
                        return false;
                }
                return model.isTypeOf(name,ZoneController.
                    class.getName());
        }

        public static boolean isRFSiteControlPath(String
            name){
                if(model == null){
                        return false;
                }
                return model.isTypeOf(name,RFSiteControlPath
                    .class.getName());
        }

        public static boolean isBTS(String name){
                if(model == null){
                        return false;
                }

                return model.isTypeOf(name,BTS.class.getName
                    ());
        }

        public static boolean isMBTS(String name){
                if(model == null){
                        return false;
                }

                return model.isTypeOf(name,MBTS.class.
                    getName());
        }

        public static boolean isBTSManager(String name){
                if(model == null){
                        return false;
                }

                return model.isTypeOf(name,BTSManager.class.
                    getName());
        }
```

```



/**
* The second group of predicates evaluate the management relationship between two network elements,
* whose names are specified by parameters manager and managedObj
*/

        public static boolean isManagedBy(String managedObj, String manager){
                if(model == null){
                        return false;
                }

                return model.isManagedBy(managedObj, manager);
        }


/**
* The 3rd group of predicates evaluate the containment relationship between two network elements,
* whose names are specified by parameters component and container
*/
        public static boolean isContainedIn(String component,String container){
                if(model == null){
                        return false;
                }

                return model.isContainedIn(component, container);
        }

/**
* The 4th group of predicates evaluate the containment relationship between two network elements,
* whose names are specified by parameters component and container
*/
                public static boolean isConnectedVia(String node,String link){
                        if(model == null){
                                return false;
                        }

                        return model.isConnectedVia(node,
                            link);
                }

                public static boolean isConnectedTo(String
                    link,String node){
                        if(model == null){
                                return false;
                        }

                        return model.isConnectedTo(link,
                            node);
                }



}
```

B.1.3 Helper.java

```
package sector;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import sector.model.ModelerConfigException;

import java.io.File;
import java.util.ArrayList;
import java.util.List;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

final public class Helper {

        private Helper(){

        }

        /**
     * Checks whether the given Object is null.
     *
```

```java
     * @param arg the argument to check
     * @param name the name of the argument to check
     *
     * @throws IllegalArgumentException if the given Object
         is null
     */
    public static void checkNull(Object arg, String name) {
        if (arg == null) {
            throw new IllegalArgumentException(name + "
                should not be null.");
        }
    }

    /**
     * Checks whether the given String is null or empty.
     *
     * @param arg the String to check
     * @param name the name of the String argument to check
     *
     * @throws IllegalArgumentException if the given string
         is empty or null
     */
    public static void checkString(String arg, String name)
       {
        checkNull(arg, name);

        if (arg.trim().length() == 0) {
            throw new IllegalArgumentException(name + "
                should not be empty.");
        }
    }

    public static String getTextContents ( Node node )
    {
      checkNull(node,"node");
      NodeList childNodes;
      StringBuffer contents = new StringBuffer();

      childNodes =  node.getChildNodes();
      for(int i=0; i < childNodes.getLength(); i++ ){
        if( childNodes.item(i).getNodeType() == Node.
            TEXT_NODE ){
                contents.append(childNodes.item(i).
                    getNodeValue());
        }
      }
      return contents.toString();
    }

    public static Element[] getDirectElementsByTagName(Node
        node, String name){
         checkNull(node,"node");
         checkString(name,"name");
         NodeList childNodes = node.getChildNodes();
         List<Element> list = new ArrayList<Element>();
         for(int i=0; i < childNodes.getLength(); i++ ){
             if( childNodes.item(i).getNodeType() == Node.
                 ELEMENT_NODE ){
                 Element element = (Element)childNodes.item(i
                     );
                 if(element.getTagName().equals(name)){
                         list.add(element);
                 }
             }
           }

         Element[] elements = new Element[list.size()];
         int i= 0;
         for(Element element : list){
                 elements[i] = element;
                 i++;
         }

         return elements;

    }

}
```

B.2 Package sector.model - Model Construction

B.2.1 MemModelImpl.java

```
package sector.model;

import java.util.HashMap;
import java.util.Map;
import sector.Helper;
import sector.ModelDescriptionException;
import sector.element.Element;
import sector.element.IllegalComponentException;
import sector.element.IllegalLinkException;
import sector.element.IllegalManagedObjException;
import sector.element.IllegalManagerException;
import sector.element.IllegalOwnerException;
import sector.element.Link;
import sector.element.ManagedObject;
import sector.element.Manager;
import sector.element.Node;



/**
 * <p>This class represents the only implementation of the
     interface Modle provided in SECTOR at the moment.
 * All model information is keeped in MemModelImpl.elements
     member field, which is defined as java.util.Map.
 *
 * This class has only one constructor having no input
     parameter.
 * </p>
 */
public class MemModelImpl implements sector.Model {

/**
 * <p>The data structure contains all model information</p>
 */
    final private Map<String,Element> elements = new HashMap
        <String,Element>();


    public MemModelImpl(){
        //do nothing in current version
    }

/**
 * <p>Add one network element into this model</p>
 * @param element
 * @throws ModelDescriptionException If there is already an
     element which has the same name as the added one,
 *          or the endpoints of added link are not pre-
     existed, then this exception is thrown
 */
    public void addElement(sector.element.Element element)
        throws ModelDescriptionException {
        Helper.checkNull(element,"element");
        if(this.getElement(element.getName())!=null){
                throw new ModelDescriptionException("There
                    is already one element called "+element.
```

```
                     getName());

        }

        //special check for Link element
        if(element instanceof Link){
                Link link = (Link) element;
                Node[] endpoints = link.getEndPoints();
                for( Node endpoint : endpoints){
                        if(getElement(endpoint.getName())==
                            null){
                                throw new
                                    ModelDescriptionException
                                    ("Failed to add Link "+
                                    element.getName()+
                                                "due to one
                                                    or both
                                                    of its
                                                    endpoints
                                                     do not
                                                    exist in
                                                    the model
                                                    ");
                        }
                }

        }

        //System.out.println("add "+element.getName()+" "+
            element.getClass().getName());
        this.elements.put(element.getName(), element);
    }

/**
 * <p>Add one management relationship into this model</p>
 * @param manager
 * @param managedObj
 * @throws ModelDescriptionException
 */
    public void addManagement(String manager, String
        managedObj) throws ModelDescriptionException {

        Element managerElement = this.getElement(manager);
        Element managedElement = this.getElement(managedObj)
            ;

        if(managerElement==null || managedElement==null){
```

```
                        throw new ModelDescriptionException("Illegal
                             Management description. Can not find the
                             Manager element "
                                        +manager+" or ManagedObject
                                            element "+managedObj);
                }

                if(!(managerElement instanceof Manager)){
                        throw new ModelDescriptionException(manager+
                            " is not a Manager");
                }

                if(!(managedElement instanceof ManagedObject)){
                        throw new ModelDescriptionException(
                            managedObj+" is not a ManagedObject");
                }

                try {
                                ((Manager)managerElement).
                                    addManagedObject((ManagedObject)
                                    managedElement);
                                ((ManagedObject)managedElement).
                                    addManager((Manager)
                                    managerElement);
                        } catch (IllegalManagedObjException e) {
                                throw new ModelDescriptionException(
                                    "Failed to create management
                                    relationship due to "+e.
                                    getMessage(),e);
                        } catch (IllegalManagerException e) {
                                throw new ModelDescriptionException(
                                    "Failed to create management
                                    relationship due to "+e.
                                    getMessage(),e);
                        }



        }

/**
 * <p>Does ...</p>

 * @param container
 * @param contained
 * @throws ModelDescriptionException
 */
```

```
    public void addContainment(String container, String
        component) throws ModelDescriptionException {
        Element containerElement = this.getElement(container
            );
        Element componentElement = this.getElement(component
            );

        if(containerElement==null || componentElement==null)
            {
                throw new ModelDescriptionException("Illegal
                    Containment description. Can not find
                    the Container element "
                                +container+" or component
                                    element "+component);
        }

        if(!(containerElement instanceof ManagedObject)){
                throw new ModelDescriptionException(
                    container+" is not a ManagedObject");
        }

        if(!(componentElement instanceof ManagedObject)){
                throw new ModelDescriptionException(
                    component+" is not a ManagedObject");
        }

        try {
                        ((ManagedObject)containerElement).
                            addComponent((ManagedObject)
                            componentElement);
                        ((ManagedObject)componentElement).
                            addContainer((ManagedObject)
                            containerElement);
                } catch (IllegalComponentException e) {
                        throw new ModelDescriptionException(
                            "Failed to create containment
                            relationship due to "+e.
                            getMessage(),e);
                } catch (IllegalOwnerException e) {
                        throw new ModelDescriptionException(
                            "Failed to create containment
                            relationship due to "+e.
                            getMessage(),e);
                }
    }

/**
```

```
* <p>Does ...</p>
*
* @poseidon-object-id [Imb633c4fm1107ed02682mm6d0e]
* @param link
* @param point1
* @throws ModelDescriptionException
*/
   public void addConnectivity(String link, String point1,
        String point2) throws ModelDescriptionException {
        Element linkElement = this.getElement(link);
        Element point1Element = this.getElement(point1);
        Element point2Element = this.getElement(point2);

        if(linkElement==null||point1Element==null ||
            point2Element==null){
                throw new ModelDescriptionException("Illegal
                    Connectivity description. Can not find
                    the Link element "
                                +link+" or one point element
                                    "+point1+" or the other
                                    point element "+point2);
        }

        if(!(linkElement instanceof Link)){
                throw new ModelDescriptionException(link+"
                    is not a Link");
        }
        if(!(point1Element instanceof Node)){
                throw new ModelDescriptionException(point1+"
                    is not a Node");
        }

        if(!(point2Element instanceof Node)){
                throw new ModelDescriptionException(point2+"
                    is not a Node");
        }

        try {
                        ((Node)point1Element).addLink((Link)
                            linkElement);
                        ((Node)point2Element).addLink((Link)
                            linkElement);
                } catch (IllegalLinkException e) {
                        throw new ModelDescriptionException(
                            "Failed to create connectivity
                            relationship due to "+e.
                            getMessage(),e);
                    }
    }

/**
 * <p>Does ...</p>
 * @param name
 * @return
 */
    public sector.element.Element getElement(String name) {

        Helper.checkString(name, "name");

        return this.elements.get(name);
    }

/**
 * <p>Does ...</p>
 * @param name
 */
    public void removeElement(String name) {
        Helper.checkString(name, "name");

        Element element = getElement(name);

        //if this element is in model
        if(element!=null){

                // it is a Manager element
                if(element instanceof Manager){
                        removeManagerRef((Manager)element);
                }

                //it is not Manager element
                else{

                        //clear out containment and
                            management relationship
                        removeContainerRef((ManagedObject)
                            element);
                        removeComponentRef((ManagedObject)
                            element);
                        removeManagedRef((ManagedObject)
                            element);

                        //clear out connectivity
                            relationshiop

```

```
                  //if it is Node
                  if(element instanceof Node){
                        Node node = (Node) element;
                        Link[] links = node.getLinks();

                        //all links which are connected to this node should be removed
                        //since this node is their endpoints
                        for(Link link:links){
                              removeElement(link.getName());
                        }

            }
                  // or if it is a link
                  else if(element instanceof Link){
                        Link link = (Link) element;
                        Node[] nodes = link.getNodes();

                        for(Node node:nodes){
                              node.removeLink(link.getName());
                        }
                  }

            }
            elements.remove(name);
            element = null;
      }

   }


/**
 * Remove this component element from elements which are containing it
 * @param mobj
 */
private void removeComponentRef(ManagedObject object) {
      //Helper.checkNull(object, "object");
      ManagedObject[] containers = object.getContainers();
      for(ManagedObject container:containers ){
            container.removeComponent(object.getName());
```

```
        }

}

/**
 * Remove this managed element from elements which are
     managing it
 * @param mobj
 */
private void removeManagedRef(ManagedObject mobj) {
        //Helper.checkNull(mobj, "mobj");
        Manager[] managers = mobj.getManagers();
        for(Manager manager:managers ){
                manager.removeManagedObject(mobj.getName());
        }

}

/**
 * Remove this container from elements which are referring
     to it
 * @param mobj
 */
private void removeContainerRef(ManagedObject mobj) {
        //Helper.checkNull(mobj, "mobj");
        ManagedObject[] components = mobj.getComponents();
        for(ManagedObject component:components ){
                component.removeContainer(mobj.getName());
        }

}

/**
 * Remove this manager from the Managed elements which are
     referring to it
 * @param manager
 */
private void removeManagerRef(Manager manager) {
        //Helper.checkNull(manager, "manager");
        ManagedObject[] managedObjs = manager.
            getManagedObjects();
        for(ManagedObject managedObj:managedObjs ){
                managedObj.removeManager(manager.getName());
        }

}

```

```
/**
 * <p>Remove the management relationship between two elements. If there is not such a relationship, do nothing</p>
 * @param manager
 * @param managedObj
 */
    public void removeManagement(String manager, String managedObj) {
          Helper.checkString(manager, "manager");
          Helper.checkString(managedObj, "managedObj");

          Manager managerEle = (Manager) this.getElement(manager);
          ManagedObject managedObjEle = (ManagedObject) this.getElement(managedObj);

          //either the manger elment or the manged element does not exist
          //which means there is no such a relationship. Thus, simply return
          if(managerEle == null || managedObjEle == null){
                return;
          }

          else{
                  managerEle.removeManagedObject(managedObj);
                  managedObjEle.removeManager(manager);
          }
    }

/**
 * <p>Remove the containment relationship between two elements. If there is not such a relationship, do nothing</p>
 * @param container
 * @param containedObj
 */
    public void removeContainment(String container, String containedObj) {
         Helper.checkString(container, "container");
         Helper.checkString(containedObj, "containedObj");

         ManagedObject containerEle = (ManagedObject) this.getElement(container);
         ManagedObject containedObjEle = (ManagedObject) this.getElement(containedObj);
```

```

        //either the container elment or the contained element does not exist
        //which means there is no such a relationship. Thus, simply return
        if(containerEle == null || containedObjEle==null){
                return;
        }

        else{
                containerEle.removeComponent(containedObj);
                containedObjEle.removeContainer(container);
        }

    }

/**
 * <p>Remove the connectivity relationshiop among two nodes and the link via which they are connected</p>
 * @param link
 * @param point1
 * @param point2
 */
    public void removeConnectivity(String link, String point1, String point2) {
         Helper.checkString(link, "link");
         Helper.checkString(point1, "point1");
         Helper.checkString(point2, "point2");

         //if this two points are connected via this link
         //then this link should be removed from model
         if(isConnectedTo(link,point1)&&isConnectedTo(link,point2)){
                 removeElement(link);
         }

    }



    public boolean isContainedIn(String component, String container) {
        Element componentElement = this.getElement(component);
        if(componentElement!=null&&(componentElement instanceof ManagedObject)){
                return ((ManagedObject)componentElement).
                    isContainedIn(container);
        }
        return false;
    }
    public boolean isManagedBy(String managedObj, String
        manager) {
        Element managedObjElement = this.getElement(
            managedObj);
        if(managedObjElement!=null&&(managedObjElement
            instanceof ManagedObject)){
                return ((ManagedObject)managedObjElement).
                    isManagedBy(manager);
        }
        return false;
    }

    public boolean isTypeOf(String name, String type) {
        Element element = getElement(name);
        try {
                        Class classObj = Class.forName(type)
                            ;
                        return classObj.isInstance(element);
                } catch (ClassNotFoundException e) {
                        return false;
                }
    }

        public boolean isConnectedVia(String node, String
            link) {
                Element nodeElement = this.getElement(node);
        if(nodeElement!=null&&(nodeElement instanceof Node))
            {
                return ((Node)nodeElement).isConnectedVia(
                    link);
        }
        return false;
        }

        public boolean isConnectedTo(String ele1, String
            ele2) {
                Element element = this.getElement(ele1);
        if(element!=null){
                if(element instanceof Link){
                        return ((Link)element).isConnectedTo
                            (ele2);
                }

                else if(element instanceof Node){
                        return ((Node)element).isConnectedTo
                            (ele2);
                }
        }
        return false;
        }

        public Element[] getElements() {
                Element[] copyElements = new Element[
                    elements.size()];
                System.out.println(this.elements.size());
                int i = 0;
                for(Element element :elements.values()){
                        copyElements[i] = element;
                        i++;
                }

                return copyElements;
        }

        public String modelInfo() {
                StringBuffer sb = new StringBuffer();
                sb.append("Model Info:\n");
                for(Element element : getElements()){
                        if(element!=null)
                        sb.append("Name:"+element.getName()+
                            "\nType:"+element.getClass().
                            getName()
                                        +"\
                                            n@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
                                            \n");
                }
                return sb.toString();
        }

        public String toString(){
                return modelInfo();
        }


        /**
         * A simple way to test if two MemModelImpl
             instances are equal or not.
         */
        public boolean equals(Object obj){
                if(obj==null)
                        return false;
                if(!(obj instanceof sector.model.MemModelImpl))
                        return false;

                if (this == obj)
                        return true;

                //simply test the equality by comparing the info
                //it may be imporved in the future
                return this.toString().equals(obj.toString());
        }

        /**
         * Clear this model
         */
        public void clear() {
                elements.clear();

        }


 }
```

B.2.2 XMLModeler.java

```

package sector.model;
import java.io.File;
import java.lang.reflect.Constructor;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.NodeList;

import sector.Helper;
import sector.Model;
import sector.ModelDescriptionException;
import sector.element.Link;

import com.topcoder.util.config.ConfigManager;
import com.topcoder.util.config.Property;
```

```java
import com.topcoder.util.config.UnknownNamespaceException;

/**
 * <p> This class represents the default implementation of
     the interface Modler in SECTOR. It extracts the model
 * information from a XML file and build the corresponding
     model according to the specific implementation of the
 * interface Model, which is specified in the configuration
     file. </p>
 */
public class XMLModeler implements sector.Modeler {

/**
 * <p> Represents the XML document from where model
     information is extracted. It is initialized in the
     constructor.
 * It is used the first time when getModel() method is
     invoked and the model information is parsed. After
     succesfully
 * constructing the model, it will be set to null. </p>
 */
        private org.w3c.dom.Document xmlDocument;

/**
 * <p> The class of the actual implementation of interface
     Model. It is used to generated the actual model
     implementation</p>
 */
        private Class<Model> modelClass;

/**
 * <p> Represents the actual implementation of interface
     Model associated with current instance.</p>
 */
        private Model model = null;

/**
 * <p> The namespace to load configuration properties to
     construct the actual Model implementation.
 * If it is not null, it will be used as a input parameter
     for the contructors of Model implementation classes.
 * If it is null, then contructors without any input
     parameters will be invoked.
 * </p>
 */
        private String modelNameSpace;

```

```
/**
 * <p>
 * A flag to indict if the configuration info has been extracted or not
 * </p>
 */
        private boolean extracted = false;


/**
 * <p>
 * The Keys will be in the configuration file for generating XMLModeler
 * </p>
 */
        private static final String MODEL_FILE = "ModelFile";
        private static final String MODEL = "Model";
        private static final String CLASS = "Class";
        private static final String NAMESPACE = "NameSpace";


/**
 * <p>
 * The names of Element Tags will be in the xml file which is parsed by XMLModeler
 * </p>
 */
        private static final String ELE_ELEMENT = "Element";

    //the names related to connectivity relationshiop
        private static final String ELE_CONNECTIVITY = "Connectivity";
        private static final String ELE_LINK = "Link";
        private static final String ELE_POINT = "Point";

    //the names related to management relationshiop
        private static final String ELE_MANAGEMENT = "Management";
        private static final String ELE_MANAGER = "Manager";
        private static final String ELE_MANAGED = "Managed";

        //the names related to containment relationshiop
        private static final String ELE_CONTAINMENT = "Containment";
        private static final String ELE_CONTAINER = "Container";
        private static final String ELE_COMPONENT = "
            Component";

        private static final String ATTR_NAME = "name";
        private static final String ATTR_CLASS = "class";

/**
 * <p>The constructor accepting a namespace string as
     parameter. </p>
 *
 * @param nameSpace: the namespace helps ConfigManager
     identify the configuration properites related to
     XMLModeler
 * @throws ModelerConfigException
 */
        public XMLModeler(String nameSpace) throws
            ModelerConfigException {
                Helper.checkString(nameSpace,"nameSpace");
                ConfigManager cm = ConfigManager.getInstance
                    ();
                String modelClassNM = null;
                try {
                        //get the name of the configuration
                            model file
                        String file = cm.getString(nameSpace
                            , MODEL_FILE);
                        xmlDocument = getXMLDocument(file);

                        //start to get info for the model
                            associated with this modeler
                        Property property = cm.
                            getPropertyObject(nameSpace,
                            MODEL);

                        //1. get the actual Model
                            implementation
                        modelClassNM = property.getValue(
                            CLASS);
                        modelClass = (Class<Model>) Class.
                            forName(modelClassNM);

                        //check if the class is an
                            implementation of Model interface
                        modelClass.asSubclass(Model.class);

                        //2. get the namespace, which is
                            used during the construction of
```

```
                                  that model, if it is specified.
                              //   otherwise null will be setted
                                  to field modelNameSpace
                              modelNameSpace = property.getValue(
                                  NAMESPACE);

                              //3. generate the associated model.
                              //Note that the model generated at
                                  this stage contains nothing
                                  configuration info
                              model = initModel();
                    } catch (UnknownNamespaceException e) {
                              throw new ModelerConfigException("
                                  Unknown name space "+nameSpace,e)
                                  ;
                    } catch (ClassNotFoundException e) {
                              throw new ModelerConfigException("
                                  Can not find the specified class
                                  "+modelClassNM,e);
                    }catch(ClassCastException e){
                              throw new ModelerConfigException("
                                  The specified class "+
                                  modelClassNM+" is not an
                                  implementation of interface Model
                                  ",e);
                    }


          }

/**
 * Initialize the associated Model. The initial  Model does
     not contain any networking configuration info. Those
     info will be
 * added into this Model during the parsing of configuration
      file.
 * @return the initial Model
 * @throws ModelerConfigException
 */
          private Model initModel() throws
               ModelerConfigException {
          Constructor constructor;
          try{
                    //namespace is not specified, try to invoke
                        the contructor without any input
                        parameter
```

```
                if(modelNameSpace==null||modelNameSpace.trim().length()==0){
                        constructor = this.modelClass.getConstructor(new Class[0]);
                        return (Model) constructor.newInstance(new Object[0]);
                }

                //namespace is specified, try to invoke the contructor(string) to generate Model instance
                else{
                        constructor = this.modelClass.getConstructor(new Class[]{String.class});
                        return (Model) constructor.newInstance(new Object[]{modelNameSpace});
                }
                }catch (Exception e) {
                        throw new ModelerConfigException("Failed to initiate the specified model due to "+e.getMessage(),e);
                }
    }


/**
 * <p>Extracts from the XML document the configuration model details and builds the corresponding model, which is an implementation
 * of interface Model. The returned model will contain valid network configuration information. A null model will never be returned.
 * otherwise, exception will be thrown. After the first succesfull invocation of this method, the XMLModeler.model member variable
 * will cache the model, and XMLModeler.xmlDocument will be set to null. Next calls to the method will return the cached model.
 * Details about the algorithm to be applied, structure of the XML document and building the model can be found in the
 * desgin specification. </p>
 *
 * @return the retrieved model
 * @throws ModelDescriptionException
 */
    public sector.Model getModel() throws ModelDescriptionException {
        if(!extracted){
                org.w3c.dom.Element root = xmlDocument.getDocumentElement();

                //retrieve all networking elements except Link elements
                NodeList nodes = root.getElementsByTagName(ELE_ELEMENT);
                for(int i= 0; i<nodes.getLength();i++){
                        parseNetworkElement((org.w3c.dom.Element)nodes.item(i));
                }
                //retrieve Link elements and the connectivity relationship among Node elements are constructed as well
                nodes = root.getElementsByTagName(ELE_CONNECTIVITY);
                        parseConnectivity((org.w3c.dom.Element)nodes.item(0));

                        //construct management relationship
                        nodes = root.getElementsByTagName(ELE_MANAGEMENT);
                        parseManagement((org.w3c.dom.Element)nodes.item(0));

                        //construct containment relationship
                        nodes = root.getElementsByTagName(ELE_CONTAINMENT);
                        parseContainment((org.w3c.dom.Element)nodes.item(0));

                        xmlDocument = null;
                        extracted = true;

        }
        return model;

    }

    private void parseContainment(org.w3c.dom.Element element) throws ModelDescriptionException {
        NodeList containers = element.getElementsByTagName(ELE_CONTAINER);
```

```
                for(int i= 0; i<containers.getLength();i++){
                        org.w3c.dom.Element containerElement
                             = (org.w3c.dom.Element)
                            containers.item(i);
                        String container = containerElement.
                            getAttribute(ATTR_NAME);

                        NodeList components =
                            containerElement.
                            getElementsByTagName(
                            ELE_COMPONENT);
                        for(int l= 0; l<components.getLength
                            ();l++){
                                org.w3c.dom.Element
                                    component = (org.w3c.dom.
                                    Element)components.item(i
                                    );
                                model.addContainment(
                                    container, component.
                                    getAttribute(ATTR_NAME));
                        }
                }

    }

        private void parseManagement(org.w3c.dom.Element
            element) throws ModelDescriptionException {
        NodeList managers = element.getElementsByTagName(
            ELE_MANAGER);
                for(int i= 0; i<managers.getLength();i++){
                        org.w3c.dom.Element managerElement =
                             (org.w3c.dom.Element)managers.
                            item(i);
                        String manager = managerElement.
                            getAttribute(ATTR_NAME);

                        NodeList managedElements =
                            managerElement.
                            getElementsByTagName(ELE_MANAGED)
                            ;
                        for(int l= 0; l<managedElements.
                            getLength();l++){
                                org.w3c.dom.Element
                                    managedElement = (org.w3c
                                    .dom.Element)
                                    managedElements.item(i);
                                model.addManagement(manager,
                                     managedElement.
                                    getAttribute(ATTR_NAME));
                        }

                }
        }

        private void parseConnectivity(org.w3c.dom.Element
            element) throws ModelDescriptionException {
        NodeList links = element.getElementsByTagName(
            ELE_LINK);
                for(int i= 0; i<links.getLength();i++){
                        org.w3c.dom.Element link = (org.w3c.
                            dom.Element)links.item(i);
                        parseLink(link);
                }

        }

        private void parseLink(org.w3c.dom.Element linkNode)
             throws ModelDescriptionException {
                String name = linkNode.getAttribute(
                    ATTR_NAME);
                String className = linkNode.getAttribute(
                    ATTR_CLASS);
                sector.element.Element[] points = new sector
                    .element.Element[2];
                Class elementClass = null;
                NodeList nodes = linkNode.
                    getElementsByTagName(ELE_POINT);
                if(nodes.getLength()!=2){
                        throw new ModelDescriptionException(
                            "Link "+name+" should have two
                            endpoints!");
                }

                for(int i= 0; i<nodes.getLength();i++){
                        String pointName = ((org.w3c.dom.
                            Element)nodes.item(i)).
                            getAttribute(ATTR_NAME);
                        points[i] = model.getElement(
                            pointName);
                        if(points[i]==null){
                                throw new
                                    ModelDescriptionException
                                    ("Endpoint "+pointName+"
                                        does not exist!");
                        }
                }

                try {
                        elementClass = Class.forName(
                            className);
                } catch (ClassNotFoundException e) {
                        throw new ModelDescriptionException(
                            "Can not parse link "+name+" due
                            to "+e.getMessage());
                }
                Constructor[] constructors = elementClass.
                    getConstructors();
                Link link = null;

                //find the right constructor to instantiate
                    a link instance
                for(Constructor constructor:constructors){
                        try {
                                link = (Link)constructor.
                                    newInstance(new Object[]{
                                    name,points[0],points
                                    [1]});

                        } catch (Exception e) {
                                ;
                        }
                }

                //failed to instantiate a link instance
                if(link==null){
                        throw new ModelDescriptionException(
                            "Can not parse link "+name+" due
                            to no right constructor for link
                            class "+className);
                }
                model.addElement(link);
                model.addConnectivity(link.getName(), points
                    [0].getName(), points[1].getName());

        }

        private void parseNetworkElement(org.w3c.dom.Element
             node) throws ModelDescriptionException {
        String name = node.getAttribute(ATTR_NAME);
```

```
               String className = node.getAttribute(
                   ATTR_CLASS);
               Class elementClass;
               try {
                       elementClass = Class.forName(
                           className);
                       Constructor constructor =
                           elementClass.getConstructor(new
                           Class[]{String.class});
                       Object obj = constructor.newInstance
                           (new Object[]{name});
                       model.addElement((sector.element.
                           Element) obj);
               } catch (Exception e) {
                       throw new ModelDescriptionException(
                           "Can not parse network element
                           due to "+e.getMessage(),e);
               }
   }



   private Document getXMLDocument(String file) throws
       ModelerConfigException {
       Helper.checkString(file, "file");
       DocumentBuilderFactory docFactory =
           DocumentBuilderFactory.newInstance();
       try {
                       DocumentBuilder builder = docFactory
                           .newDocumentBuilder();
                       return builder.parse(new File(file))
                           ;
               }catch(Exception e){
                       throw new ModelerConfigException("
                           Can not get the specified XML
                           document from "+file,e);
               }
   }


}
```

B.3 Package sector.registrator - Event Registration

B.3.1 DefaultEventSpec.java

```

package sector.registrator;

import java.io.File;
import java.lang.reflect.Constructor;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

import sector.EventSpecException;
import sector.EventSubscriber;
import sector.Helper;

/**
 * <p>DefaultEventSpec class is the implementation of
     EventSpec interface. It can either parse
 * the event specification from a XML file or
     programatically create an event specification.</p>
 *
 */
public class DefaultEventSpec implements sector.EventSpec {


private static final String NAME = "name";
private static final String CLASS = "class";
private static final String EVENT = "Event";

private static final String DEFINITION = "Definition";
private static final String SUBSCRIBER = "Subscriber";

private static final String NAMESPACE = "namespace";

```

```

/**
 * <p>Represents the xml document which contains the event
      specification</p>
 */
    private Document xmlDoc = null;

    //private String alarmName = null;
    //private Map<String, Class> alarmProperties = new
        HashMap<String, Class>();

    /** The variable events contains all information related
         to events defined in the specification.
      * It includes their names and definitions. It is a
         LinkedHashMap in order to preserve the order.
      * Note that the event definition order is important to
         determine the validity of one event specification
      */
    private Map<String, String> events = new LinkedHashMap<
        String, String>();
    //private Map<String, EventSubscriber> subscribers = new
         HashMap<String, EventSubscriber>();

    //contains subscription information: event with its
        associated subscribers
    private Map<String, Set<EventSubscriber>> subscription =
         new HashMap<String, Set<EventSubscriber>>();


/**
 * <p>Construct an instance which parses event specification
      from a xml document and stores it</p>
 *
 * @param file The name of that XML document
 * @throws EventSpecException It will be thrown when the
      sytax or semantic of evnet specification is invalid
 * @throws EventSpecConfigException It is thrown when the
      provided XML document can not be found
 */
    public  DefaultEventSpec(String file) throws
        EventSpecConfigException, EventSpecException {
         Helper.checkString(file, "file");
         xmlDoc = getXMLDocument(file);
         parseSpecification(xmlDoc);
         xmlDoc = null;
    }

```

```java
/**
 * <p>Construct an instance to represent an event
      specification, the info about event specification can be
 * programmaticlly added into this instance</p>
 */
    public  DefaultEventSpec() {
        //do nothing
    }


    public Map<String,String> getEvents() {

        return this.events;
    }


    public Map<String, Set<EventSubscriber> >
        getSubscriptions() {

        return this.subscription;
    }




    /**
     *
     */
    public void addEvent(String name, String def)throws
        EventSpecException {
        Helper.checkString(name,"name");
        Helper.checkString(def,"def");
        //fixme if the event name is already there
        if(this.events.containsKey(name)){
                //System.out.println("Warning: an event "+
                    name+" is already defined and its
                    definition is overwritten" +
                        //       " now.");
                throw new EventSpecException("Warning: an
                    event "+name+" is already defined");
        }
        this.events.put(name, def);

    }

    /**
     * Add a subscription to that specification.
     * @param eventName
     * @param subscriber
     * @throws EventSpecException
     */
    public void addSubscription(String eventName, EventSubscriber subscriber) throws EventSpecException {
        Helper.checkString(eventName,"eventName");
        Helper.checkNull(subscriber,"subscriber");

        //if there is that event
        if(containsEvent(eventName)){
                Set<EventSubscriber> subscribers = (Set<EventSubscriber>) getSubscription(eventName);
                //If it is the first subscriber of that event
                if(subscribers == null){
                        subscribers = new HashSet<EventSubscriber>();
                }

                subscribers.add(subscriber);
                this.subscription.put(eventName, subscribers);
                //System.out.println("addSubscription one");
        }

        else{
                throw new EventSpecException("Subscription is not correct since event "+eventName+" does not exist!");
        }

    }


    public boolean containsEvent(String eventName) {
        Helper.checkString(eventName, "eventName");
        return this.events.containsKey(eventName);
    }

    /**
     * Get an event's subscription information by listing all its subscribers
     * @param event
     * @return
     */
    public Set<EventSubscriber> getSubscription(String event
        ) {
        Helper.checkString(event, "event");
        return this.subscription.get(event);
    }

    public String getEventDef(String eventName) {
        Helper.checkString(eventName, "eventName");
        return this.events.get(eventName);
    }

    public void removeEvent(String eventName) {
        Helper.checkString(eventName, "eventName");

        //if there is that event
        if(containsEvent(eventName)){
                //first remove that event definition
                this.events.remove(eventName);

                //then remove the subscription of that event
                removeSubscription(eventName);
        }

    }

    /**
     * Remove all current subscribers from that event
     * @param eventName
     */
    public void removeSubscription(String eventName) {
        Helper.checkString(eventName, "eventName");
        Set<EventSubscriber> subscribers = this.
            getSubscription(eventName);

        if(subscribers!=null){
                subscribers.clear();
                subscribers = null;
                this.subscription.remove(eventName);
        }

    }

    public void removeSubscription(String eventName,
        EventSubscriber subscriber) {
        Helper.checkString(eventName, "eventName");
        Helper.checkNull(subscriber, "subscriber");
        Set<EventSubscriber> subscribers = this.getSubscription(eventName);

        if(subscribers!=null){
                subscribers.remove(subscriber);
        }

    }

    public boolean isSubscribledBy(String event, EventSubscriber subscriber) {
        Set<EventSubscriber> subscribers = getSubscription(event);
        if(subscribers!= null){
                for(EventSubscriber sub:subscribers){
                        if(sub==subscriber){
                                return true;
                        }
                }
        }
        return false;
    }


/**
 * <p>Parse the XML file</p>
 * @param xmlDoc
 * @throws EventSpecException
 */

        private void parseSpecification(Document xmlDoc) throws EventSpecException {
        org.w3c.dom.Element root = xmlDoc.getDocumentElement();
                //NodeList nodes = root.getElementsByTagName(EVENT);

                Element[] elements = Helper.getDirectElementsByTagName(root, EVENT);
                //System.out.println("length "+elements.length);
                for(Element element : elements){
                        parseEvent(element);
                }

                //parse subscribers
```

```
			//nodes = root.getElementsByTagName(
			    SUBSCRIBER);
			elements = Helper.getDirectElementsByTagName
			    (root, SUBSCRIBER);
			for(Element element : elements){
				parseSubscriber(element);
			}

			/*nodes = root.getElementsByTagName(
			    SUBSCRIPTION);
			for(int i= 0; i<nodes.getLength();i++){
				parseSubscription((org.w3c.dom.
				    Element)nodes.item(i));
			}*/

	}



	/**
	 * Parse subscription information according subscriber
	     definition defined in xml file
	 * @param element
	 * @throws EventSpecException If the subscriber is
	     specified incorrectly or in a unexpected format
	 */
		private void parseSubscriber(org.w3c.dom.Element
		    element) throws EventSpecException {
			//String subscriberName = element.
			    getAttribute(NAME);
			String className = element.getAttribute(
			    CLASS);
			String namespace = element.getAttribute(
			    NAMESPACE);

			EventSubscriber subscriber;
			try {
				Class classInstance = Class.forName(
				    className);
				//check if the class is an
				    implementation of Model interface
				classInstance.asSubclass(
				    EventSubscriber.class);
				Constructor con;

				//no namespace is specified, thus
				    the constructor without any
```

```
                    parameters is invoked
                if(namespace== null||namespace.trim
                    ().length()==0){
                        con = classInstance.
                            getConstructor(new Class
                            [0]);
                        subscriber = (
                            EventSubscriber) con.
                            newInstance(new Object
                            [0]);
                        //fixme
                        //System.out.println("
                            parseSubscriber"+
                            className);
                }
                //invoke the constructor with a
                    String input parameter
                else{
                        con = classInstance.
                            getConstructor(new Class
                            []{String.class});
                        subscriber = (
                            EventSubscriber) con.
                            newInstance(new Object[]{
                            NAMESPACE});
                }

        } catch(ClassCastException e){
                throw new EventSpecException("The
                    specified class "+className+" is
                    not an implementation of
                    interface sector.EventSubscriber"
                    ,e);
        }catch (Exception e) {
                throw new EventSpecException("Can
                    not instantiate a new subscriber
                    due to "+e.getMessage(),e);
        }

        //get all events that subscriber subscribes
        NodeList eventNodes = element.
            getElementsByTagName(EVENT);
        for(int i = 0; i<eventNodes.getLength(); i
            ++){
                String eventName = ((Element)
                    eventNodes.item(i)).getAttribute(
                    NAME);
```

```java
            try{
                addSubscription(eventName, subscriber);
            }catch(Exception e){
                throw new EventSpecException("This new subscriber can not subscribe to event "+eventName+
                                        " due to "+e.getMessage(),e);
            }

        }

    }

/**
 * Parse event information from xml file
 * @param element
 * @throws EventSpecException If the event is specified incorrectly or in a unexpected format
 */
    private void parseEvent(org.w3c.dom.Element element) throws EventSpecException {
        //get event name
        String eventName = element.getAttribute(NAME);
        //get event definition
        NodeList nodes = element.getElementsByTagName(DEFINITION);

        //can not find event definition
        if(nodes.getLength()==0){
            throw new EventSpecException("Without definition for event "+eventName);
        }

        org.w3c.dom.Node defNode = nodes.item(0);
        String def = Helper.getTextContents(defNode);

        addEvent(eventName,def);
    }

```

```
        /**
         * Parse the content of the given file as an XML
             document and return a new DOM Document object
         * @param file The name of xml document
         * @return the DOM Document object
         * @throws EventSpecConfigException if the specified
             xml document does not exist
         */
        private Document getXMLDocument(String file) throws
            EventSpecConfigException {
                Helper.checkString(file, "file");
                DocumentBuilderFactory docFactory =
                    DocumentBuilderFactory.newInstance();
                try {
                        DocumentBuilder builder = docFactory
                            .newDocumentBuilder();
                        return builder.parse(new File(file))
                            ;
                }catch(Exception e){
                        throw new EventSpecConfigException("
                            Can not get the specified XML
                            document from "+file,e);
                }
        }


}
```

B.3.2 DefaultEventRegistrator.java

```

package sector.registrator;

import java.util.Map;
import java.util.Set;

import sector.EventRegistrationException;
import sector.EventRegistrator;
import sector.EventSubscriber;
import sector.Helper;
import net.esper.client.EPAdministrator;
import net.esper.client.EPServiceProvider;
import net.esper.client.EPStatement;

/**
 * <p>DefaultEventRegistrator class is the default
     implementation of EventRegistrator provided in this
     sector component.
 *
 * It contains one member variable 'epAdmin', which is the
     Administrative interface to the Esper engine. Through
     that interface,
 *
 * event definition defined in EQL patterns and EQL
     statements can be registered into Esper engine. The
     subscriber components which will be
 *
 * informed when certain events occur will be registered
     into Esper engine through that interface.</p>
 */
public class DefaultEventRegistrator implements
    EventRegistrator {

        //private EPServiceProvider epService;
        private EPAdministrator epAdmin;

/**
 * <p></p>
 *
 * @param spec
 */
    public void register(sector.EventSpec spec) throws
        EventRegistrationException{
        Map<String,String> events = spec.getEvents();

        for(String name: events.keySet()){
                String eventDef = events.get(name);
                EPStatement statement = null;
                try{
                        statement = (EPStatement) epAdmin.
                            createEQL(eventDef);
                }catch(Exception e){
                        throw new EventRegistrationException
                            ("Failed to register event "+name
                            +" due to "+e.getMessage(),e);
                }

                //System.out.println("name: "+name);
                Set<EventSubscriber> subscribers = spec.
                    getSubscription(name);
                if(subscribers!=null){
                        for(EventSubscriber subscriber:
                            subscribers){
                        statement.addListener(subscriber);
                    }

                    }

            }
        }

/**
 * <p>Constructor</p>
 */
        public  DefaultEventRegistrator(EPServiceProvider
             epService) {
             Helper.checkNull(epService,"epService");
             //this.epService = epService;
             this.epAdmin = epService.getEPAdministrator();
        }
 }
```

B.4 Package sector.adaptor - Event Adaptation

B.4.1 CSVEventAdaptor.java

```

package sector.adaptor;

import java.util.HashMap;
import java.util.Map;

import net.esper.adapter.AdapterInputSource;
import net.esper.adapter.InputAdapter;
import net.esper.adapter.csv.CSVInputAdapter;
import net.esper.adapter.csv.CSVInputAdapterSpec;
import net.esper.client.EPServiceProvider;
import sector.EventAdatporRunTimeException;
import sector.Helper;

import com.topcoder.util.config.ConfigManager;
import com.topcoder.util.config.Property;

/**
 * <p>CSVEventAdaptor class is the implementation of
      EventAdaptor interface. It gets alarms from a CSV file.
 *
 * It is a wrapper of Esper InputAdapter instance. Thus,
      this class simply calles methods provided by Esper
 * InputAdapter instance to offer funcationality defined in
     interface EventAdaptor.
 * </p>
 */
public class CSVEventAdaptor implements sector.EventAdaptor
    {

        /*<p>The Esper InputAdapter instance which provide
             the actual service</p>*/
        private CSVInputAdapterSpec spec;

        /* The alias of alarm stream generating from the CSV
              file. This alias is registered in Esper Engine
             */
        private String alias;

        /* This map variable stores all attributes of one
             alarm log*/
        private Map<String, Class> eventProperties = new
             HashMap<String, Class>();


        /* The strings in the configuration file */
        private static final String SOURCE = "Source";
        private static final String EVENTALIAS = "EventAlias
             ";
        private static final String TIMESTAMPCOLUMN = "
             TimestampColumn";
        private static final String ALARM = "Alarm";



/**
 * <p>Constructor</p>
 * @param nameSpace
 * @throws CSVEventAdaptorCreationException
 */
    public  CSVEventAdaptor(String nameSpace) throws
         CSVEventAdaptorCreationException {
        Helper.checkString(nameSpace,"nameSpace");
        ConfigManager cm = ConfigManager.getInstance();
        String file = null;
                try {
                        file = cm.getString(nameSpace,
                             SOURCE);

                        Property alarmFormat = cm.
                            getPropertyObject(nameSpace,
                            ALARM);
                        if(alarmFormat == null){
                                throw new
                                    CSVEventAdaptorCreationException
                                    ("Do not define the
                                    format of alarms!");
                        }
                        parseEventProperties(alarmFormat);
                        alias = cm.getString(nameSpace,
                            EVENTALIAS);


                        spec = new CSVInputAdapterSpec(new
                            AdapterInputSource(file), alias);

                        String timeStampColumn = cm.
                            getString(nameSpace,
                            TIMESTAMPCOLUMN);
                        if(timeStampColumn!=null&&
                            timeStampColumn.trim().length()
                            >0){
                                spec.setTimestampColumn(
                                    timeStampColumn);
                        }

                } catch(CSVEventAdaptorCreationException e){
                        throw e;

                }catch (Exception e) {
                        throw new
                            CSVEventAdaptorCreationException(
                            "Faild to create CSVEventAdaptor
                            instance due to "+e.getMessage(),
                            e);
                }


        }

        /**
         * <p>Start the sending of events into the Esper egine.
             </p>
         * @param epService The reference to Esper Egnie
         * @throws EventAdatporRunTimeException
         */
```

```
    public void start(EPServiceProvider epService) throws
        EventAdatporRunTimeException {
                Helper.checkNull(epService, "epService");
        try{
                InputAdapter inputAdapter = new
                    CSVInputAdapter(epService, spec);
                inputAdapter.start();
        }catch(Exception e){
                throw new sector.
                    EventAdatporRunTimeException ("Errors
                    occur during processing the events: "+e.
                    getMessage(),e);
        }
    }

    public String getEventAlias(){
        return this.alias;
    }

    public Map<String,Class> getEventProperties(){
        return eventProperties;
    }

    private void parseEventProperties(Property
        alarmProperties) throws ClassNotFoundException {
        java.util.Enumeration names =  alarmProperties.
            propertyNames();

        while(names.hasMoreElements()){
                String name = (String) names.nextElement();
                //String name = ((Property)property).
                    getValue(NAME);
                String classNM = alarmProperties.getValue(
                    name);
                Class type = getType(classNM);
                //System.out.println("name "+name+" class "+
                    classNM);
                this.eventProperties.put(name, type);
        }

    }

        private Class getType(String classNM) throws
            ClassNotFoundException {
                if(classNM==null){
                        throw new ClassNotFoundException("
                            The type of property should be
```

```
                                specified");
                }

                if(classNM.equals("int")){
                        return int.class;
                }
                else if(classNM.equals("double")){
                        return double.class;
                }

                else if(classNM.equals("float")){
                        return float.class;
                }
                else if(classNM.equals("boolean")){
                        return boolean.class;
                }
                else if(classNM.equals("char")){
                        return char.class;
                }
                else if(classNM.equals("byte")){
                        return byte.class;
                }
                else if(classNM.equals("short")){
                        return short.class;
                }else if(classNM.equals("long")){
                        return long.class;
                }

                return Class.forName(classNM);
        }

}
```

B.5 Package sector.test - Unit Test Classes

B.5.1 MemModelImplTest.java

```
package sector.test;

import sector.ModelDescriptionException;
import sector.element.BTS;
import sector.element.BaseRadio;
import sector.element.EBTS;
import sector.element.Element;
import sector.element.IllegalComponentException;
```

```java
import sector.element.IllegalLinkException;
import sector.element.IllegalManagedObjException;
import sector.element.RFSiteControlPath;
import sector.element.BTSManager;
import sector.element.ZoneController;
import sector.model.MemModelImpl;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
/**
 * Tests the behavior of MemModelImpl class
 *
 * @author  Xin Hu
 * @version 1.0
 */
public class MemModelImplTest extends TestCase {

        //the MemModelImpl instance to test on
        private MemModelImpl model;

        protected void setUp() throws Exception {
                model = new MemModelImpl();
        }

        protected void tearDown() throws Exception {
                model.clear();
                model = null;
        }

        /**
     * <p>
     * Creates a test suite of the tests contained in this
        class.
     * </p>
     * @return a test suite of the tests contained in this
        class.
     */
    public static Test suite() {
        return new TestSuite(MemModelImplTest.class);
    }
        public void testConstructor(){
                assertNotNull(model);
        }

        /*
         * Test to add one BaseRadio element into the model.
                After the add, this element can be retrieved by
         * getElement method provide that its name (ID)
         */
        public void testAddBaseRadio() throws Exception{
                Element element = new BaseRadio("br");
                model.addElement(element);

                assertEquals(element, model.getElement(element.getName()));
        }

        /*
         * Test to add two BaseRadio elements with the same name(ID) into the model.
         *
         * The 2nd add will fail
         */
        public void testAddBaseRadioWithSameNames() throws Exception{
                Element element = new BaseRadio("br");
                model.addElement(element);

                assertEquals(element, model.getElement(element.getName()));

                Element duplicate = new BaseRadio("br");
                try{
                        model.addElement(duplicate);
                        fail("Should throw ModelDescriptionException");
                }catch(ModelDescriptionException e){

                }

        }

        /*
         * Test to add one BTS element into the model. After the add, this element can be retrieved by
         * getElement method provide that its name (ID)
         */
        public void testAddBTS() throws Exception{
                Element element = new BTS("bts");
                model.addElement(element);

                assertEquals(element, model.getElement(element.getName()));
        }


        /*
         * Test to add one good RFSiteControlPath element
             into the model. After the add, this element can
             be retrieved by
         * getElement method provide that its name (ID)
         */
        public void testAddRFSiteControlPath() throws
            Exception{
                //one link is considered to be in the model
                    only its two endpoints are in the model
                //Thus, construct two endpoints first
                BTS endpoint1 = new BTS("bts");
                ZoneController endpoint2 = new
                    ZoneController("zc");
                model.addElement(endpoint1);
                model.addElement(endpoint2);

                //then construct the RFSiteControlPath
                Element path = new RFSiteControlPath("RFPath
                    ",endpoint1,endpoint2);
                model.addElement(path);

                assertEquals(path, model.getElement(path.
                    getName()));
        }

        /*
         * Test to add one bad RFSiteControlPath element
             into the model.
         * This RFSiteControlPath is bad because none of its
              endpoints are pre-existed
         */
        public void testAddBadRFSiteControlPath() throws
            Exception{
                //one link is considered to be in the model
                    only its two endpoints are in the model
                BTS endpoint1 = new BTS("bts");
                ZoneController endpoint2 = new
                    ZoneController("zc");
                //the construction is ok
                Element path = new RFSiteControlPath("RFPath
                    ",endpoint1,endpoint2);

                //but the two endpoints are not in model
                    since they have not been added by calling
                model.addElement()
            //exception will be thrown
            try{
                model.addElement(path);
                fail("Should be
                    ModelDescriptionException");
            }catch(ModelDescriptionException e){

            }
        }

        /**
         * Test to add one legal management relationship
         */
        public void testAddManagement() throws Exception{
            Element zconBTS = new BTSManager("zconbts");
            Element bts = new BTS("bts");
            model.addElement(zconBTS);
            model.addElement(bts);

            model.addManagement(zconBTS.getName(), bts.
                getName());

            assertTrue(model.isManagedBy(bts.getName(),
                zconBTS.getName()));
        }

        /**
         * Test to add one bad management relationship: a
             base radio manages a bts
         */
        public void testAddBadlManagement1() throws
            Exception{
            Element br = new BaseRadio("br");
            Element bts = new BTS("bts");
            model.addElement(br);
            model.addElement(bts);

            try{
                model.addManagement(br.getName(),
                    bts.getName());
                fail("should throw
                    ModelDescriptionException");
            }catch(ModelDescriptionException e){
            }
        }

        /**
         * Test to add one bad management relationship: a
             ZConBTS manages a base radio
         */
        public void testAddBadManagement2() throws Exception
            {
                Element br = new BaseRadio("br");
                Element zconBTS = new BTSManager("zconbts");
                model.addElement(br);
                model.addElement(zconBTS);

                try{
                        model.addManagement(zconBTS.getName
                            (),br.getName());
                        fail("should throw
                            ModelDescriptionException");
                }catch(ModelDescriptionException e){
                        assertTrue(e.getCause() instanceof
                            IllegalManagedObjException);
                }
        }

        /**
         * Test to add one bad management relationship: a
             ZConBTS manages a non-existing BTS
         */
        public void testAddBadManagement3() throws Exception
            {
                Element zconBTS = new BTSManager("zconbts");
                model.addElement(zconBTS);

                try{
                        model.addManagement(zconBTS.getName
                            (),"non-existing bts");
                        fail("should throw
                            ModelDescriptionException");
                }catch(ModelDescriptionException e){
                }
        }

        /**
         * Test to add one good management relationship: a
             BTS contains a base radio
         */
        public void testAddGoodContainment() throws
            Exception{
                Element bts = new BTS("zconbts");
                model.addElement(bts);

                Element br = new BaseRadio("br");
                model.addElement(br);

                model.addContainment(bts.getName(), br.
                    getName());

                assertTrue(model.isContainedIn(br.getName(),
                     bts.getName()));
        }

        /**
         * Test to add one bad management relationship: a
             base radio contains a BTS
         */
        public void testAddBadContainment() throws Exception
            {

                Element br = new BaseRadio("br");
                model.addElement(br);

                Element bts = new BTS("zconbts");
                model.addElement(bts);

                //illegal management relationship: a base
                    radio contains a BTS
                try{
                        model.addContainment(br.getName(),
                            bts.getName());
                        fail("Should be
                            ModelDescriptionException");
                }catch(ModelDescriptionException e){
                        assertTrue(e.getCause() instanceof
                            IllegalComponentException);
                }

        }

        /**
         * Test to add one good connectivity relationship: a
              Zone Controller connects to
         * a ebts via the RF control path
         */
        public void testAddGoodConnectivity() throws
            Exception{
                //create a BTS element and add it into model
                BTS bts = new BTS("bts");
                model.addElement(bts);

                //create a ZoneController element and add it into model
                ZoneController zc = new ZoneController("zc");
                model.addElement(zc);

                //create a RFSiteControlPath element and add it into model
                Element link = new RFSiteControlPath("RFPath",bts,zc);
                model.addElement(link);

                //add good connectivity relationship among the link, bts, zc elements created before
                model.addConnectivity(link.getName(), bts.getName(), zc.getName());


                //test after adding the connectivity relationship
                //a. link is connected to bts and zc
                assertTrue(model.isConnectedTo(link.getName(), bts.getName()));
                assertTrue(model.isConnectedTo(link.getName(), zc.getName()));
                //b. bts and zc are connected via link
                assertTrue(model.isConnectedVia(bts.getName(),link.getName()));
                assertTrue(model.isConnectedVia(zc.getName(),link.getName()));
                //c. bts and zc are connected
                assertTrue(model.isConnectedTo(bts.getName(), zc.getName()));
                assertTrue(model.isConnectedTo(zc.getName(), bts.getName()));
        }

        /**
         * Test to add one bad connectivity relationship: a Zone Controller tries to connect to a base radio
         * via the RF control path
         */
        public void testAddBadConnectivity1() throws Exception{
                //create a BTS element and add it into model
                BTS bts = new BTS("bts");
                model.addElement(bts);

                //create a ZoneController element and add it
                     into model
                ZoneController zc = new ZoneController("zc")
                    ;
                model.addElement(zc);

                //create a RFSiteControlPath element and add
                     it into model
                Element link = new RFSiteControlPath("RFPath
                    ",bts,zc);
                model.addElement(link);

                //create a Base Radio element and add it
                    into the model
                Element br = new BaseRadio("br");
                model.addElement(br);

                //add a bad relationship: a Zone Controller
                    tries to connect to a base radio via the
                    RF site control path
                try{
                        model.addConnectivity(link.getName()
                            , br.getName(), zc.getName());
                        fail("Should throw
                            ModelDescriptionException");
                }catch(ModelDescriptionException e){
                        assertTrue(e.getCause() instanceof
                            IllegalLinkException);
                }

        }

        /**
         * Test to add one bad connectivity relationship: a
              Zone Controller tries to connect to a EBTS
         * via a base radio, which is not a link element
         */
        public void testAddBadConnectivity2() throws
            Exception{
                //create a BTS element and add it into model
                BTS bts = new BTS("bts");
                model.addElement(bts);

                //create a ZoneController element and add it into model
                ZoneController zc = new ZoneController("zc");
                model.addElement(zc);

                //create a Base Radio element and add it into the model
                Element br = new BaseRadio("br");
                model.addElement(br);

                //add a bad relationship: a Zone Controller tries to connect to a base radio via the RF site control path
                try{
                        model.addConnectivity(br.getName(), zc.getName(), bts.getName());
                        fail("Should throw ModelDescriptionException");
                }catch(ModelDescriptionException e){
                }

        }

        /**
         * Test to add one bad connectivity relationship: a Zone Controller tries to connect to a RF site control path, which is not a node element
         * via another RF site control path
         */
        public void testAddBadConnectivity3() throws Exception{
                //create a BTS element and add it into model
                BTS bts = new BTS("bts");
                model.addElement(bts);

                //create a ZoneController element and add it into model
                ZoneController zc = new ZoneController("zc");
                model.addElement(zc);

                Element path = new RFSiteControlPath("path",bts,zc);
                model.addElement(path);

```

```
                //add a bad relationship: a Zone Controller tries to connect to a RF site control path, which is not a node element
                //via another RF site control path
                try{
                        model.addConnectivity(path.getName(), zc.getName(), path.getName());
                        fail("Should throw ModelDescriptionException");
                }catch(ModelDescriptionException e){
                }

        }

        /**
         * Test get an element from the model by providing its name
         */
        public void testGetElement() throws Exception{
                //create a BTS element and add it into model
                BTS bts = new BTS("bts");
                model.addElement(bts);

                assertEquals(model.getElement(bts.getName()),bts);

        }

        /**
         * Test get an element from the model by providing its name
         */
        /*public void testRemoveElement() throws Exception{
                //create a BTS element and add it into model
                BTS bts = new BTS("bts");
                model.addElement(bts);

                assertEquals(model.getElement(bts.getName()),bts);

        }*/


        /**
         * Test to remove a containment relationship from model
         */
        public void testRemoveContainment() throws Exception
            {
                //create a Base Radio and add it into model
                BaseRadio br = new BaseRadio("br");
                model.addElement(br);

                //create a EBTS and add it into model
                EBTS ebts = new EBTS("ebts");
                model.addElement(ebts);
                //add the contaimnet relationship
                model.addContainment(ebts.getName(), br.
                    getName());
                //so the isContainedIn should return true
                assertTrue(model.isContainedIn(br.getName(),
                     ebts.getName()));

                //call removeContaiment()
                model.removeContainment(ebts.getName(), br.
                    getName());
                //now the inContainedIn should return false
                assertFalse(model.isContainedIn(br.getName()
                    , ebts.getName()));

        }

        /**
         * Test to remove a management relationship from
             model
         */
        public void testRemoveManagement() throws Exception{
                //create a EBTS and add it into model
                EBTS ebts = new EBTS("ebts");
                model.addElement(ebts);

                //create a ZConEBTS and add it into model
                BTSManager zconebts = new BTSManager("
                    zconebts");
                model.addElement(zconebts);

                //add the management relationship
                model.addManagement(zconebts.getName(), ebts
                    .getName());
                //so the isManagedBy should return true
                assertTrue(model.isManagedBy(ebts.getName(),
                    zconebts.getName()));

                //call removeManagment()
                model.removeManagement(zconebts.getName(),
                    ebts.getName());
                //now the isManagedBy should return false
                assertFalse(model.isManagedBy(ebts.getName()
                    ,zconebts.getName()));

        }

        /**
         * Test to remove a connectivity relationship from
             model
         */
        public void testRemoveConnectivity() throws
            Exception{
                //create a EBTS and add it into model
                EBTS ebts = new EBTS("ebts");
                model.addElement(ebts);

                //create a Zone Controller and add it into
                    model
                ZoneController zc = new ZoneController("zc")
                    ;
                model.addElement(zc);

                //create a link which is from zc to ebts and
                     add it into model
                String linkName = "link";
                RFSiteControlPath link = new
                    RFSiteControlPath(linkName,ebts,zc);
                model.addElement(link);

                //add the connectivity relationship
                model.addConnectivity(link.getName(), ebts.
                    getName(), zc.getName());

                //so the isConnectedTo should return true
                assertTrue(model.isConnectedTo(ebts.getName
                    (),zc.getName()));
                assertTrue(model.isConnectedTo(link.getName
                    (),zc.getName()));
                assertTrue(model.isConnectedTo(link.getName
                    (),ebts.getName()));

                //so the isConnectedVia should return true
                assertTrue(model.isConnectedVia(zc.getName()
                    , link.getName()));
                assertTrue(model.isConnectedVia(ebts.getName
                    (), link.getName()));

                //call removeConnectivity
                model.removeConnectivity(link.getName(),zc.
                    getName(), ebts.getName());
                //now the isConnectedTo should return false
                assertFalse(model.isConnectedTo(ebts.getName
                    (),zc.getName()));
                assertFalse(model.isConnectedTo(linkName,zc.
                    getName()));
                assertFalse(model.isConnectedTo(linkName,
                    ebts.getName()));
                assertFalse(model.isConnectedVia(zc.getName
                    (), linkName));
                assertFalse(model.isConnectedVia(ebts.
                    getName(),linkName));
                assertNull(model.getElement(linkName));

        }

        /**
         * Test a if the model can know a component's type
             acoording to the component's name
         */
        public void testIsTypeOf() throws Exception{
                //create a Base Radio and add it into model
                BaseRadio br = new BaseRadio("br");
                model.addElement(br);

                //The isTypeOf should return true since the
                    component is a base radio
                assertTrue(model.isTypeOf(br.getName(),
                    BaseRadio.class.getName()));

                assertFalse(model.isTypeOf("no this
                    component",BaseRadio.class.getName()));
                assertFalse(model.isTypeOf(br.getName(),EBTS
                    .class.getName()));

                //create a EBTS and add it into model
                EBTS ebts = new EBTS("ebts");
                model.addElement(ebts);

                //The isTypeOf should return true since the
                    component is a base radio
```

```java
                    assertTrue(model.isTypeOf(ebts.getName(),
                        EBTS.class.getName()));

                    //create a Zone Controller and add it into
                        model
                    ZoneController zc = new ZoneController("zc")
                        ;
                    model.addElement(zc);

                    //The isTypeOf should return true since the
                        component is a base radio
                    assertTrue(model.isTypeOf(zc.getName(),
                        ZoneController.class.getName()));

                    //create a Zone Controller and add it into
                        model
                    RFSiteControlPath link = new
                        RFSiteControlPath("link",zc,ebts);
                    model.addElement(link);

                    //The isTypeOf should return true since the
                        component is a base radio
                    assertTrue(model.isTypeOf(link.getName(),
                        RFSiteControlPath.class.getName()));


        }

        //fixme miss test onremoveElement


}
```

B.5.2 DefaultEventSpecTest.java

```java
package sector.test;

import java.io.File;
import java.util.Set;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import sector.EventSpecException;
import sector.EventSubscriber;
import sector.client.BRAlert;
import sector.registrator.DefaultEventSpec;
```

```

/**
 * Tests the behavior of DefaultEventSpec class
 *
 * @author  Xin Hu
 * @version 1.0
 */
public class DefaultEventSpecTest extends TestCase {

    /**
     * The xml config file storing event specification.
     */
    private File file = null;

    /**
     * The DefaultEventSpec to test on.
     */
    private DefaultEventSpec eventSpec = null;

    /**
     * Set up testing environment.
     *
     * @throws Exception to JUnit.
     */
    protected void setUp() throws Exception {
        eventSpec = new DefaultEventSpec();

    }

    /**
     * Tear down testing environment.
     *
     * @throws Exception to JUnit.
     */
    protected void tearDown() throws Exception {
        eventSpec = null;
        if(file!=null){
                file.delete();
                file = null;
        }
    }

    /**
     * <p>
     * Creates a test suite of the tests contained in this
         class.
     * </p>
     * @return a test suite of the tests contained in this class.
     */
    public static Test suite() {
        return new TestSuite(DefaultEventSpecTest.class);
    }

    /**
     * Test create DefaultEventSpec by calling DefaultEventSpec(String)
     */
    public void testGoodConstructor() throws Exception {
        String eventName1 = "BRLocked";
        String def1 = "insert into BRLocked select * from AlarmLog" +
                        "where message like '%(3)%DISABLED%(3004)%LOCKED%'  and Predicater.isBaseRadio(nodename)";

        String eventName2 = "EBTSDown";
        String def2 = "insert into EBTSDown select * from AlarmLog "+
                        "                where Predicater.isEBTS(nodename) and message like '%(31)%NO TRUNKING%(31004)%NO CONTROL CHANNEL%' ";

        String eventName3 = "ZCEBTSDown";
        String def3 = "         insert into ZCEBTSDown select * from AlarmLog "+
                "                where Predicater.isZConBTS(nodename) and message like '%(101)%NOT WIDE TRUNKING%(101005)%NO CONTROL CHANNEL%' ";

        String eventName4 = "BRLockedAlert";
        String def4 = "         insert into BRLockedAlert(nodename,message,event_time) select A.nodename, A.message, A.event_time " +
                        "       from pattern [every (A=BRLocked -> B=ZCEBTSDown -> C=EBTSDown where timer:within(30 sec) )] where " +
                        "       Predicater.isContainedIn(A.nodename,C.nodename) and Predicater.isManagedBy(C.nodename, B.nodename)  ";

        String content = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" +
                        "<EventSpecification> "+
                        "  <Event name='"+eventName1+"'> "+
                        "              <Definition> "+ def1 +
                        "       </Definition>"+
                        "  </Event>"+
                        "  <Event name='"+eventName2+"'>"+
                        "              <Definition>"+ def2+
                        "              </Definition> "+
                        "  </Event> "+
                        "  <Event name='"+eventName3+"'> "+
                        "              <Definition> "+ def3 +
                        "              </Definition>   " +
                        "</Event> "+

                        "<Event name='BRLockedAlert'> " +
                        "       <Definition> " + def4+
                        "       </Definition> " +
                        "</Event> " +

                        "<Subscriber class ='sector.client.BRDown'> " +
                        "       <Event name ='"+eventName1+"' />              " +
                        "       <Event name ='"+eventName2+"' />              " +
                        "       <Event name ='"+eventName3+"' />       " +
                        "</Subscriber>" +

        "<Subscriber class ='sector.client.BRAlert'> " +
        "       <Event name ='"+eventName4+"' /> " +
        "</Subscriber> " +
        "</EventSpecification>";
        file = TestHelper.createTestXMLFile(this.getName(), content);
        this.eventSpec = new DefaultEventSpec(file.getAbsolutePath());

        assertNotNull(this.eventSpec);

        assertTrue(eventSpec.containsEvent(eventName1));
        assertTrue(eventSpec.containsEvent(eventName2));
        assertTrue(eventSpec.containsEvent(eventName3));
        assertTrue(eventSpec.containsEvent(eventName4));

        assertTrue(eventSpec.getEventDef(eventName1).indexOf
            (def1)>=0);
        assertTrue(eventSpec.getEventDef(eventName2).indexOf
            (def2)>=0);
        assertTrue(eventSpec.getEventDef(eventName3).indexOf
            (def3)>=0);
        assertTrue(eventSpec.getEventDef(eventName4).indexOf
            (def4)>=0);

        Set <EventSubscriber> subscribers = eventSpec.
            getSubscription(eventName1);
        assertTrue(subscribers.iterator().next() instanceof
            sector.client.BRDown);

        subscribers = eventSpec.getSubscription(eventName2);
        assertTrue(subscribers.iterator().next() instanceof
            sector.client.BRDown);

        subscribers = eventSpec.getSubscription(eventName3);
        assertTrue(subscribers.iterator().next() instanceof
            sector.client.BRDown);

        subscribers = eventSpec.getSubscription(eventName4);
        assertTrue(subscribers.iterator().next() instanceof
            sector.client.BRAlert);
    }

    /**
     * Test create DefaultEventSpec by calling
         DefaultEventSpec(String) with wrong config file:
         subscriber
     * try to subscribe to the non-existing events
     */
    public void testBadConstructor1() throws Exception {

        String content = "<?xml version=\"1.0\" encoding=\"
            UTF-8\"?>" +
        "<EventSpecification> "+

                        "<Subscriber class ='sector.client.
                            BRDown'> " +
                        "       <Event name ='non-existing'
                            />          " +
                        "</Subscriber>" +

        "</EventSpecification>";
        file = TestHelper.createTestXMLFile(this.getName(),
            content);

        try{
                this.eventSpec = new DefaultEventSpec(file.
                    getAbsolutePath());
                fail("Should EventSpecException");
        }catch(EventSpecException e){

        }

    }

    /**
     * Test create DefaultEventSpec by calling
         DefaultEventSpec(String) with wrong config file: the
          specified subscriber
     * is not a tyep of sector.EventSubscriber
     */
    public void testBadConstructor2() throws Exception {
        String eventName1 = "BRLocked";
        String def1 = "insert into BRLocked select * from
            AlarmLog" +
                        "where message like '%(3)%DISABLED
                            %(3004)%LOCKED%'  and Predicater.
                            isBaseRadio(nodename)";

        String content = "<?xml version=\"1.0\" encoding=\"
            UTF-8\"?>" +
        "<EventSpecification> "+
        "  <Event name='"+eventName1+"'> "+
                "               <Definition> "+ def1+
                "       </Definition>"+
                "  </Event>"+

                        "<Subscriber class ='wrongclass'> "
                            +
                        "       <Event name ='"+eventName1+"
                            ' />                " +
                        "</Subscriber>" +

        "</EventSpecification>";
        file = TestHelper.createTestXMLFile(this.getName(),
            content);

        try{
                this.eventSpec = new DefaultEventSpec(file.
                    getAbsolutePath());
                fail("Should EventSpecException");
        }catch(EventSpecException e){

        }

    }

    /**
     * Test create DefaultEventSpec by calling
         DefaultEventSpec(String) with wrong config file: the
          event definition is missing
     */
    public void testBadConstructor3() throws Exception {
        String eventName1 = "BRLocked";

        String content = "<?xml version=\"1.0\" encoding=\"
            UTF-8\"?>" +
        "<EventSpecification> "+
        "  <Event name='"+eventName1+"'> "+
                "  </Event>"+

                        "<Subscriber class ='wrongclass'> "
                            +
                        "       <Event name ='"+eventName1+"
                            ' />                  " +
                        "</Subscriber>" +

        "</EventSpecification>";
        file = TestHelper.createTestXMLFile(this.getName(),
            content);

        try{
                this.eventSpec = new DefaultEventSpec(file.
                    getAbsolutePath());
                fail("Should EventSpecException");
        }catch(EventSpecException e){

        }

    }

    /**
     * Test addEvent by providing valid parameter
     */
    public void testGoodAddEvent() throws Exception {
        String eventName = "test";
        String definition = "insert into test select * from
            AlarmLog";

        //call addEvent()
        eventSpec.addEvent(eventName, definition);

        //test after calling addEvent()
        assertEquals(definition,eventSpec.getEventDef(
            eventName));
        assertTrue(eventSpec.containsEvent(eventName));

    }

    /**
     * Test addEvent by providing invalid parameter: the
         added event has the same name as a particular event
     * which is already in event specification
     */
    public void testBadAddEvent() throws Exception {

        //create one event and add it into event
            specification
        String eventName = "test";
        String definition = "insert into test select * from
            AlarmLog";
        eventSpec.addEvent(eventName, definition);

        //create another event with the same name as the
            first one's and add it into event specification
        String eventName2 = "test";
        String definition2 = "insert into test select * from
             AlarmLog where name = 'baseradio' ";
        try{
                eventSpec.addEvent(eventName2, definition2);
                fail("should be EventSpecException");
        }catch(EventSpecException e){

        }

    }

    /**
     * Test addSubscription by providing valid parameter
     */
    public void testGoodAddSubscription() throws Exception {

        //create one event and add it into event
            specification
        String eventName = "test";
        String definition = "insert into test select * from
            AlarmLog";
        eventSpec.addEvent(eventName, definition);

        //create one event subcriber
        EventSubscriber subscriber = new BRAlert() ;

        //call addSubscription
        eventSpec.addSubscription(eventName, subscriber);
        assertTrue(eventSpec.isSubscribledBy(eventName,
            subscriber));

        //try to add one more event subscriber
        EventSubscriber subscriber2 = new BRAlert() ;
        eventSpec.addSubscription(eventName, subscriber2);
        assertTrue(eventSpec.isSubscribledBy(eventName,
            subscriber2));


    }

    /**
     * Test addSubscription by providing invalid parameter:
         the event that one subscriber wants to
     * subscribe does not exist
     */
    public void testBadAddSubscription() throws Exception {

        //create one event subcriber
        EventSubscriber subscriber = new BRAlert() ;

        try{
                //try to bind this subscriber with a non-
                    exist event
                eventSpec.addSubscription("non-exist",
                    subscriber);
                fail("should be EventSpecException");
        }catch(EventSpecException e){

        }
    }

    /**
```

```
     * Test removeEvent
     */
    public void testRemoveEvent() throws Exception {
        String eventName = "test";
        String definition = "insert into test select * from
            AlarmLog";

        //call addEvent()
        eventSpec.addEvent(eventName, definition);

        //test after calling addEvent()
        assertTrue(eventSpec.containsEvent(eventName));

        eventSpec.removeEvent(eventName);
        //now the event specification does not contain that
            event
        assertFalse(eventSpec.containsEvent(eventName));
        //and there should no subscribers subscribing that
            event
        assertNull(eventSpec.getSubscription(eventName));

    }

    /**
    * Test removeSubscription
    */
    public void testRemoveSubscription() throws Exception {

        //create the event definition and add it into event
            specification
        String eventName = "test";
        String definition = "insert into test select * from
            AlarmLog";
            //call addEvent()
        eventSpec.addEvent(eventName, definition);

        //create the subscriber which is subscribing that
            event
        EventSubscriber subscriber = new BRAlert();
        eventSpec.addSubscription(eventName, subscriber);

        //test prior to calling removeSubscription()
        assertNotNull(eventSpec.getSubscription(eventName));


        eventSpec.removeSubscription(eventName);
        //after calling removeSubscription()
        assertNull(eventSpec.getSubscription(eventName));
        assertFalse(eventSpec.isSubscribedBy(eventName,
            subscriber));

    }



}
```

B.6 Package sector.test.integration.suppression - Test Classes for Console Login Failed scenario

B.6.1 LoginFailedTest.java

```
package sector.test.integration.suppression;

import java.util.Date;

import sector.Sector;
import sector.SectorCreationException;
import sector.test.TestHelper;

public class LoginFailedTest {

        public LoginFailedTest(){

        }

        public static void main(String[] args){
                try {

                        TestHelper.loadMultipleXMLConfig("
                            sector.Sector", "test_files/
                            LoginFailedConfig.xml");
                        TestHelper.loadMultipleXMLConfig("
                            sector.model.XMLModeler", "
                            test_files/LoginFailedConfig.xml"
                            );
                        TestHelper.loadMultipleXMLConfig("
                            sector.adaptor.CSVEventAdaptor",
                            "test_files/LoginFailedConfig.xml
                            ");

                        Sector sector = new Sector("sector.
                            Sector");
                        sector.buildModel();

                        sector.setUpPredicater();
                        sector.registerEvent();
                        System.out.println(new Date()+"-----
                            Start test-----");
                        sector.start();
                } catch (SectorCreationException e) {
                        e.printStackTrace();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

}
```

B.6.2 LoginFailedAlert.java

```
package sector.test.integration.suppression;

import java.util.Date;

import net.esper.event.EventBean;
import sector.EventSubscriber;

public class LoginFailedAlert implements EventSubscriber {
        public void update(EventBean[] newEvents, EventBean
            [] oldEvents) {
                if(newEvents == null){
                        return;
                }

                 EventBean event = newEvents[0];
              System.out.println(new Date()+"#########
                  Detected a fault#########:\nConsole("
                         + event.get("nodename").toString()
                            +") login failed at "+event.get(
                            "event_time"));

        }

}
```

B.7 Package `sector.test.integration.br` - Test Classes for Base Radio Locked scenario

Note that only important class is listed

B.7.1 BaseRadioLockedAlert.java

```
package sector.test.integration.br;

import java.util.Date;

import net.esper.event.EventBean;
import sector.EventSubscriber;

public class BaseRadioLockedAlert implements EventSubscriber
     {
        public void update(EventBean[] newEvents, EventBean
            [] oldEvents) {
                if(newEvents == null){
                        return;
                }

                 EventBean event = newEvents[0];
             System.out.println(new Date()+"#########
                 Detected a fault#########:\nBase Radio " +
                 event.get("nodename").toString() +
                          " is disabled");

        }

}
```

B.8 Package `sector.test.integration.ebts` - Test Classes for EBTS Disabled scenario

Note that only important classes are listed

B.8.1 BothSitePathDownAlert.java

```
package sector.test.integration.ebts;

import java.util.Date;

import net.esper.event.EventBean;
import sector.EventSubscriber;

public class BothSitePathDownAlert implements
    EventSubscriber {
        public void update(EventBean[] newEvents, EventBean
            [] oldEvents) {
                if(newEvents == null){
                        return;
                }

                 EventBean event = newEvents[0];
              System.out.println(new Date()+"---Detected an
                  event---:\nBoth RFSiteControlPaths (" +
                  event.get("path1").toString() +
                              " and "+event.get("path2").
                                  toString()+") are down" +
                                                  "\nevent_time="+
                                                      event.get("time2"
                                                      ));

        }

}
```

B.8.2 EBTSDisabledAlert.java

```
package sector.test.integration.ebts;

import java.util.Date;

import net.esper.event.EventBean;
import sector.EventSubscriber;

public class EBTSDisabledAlert implements EventSubscriber {
        public void update(EventBean[] newEvents, EventBean
            [] oldEvents) {
                if(newEvents == null){
                        return;
                }

                 EventBean event = newEvents[0];
                System.out.println(new Date()+"#########
                    Detected a fault#########:\nEBTS " + event.
                    get("nodename").toString() +
                                " is disabled");

          }

}
```

Appendix C

XML description files

C.1 The DTD of XML model description file

```
<!DOCTYPE NetworkConfig [

<!ELEMENT NetworkConfig (Element+, Connectivity, Containment, Management)>

<!-- NE declaration section-->
<!ELEMENT Element >
<!ATTLIST Element name CDATA #REQUIRED>
<!ATTLIST Element class CDATA #REQUIRED>

<!-- connectivity dependency section-->
<!ELEMENT Connectivity (Link*)>
<!ELEMENT Link (Point+)>
<!ATTLIST Link name CDATA #REQUIRED>
<!ATTLIST Link class CDATA #REQUIRED>
<!ELEMENT Point >
<!ATTLIST Point name CDATA #REQUIRED>

<!-- Containment dependency section-->
<!ELEMENT Containment (Container*)>
```

```
<!ELEMENT Container (Component+)>
<!ATTLIST Container name CDATA #REQUIRED>
<!ELEMENT Component >
<!ATTLIST Component name CDATA #REQUIRED>

<!-- Management dependency section-->
<!ELEMENT Management (Manager*)>
<!ELEMENT Manager (Managed+)>
<!ATTLIST Manager name CDATA #REQUIRED>
<!ELEMENT Managed >
<!ATTLIST Managed name CDATA #REQUIRED>
```

C.2 The DTD of XML event specification file

```
<!DOCTYPE EventSpecification [

<!ELEMENT EventSpecification (Event+, Subscriber+)>

<!-- event definition section-->
<!ELEMENT Event (Definition) >
<!ATTLIST Event name CDATA #REQUIRED>
<!ELEMENT Definition (#PCDATA) >


<!-- event subscription section-->
<!ELEMENT Subscriber (Event+)>
<!ATTLIST Subscriber class CDATA #REQUIRED>
```

Appendix D

Testing

All the tests are based on a basic Dimetra system. This system contains a zone controller and a EBTS site which are connected via two site control links. Furhermore, that EBTS site contains a base radio and has a manager. This system can be modeled in an XML document as the follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<NetworkConfig>
<Element name='EbtsBaseRadio_1.1:zone11'
class='sector.element.BaseRadio'/>

<Element name='ZC1:zone11'
class='sector.element.ZoneController'/>

<Element name='Zcz11ebts01:zone11'
 class='sector.element.BTSManager'/>

<Element name='z11ebts01:zone11'
 class='sector.element.EBTS'/>

<Connectivity>
  <Link name='RFSiteControlPath_0.1.1:zone11'
class='sector.element.RFSiteControlPath'>
```

```
  <Point name='z11ebts01:zone11'/>
  <Point name='ZC1:zone11'/>
  </Link>
  <Link name='RFSiteControlPath_0.1.2:zone11'
class='sector.element.RFSiteControlPath'>
          <Point name='z11ebts01:zone11'/>
          <Point name='ZC1:zone11'/>
  </Link>
</Connectivity>

<Containment>
  <Container name='z11ebts01:zone11'>
<Component name='EbtsBaseRadio_1.1:zone11'/>
  </Container>
</Containment>

<Management>
  <Manager name='Zcz11ebts01:zone11'>
<Managed name='z11ebts01:zone11'/>
  </Manager>
</Management>

</NetworkConfig>
```

D.1 Console Login Failed Testing

D.1.1 Event Definitions

- **LoginFailed**

  ```
  insert into LoginFailed select * from AlarmLog
  where
  message like '%Audit: User Login Failed:%The last opera-
  tion was CONSOLE Login Failed.'
  and nodename = 'ccgw01.vortex1.zone11'
  ```

- **SuppressedLoginFailed**

  ```
  insert into SuppressedLoginFailed
  select * from LoginFailed output first every 300 seconds
  ```

D.1.2 Results

Refer to Fig. 7.3.

D.2 Base Radio is Locked Testing

D.2.1 Event Definitions

- **BRLocked**

```
insert into BRLocked select * from AlarmLog
where message like '%(3)%DISABLED%(3004)%LOCKED%'
and Predicater.isBaseRadio(nodename)
```

- **EBTSDown**

```
insert into EBTSDown select * from AlarmLog
where Predicater.isEBTS(nodename) and
message like '%(31)%NO TRUNKING%(31004)%NO CONTROL CHANNEL%'
```

- **ZCEBTSDown**

```
insert into ZCEBTSDown select * from AlarmLog
where Predicater.isBTSManager(nodename) and
message like '%(101)%NOT WIDE TRUNKING%(101005)%NO CONTROL CHANNEL%'
```

- **BRLockedAlert**

```
insert into BRLockedAlert(nodename,message,event_time)
select A.nodename, A.message, A.event_time from
pattern [every (A=BRLocked -> B=ZCEBTSDown -> C=EBTSDown
where timer:within(30 sec) )]
where Predicater.isContainedIn(A.nodename,C.nodename)
and Predicater.isManagedBy(C.nodename, B.nodename)
```

D.2.2 Results

Refer to Fig. 7.4.

D.3 EBTS is Disabled Testing

D.3.1 Event Definitions

- **SitePathDown**

```
insert into SitePathDown select * from AlarmLog
where message like ’%(1)%CONFIGURING%(1003)%CONNECTION DOWN%’
and Predicater.isRFSiteControlPath(nodename)
```

- **ActiveBackupSitePath**

```
insert into ActiveBackupSitePath select * from AlarmLog
where message like ’%(3)%ACTIVE%(3006)%PREFERRED N/A%’ and
Predicater.isRFSiteControlPath(nodename)
```

- **BothSitePathDown**

```
insert into BothSitePathDown(path1,message1,time1,path2,message2,
time2)
select A.nodename, A.message, A.event_time, C.nodename, C.message,
C.event_time
from pattern [every (A=SitePathDown -> B=ActiveBackupSitePath ->
C=SitePathDown where timer:within(5 sec) )]
where A.nodename != B.nodename and B.nodename = C.nodename
```

- **ZCEBTSPathDown**

```
insert into ZCEBTSPathDown select * from AlarmLog
where Predicater.isBTSManager(nodename) and
message
like ’%(101)%NOT WIDE TRUNKING%(101006)%SITE CONTROL PATH DOWN%’
```

- **EBTSUnreacherable**

```
insert into EBTSUnreacherable select * from AlarmLog
where Predicater.isEBTS(nodename) and message
like ’%(9994)%UNKNOWN%(9994)%UNREACHABLE FROM MANAGER (SNMP TIMEOUT)%’
```

- **EBTSDownAlert**

```
insert into EBTSDownAlert(nodename,message,event_time)
select C.nodename, C.message, C.event_time from
```

```
pattern [every (A=BothSitePathDown -> B=ZCEBTSPathDown ->
C=EBTSUnreacherable where timer:within(75 sec) )]
where Predicater.isConnectedTo(A.path1,C.nodename) and
Predicater.isConnectedTo(A.path2,C.nodename) and
Predicater.isManagedBy(C.nodename, B.nodename)
```

D.3.2 Results

Refer to Fig. 7.5.

Bibliography

[1] K. Houck, S. Calo, 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 [86].

[2] M. Steinder and A. S. Sethi, *A Survey of Fault Localization Techniques in Computer Networks.* [Elsevier] Science of Computer Programming, S.I. on Network and System Administration, 2004.

[3] G. Liu, A.K. Mok, E.J. Yang, *Composite events for network event correlation*, in: M. Sloman, S. Mazumdar, E. Lupu (Eds.), Integrated Network Management VI, IEEE, 1999, pp. 247-260 [89].

[4] M. Hasan, B. Sugla, and Viswanathan R., *A Conceptual Framework for Network Management Event Correlation and Filtering Systems*, IEEE/IFIP Symposium on Integrated Network Management, 1999, 233-246.

[5] Dilmar Malheiros Meira, *A Model For Alarm Correlation in Telecommunications Networks*, Ph.D. Thesis, Federal University of Minas Gerais, 1997.

[6] Gabriel Jakobson, Mark D.Weissman, *Alam Correlation*, IEEE Network, 1993.

[7] G. Jakobson, M.D. Weissman, *Real-time telecommunication network management: Extending event correlation with temporal constraints*, in: A.S. Sethi, F. Faure-Vincent, Y. Raynaud (Eds.), Integrated Network Management IV, Chapman and Hall, London, 1995, pp. 290-302 [86].

[8] S. Kätker, M. Paterok, *Fault isolation and event correlation for integrated fault management*, in: A. Lazar, R. Sarauo, R. Stadler (Eds.), Integrated Network Management V, Chapman and Hall, London, 1997,pp. 583-596 [60].

[9] M. Steinder, 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 [91].

[10] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, 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 [86].

[11] S.A. Yemini, S. Kliger, E. Mozes, Y. Yemini, D. Ohsie, *High speed and robust event correlation*, IEEE Communications Magazine 34 (5) (1996) 82-90.

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

[13] P. Wu, R. Bhatnagar, L. Epshtein, M. Bhandaru, Z. Shi, *Alarm correlation engine (ACE).* In Proc. Network Operation and Management Symposium, NOMS'98, New Orleans, LA, 1998, pp. 733-742 [77].

[14] M. Klemettinen, H. Mannila, H. Toivonen, *Rule discovery in telecommunication alarm data*, Journal of Network and Systems Management 7 (4) (1999) 395-423.

[15] 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 [36]

[16] S. Sengupta, A. Dupuy, J. Schwartz, O. Wolfson, Y. Yemini, *The Netmate Model for Network Management*, in: IEEE Network Operations and Management Symposium (NOMS), San Diego, CA, 1990, pp. 11-14.

[17] A. Dupuy, S. Sengupta, O. Wolfson, Y. Yemini, *Design of the Netmate network management system*, in: I. Krishnan, W. Zimmer (Eds.), Integrated Network Management II, North-Holland, Amsterdam, 1991, pp. 639-650 [59].

[18] Denise W. Gurer, Irfan Khan, Richard Ogier, *An Artificial Intelligence Approach to Network Fault Management*, http://citeseer.ist.psu.edu/105695.html

[19] Terrestrial Trunked Radio, http://en.wikipedia.org/wiki/Terrestrial_Trunked_Radio

[20] Motorola System Documentation, *Dimetra IP Compact/ Scalable Dimetra IP - Understanding Your System (6802800U76-A)*, September 2006

[21] Motorola System Documentation, *Dimetra IP Compact/ Scalable Dimetra IP - Fault Management (6802800U77-A)*, September 2006

[22] Motorola System Documentation, *Dimetra IP Compact/ Scalable Dimetra IP - Diagnostics and Troubleshooting (6802800U81-A)*, September 2006

[23] Extensible Markup Language (XML), http://en.wikipedia.org/wiki/XML

[24] Esper, http://esper.codehaus.org/

[25] SQL, http://en.wikipedia.org/wiki/SQL

[26] Java Technology, http://java.sun.com/

[27] Unified Modeling Language, http://www.uml.org/

[28] *Poseidon for UML*, http://www.gentleware.com/

[29] *Rational Rose*, http://www-306.ibm.com/software/awdtools/developer/rose/

[30] *Eclipse*, http://www.eclipse.org/

[31] *Design pattern (computer science)*, http://en.wikipedia.org/wiki/Design_pattern_(computer_science)

[32] *Strategy pattern*, http://en.wikipedia.org/wiki/Strategy_pattern

[33] *Observer pattern*, http://en.wikipedia.org/wiki/Observer_pattern

[34] *Java Interface*, http://en.wikipedia.org/wiki/Interface_(Java)

[35] *Java package*, http://en.wikipedia.org/wiki/Java_package

[36] *White box testing*, http://en.wikipedia.org/wiki/White_box_testing

[37] *JUnit*, http://www.junit.org/index.htm

[38] *Black box testing*, http://en.wikipedia.org/wiki/Black_box_testing

[39] *Java keyword: static*, http://java.sun.com/docs/books/tutorial/java/javaOO/classvars.html

[40] 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 [36].

[41] *Simple Network Management Protocol*, http://en.wikipedia.org/wiki/Simple_Network_Management_Protocol

[42] R.D. Gardner, D.A. Harle, *Methods and systems for alarm correlation*, in: Proc. of GLOBECOM, London, UK, November 1996, pp. 136-140.

[43] Context-free grammar,
http://en.wikipedia.org/wiki/Context-free_grammar

[44] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. *Failure diagnosis using decision trees.* In Proc. Intl. Conference on Autonomic Computing, New York, NY, 2004

[45] S. Russell, *Machine learning*, in: M.A. Boden (Ed.), Artificial Intelligence, second ed., Handbook of Perception and Cognition, Academic Press, New York, 1996, pp. 89-133 (Chapter 4)