City University, London
Department of Computing

Diagnosing Runtime Violations of Security & Dependability Properties

Theocharis Tsigkritis

Submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in Computing at City University, London, April 2010

TABLE OF CONTENTS

List of Tables

List of Figures

Acknowledgements

This thesis is the result of a long going process which revealed the author's weaknesses and sensitivity to external phenomena. Therefore, the author would like to thank the people that supported, encouraged and assisted him to overcome all the occurred obstacles. Without these people, the present thesis would have not been existed.

Firstly, the author would like to extend his gratitude and appreciation to his supervisor and mentor, Professor George Spanoudakis. Professor G. Spanoudakis was constantly and patiently encouraging, supporting and revealing multiple ways for addressing any issues that happened to occur by respecting the author's judgment and freedom of choice. His insistence and emphasis on analyzing, evolving and expanding thoughts and ideas was quite painful sometimes, however it was a way to awake the author's focus and awareness when necessary. The discussions between Professor G. Spanoudakis and the author will be unforgettable for the author. For the short number of reasons mentioned above, author is honoured to have cooperated with Professor G. Spanoudakis.

In addition to his supervisor, the author would like to express his appreciation to his co-supervisor Dr. Christos Kloukinas for his valuable advices, his constant support and his relaxing way to discuss any emerged issue.

Two more people that the author would like to express his sincere thankfulness are Dr. Costas Lambrinoudakis and Dr. Stefanos Gkritzalis. Without their prompt and counsel, the author would have never embarked upon the adventure behind this thesis. The author would like also to give his thanks to Dr. Andrea Zisman for her support and concern especially in the last phase of the thesis preparation, and Dr. Artur d'Avila Garcez for his valuable comments during the initial stages of the research work presented in this thesis.

The author would like to thank the School of Informatics for the financial support for this research, as well, as the members of the Technical Support Team (TST) in the School of Informatics for their constant and tireless services.

In addition to the above people, there are numerous colleagues and friends who were extremely helpful over the years, each in their own way. These include Nikos Konstantoudakis, Dr. Khaled Mahbub, Dr. Waraporn Zirapathong, Dr. Gilberto

Cysneiros Filho, Dr. Shant Narsesian, Mark Firman, Amalia, Andreas and Eirini Spyrou, Dr. Kelly Androutsopoulos, Dr. Jameel Syed, Anestis Benavidis, Dr. Vasiliki Efstathiou, Anna Thanou, Leonidas Skoutas, Dr. Davide Lorenzoli, Dr. Marco Comuzzi, Ricardo Contreras, Dr. Igor Siveroni, Kostas Poulios, Dr. Eirini Nedelkopoulou, Stavros Fakanas, and Stelios Papakonstantinou.

Besides all the aforementioned people, the author feels mostly obliged to his family. The author therefore is especially grateful to his parents, Athanasios and Loukia, for supporting him in any possible way, whether it was love or advice, wherever and whenever. Also, the author would like to express his sincere thankfulness to his brother and cousin, Petros T. and Petros P., whose joyfulness made author's hard times smoother, and his aunt Marina, who hosted him in Kefallonia - a vitalising and rejuvenating piece of land for the author. It would be impossible for the author to overstate how valuable the contributions of his family were over these years.

Given the chance, the author would like to dedicate this thesis to his grandparents, who are not with him in this life anymore. The author dedicates humbly and equally this piece of work to the memory of Petros and Anastasia expressing his true gratitude for everything that had lavishly given to him, and to the memory of Theocharis and Sofia, even if the author did not have the chance to meet them alive and live with them. Both couples of author's grandparents are always in author's mind considered as the very origin of his life.

Last but not least, in addition to all people the author has thanked, the author would like to thank humbly God for all the strength and enlightenment He had revealed within the author during difficult times.

Declaration

The author grants powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies make for study purposes, subject to normal conditions of acknowledgement.

Abstract

Monitoring the preservation of security and dependability (S&D) properties of complex software systems is widely accepted as a necessity. Basic monitoring can detect violations but does not always provide sufficient information for deciding what the appropriate response to a violation is. Such decisions often require additional diagnostic information that explains why a violation has occurred and can, therefore, indicate what would be an appropriate response action to it. In this thesis, we describe a diagnostic procedure for generating explanations of violations of S&D properties developed as extension of a runtime monitoring framewoek, called *EVEREST*. The procedure is based on a combination of abductive and evidential reasoning about violations of S&D properties which are expressed in *Event Calculus*.

Chapter 1: Introduction

1.1 Overview

Monitoring security and dependability (S&D) properties of software systems at runtime is widely accepted as a measure of increased resilience to dependability failures and security attacks, and several approaches have been developed to support it (see [99] for a survey). Whilst basic monitoring provides the core functionality for detecting violations of such properties, it cannot always provide the information that is necessary in order to understand the reasons that underpin the violation of a property and decide what would be an appropriate reaction to it.

In this thesis, we present a diagnosis system that we have developed as extension of a monitoring framework [109, 153, 155], called *EVEREST* (*EVE*nt *REaS*oning *T*oolkit). *EVEREST* supports the specification and monitoring of properties expressed in *Event Calculus* (EC) [149] as rules. The provision of diagnostic information is based on the generation of alternative *explanations* for the events which are involved in the violations of rules, and the assessment of the plausibility of these explanations based on whether their effects correspond to events recorded during the operation of the monitored system as already presented in [162, 163, 164]. The key characteristic of our approach is the use of abductive reasoning [42, 122, 136] for the generation of explanations and belief based reasoning [146] for the assessment of explanation plausibility.

1.2 The need for diagnosis

To appreciate the need for diagnosing the reasons underpinning violations of security and dependability properties, consider an air traffic management system, referred to as ATMS in the following. ATMS uses different radars to monitor the trajectories of airplanes in different air spaces. It is also connected with a system that keeps a record of flight plans which are submitted by different planes ahead of flights to indicate the expected route of a flight and request flight permission.

The operations of ATMS may be monitored at runtime to ensure the integrity of its components and the information generated by them. Monitoring, for example, may focus on properties related to: (i) the liveness of the radars connected to ATMS, and (ii) the generation of mutually consistent information by them. An example of property of this kind relates to cases where air spaces are covered by different radars or have overlapping

areas covered by different radars. In such cases, to check the integrity of the information that is provided by the different radars which cover an airspace, we can monitor a rule requiring that if one of these radars sends a signal indicating that an airplane is in the airspace, every other radar that covers the same space should also send a signal indicating the presence of the plane in it within a certain time period of time after the receipt of the initial signal. Such a rule is violated in all cases where the monitor receives a signal event by one of the radars of ATMS that cover a specific airspace but not the other. Clearly, whilst in such cases, knowing that the rule has been violated is important for the operation of ATMS but the violation report on its own is not sufficient for establishing the reasons why the second expected signal was not received and taking appropriate action (if possible). In fact, the violation may have been due to several reasons, including the following:

- The radar that did not send the expected signal was malfunctioning (Cause 1).
- The communication link between the radar that did not send the expected signal and the monitor was malfunctioning or an intruder captured the signal and prevented it from reaching the monitor (Cause 2).
- The radar that sent the expected signal was malfunctioning or its identity was faked by an intruder that sent a fake signal to the monitor (Cause 3).

Although the above list of possible causes is not exhaustive, it demonstrates that a decision about what would be an appropriate reaction to the violation depends on the reason(s) that have caused it and, therefore, the selection of the appropriate response action cannot be made solely on the basis of knowledge about the violation but requires additional diagnostic information. Therefore, identifying which of the above reasons has caused the violation is important for taking actions that would restore the integrity of the operation of ATMS.

1.3 Dynamic Verification and Diagnosis

Dynamic verification enables a software system to improve its dependability [14], by checking whether its behaviour satisfies specific dependability properties while it is running. On the other hand, diagnosis can be considered as the process whose aim is the identification of the causes of symptoms and observations might occur during the operation of an examined system. For instance, regarding the motivating example in

Section 1.2, dynamic verification solutions used in ATMS would be responsible to check at runtime the liveness of the radars connected to ATMS. In case that radar liveness violations are detected, diagnosis tools could be used for providing additional information by identifying the reasons have caused the detected violations.

However, in the context of software system dynamic verification, there are many approaches [22, 66, 128, 139, 161] that consider diagnosis as the identification of system observations, which resulted to a failure, fault or violation. For instance, in [22, 66, 128, 139, 161], synchronization of automata that model the expected behaviour of the monitored system and the generated event sets are used for carrying out diagnostic tasks. More specifically, Pencolé and Cordier [128] propose a decentralised fault diagnosis approach that synchronization of automata is performed for individual system components (fault detection at system components level), and then aggregated for the global system (fault detection at system level). Also, in both approaches given by Tripakis [161] and Bouyer et al. [22], the fault diagnosis problem is treated by generating algorithms (diagnosers) that act as fault detectors of internal faults for any given sequence of events generated by the system, which is modelled as a timed automaton [4].

In the present thesis, we assume a distinct separation between the dynamic verification task and the diagnosis task. The goal of monitoring task is the detection of inconsistencies with respect to the intended behaviour of a system with respect to some dependability properties. Therefore, the approaches [22, 66, 128, 139, 161] mentioned above could be considered as possible solutions to the monitoring task. An extended set of approaches that focus on system runtime monitoring, as for instance the work by Spanoudakis et al. [109, 153, 155] that the diagnostic approach presented in this thesis is built upon, are discussed below in the document, in particular in Chapter 2.

Upon inconsistencies with respect to the intended behaviour of the system detected by dynamic verification approaches, the diagnostic task comes to play for identifying the reasons that have caused the detected inconsistencies. As discussed in Section 1.1, it should be noted that the output of the diagnostic task is important for taking actions that would restore the integrity of the operation of the monitored system. Examples of research effort that focus on the diagnostic task, as the work by Console et al. [42] and Santos [140], are discussed again in Chapter 2.

1.4 The Diagnostic Approach

The overall aim of the diagnostic approach this thesis presents is the identification of possible *explanations* for the violations of S&D properties [162, 163, 164] in order to aid the selection of appropriate reactions to these violations. To generate such explanations, the diagnostic mechanism uses *abductive reasoning* [42, 122, 136]. Then, following the identification of possible explanations, the diagnosis mechanism also assesses the plausibility of explanations by identifying any effects that they would have generated from the explanations and checking whether these effects correspond to observations that have occurred and are genuine. The assessment of the genuineness of the explanation effects and the validity of explanations is based on the computation of beliefs using functions that we have defined for this purpose. These functions have been defined using the axiomatic framework of the Dempster-Shafer theory of evidence [146].

The diagnostic framework has been designed as an extention of *EVEREST* monitoring framework described in [109, 153, 155]. *EVEREST* is able to monitor whether the behaviour of a system is consistent with tis intended behaviour in parallel with the operation of the monitored system without intervening with this operation in any form. In *EVEREST*, the properties to be checked are specified as rules and ssumptions in a language called *EC-Assertion* [109, 153, 155] that has its formal foundation in *Event Calculus* [149]. *EVEREST* finally provides the infrastructure for storing properties violations and other observations occurring during the operation of the monitored system that are necessary inputs for the diagnostic process.

1.5 Contributions

The diagnostic approach this thesis presents is the result of the research work that evolved as shown throughout our earlier publicatations [162, 163, 164]. In these publications, the concept of event genuineness, as a criterion for event trustworthiness, and its assemssment mechanisms have been introduced and presented to the outer research world, starting from our initial intuitions, thoughts and formulations regarding diagnosis via event trustworthiness assessment and evolving to concrete evidential mechanisms in the same direction. What is presented in this thesis is the latest version of the aforementioned concept and its assessment mechanism after reconsiderations and corrections have been made. The main contributions of our research work are as follows:

- Design of a diagnostic framework for runtime S&D violations.

The diagnostic framework firstly presented in [162, 163, 164] and discussed in this thesis is designed to identify possible *explanations* for runtime S&D violations rules specified in *Event Calculus* [149] in order to aid the selection of appropriate reactions to these violations. The violations of S&D properties are detected by a non-intrusive monitoring[1] framework [109, 153, 155] at the runtime of the monitored system. The diagnostic framework processes the detected violations by taking into account other observations from the monitored system. The diagnostic information that is produced refers to the cause of the detected violations in terms of belief measurements in the genunieness of explanations effects and the validity of explanations that were generated for the examined violations.

- Abductive algorithm for generating explanations.
  To compute the possible explanations of runtime violations of S&D properties specified in *Event Calculus* [149], an algorithm that reasons abductively on the observations involved in the examined violations has been devised (see Section 5.2.1 and [162, 163, 164]). Another key characteristic of the abductive algorithm is that it reasons on the temporal aspects of violation observations against the intended behaviour of the examined system by treating any occurring time constraint satisfaction problem as linear programming problem. The output of the algorithm is a list representing the alternative explanations for a particular violation observation.

- Deductive algorithm for identifying expected consequences.
  To assess the plausibility and validity of the abductive explanations, the diagnosis process computes the explanations effects. For undertaking this task, a deductive algorithm that reasons on the abductive explanations against the intended behaviour of the system has been designed (see Section 5.3.1 and [162, 163, 164]). The deductive algorithm reasons again on the temporal aspects of the abductive explanations by treating any occurring time constraint satisfaction problem as linear programming problem. The output of the algorithm consists of

1 It should be noted that the term "non intrusive monitoring" here signifies a form of monitoring that is carried out by a computational entity that is external to the system that is being monitored, is carried out in parallel with the operation of this system and does not intervene with this operation in any form.

an exhaustive list of the observations that could be derived from the abductive explanations and the intended behaviour model of the system.

- Plausibility assessment scheme based on evidential reasoning.
  An assessment scheme for observation genuineness based on the plausibility and correctness of the alternative explanations that are generated for a violation observation in the previous steps of the diagnostic process has been designed (see Section 5.4). As presented in [162, 163, 164], the assessment of explanation plausibility is based on the hypothesis that if the expected effects of an explanation match with observations that have occurred (and recorded) during the operation of the system that is being monitored, then there is evidence about the validity of the explanation. However, there is the possibility that we would not be able to confirm or disconfirm the validity of the explanation at the time that diagnostic process searches for evidence. To deal with this uncertainty, the diagnosis mechanism advocates an approximate reasoning approach which generates degrees of belief in the membership of observations in the log of the monitor and the existence of some valid explanation for it rather than strict logical truth values. These degrees of belief are computed by functions founded in the axiomatic framework of the Dempster-Shafer theory of evidence [146].

- Implementation of a diagnostic prototype.
  A prototype based on the diagnostic framework this thesis presents has been implemented. The diagnostic prototype has been integrated with the underlying monitoring framework that is easy to set up and provides user flexibility in quite satisfying levels. The underlying monitoring framework extended with diagnostic capabilities allows the user to specify monitoring rules and assumptions of S&D properties and indicate whether diagnostic results are needed.

1.6 Outline of the Thesis

The rest of the thesis is structured as follows.

Chapter 2 provides the reader with an overview of security and dependability properties from the perspective of software system engineering and the technical background on security and dependability dynamic verification or runtime monitoring. This is important since the diagnosis approach discussed in this thesis focuses on security and dependability proeperties and is based on a runtime monitoring framework. Also,

Chapter 2 presents a short literature review on abductive reasoning, as our diagnosis approach uses abductive reasoning as a technique for generating explanations that are used as diagnostic information.

Chapter 3 focuses on the theoretical background that underpins our diagnosis approach. More specifically, Chapter 3 presents the language that the reasoning mechanisms of our diagnosis approach use. Also, the monitoring framework, which the diagnosis approach in based on, is discussed by highlighting the formal specifications the framework uses. Finally, a short overview of the axiomatic framework that the diagnosis approach uses for handling uncertainty with respect to possible explanations of S&D violations on the basis of the available evidence is also provided.

Chapter 4 discusses the basic formulation of the diagnostic task as it is carried out by the presented diagnosis approach. The basic formulation is provided as formal specifications extensions to the monitoring framework that the diagnosis approach is based on. Having provided the basic formal characteristics of the diagnostic task, Chapter 4 concludes by presenting the formal specifications of the ATMS motivating example that is discussed in Section 1.2. This example is used in following chapters in order to demonstrate the technical aspects of the diagnosis approach.

In Chapter 5, a detailed description of our diagnostic approach is given. More specifically, we give an overview of the diagnosis process and describe in detail the algorithms and mathematical formulas used to carry out the forms of analysis which are used in the different stages of this process.

Chapter 6 presents the set up and results of an experimental evaluation of the diagnostic prototype that was implemented as a proof of concept for our diagnostic approach. The experimental evaluation has been based on the monitoring of a simulated case study.

Chapter 7 provides the reader with some of our insights regarding the open issues that emerged from our work in diagnosis of security and dependability properties violations.

Finally, Chapter 8 concludes with the summary of the diagnosis approach that is described in this thesis, the contributions of the research underpinning it, as well as some limitations of the diagnostic approach and the prototype that implements it.

Chapter 2: Related Literature

2.1 Overview

The purpose of this chapter is firstly to give an overview of the notions of security and dependability properties from the perspective of software engineering and provide the readers with the technical background on security and dependability dynamic verification or runtime monitoring. Secondly, this chapter presents a short state of the art on abductive reasoning as a technique for generating explanations.

More specifically, Section 2.2 covers the technical background on dynamic verification or monitoring of system dependability and security by providing initially a short overview of the security and dependability properties themselves, mostly based on definitions given by Avizienis et al. [14]. Having provided definitions of security and dependability properties, we present critical analysis of current research on dynamic verification by presenting general purpose and security oriented dynamic verification approaches. We also present comparative discussion on the presented security and dependability dynamic verification lines of research.

Section 2.3 provides an overview of research in the area of abductive reasoning that it is a key characteristic of the approach presented by the present thesis. Therefore, by highlighting the basic aspects of abductive reasoning and the recent relevant research approaches, Section 2.3 aims to enable the reader to understand the relationship of our approach to abductive reasoning as a technique for generating explanations.

2.2 Dynamic Verification of S&D properties

The objective of this section is twofold: firstly to discuss about the notions of security and dependability properties, and secondly to provide a review of the state of the art in security and dependability dynamic verification or monitoring.

2.2.1 Security and Dependability Properties: An Overview

In order to be able to provide an overview of the security properties from the perspective of software engineering, it would be necessary and quite helpful to examine and discuss about the security requirements that appear across relevant bibliography [55, 62, 157]. According to [55, 62, 157], security requirements cover issues related to:

- *Confidentiality* is the ability to maintain the secrecy of data and the messages exchanged between a system and its collaborating actors over communication channels.
- *Integrity* is the ability to ensure the accuracy and completeness of the data stored and the messages exchanged by a system. Maintaining integrity involves allowing only authorised users to change or create data and messages and applying controls to ensure the correctness of these messages and data.
- *Availability* is concerned with ensuring that access to a system is possible when required.
- *Non-repudiation* is concerned with making it impossible for an entity that has participated in some communication with a system to deny this participation. In message exchange, for instance, non- repudiation guarantees that the sender and the receiver of a message cannot deny the dispatch and receipt of the message, respectively.
- *Authentication* is the ability to determine whether an actor interacting with the system has the identity that it claims to have.
- *Authorization* is concerned with the assignment of the right permissions to an already authenticated entity.
- *Privacy* is the ability of a system to prevent personal information from becoming known to entities other than those which own the information or the information is about.

Regarding dependability properties, Avizienis et al. [14] have defined dependability as "the ability of a (computer) system to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s)" and "deliver service that can be justifiably trusted". The notion of service in this definition corresponds to the system behaviour as viewed by the user, who may be a human interacting with the system or another system. A service delivery is acceptable if it implements the required system behaviour and satisfies certain quality constraints while failures relate to events that make the service deviate from what is perceived to be a correct delivery. According to Avizienis et al. [14], some of the aforementioned security requirements (confidentiality, availability and integrity) are considered as attributes of

dependability. In the same context, we could generalise and consider that security properties are a subset of the dependability properties set, as the security properties of a system aim to prevent from unaccepted leak of private information (man in the middle attack [106]) and/or unaccepted delays in the deliverance of the provided services (denial of service attack [2]).

An important element in the above definition of dependability is the notion of "justifiable trust" which requires the ability to objectively verify that the delivered system service does not deviate from the required system behaviour and associated quality constraints. The development of system verification capabilities (i.e., the ability to verify that a system satisfies certain properties) has been the focus of significant research over the last few decades and has resulted in the development of a wide spectrum of, typically tool-supported, methods that offer such capabilities. These methods are distinguished into *static* and *dynamic*.

Static verification methods aim to show that the desired properties of a system will always hold based solely on the specification of the system without considering its actual run-time behaviour. Dynamic verification methods, on the other hand, aim to show that desired properties hold based on observation of the run-time behaviour of a system and its interactions with its operational environment.

Due to the fact that static verification is not the main area of interest of this thesis, we are not providing an overview of methods that fall in this category. On the other hand, a brief overview of the dynamic verification of systems is following in the rest of the sections of the chapter.

2.2.2 Dynamic verification

Dynamic verification enables a software system to improve its dependability (and therefore security) [14], by checking whether its behaviour satisfies specific dependability and security properties while it is running. This can be accomplished by a software module, which monitors the execution of the system and checks its conformity with the specification of the relevant properties. This module can be either an external or an internal module of the monitored system.

Software systems are increasingly becoming ubiquitous and heterogeneous and rely on technologies such as mobile code and components off the shelf (COTS). Static

verification and testing of dynamically adapted entities cannot provide adequate results, each one for different reasons. Static verification is a formal method and can prove that a system (or to be more accurate its model) is correct but is very time consuming and demands substantial education and experience from practitioners. Testing [101] on the other hand is an informal method which cannot prove a system correct since it can never offer a complete coverage of all its possible executions but can be easily applied even from inexperienced practitioners.

Being situated somewhere between static verification and testing, dynamic verification techniques aim to achieve the benefits of both approaches, by merging testing and formal specification. Thus, dynamic verification is considered to be a formal method applied to the implementation of the system that avoids the pitfalls of ad hoc testing and the complexity of full blown static verification techniques (model checking, theorem proving).

Dynamic system verification has been investigated in the context of different areas including requirements engineering, program verification, safety critical systems and service centric systems.

According to literature on dynamic verification [20, 47, 75, 153], the basic stages of dynamic verification are: (i) the development of a formal specification of a system including various types of properties, like safety and security properties, (ii) the application of methods for capturing events of interest and (iii) checking for violations by a monitor which can verify whether the observed behaviour of a system satisfies the required properties.

It should be noted that there are cases such as Aspect Oriented Programming [90] and Monitoring Oriented Programming [34] in which a monitor is generated automatically and inserted into the code that has to be monitored. Thus, in such cases, the second stage includes the monitor generation as well. On the other hand, in all the other cases, monitors are considered to be software modules, which have to be implemented [12, 75] separately from the monitored system. The monitor inputs are the formal specification of the system (product of first stage) and the flow of events generated during the execution of the system. The monitor then reasons about the conformance of the captured runtime

behaviour of the system (events flow) against the indented system behaviour (formal specification).

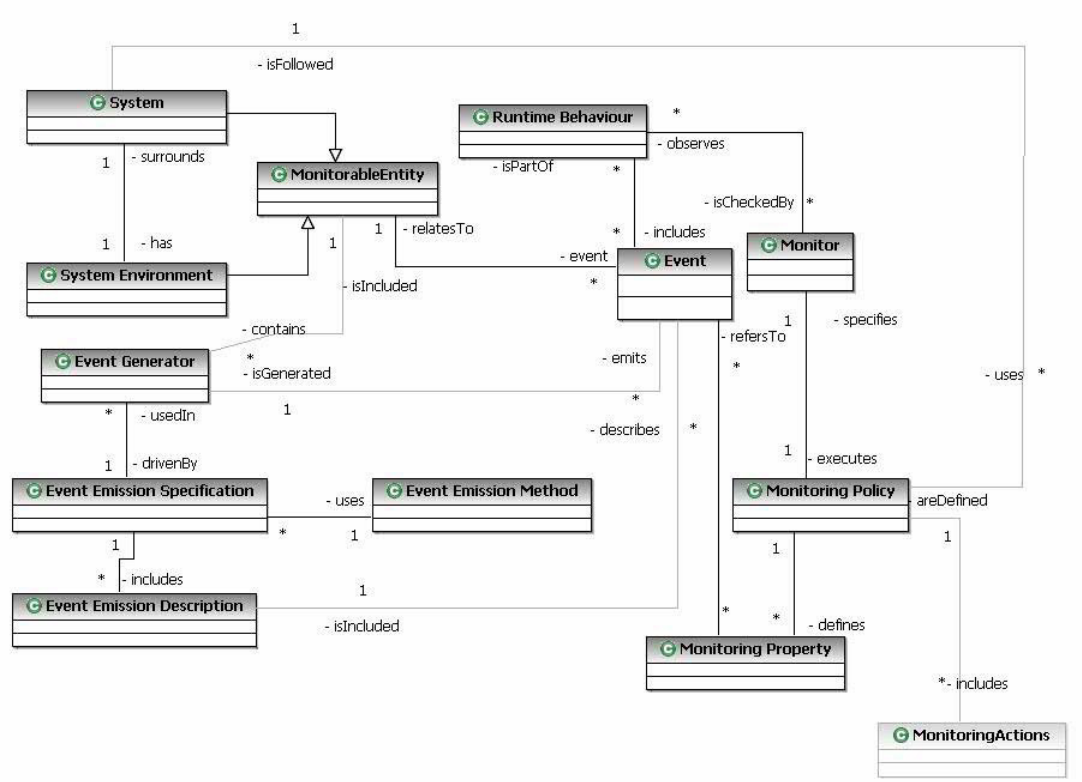

Figure 2-1 – Conceptual Model for Dynamic Verification

Figure 2-1 shows the conceptual model we have constructed to indicate the entities involved in dynamic verification. According to this model, the subject of dynamic verification that is signified by the class *MonitorableEntity* can be either a *System* or a *System's Environment*. Dynamic verification is carried out by a *Monitor* which observes the *Runtime Behaviour* of a system or its environment. The *RuntimeBehaviour* is a set of *events* generated during the operation of the monitorable entities. These events are generated by one or more *Event Generator* according to different *Event Emission Specifications*. An event emission specification describes the particular *Event Emission Method* to be used and one or more *Event Emission Descriptions*, which describe the exact types of events which should be generated. The observation of the events in a *Runtime Behaviour* by the *Monitor* is carried out according to a specific *Monitoring Policy* which specifies the *Monitoring Properties* that should be verified at runtime and the set of *Monitoring Actions* the *Monitor* should perform to enable the system control and/or recover from violations of the monitoring properties.

Figure 2-2 presents a taxonomy of monitor and event generation features. This taxonomy has three layers which differentiate monitoring and event generation capabilities according to (a) the controlling capabilities of a monitor, (b) the time of the event emission with respect to the occurrence of the action described by the event, and (c) the communication type between the monitor and the system.

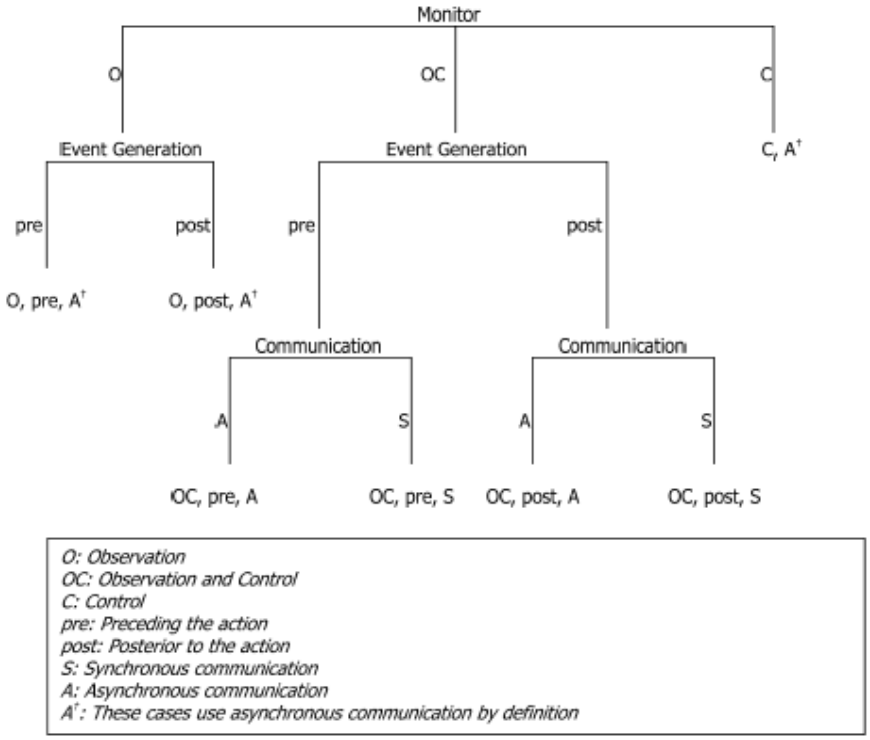

Figure 2-2 – Taxonomy of Monitor and Event Generation Features

More specifically at the first layer a distinction is made based on whether the monitor has observation only, observation and control or control only capabilities. These capabilities can be summarised as follows:

- Observation (O): The monitor observes the runtime behaviour of the system by receiving the generated events and it checks whether the monitoring properties hold at runtime.
- Observation and Control (OC): The monitor observes the runtime behaviour of the system by receiving the generated events, it checks whether the monitoring properties hold at runtime and forces the system to execute specific actions. These actions can be either preventive or perform recovery. This class is also known as closed-loop control.

- Control (C): The monitor forces the system to execute actions without needing to observe the actual state of the system. This class is also known as open-loop control.

The second layer of the taxonomy presents a distinction according to the time of the event emission with respect to the occurrence of the action described by the event. According to the criterion, we can distinguish between two cases:

- Emission preceding the action (pre): The event precedes the action which it describes. For example, the event generator sends an event to the monitor informing it that the system wishes to lock some resource before the system locks it.
- Emission posterior to the action (post): The event follows the action which it describes. For example, the event generator sends an event to the monitor informing it that the system has completed some transaction.

Finally, the third layer of the taxonomy refers to the type of the communication between the monitored system and the monitor. According to this criterion, we distinguish between the following two types of communication:

- Synchronous communication (S): The event generator uses a blocking send primitive to communicate with the monitor, waiting for a reply from it. This is only used when the monitor can exert control over the system.
- Asynchronous communication (A): The event generator uses a non-blocking send primitive to communicate with the monitor. It is mainly used when the monitor cannot exert any control over the system or when the control actions can be applied asynchronously. For example, the monitored system may notify the monitor that it will attempt to perform some action and start performing it without waiting for a permission to do so, as in optimistic transactions. If the monitor subsequently decides that this action is undesirable it can send a signal to the system to abort the action.

2.2.2.1 Formalisation of Properties for Dynamic Verification

2.2.2.1.1 General Purpose Systems

In most of cases, the formal specification of the requirements that are to be dynamically verified is based on Linear Temporal Logic (LTL) [129] and variations of it including past and future time LTL (ptLTL and ftLTL respectively). Past and future time Linear Temporal Logics are modal logics for specifying properties of concurrent reactive systems and are used for analysing traces of execution of such systems. ptLTL provides temporal operators that refer to the past states of an execution trace, while ftLTL provides temporal operators that refer to the future/remaining part of an execution trace. In particular, the Temporal Rover (TR) tool [57] supports a future and past time Metric Temporal Logic (MTL). MTL [32] extends LTL with relative time and real time constraints. All four LTL future time operators can be constrained by relative time and real time constraints specifying the duration of the temporal operator. MTL constraints can specify lower bounds, upper bounds, and ranges for relative time and real time constraints.

In the context of monitoring oriented programming (MoP), any monitoring formalism can be added to the system. ptLTL, ftLTL and extended regular expressions (ERE), which can express patterns in strings in a compact way [145], have been used to formalise properties to be monitored [34]. The proposed algorithms use binary transition tree finite state machines (BTT-FSMs) to monitor ftLTL properties [34], as well as, formulas written in a logic based on EREs [145].

Havelund et al. [72, 73, 74] have developed several algorithms, which are relative to temporal logic generation and monitoring. For instance, they propose algorithms for past time logic generation by using dynamic programming [74]. Also they have used the MAUDE rewriting engine [37], for monitoring future time logic [72, 73] and have proposed algorithms that generate Büchi automata adapted to finite trace LTL [64].

Other logics/languages used for formalising properties are EAGLE [20] and HAWK [47]. EAGLE is a ruled-based language, which essentially extends the μ-calculus with data parameterization and past time logic. HAWK can be viewed as a specialization of EAGLE for JAVA, as it supports data binding and object reasoning. HAWK further extends EAGLE with event expressions, where events are restricted to method calls and returns. The integration of programming and logic as well as the notation and semantics

of event expressions are similar to those used in modal logics like the π-calculus. HAWK also supports extended regular expressions.

According to the concept of Design by Contract (DBC) technique, introduced by Meyer [112] as a built-in feature of the Eiffel programming language, specifications of pre-conditions and post-conditions can be associated with a class in the form of assertions and invariants and subsequently be compiled into runtime checks. Jass [115] and jContractor [1] are two Java-based DBC systems. Jass is a pre-compiler, which turns the assertion comments into Java code. The JASS sub-language for specifying trace-assertions is similar to CSP [78], and its syntax is more like a programming language. jContractor is implemented as a Java library which allows programmers to associate contracts, consisting of pre/post-conditions and invariants, with any Java class or interface.

The Monitoring and Checking (MaC) framework [102] is based on a logic that combines a form of past time LTL and models real-time via explicit clock variables. JAVA MAC [93], a prototype implementation of the MaC framework for monitoring and controlling applications written in Java, defines an event-based language to describe monitors. Note that, in the context of the Java MaC framework, events refer to information that holds instantly during the system runtime, while conditions are defined to illustrate information that holds for a time period. The Java MaC framework is composed of two specification event-based languages: the Primitive Event Definition Language (PEDL) and the Meta Event Definition Language (MEDL). PEDL is used for writing low-level specifications and is tightly related to the programming language. As such it deals with primitive events and conditions that might occur during the program execution, which are defined using program entities such as variables and methods. The operations on events and conditions can be used to construct more complex events and conditions from the primitive ones. A MEDL specification then makes use of these primitive events and conditions in order to state high-level requirements. Using MEDL, a user can specify the correctness requirements declaratively, without worrying about operational issues related to the monitor. The MaC framework also supports the declaration of variables of primitive types which can be updated by user-defined assignment statements upon arrival of new events. These variables can be referred to in formulas.

Mahbub and Spanoudakis [109] have developed a framework for monitoring the behaviour of service centric systems which expresses the requirements to be verified against this behaviour in event calculus [149]. In this framework, event calculus is used to specify formulas describing behavioural and quality properties of service centric systems, which are either extracted automatically from the coordination process of such systems (this process is expressed in WS-BPEL) or are provided by the user.

In the area of component based programming Barnett and Schulte [19] have proposed a framework which uses executable interface specifications and a monitor to check for behavioural equivalence between a component and its interface specification. In this framework, there is no need for recompiling, re-linking, or any sort of invasive instrumentation at all, due to the fact that a proxy module is used for event emission. The component's interface specifications are written in the Abstract State Machine Language (AsmL) [70], which is based on Abstract State Machines (ASM) [69]. This language is executable and supports non-deterministic specifications. Having native COM connectivity, one can not only specify and simulate components in AsmL but also substitute low-level implementations by high-level specifications. Specifications written in AsmL are operational specifications of the behaviour expected of any implementation. They provide a minimal model by constraining implementations as little as possible.

Robinson [137] has proposed a framework for requirements monitoring based on code instrumentation in which the high-level requirements to be monitored are expressed in KAOS. KAOS [48] is a framework for goal oriented requirements specification which is based on temporal logic. The KAOS modelling language can support all the phases of requirements acquisition and modelling, starting from initial functional and non-functional goals, formalising the meaning of such goals using temporal logic formulas and assigning the responsibility for the achievement of these goals to potential agents which may signify the system in question, systems that interoperate with it, and human actors interacting with the system. KAOS has also been used by Feather et al [61] in a framework that they have developed to monitor system requirements at runtime and which incorporates some capabilities regarding the reconciliation of requirements with the runtime system behaviour.

In the context of software system monitoring, diagnosis focuses on the detection of system failures. Diagnosis typically involves the identification of trajectories of system observations, which have led to a failure. By using automata that recognise faulty

behaviour [22, 66, 128, 139, 161], diagnosis is carried through the synchronisation of automata modelling the expected behaviour of a monitored system and the events captured from it. Pencolé and Cordier [128] propose a similar but decentralised approach where synchronization is performed for individual system components and then aggregated for the global system.

The problem of fault diagnosis, concerning time, has been studied and analysed by Tripakis [161] and Bouyer et al. [22], where the system is modelled as a timed automaton. Timed automata extend the finite state machine model with real time clocks [4]. In both [22] and [161], the goal is the devising of algorithms (diagnosers) that would function as efficient online fault detectors of internal faults for any given sequence of observable events generated by the system. Tripakis has worked on the diagnosability of a timed system (plant). In particular, Tripakis has shown that the problem of checking whether a given timed system is diagnosable or not is a decidable problem and a diagnoser can be constructed as an online algorithm in case that the system is indeed diagnosable. The Δ-diagnosability diagnosis algorithm proposed by Tripakis is based on state estimation in order to decide whether a fault has occurred and report the fault at most Δ time units after its occurrence, for a given set of observations. In particular, the Δ-diagnosability algorithm keeps track of several possible control states and time ranges (zones) that the clock values can be in. The Δ-diagnosability problem for timed automata is PSPACE-complete. The complexity to diagnose faults from an observation is doubly exponential with respect to the final states of the system and to the size of the observations.

Due to the high complexity of the Δ-diagnosability algorithm by Tripakis [161], Bouyer et al. [22] describe a diagnoser, with lower complexity, more appropriate for online diagnosis. Bouyer et al. suggest two deterministic timed automata for realizing an efficient online diagnoser. On one hand, Bouyer et al. consider general deterministic timed automata (DTA) for realizing efficient online diagnosers. Bouyer et al. have proved that the problem of checking whether there is a realizable DTA diagnoser for a given timed system, provided that the number of clocks and the set of constants are well defined and available to the diagnoser, is a decidable problem and is 2EXPTIME-complete. On the other hand, Bouyer et al. study the fault diagnosis problem considering a subclass of DTAs called Event Recording Automata (ERA) [4]. In the context of ERA, there is an implicit clock attached to each action. The problem of checking whether there

is a diagnoser realizable as an ERA, provided that the number of clocks and the set of constants are well defined and available to the diagnoser, is decidable and PSPACE-complete.

In [128], a decentralised model-based approach for diagnosing discrete event systems has been proposed. In particular, the proposed formal framework is based on communicating automata for computing online diagnosis of large discrete event systems. According to the authors, the diagnosis is defined as the identification of failure events and their propagations, which can explain the system observations. The system observations are split into temporal windows. For each temporal window, diagnosis (subsystem diagnosis) is performed for each well defined subsystem of the system. The subsystem diagnoses are, then, merged to build the overall diagnosis for the system (global diagnosis). Each subsystem is modelled as a communicating finite state machine. The explicit behaviour of each subsystem can be computed by using a synchronization operation, which is based on a transition system product [9] and applied to the component models of the subsystem.

2.2.2.1.2 Security Oriented Systems

Some of the logics and languages reviewed in the previous section have also been used either as they were initially proposed or with some semantic modifications and extensions for the formalisation of security properties. Naldurg et al [117], for instance, have proposed a framework for intrusion detection which takes advantage of the capabilities of the EAGLE language for specifying the attack-safe behaviour of a system. EAGLE is suitable for expressing temporal patterns that involve reasoning about the data values observed in individual events and thus it allows the description of attacks whose signatures appear to have statistical properties, e.g., password guessing or denial of service attacks. For such attacks there is no clear distinction between an intrusion and a normal behaviour and the detection of intrusions involves collecting statistics during runtime and using them to evaluate the probability of the occurrence of an attack.

In the area of intrusion detection (see [99] for a survey), Ko et al [95] have proposed a specification-based approach, which uses dynamic verification techniques to detect exploitations of vulnerabilities in security-critical programs. According to this framework, one has to specify a *trace policy* which describes the intended behaviour of programs with regards to security properties. A trace policy determines security-valid

operation sequences of the execution of one or more programs. For specifying such trace policies, Ko et al. [95] have developed a grammar, called "parallel environment grammar (PE-grammar)" whose alphabet consists of system operations. A PE-grammar can express various classes of security trace policies, including behaviour related to access to system objects, synchronization, and operation sequencing and race conditions in concurrent or distributed programs.

Schneider [143] has developed a system called *Execution Monitoring (EM)* which can monitor violations of security policies by monitoring the execution steps of a system. This system is based on the security automata of Alpern and Schneider [3], which are a special type of Büchi automata. EM also incorporates mechanisms that can terminate the system execution if it is about to violate its security policy. Following the same automata-based formalism, Ligatti et al [104] extended the control capabilities of security automata by proposing edit automata, which can remove and add letters (i.e., system actions) to the words (i.e. execution traces) they recognise.

Having proposed a security-policy enforcing model which follows the general dynamic verification approach, Bandara et al. [15] have specified a language based on Event Calculus to model the system behaviour and write security policy specifications. The form of EC, which is used in this work, was presented in [138] and consists of: (i) a set of time points (that can be mapped to the non-negative integers), (ii) a set of properties that can vary over the lifetime of the system (fluents), and (iii) a set of event types. System operations and domain-independent rules for policy enforcement were specified in this approach using these constructs. According to Bandara et al. [15], one can use EC to express system-models containing a combination of authorisation, obligation and refrain policies.

Janicke et al [82] have proposed a security model that allows expressing dynamic access control policies, which can be either time or event-driven. A system's overall security policy can then be composed out of smaller policies which capture specific requirements and which can be verified individually. The advantage of the access control model used in this work is that it allows expressing both parallel and sequential composition. Janicke et al. [82] based their security model on Interval Temporal Logic (ITL), a flexible notation for both propositional and first order reasoning about intervals of time. ITL allows expressing properties for safety, liveness and timeliness. The policy model of Janicke et al. [82] provides a wide range of operators, for example to allow the

dynamic addition/deletion of rules or to select different sub-policies based on to the occurrence of an event or a time-out. An important reason of choosing ITL was the availability of an executable subset of the logic, known as Tempura [116]. The use of ITL, together with its subset of Tempura, offers the benefits of traditional proof methods with the speed and convenience of computer-based testing through execution and simulation.

Brisset [24] has worked on establishing and ensuring the correct operation of a Java platform security mechanism for runtime authorization of not trusted applications in remote hosts. The resulting Java security mechanism, which is called SecurityManager and belongs to the JAVA runtime library, essentially embodies the security policy of the virtual machine. The verification technique used a CTL-based language, which extends CTL with JVM-specific atomic propositions. Thus, JVM-specific atomic formulas can be used for runtime authorization of not trusted applications. In order to verify an application against these formulas its byte-code is translated into pre/post-condition generators for CTL formulas on-the-fly.

Sekar et al. [144] presented an approach called model-carrying code (MCC) for mobile code security. The main components of MCC are: (a) a policy language for specifying security policies and a compiler for this language, (b) a language for specifying program behaviour models and techniques for extracting them, and (c) a policy refinement component which is based on model-checking techniques. Their language for policies and behaviour models is called Behaviour Monitoring Specification Language (BMSL) and it is compiled into extended finite state automata (EFSA). EFSA are standard finite state automata (FSA) augmented with the ability to store values in a fixed number of state variables. These state variables are capable of storing values over both finite and infinite domains. The state of the EFSA is then characterized by its control state (which has the same meaning as the notion of state in the case of FSA), plus the values of these state variables. Each transition in the EFSA is associated with an event, an enabling condition involving the event arguments and state variables, and a set of assignments to state variables. For a transition to be taken, the associated event must occur and the enabling condition must hold. The assignments associated with the transition are performed when the transition is taken. In usual behavioural models the event alphabet of the EFSA consists of system-call names. On the other hand, security policies need to refer to particular uses of such system calls and be able to examine their

respective arguments. These uses, for instance "read(sensitive_file)", augment the alphabet of EFSA with parameters to the initial system call names event alphabet. The resulting language is therefore able to distinguish the difference between the opening of a temporary file and the opening of a password file. Moreover, EFSA can also represent properties that refer to the arguments to system calls in the past, e.g. a program opens a file, whose name was given as an argument in the command line in the past.

For thoroughness we shall also mention certain higher-level languages and frameworks, which have been proposed for security requirements and policies. The KAOS framework, which we have already examined in the previous section on general-purpose formalisms, has been extended for modelling, specifying and analysing security requirements [166] by including the classical security concepts:

- *Adversaries/attackers* which are the malicious agents in the environment,
- *Threats* which are obstacles (anti-goals) intentionally set up by adversaries, and
- *Assets*, which must be protected against *threats*, are illustrated as passive or active objects.

The *Confidentiality*, *Integrity*, *Availability, Privacy, Authentication* and *Non-repudiation* requirements are sub-classes of the meta-class SecurityGoal in KAOS. Finally, the formal first-order, real-time, linear temporal logic of KAOS has been augmented with epistemic operators (Knows, Belief), which are needed in security-related properties (e.g. Authorized, UnderControl, Integrity or Using predicates).

Damianou et al. [46] have defined a declarative, object-oriented language, called Ponder, to specify security policies which can be monitored and applied at runtime. Ponder can be used to specify security policies regarding role-based access control to system resources, and general-purpose system management policies. Security policies are distinguished by Damianou et al. [46] in authorisation, obligation, refrain and delegation policies. Authorisation policies specify whether a subject is permitted to perform a particular action on a target; obligation policies specify management operations that must be performed when a particular event occurs and some supplementary guarding conditions are true; refrain policies allow system administrators to specify conditions under which certain operations should not be performed; and delegation policies specify which actions subjects are allowed to delegate to others. Ponder has been designed with the intention to be an extensible security policy specification language that would be able

to cater for future types of policies and, rather than assuming a particular implementation platform, it could map to, and co-exist with, different underlying platforms.

In Service Oriented Computing, Baresi and Guinea [16] have proposed a framework for runtime monitoring of WS-BPEL processes. Monitoring rules are weaved at runtime into the process they must monitor and a proxy module supports their dynamic selection and execution [18]. Finally, they proposed a user-oriented language to integrate data acquisition and analysis into monitoring rules. Their monitoring rules define runtime constraints on WS-BPEL process executions and are expressed using the WSCoL language (Web Service Constraint Language). This development of this language has been inspired by the Java Modelling Language (JML) of Leavens, Baker and Ruby [100]. WS-CoL is a domain-independent policy assertion language for specifying user requirements (constraints) on the execution of Web services, which can be used within the framework of WS-Policy [142] and WS-Security [88]. WS-CoL is an assertion language augmented with features for allowing one to retrieve information that originates outside the process. It distinguishes between *data collection* and *data analysis* to differentiate the phase in which information is collected (data collection), from the phase in which this data is analysed (data analysis). Data can be collected from the process directly (e.g., values of internal variable) but they can also come from external sources (e.g., exchanged SOAP messages). An example of a monitoring rule in this language could specify that all exchanged messages must be encrypted using the 3DES encryption algorithm.

2.2.2.1.3 Summary of specification languages for security and other system properties for dynamic verification

Table 2-1 gives a summary of various formal notations which have been used by different dynamic verification methods to express the properties to be verified and other functional and non functional characteristics of the systems and identifies languages and notations that have been specifically developed for expressing and verifying security properties.

As shown in the table most of the approaches deploy languages which are based on some form of temporal logic as these languages provide the necessary operators for expressing conditions about the temporal ordering and boundaries of occurrence of events which is required for the expression of most of the properties that need to be verified at runtime. The most popular formal notation for expressing security properties is

Linear Temporal Logic (LTL) or extensions of it and languages with similar expressive power such as Event Calculus.

Table 2-1 - Summary of formal languages used for dynamic verification

Languages for expressing security properties for dynamic verification	Languages for expressing all types of properties for dynamic verification
Behaviour Monitoring Specification Language (BMSL) and Extended finite state automata (EFSA)	EAGLE and HAWK
EAGLE	CSP – like specification
PE Grammar	LTL and its extensions
ITL	PEDL and MEDL
CTL (extended)	AsmL
Security automata	Event Calculus
Ponder	Ponder
KAOS	KAOS
Event Calculus	EC-Assertion

Some dynamic verification techniques reason about systems at both low and high level of abstraction, such as Primitive Event Definition Language (PEDL) and Meta Event Definition Language (MEDL) in Java Monitoring and Checking (JavaMaC) framework [102]. PEDL is used for writing low-level specifications and is tightly related to the programming language, while MEDL specification makes use of primitive events and conditions in order to state high-level requirements.

2.2.2.2 Methods for Capturing Events

In the second stage of the general dynamic verification process, the goal is to apply techniques so as to capture the real behaviour of the system during its execution.

As shown in Figure 2-3, existing event emission methods can be divided into *code modifying* and *non code modifying ones*. Code modifying event emission methods require direct access to the source or binary code of a system in order to insert code statements that will generate the events of interest. Code instrumentation is an example of a code modifying event emission method in which event generation statements are inserted

manually into the code of a system. Aspect Oriented Programming (AOP) has also been used to generate events (through the weaving of aspects into binary or source system code). AOP is a *code modifying event emission method*, which can be considered as a subcategory of code instrumentation. Monitoring Oriented Programming [34] and Design by Contract [112] are also code modifying event emission methods which can be regarded as subcategories of Aspect Oriented Programming [90].

Non code modifying event emission methods generate events without altering the code of a system. Such methods access, modify and/or take advantage of capabilities of the general computational environment in which a system is executed, in order to generate the events flow. Reflective middleware approaches [30, 31, 110], proxy-based architectures [19] and the use of application programming interfaces (APIs) [12, 26, 109] constitute examples of event emission methods which belong to this category.

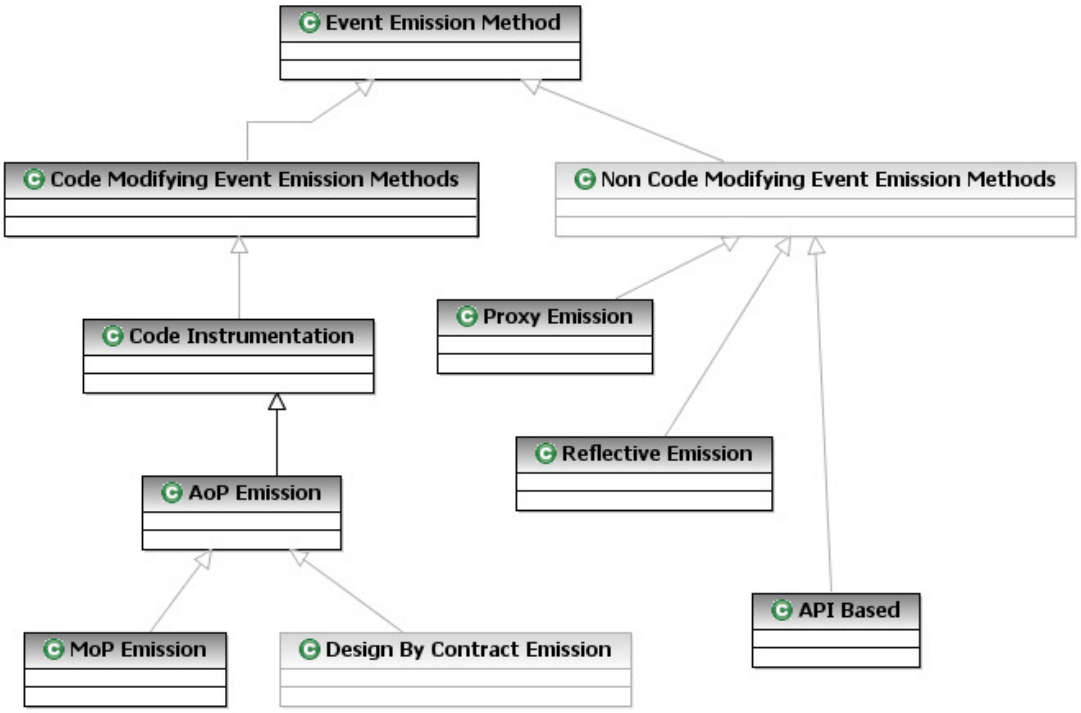

Figure 2-3 – Taxonomy of Event Emission Methods

2.2.2.2.1 Code-Modifying Event Capture Methods

2.2.2.2.1.1 Code Instrumentation

The technique of *code instrumentation* can be described [137] as the insertion of statements into the system's code (source or binary code) for monitoring purposes. Instrumentation can be done manually or automatically e.g. by using Jtrek-JSpy [65] or Joie [39] which automatically instrument Java byte code. During the execution of the instrumented code, an event stream is generated. The generated events can then be passed directly to external monitors or pre-processed before they reach the verification stage.

A tool using code instrumentation for capturing events in Java-based systems is RMon [137]. In Rmon, requirements are initially expressed in the KAOS framework [48], which provides a goal-oriented formal specification language based on temporal logic. Requirements are thus specified as high level goals which must be achieved by the system. These goals must then be mapped onto low-level events which can be monitored at runtime. The system's code is then instrumented in order to capture these low level events, using the Joie framework [39].

In the initial phase of the Java MaC architecture [93], low-level specifications (written in PEDL) are inserted into the byte code of the monitored program through an automatic instrumentation procedure. Furthermore, in the MONID tool [117] system-level events are generated by appropriately instrumented source code.

2.2.2.2.1.2 Aspect Oriented Programming

Aspect Oriented Programming (AOP) [90], also called Aspect Oriented Software Development (AOSD), was proposed to support the advanced identification, illustration and separation of non-functional concerns, which crosscut the system's main functionality. Complex programs include various crosscutting concerns (properties of interest such as QoS, energy consumption, fault tolerance, and security). While object-oriented programming abstracts out commonalities among classes in an inheritance tree, crosscutting concerns are scattered among different classes, complicating the development and maintenance of applications. As depicted in Figure 2-4, AOP enables the separation of crosscutting concerns during the development of the software. Specifically, the code implementing crosscutting concerns of the system, called aspects, is developed separately from other parts of the system. In AOP, locations in the program where aspect code can be woven, called *pointcuts*, are typically identified during development. Later, for example during compilation, an aspect weaver can be used to weave different aspects of the program together so as to form a program with new

behaviour. AOP proponents argue that disentangling crosscutting concerns leads to simpler development, maintenance, and evolution of software [90]. Examples of AOP approaches include AspectJ [91] and Hyper/J [159].

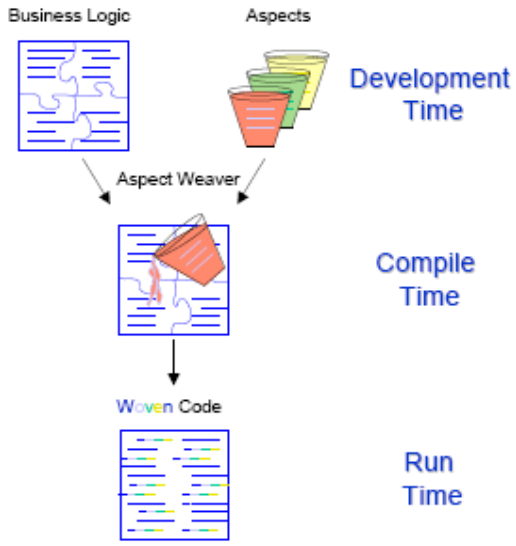

Figure 2-4 – Conceptual Representation of Aspect Weaving [90]

AOP supports dynamic re-composition in three major ways. First, most adaptations are relative to some crosscutting concern, such as quality-of-service or fault tolerance. AOP enables the code associated with these aspects to be written and managed independently of the application code as well as other parts of the system, such as traditional middleware platforms. Such a separation is needed in order to dynamically replace one instantiation of a particular solution for a concern with another. Second, although compile-time aspect weaving produces a tangled executable that cannot easily be reconfigured, delaying the weaving process until runtime provides a systematic way to realize dynamic re-composition [77, 169]. Finally, if adaptability itself is considered as a "generic" aspect [49, 170], then runtime weaving can be used to enhance the program with adaptive behaviour, not necessarily anticipated during the original development (e.g. to tolerate newly discovered faults or to detect and respond to new security attacks). This kind of upgrading is especially important in situations where the application is required to run continuously and cannot be easily halted for upgrade. However, the need of a formal aspect specification written in a domain-specific knowledge language or using logic, rather than the host programming language itself, is expressed in [34]. The mapping from specification to implementation, with the support of automatic code generation can then be formally verified.

In particular, AspectJ [91] provides an approach to implementing cross-cutting features in Java. AspectJ provides a pattern mechanism, called pointcuts, for capturing groups of events, called joinpoints, that may occur during a program's operation (such as method calls/receptions, constructor calls, field accesses, and exception events). The pattern matching mechanism includes regular expression matching, with wild-carding over fragments of method names, argument names, types etc. Extra code, called advices, can be associated with pointcuts, and is inserted by the AspectJ compiler into the join-points. Advices can inspect and modify data that are available at joinpoint events (e.g. method-call arguments and return values), and can create new data dynamically that is only shared with other advice.

For instance, Dingwall-Smith and Finkelstein [56] have developed an aspect oriented approach, in which system providers specify instrumentation code in separate classes, and define composition rules which determine how this code is to be merged with the application code, by using Hyper/J [159]. Also, Baresi and Guinea [17] have proposed a framework for runtime monitoring of WS-BPEL processes, in which monitoring rules are specified and weaved dynamically into the process they belong to. Furthermore, the instrumentation module of the JpaX framework performs a script-driven automated instrumentation of the program to be verified. JSpy [65] is the automated AOP environment package, which is used in JPaX [72, 75].

2.2.2.2.1.3 Design by Contract

Design by Contract (DBC), as proposed by Meyer [112] for the object-oriented language Eiffel, is a practical approach to runtime checking in applications. DBC is a lightweight formal technique, which allows one to add semantic information to a program by specifying assertions regarding the program's runtime state. Then, checks for specification violations are carried out at runtime. Such a technique stresses the importance of explicitly specifying the constraints that hold before (pre-conditions) and after a program is executed (post-conditions). The technique's name refers to a contract, which is made between the client and the supplier of a system module and defines conditions before and after the execution of the module. Thus, for monitoring reasons the entry and exit points of the module become the events that we want to observe.

In the context of the Eiffel object-oriented language, specifications of pre/post-conditions can be associated with a class in the form of assertions and invariants.

Subsequently, inserted specifications can be compiled into monitoring code. In the Java language, there are two approaches which are based on DBC. Jass [21] is a pre-compiler which turns the assertion comments into Java code. Properties in Jass are called trace assertions and they specify permissible sequences of method calls in a CSP-like notation. Thus, processes, parallelism, conditionals and data exchange among processes can also be expressed. However, the trace assertions are interpreted loosely; no formal semantics is provided. The Jass pre-compiler translates the trace assertions into runtime checks.

2.2.2.2.1.4 Monitoring Oriented Programming

Monitoring Oriented Programming (MoP) is a paradigm which combines a formal specification with an implementation in order to form a system. In particular, it provides a light-weight formal method for runtime specification checks against the behaviour of the implementation. By using MoP, logical statements can be inserted anywhere in the program. These statements are simply Boolean expressions which can refer to past and future states of the program. A MoP user can insert such statements for different reasons e.g. to guide the system's execution, terminate the program or throw exceptions. Thus, MoP can increase the dependability of a system by monitoring its requirements at runtime and controlling it at the same time.

In particular, the statements, which can be inserted as annotations into the code, can be divided into three parts. The first part consists of a keyword defining the logic in which the rest of the inserted statements are expressed in. The second part comprises the definitions of the predicates and the formula to be monitored. Finally, user defined code which will be executed in case the monitored formula is violated is included in the third part, called a violation handler.

The general MoP paradigm is language and specification formalism independent. According to Chen and Rosu [34], a MoP environment should provide the capability of adding any logic framework on top of any target programming language via logic plug-ins, which can be publicly accessed. A logic plug-in consists of two modules, namely the logic engine and the target language shell. Logic engines translate formulae into monitors, encoded in an abstract representation (pseudocode). Then the language shell transforms the monitor pseudocode into the target language code. Thus, the logic plug-in can be considered as the code generator of the monitor.

Moreover, a MoP environment allows users to specify whether the monitoring code will be executed using the resources of the monitored program (internal monitor) or within a different process (external monitor). In the first case, the inserted logic statements contain the monitor's specifications which are replaced by the generated monitoring code in the end. Note that internal monitors, in general, cannot check for program deadlocks and unexpected terminations. In case that a monitor is executed as a different process, the inserted statements are replaced by instrumentation code which operates as an event generator. The user can specify whether the monitor should be executed synchronously or asynchronously with the monitored system and whether it should be executed on the same machine with the system or a different one.

2.2.2.2.2 Non Code Modifying Event Capture Methods

2.2.2.2.2.1 Reflective Middleware

Middleware technologies [58] have been designed to support the development of distributed systems. Their success has been mainly due to their ability of making distribution transparent to both users and software engineers, so that systems appear as single integrated computing facilities. However, hiding the implementation details from the application completely is very difficult in a mobile setting and not even always desirable, since mobile systems need to quickly detect and adapt to changes in their environment. A new form of awareness is needed to allow application designers to inspect the execution context and adapt the behaviour of the middleware accordingly.

Reflection and metadata can be successfully exploited to develop middleware targeted to mobile settings. By using metadata, we separate the middleware in two parts: what the middleware does and how the middleware does it. Reflection allows applications to inspect and adapt their metadata. In this way, applications can influence the way their middleware behaves, according to their current context of execution.

Capra, Emmerich and Mascolo [31] proposed a framework designed to ease the adaptation of applications to changing execution conditions. The model considers different layers (operating system, middleware, application, and user), each of which is described using metadata in order to ease their interaction. When the application invokes a service, the middleware uses both the application metadata and the metadata reflecting the current execution conditions to decide how to offer the requested service.

Applications can also ask the middleware to be notified when specific execution conditions occur. This system allows for a fine adaptation of applications, but it requires that service calls be coded explicitly in the applications. However, a complete transparency is not possible if adaptation (which requires awareness) is desired.

CARISMA [31] is a context-awareness based reflective middleware. It includes a reflective API, which allows applications to dynamically inspect their current configuration and alter it to best suit the current environment. CARISMA maintains a representation of the execution context by interacting with the underlying network operating system. Based upon this representation, the application may behave in different ways. For example, an application attempting to send messages in low bandwidth availability may compress messages before emitting them, whereas it would send them uncompressed when bandwidth availability is high. The behaviour of the middleware with respect to the application is referred to as an application profile. There are two main aspects of an application profile, services and policies. Services describe the services offered to the application and which the middleware can customize. Policies describe the different variations in which the services can be delivered. In the prior example, the service the application is using is sending messages, and the different policies to deliver the service are sending either compressed or uncompressed messages based upon the context environment (high or low bandwidth). In CARISMA, each time a service is invoked, the middleware examines the application profile. Based upon the context of the application, the middleware determines which policy is best suited for the current context. This relieves the application of the burden of determining how to optimise its own behaviour.

XMIDDLE [110] is a middleware for mobile computing that focuses on the synchronization of replicated XML documents. In order to enable application-driven conflict detection and resolution, XMIDDLE supports the specification of conflict resolution policies through meta-data definitions using an XML schema.

2.2.2.2.2.2 Proxy Architecture

A proxy module acts as an intermediate between the monitored system and its environment, capturing their interaction and emitting the corresponding events. Thus, there is no need for code recompiling, re-linking or any other sort of invasive instrumentation at all.

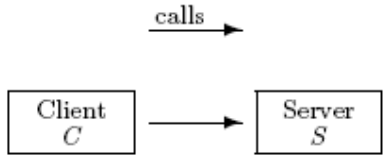

Figure 2-5 – A client-server architecture [19]

For component based programming, Barnett and Schulte [19] have proposed a framework which uses executable interface specifications and a monitor to check for behavioural equivalence between a component and its interface specification. Let us assume that a client–server architecture is used, like the one illustrated in Figure 2-5.

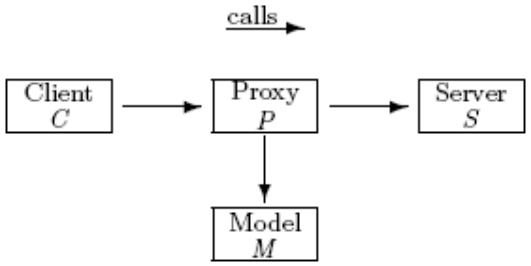

Figure 2-6 – Proxy architecture [19]

A component, P, which essentially operates as a proxy, is inserted between the client C and the server S as shown in Figure 2-6. Using a proxy allows the interaction of the client C and the server S to be observed without having to modify either component. P can be created automatically from the definition of the interfaces, which C and S use in order to interact. The proxy forks all of the calls made from C to S so that they are delivered to both S and the (AsmL specification based) model, M, managing the concurrent execution of M and S. Then P compares the results from components M and S. P checks at each interface whether the results agree in terms of their success/failure codes as well as any return values. As long as, the results are the same, they are sent to C. In any other case, S and M are deemed not to be behaviourally equivalent.

2.2.2.2.2.3 API-based Event Capturing

In the last non code modifying event emission subcategory, one finds approaches which make use of specific APIs for capturing and emitting events.

For instance, the JNuke tool takes advantage of its virtual machine's (VM) specific API in order to observe the runtime behaviour of the monitored system. In particular, the event-based runtime verification API of JNuke's VM serves as a platform for various runtime algorithms. This API provides access to events occurring during program

execution. Event listeners can then query the VM for detailed data about its internal state and thus implement any runtime verification algorithm, including detection of high-level data races [10] or stale-value errors [11] (see Section 2.2.2.4.7 for more details).

In the same family of event capturing methods is the prototype implementation of the specification based intrusion detection system, proposed by Ko et al. [95], which takes advantage of audit trails provided by the operating system. The prototype runs under the Solaris 2.4 operating system and uses the auditing services of the Sun BSM audit subsystem. The BSM audit subsystem provides a log of the activities that occur in the system. A BSM audit record contains information such as the process ID and the user ID of the process involved, as well as, the path name and the permission mode of the files being accessed. However, it does not contain information about the program the process is running. Therefore, an audit record pre-processor is used to associate the program identification with each audit record. The audit record pre-processor actually filters audit records that are irrelevant to the monitoring system and translates the BSM audit records into the format required by the monitoring system.

2.2.2.3 Checking for violations

The third stage of dynamic verification is concerned with the checks that a monitor carries out to identify whether the runtime behaviour of a system conforms to certain properties. According to the taxonomy of Figure 2-2, the monitors with the most advanced capabilities are the "OC−pre−S" monitors. This category describes monitors, which verify the system's correct behaviour based on events describing the system's state before the execution of some action. The check is carried out while the system is halted, waiting for the monitor's reply. Once the monitor assures that the monitored properties hold, it allows the system to continue with its normal execution. If however a violation is reported, the monitor can force the system to execute some other action so as to remedy the current violation.

2.2.2.3.1 Checks for Admission

A widely used type of runtime checks is the check for admission. In this check a monitor checks an incoming request/application for admission, before actually honouring/executing it. In the following we shall examine some of the solutions for performing admission checks.

2.2.2.3.1.1 Signed Code

Another technique for protecting a system, which is allowed to host mobile code, is by signing code with a digital signature. Using digital signatures, one can confirm the authenticity of the code, its origin, and its integrity. Typically the code signer is either the code producer or a trusted entity that has reviewed the code. Especially in mobile agents systems, where an agent can operate on behalf of an end-user or organization [158], the signature of an agent is used as an indication of the authority under which the agent operates.

Code signing is tightly bound with public key cryptography, which relies on a pair of keys (private and public) associated with an entity. One key is kept private by the entity and the other is made publicly available. Digital signatures benefit greatly from the existence of a public key infrastructure (PKI), since certificates containing the identity of an entity and its public key (i.e., a public key certificate) can be readily located and verified. The code signer applies an irreversible hash function to the code. The result of this function is a unique message digest of the code, which the code producer encrypts with his private key, thus forming a digital signature of the code. Because the message digest is unique, and thus bound to the code, the resulting signature also serves as an integrity protection against any malicious code modifications. The produced signature and the public key certificate can then be sent along with the code to the code consumer. The code consumer can easily verify the source and authenticity of the code by using the same hash function and the appropriate decrypting mechanism, which the code producer used to sign the code. If the signature verification succeeds, the code consumer can execute the code.

Note that the meaning of a signature may be different depending on the policy associated with the signature scheme and the party who signs. For example, the code producer, either an individual or an organization, may use a digital signature to indicate who produced the code, but not to guarantee that the code will be executed without faults.

Microsoft's Authenticode [67], enables Java applets or Active X controls to be signed, ensuring consumers that the software has not been tampered with and that the identity of the code producer is verified. Moreover, JDK 1.1 introduced the capability to digitally sign Java byte code (at least byte code files placed in a Java archive, called a JAR file), which expanded more with Java 2 [111]. From a certificate authority perspective,

VeriSign provided a solution which addressed signed code issues for specific Netscape objects [167].

2.2.2.3.1.2 Proof Carrying Code

Proof Carrying Code (PCC) [118] can be used to increase security in systems executing not-trusted, mobile code. With PCC, a program is supplied along with a proof of its correctness and this proof is in a form which can be easily verified mechanically before the program's execution. Therefore, it is now the code producer's responsibility to formally prove that the program will assure the safety properties specified by the code consumer, honouring the security policy of the underlying platform/system. Then, both the code and its proof are sent to the code consumer, where the safety properties are verified. A safety predicate is also generated directly from the native code to ensure that the accompanying proof does in fact correspond to the code sent. Once verified, the code can execute without any further checking. Any attempts to tamper with either the code or the safety proof result in a verification error.

The PCC binary life-cycle includes three stages:

- *Certification*: During this stage, the code producer compiles and generates a proof for the code, proving that the source program adheres to the safety policy of the code consumer. The proof can be produced by theorem proving.
- *Verification*: This stage is performed in the code consumer side. The code consumer verifies the proof part of the PCC binary code. The verification is performed by a simple algorithm, which is trusted by the consumer.
- *Execution*: The code consumer can execute the code without any further run-time checks.

For expressing safety policies Necula [118] has used first-order predicate logic, extended with predicates for type-safety and memory-safety. The not-trusted code is in the form of machine code. For relating machine code to specifications they used a form of Floyd's verification-condition generator. Such a generator extracts the safety properties of a machine code program as a predicate in first-order logic. This predicate must then be proved by the code producer using axioms and inference rules supplied by the code consumer as part of the safety policy. For generating the safety proof, a theorem prover can be used, in the code producer's side.

Proof encoding can adequately be expressed using the Edinburgh Logical Framework (LF). LF is general and can easily encode a wide variety of logics, including higher-order logics. Another compact representation of proofs is a form of oracles, which guide a simple non-deterministic theorem prover in verifying the existence of the proof. For validating proofs encoded in LF, an LF type checker can be used. A non-deterministic logic interpreter can be used in the case that a proof is encoded as an oracle.

Initial research has demonstrated the applicability of PCC for fine-grained memory safety and shown the potential of it for other types of safety policies, such as controlling resource use.

PCC is based on principles from logic, type theory, and formal verification. There are, however, some potentially difficult problems to be solved before the approach is considered practical. These include a standard formalism for describing security policies, automated assistance for the generation of proofs and techniques for limiting the potentially large size of proofs that in theory can arise. In addition, the technique is tied to the hardware and operating environment of the code consumer, which may limit its applicability.

Comparing PCC to signed code, PCC is a prevention technique, while code signing is an authentication and identification technique used to deter the execution of unsafe code. Furthermore, the proof is structured in such a way that simplifies its verification, since it must be carried out efficiently without using any external assistance.

2.2.2.3.1.3 Model Carrying Code

Model Carrying Code (MCC) is an approach for supporting the safe execution of not-trusted mobile code [144]. The central idea of MCC is that the code producer sends the code along with a high-level model, which describes the code's security-relevant behaviour. It should be noted that the generated model has to be usable by all code consumers. The automated model generation is based on model extraction via machine learning from execution traces. In the consumer's side, the model is checked for compliance with the consumer's security policy. If the security policy is satisfied, the code can be executed. In case there are conflicts, the consumer's policy can be refined, taking into consideration the code's functionality. When the code is executed, runtime verification methods are used to guarantee that the consumer's (refined) policy is not violated by the code (Figure 2-7).

By these means, the model bridges the semantic gap between the low-level binary code and the high-level security policies of the consumer. Moreover, the code producer does not have to know the consumer's security policies (as in PCC). Assuming that a model can be much less complex than the corresponding program, it is feasible for a consumer to automatically determine whether a model conforms to his security policies.

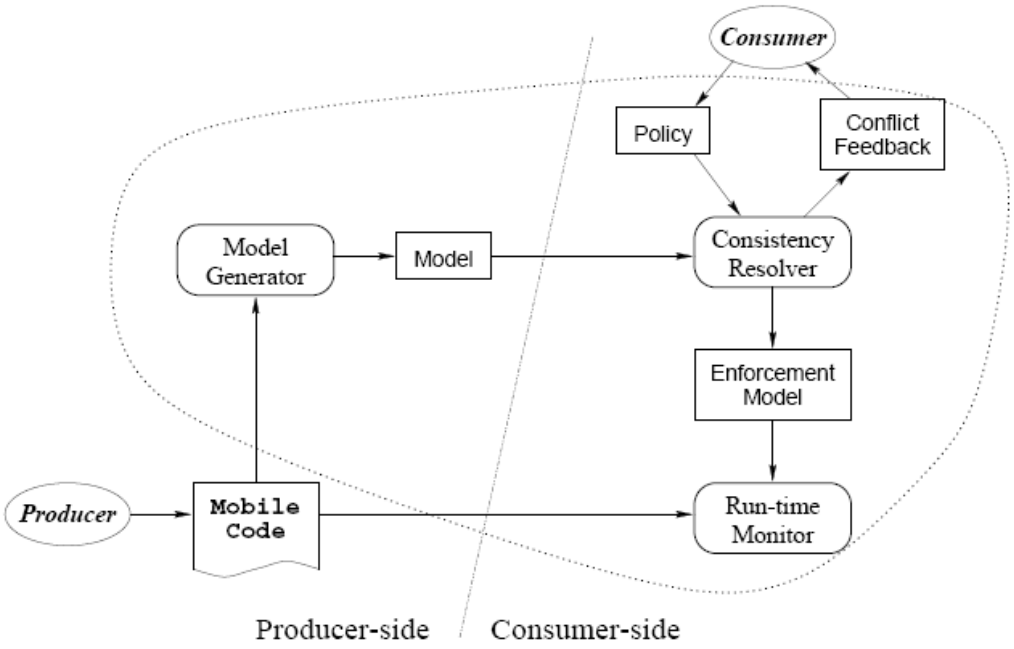

Figure 2-7 - The Model-Carrying Code framework [144]

2.2.2.3.1.4 Java Virtual Machine Byte-Code Verifier

The basic Java Virtual Machine (JVM) security model provides the capability of carrying out checks for admission for not trusted code, via a byte-code verifier [105]. In general the basic JVM security model comprises three related parts, namely the byte-code verifier, the class loader and the security manager. The JVM verifies all byte-code before execution.

The byte-code verifier reconstructs type information by inspecting the byte-code [171]. The types of all parameters of all byte-code instructions must be checked. The JVM specification lists what must be checked and what exceptions may result from a failed check. However, the JVM specification does not define when and how type verification should be done. Thus, while the process of verification in Java is defined to allow different implementations of the JVM, most Java implementations take a similar approach to verification. The most common verification process consists of byte-code

checks on the class file itself and runtime checks, which confirm whether the referenced classes, fields and methods are existing and compatible to their attempted use.

The byte-code checks establish a basic level of security guarantees. In particular, the class file format is checked whether it is correct. This check is carried out with the class loader's cooperation. The code is also verified for the correct hierarchical structure of its classes. Thus, every class must have a super-class, final classes cannot have subclasses, final methods cannot be overridden and all field and method references in the constant pool (a heterogeneous array composed of five primitive types) must have legal names and classes. Moreover, the byte-code is verified by using data-flow analysis. By this means, it can be ensured that the operand stack can not be overflowed or underflowed, variables are properly initialised, register access is checked for using the proper value type, that method calls are done with the appropriate number and type of arguments, fields are updated with the appropriate type and all opcodes have the proper type of arguments on the stack and in the registers.

During the class execution runtime checks can occur, since some aspects of Java's type system cannot be statically checked, like dynamic linking. Java loads each class only when it is actually needed at runtime (dynamic linking). Thus, whenever an instruction calls a method, or modifies a field, the runtime checks ensure that the method or field exists, type-checks the call and checks that the executing method has the appropriate access privileges.

2.2.2.3.2 Post – Mortem Checks

Monitors which can only observe the runtime behaviour of a system ("O, pre, A" and "O, post, A") perform post-mortem checks. Post-mortem checks deal with properties which might not be of high importance. Proposed monitoring architectures for this category of monitors, like AMOS [38] and FLEA [60] maintain event logs and offer proprietary event pattern specification languages, or store events in relational databases and deploy standard SQL querying for detecting requirement violations [137].

2.2.2.4 General Purpose Dynamic Verification Tools

2.2.2.4.1 The Java PathExplorer (JPaX) framework

The Java PathExplorer (JPaX) is a tool for monitoring systems at their runtime [72, 75]. By using JPaX one can automatically instrument code and observe the system's runtime behavior. It can be used during development to provide more robust verification. It can also be used in an operational setting, to help optimize and maintain systems as they mature. Figure 2-8 illustrates an overview of JPaX architecture.

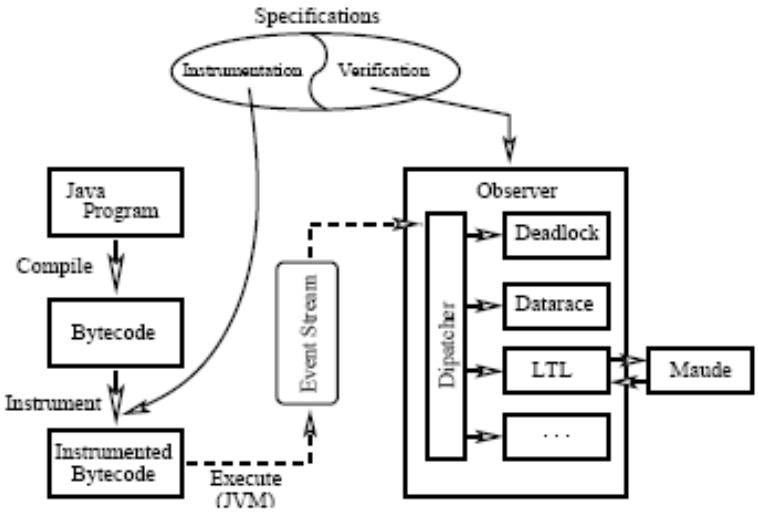

Figure 2-8 – The JPaX architecture [75]

JPaX consists of three modules:

1. Instrumentation module: It performs a script-driven automated instrumentation of the program to be verified, through which the byte-code is automatically instrumented.
2. Interconnection module: Its responsibility is to receive events about potential errors and transmit them to the observer module.
3. Observer module: It performs two kinds of verification:
    - Checks events against a user-provided requirement specification written in Maude, a formal, modularized specification and verification language. JPaX supports linear temporal logic (LTL), for both future and past time. Future time LTL uses execution traces as an underlying model making it convenient for program monitoring. Past time is useful for verification of safety properties.
    - Carries out error pattern analysis by exploring an execution trace and detecting potential problems such as error-prone programming

techniques, e.g. locking practices that may lead to data races and/or deadlocks. The important and appealing capability of the error pattern analysis algorithms is that they can find potential errors, even in the case where errors do not explicitly occur in the examined execution trace. However, error pattern analysis may sometimes find errors which cannot exist. Two algorithms focusing on concurrency errors are implemented for JPaX:

I. The "Eraser" data race analysis algorithm. A data race occurs when two or more concurrent threads access a shared variable simultaneously without any locking mechanism and at least one thread intends to write in the variable. The "Eraser" keeps track of thread locks and variables to find data race conditions.

II. Deadlock analysis algorithm. Deadlocks occur when multiple threads take locks in different order. For example, a deadlock condition occurs when:

   - Thread A acquires Lock 1 while Thread B acquires Lock 2
   - Thread A retains Lock 1 and asks for Lock 2 while Thread B retains Lock 2 and asks for Lock 1. JPaX monitors locks during program execution to find potential deadlocks.

Using JPaX, a Java program byte-code is automatically instrumented using instructions from a user-provided script. This script defines what kind of error pattern detection algorithms should be activated and what kind of logic-based monitoring should be performed. The automated instrumentation tool, which is used in JPaX, is JSpy [65]. JSpy can be seen as an Aspect Oriented Programming tool in the sense that it is guided by rules, or aspects, which specify how a program should be transformed to achieve additional functionality. However, the main purpose of these aspects is to extract information from a running program. JSpy itself is built on top of the low-level JTrek instrumentation package [40].

As aforementioned, JPaX makes use of the Maude system [37]. Maude is a specification and verification system which supports rewriting logic. Rewriting logic is

appropriate for expressing concurrent changes, which can naturally deal with state and with concurrent computations. Therefore, rewriting logic can be used like a universal logic, due to the fact that the syntax and operational semantics of other logics (such as temporal logics) can be expressed in rewriting logic.

The Maude rewriting engine can be used as:

- A monitoring engine during program execution. In JPaX, execution events are submitted to the Maude program that evaluates them against the requirements specification.
- Translation engine before execution. In JPaX, the specification is translated into a data structure optimised for program monitoring. This data structure is then used within the Java program to check the events at runtime.

JPaX produces either no output (when no errors are found) or a set of warnings. The warnings deal not only with runtime violations of high-level requirements written in temporal logic formulae but also with low-level error-detection procedures like concurrency related problems such as deadlock and data race algorithms.

The JPaX Java instrumentation module can be replaced with a C++ module to monitor C++ code. Experiments were conducted by the NASA Ames Robotic group on C++ code to check for deadlocks. JPaX located a potential deadlock that had not been previously detected during other testing [23].

To conclude, JPaX can also find potential errors, even in the case where errors do not explicitly occur in the examined execution trace. However, its logic-based monitoring adds an overhead to the normal execution of programs. Moreover, its error pattern runtime analysis can detect problems that do not really exist (called false positives).

2.2.2.4.2 The Java Monitoring and Controlling Framework (Java-MaC)

The Java Monitoring and Controlling (Java-MaC) framework uses formally specified properties to monitor Java programs at runtime [93]. Its architecture is shown in Figure 2-9. It can be divided in two main parts: the *static phase* (before a monitored entity runs) and the *runtime phase* (while the monitored entity is executing). During the static phase, the runtime modules, namely a *filter* (event generator), an *event recognizer* (event processing module), and a *run-time checker* (external monitor), are automatically generated from a formal requirements specification. During the runtime phase, events

from the execution of the monitored program are collected and checked against the given requirements specifications.

The static phase of the Java-MaC architecture starts with a formal requirements specification, which is written in both high-level and low-level specifications. Java-MaC makes use of two event-based formal languages, the Primitive and the Meta Event Definition Languages (PEDL and MEDL), which are used for writing low and high level specifications respectively. PEDL is tightly related to the programming language. Specifications written in PEDL contain the definitions of primitive events and conditions expressed using these events. Such definitions are given in terms of program entities such as program variables and program methods and their purpose is to assign meanings to the program entities. MEDL specifications consist of required safety properties. Primitive events and conditions are used to express these safety properties. Intuitively, a condition is a state predicate and an event is an instantaneous state change. The reporting capabilities of the runtime checker are described in the MEDL specifications, as well. MEDL uses alarms to express a violation of a property. An alarm is an event that should not occur during an execution. If an alarm fires during an execution, then a user notification is issued.

Once the specifications are written, the next step is the generation of the runtime modules. Low-level specifications generate a filter that is inserted into the byte-code of the monitored program using an automatic instrumentation procedure. An event recognizer is also generated automatically by translating the PEDL specification. Similarly, a runtime checker is generated automatically from the higher-level MEDL specification.

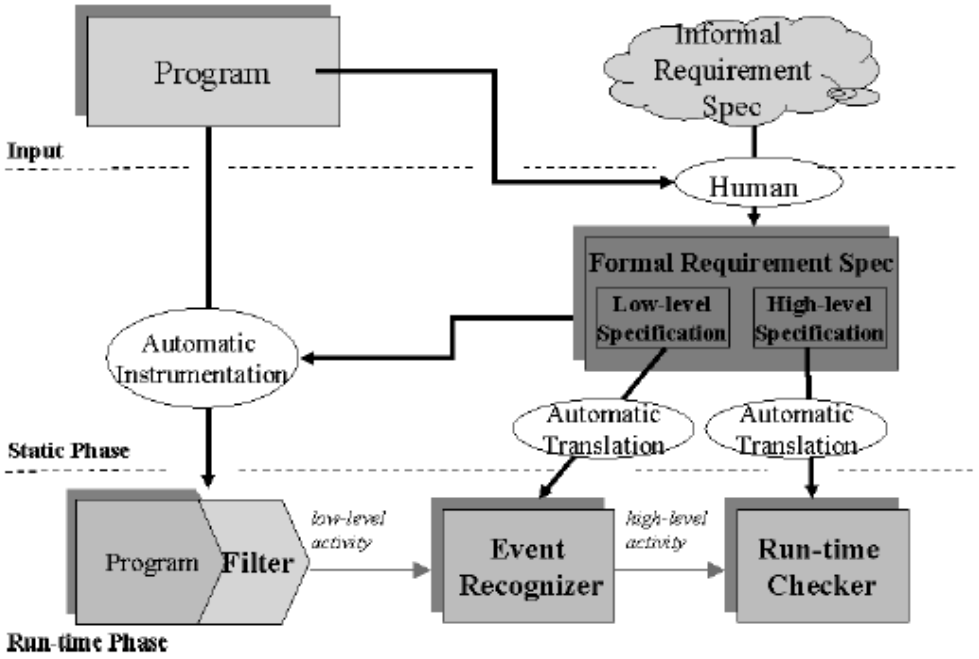

Figure 2-9 – The Java-MaC architecture [93]

During the runtime phase, the instrumented program is been monitored and checked against the requirements specification. The filter keeps track of changes of monitored objects and sends relevant information about the execution trace to the event recogniser. The event recognizer detects events from the state information received by the filter. An event can be either a primitive event (such as a method call) or a change in the state of a condition. Recognized events are sent to the run-time checker, which determines whether or not the current execution trace satisfies the requirements specification and raises an alarm if a violation is detected.

2.2.2.4.3 The Java Monitoring-Oriented Programming Framework (Java-MoP)

Chen and Rosu [34] proposed a development and analysis framework for Java, the Java Monitoring-Oriented Programming (Java-MoP). Java-MoP follows the MOP paradigm and thus monitoring is one of its fundamental principles. It also provides the capability of recovering from errors (specification violations) at runtime.

According to its proposed distributed architecture, annotations formally describing requirements on past, current and future states, have first to be inserted into the monitored Java source code, in the client side. Java annotation processors send these annotations to the appropriate logic plug-ins, which reside at the server side. Essentially, each of the

logic plug-ins implements an algorithm for synthesising monitoring code for a specific formalism. Logic plug-ins support past and future time variants of temporal logics, as well as, extended regular expressions. Furthermore, Jass [21, 115] and JML [100] annotations can be used. These specific annotations do not require a special logic plug-in, only a Java shell to transform them into Java executable code.

Once the annotations have been transformed into Java executable code at the server side, they are sent to the client side. Java assertion processors integrate the received code in the system, according to the configuration attributes of the monitor. In addition, the client side modules are also responsible for the system's code instrumentation for emitting events, in case of an external monitor. In this Java-MoP implementation, AspectJ [91] is used as the instrumentation mechanism.

The checks, which can be carried out by using Java-MoP, depend on the monitoring properties. Thus, a monitor implemented in Java-MoP can check for class invariants at every change of class state or for method pre/post-conditions. Also, a monitor can be configured to halt the program's execution while it carries out specific checks which deal with critical properties (synchronised keyword). In case that a non-critical property must be checked, a monitor's reply may not be important, so the system keeps running during the check (not synchronised keyword).

MOP allows one to control the execution of a monitored program. By allowing users to specify handlers for the violation or validation of monitored properties, Java-MoP can support the runtime control and recovery of a monitored Java program. These handlers can either simply report errors and throw exceptions or take more complicated actions, like resetting states and performing alternative, error-correcting computations.

2.2.2.4.4 The Jassda Framework

As an alternative approach to Jass [21, 115], Jassda framework [25, 26] checks assertions on traces by observing the events generated for debuggers through the Java Debug Interface (JDI). An obvious shortcoming of this alternative is that the monitored program must be running in the debugging mode.

The Jassda tool allows the dynamic verification of a system written in Java against a CSP-like specification. The events from the monitored system are obtained through a general event extraction and dispatching facility, the Jassda framework [25, 26]. This

framework can also be used for other purposes, e.g., to log events or to stimulate a program for testing purposes.

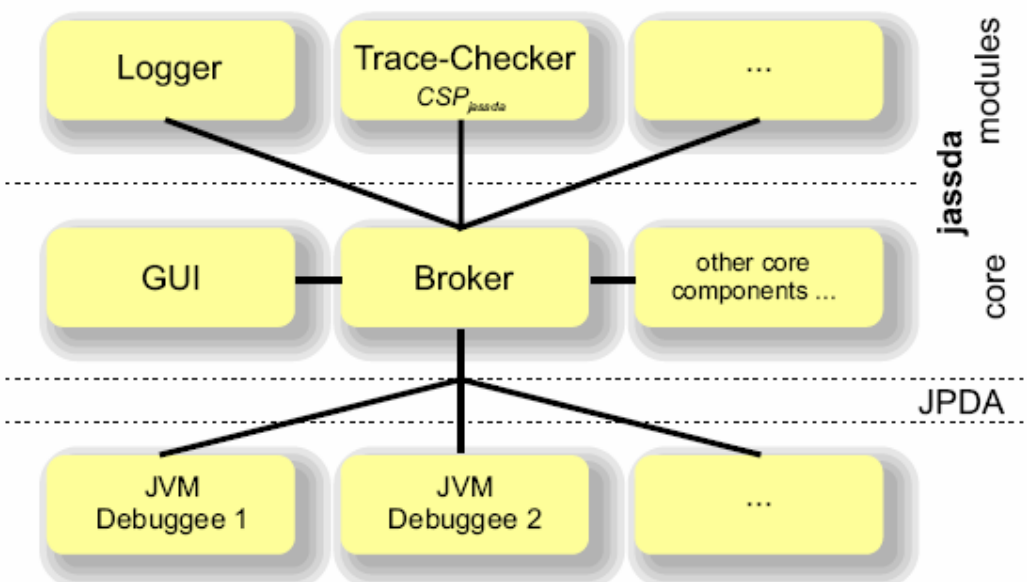

Figure 2-10 – The architecture of Jassda framework [25]

The architecture of the Jassda framework is shown in Figure 2-10. At the lowest level JVMs execute the monitored system's code (debuggees). These debuggees are connected to the Broker, which is the central component of the Jassda framework. The "Registry" database, an optional graphical user interface and the Broker build the Jassda core. Other Jassda modules connect to this core requesting and consuming events. The connection between the debuggees and the Jassda core transports the events that we want to observe. This connection is established by using the Java Platform Debugger Architecture (JPDA). The Jassda tool development aimed to achieve a method for monitoring Java programs which would be as less code intrusive as possible. The Java Debug Interface (JDI) [157] was used for this purpose.

During runtime the debuggees can be configured to generate events for several situations, e.g. a method has started or terminated, an exception has occurred, a breakpoint is reached, a class is loaded/unloaded, read/write access to a variable, a thread was started/stopped. After having emitted an event, the debugging VM can be configured to suspend execution and thus allow a deep view into the VM. For example, for each currently running thread its stack trace can be analyzed or for each class its inner structure (like super-classes and implemented interfaces) can be read. Even the byte-code of every method can be accessed for further analysis.

The Logger module logs the execution of a Java system by writing its sequence of events into a file. The amount of information that can be derived from an event as well as the alphabet of events can be configured by an XML-based configuration file. The most important event listening module is the Jassda Trace–Checker. The Trace–Checker reads one or more trace specifications written in CSPJassda and builds an internal process representation for the set of legal traces. With every received event the Trace–Checker will ensure that this actual sequence of events is a legal trace of the specification's process representation or stop the program and inform the user about the violation.

2.2.2.4.5 The Temporal Rover Toolset

The Temporal Rover [57] is a commercial toolset, which performs dynamic verification of temporal properties over programs written in Java, C, C++, VHDL, Verilog, and ADA. This is achieved by adding extra LTL/MTL assertions to the program source code. These assertions are embedded as comments into the source code. The Temporal Rover parser converts program files into new files, which are identical to the original files except for the assertions that are now implemented in source code.

The Temporal Rover adopts a coarse-grained view of the state model. A state constitutes the values of variables within the scope of a given method. Method execution is viewed as an event that causes transition between states, and properties are evaluated only at the completion of a method execution. Clearly, it misses invalid states that may occur during the execution of a method. Properties are written inside methods and predicates map to the variables within the scope of the method. Consequently, each property has a unique perspective of the environment that it is validating and properties may not be composed. For example, even though one would imagine that two contradicting properties could be composed and reduced to "false", this is not the case under the Temporal Rover's state model. A property's notion of time refers to the next execution of the method containing it. Two properties may therefore carry different semantics for the next-state operator. Another limitation of Temporal Rover is that under its state model one can not reason about control intensive properties such as method x must never execute after method y. The DBRover is a distributed-monitor version of the Temporal Rover where assertions are monitored on a remote machine, using HTTP, sockets or serial communication with the underlying target application.

2.2.2.4.6 The Java PathFinder (JPF) Framework

Java PathFinder (JPF) [168] is a model checker for Java byte code. More specifically, it is a specialized Java Virtual Machine (JVMJPF), which runs on top of the underlying host JVM. In contrast to the standard JVM, JVMJPF executes the program in all possible ways. The state space of a program is thus the resulting computational tree, whose branches are determined by the threads' instructions and possible values of input data. JPF supports depth-first, breadth-first as well as heuristic search strategies to guide the model checker's search in cases where the state explosion problem is too severe [168]. JPF contains no mechanism of its own to specify user-defined properties, but rather integrates with the Bandera toolset [44] and accepts the Bandera Specification Language (BSL) [43]. Even though JPF carries an elaborate state model (being able to capture every state of the JVM), temporal property specification is limited to the capabilities of BSL. Figure 2-11 depicts the JPF architecture.

Like other model checkers for concurrent programs, JPF supports the partial order reduction (POR) [36]. The purpose of this technique is to lower the state space size via including in the state space only one interleaving of instructions that are both independent and executed by different threads. The consequence is that JPF actually traverses a reduced state space where each state is associated with one of the following events ("points") in the byte code execution:

- Scheduling point: The current instruction is thread scheduling relevant (e.g. it accesses a shared variable, starts/stops a thread, blocks a thread, etc.)
- Value point: A value selection takes place.

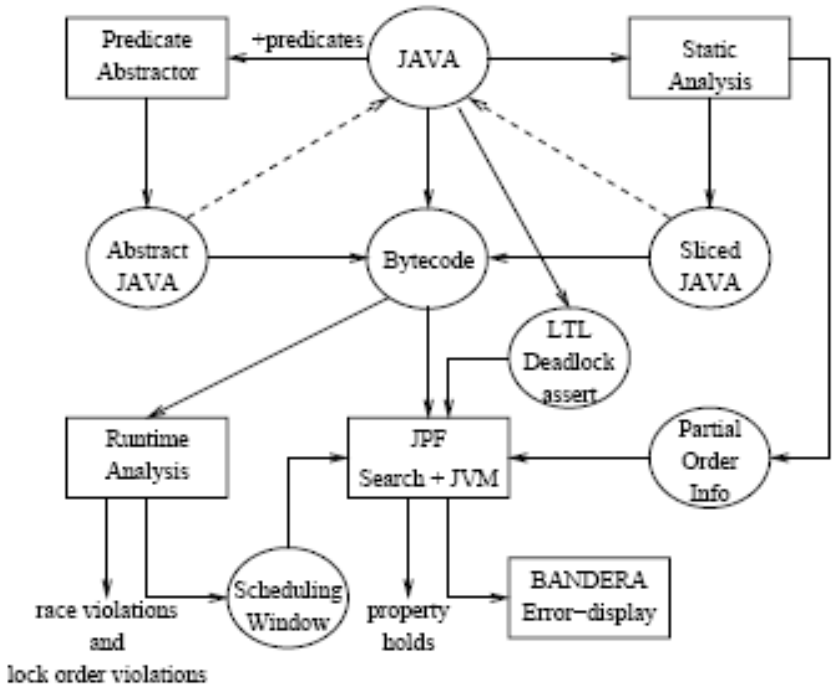

Figure 2-11 – The JPF Architecture [168]

In order to check a code unit (e.g. a method) for different values of input data, JPF contains a static class Verify which provides methods for a systematic selection of values of virtually any type. The methods of Verify are to be called in the checked code. For example, if the checked code unit executes Verify.random(3), an integer value from the range 0..3 is selected. However, after reaching an end state, JPF backtracks up to the Verify.random(3) call and selects another value from 0..3; this is repeated until all the values from this interval have been used for execution. By employing methods of Verify the state space size increases since each selected value creates a different branch in the state space.

By default, JPF searches the state space of the checked program for "low-level" properties like deadlocks, unhandled exceptions and failed assertions. However it is extensible via the publisher/listener pattern and as such it allows for observing more general properties. Since Java code assertions must always hold, temporal properties specified outside of BSL can be checked as well. This way, listeners can check for specific state-based properties.

Each state of a checked program in JPF consists of the heap, static area and stacks of all threads. When traversing the state space, JPF checks whether the current state has already been visited. If this is so, it backtracks to the nearest scheduling/value point, for

which an unexplored branch exists and continues along that. This backtracking is based on keeping a stack representing the currently explored path in the state space (an item in the stack determines the list of yet unvisited branches).

The Bandera toolset [71] is a collection of program analysis, transformation, and visualization components designed to allow experimentation with model-checking of Java programs. Bandera takes as input a Java source code and a program specification written in Bandera's temporal Specification Language (BSL), and produces a program model and a specification as input to model-checking applications, like Spin [80] and Java PathFinder [168]. Then, Bandera uses the corresponding model-checker to prove whether the model satisfies the required specification (i.e. whether the Java program satisfies the BSL specification). If the specification is not satisfied, then a counter example trace is returned. Bandera uses this to show the problematic execution path directly in the original Java code. Bandera deals with the state explosion problem and the fact that the program state models must be finite by providing data abstraction and program slicing methods when customizing the model. These features help produce a much smaller finite state model of the Java program.

In particular, Bandera consists of five major components:

- Property specification is supported in Bandera through the use of global properties (e.g., deadlock) and application specific properties (e.g., assertions and temporal logic formulas). Users define observations of the execution state of a Java program, as predicates over program locations and data values in the program. Assertions and temporal formulas are then defined in terms of those observations.

- Program slicer: Automates the elimination of program components that are irrelevant for the property under analysis. Slicing criteria are automatically extracted from the observable predicates that are referenced in the property. Bandera's Java slicer treats multi-threaded programs [71] and is based on calculation of the program's data dependence graph.

- Program abstraction which can be summarized as: (i) definition of an abstraction mapping, which is appropriate for the property being verified, (ii) use of the abstraction mapping to transform the temporal property into an abstract property, (iii) use of the abstraction mapping to transform the concrete program into an

abstract program, (iv) checking whether the abstract program satisfies the abstract property, (v) reasoning about the satisfaction of the concrete property by the concrete program.

- Verification code generator: Transforms the sliced, abstracted program into the input format of a selected model checker. This component is also responsible for establishing the correspondence between the states of the produced model and the states of the original program.
- Counter-example interpreter: Involves the mapping of low-level, verifier-specific counter-examples back to the Java source code. Facilities for navigating through the counter-example and displaying the values of both stack and heap allocated data are provided through a debugger-like interface.

2.2.2.4.7 The JNuke Tool

JNuke is a framework for static and dynamic analysis of Java programs [12, 13]. It was originally designed for dynamic analysis, including explicit-state software model checking and runtime verification.

JNuke's virtual machine (VM) is the core element of the framework. For generic runtime verification, the engine executes only the program once according to a given scheduling algorithm. The VM API allows for event-based runtime verification through various runtime algorithms. This API provides access to events occurring during the program execution. Event listeners can, then, query the VM for detailed data about its internal state and thus implement any runtime verification algorithm, including detection of high-level data races [10] and stale-value errors [11].

Before the execution of the monitored program, the class loader transforms the Java byte code into a simplified form containing only 27 instructions, which is then transformed into a register-based version [13]. Execution of the program generates an event trace. During execution, the runtime verification API allows event listeners to capture this event trace. These listeners are used to implement scheduling policies and runtime verification algorithms, like Eraser [141] and detection of high-level data races [10]. The verification algorithm is responsible to copy data it needs for later investigation, as the VM is not directly affected by the listeners and thus may choose to

free data not used anymore. Figure 2-12 presents an overview of the JNuke VM and how a runtime verification algorithm can be executed by using callback functions in the VM.

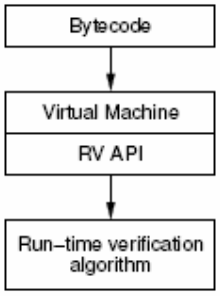

Figure 2-12 – Runtime verification in JNuke [13]

JNuke was expanded with static analysis capabilities at a later stage. Static analysis is usually faster than dynamic analysis but less precise, approximating the set of possible program states. In static analysis, iterations over these approximated states are carried out until a fix point of them is computed [45]. Properties checked with static analysis require summary information of dependent methods or modules. Figure 2-13 illustrates the separate classical approaches for dynamic and static analysis.

On the other hand, dynamic analysis examines properties against an event trace originating from a program execution. By using a free data flow analysis graph [113] static analysis can work similarly to the dynamic execution. Analysis algorithms based on such a graph can allow for non-deterministic control-flow and use sets of states rather than single states in its abstract interpretation [13]. Moreover, in such a graph data locality is improved because an entire path of computation is followed as long as valid new successor states are discovered. Thus, all Java methods can be executed, allowing for a generic analysis algorithm to be executed under both static and dynamic analyses. The chosen analysis algorithm runs until an abortion criterion is met or the full abstract state space is exhausted.

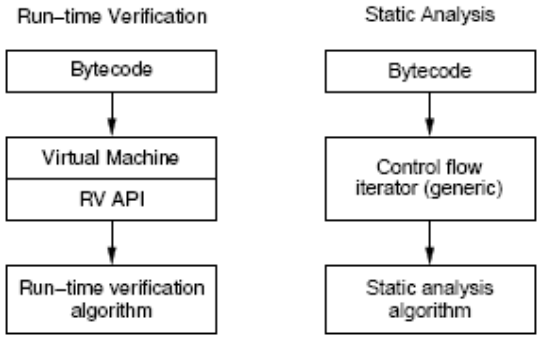

Figure 2-13 – Classical approach for dynamic and static analysis [13]

A generic analysis represents a single program state or a set of program states at a single program location. It also includes a number of event handlers, which model the semantics of byte code operations. Both static analysis and runtime analysis trigger an intermediate layer, which evaluates the events. The environment hides its actual nature (static or dynamic) from the generic algorithm and maintains a representation of the program state that is suitably detailed.

Figure 2-14 shows the generic analysis principle. Run-time verification is driven by a trace, a series of events e emitted by the runtime verification API. An event represents a method entry or exit, or execution of an instruction at location l. Runtime analysis examines these events directly. The dynamic environment, on one hand, uses the event information to maintain a context c of algorithm-specific data before relaying the event to the generic analysis. This context is used to maintain state information s that cannot be updated uniformly for the static and dynamic case. It is updated similarly by the static environment, which also receives events e, determining the successor states at location l which are to be computed. The key difference for the static environment is that it updates c with sets of states S. Sets of states are also stored in components used by the generic algorithm. Operations on states (such as comparisons) are performed through delegation to component members. Therefore the "true nature" of state components, whether they embody single concrete states or sets of abstract states, is transparent to the generic analysis algorithm, which can thus be used either statically or dynamically.

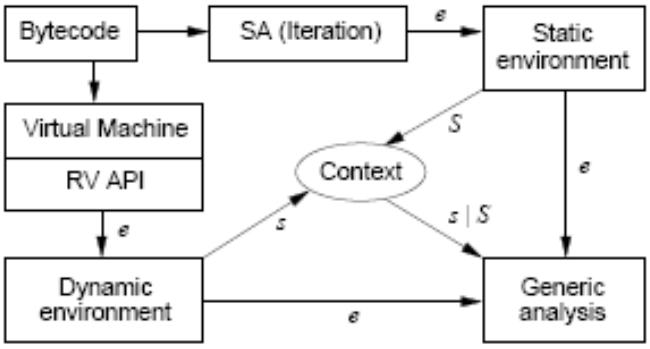

Figure 2-14- Generic analysis for both a static & dynamic environment [13]

The abstract domain for the static analysis is chosen based on the features required by the generic analysis algorithm to evaluate given properties. Both the domain and the properties are implemented as an observer algorithm in JNuke. Future algorithms may include an interpreter for logics such as LTL. Interpretation of events with respect to

temporal properties would then be encoded in the generic analysis while the event generation would be implemented by the static and dynamic environment, respectively.

2.2.2.4.8 Summary of Dynamic Verification Tools

The following table summarizes the surveyed verification tools in terms of the general dynamic verification approach steps of Table 2-2.

Table 2-2 – Summary of Dynamic Verification Tools

Tool	Language for Properties Formalization	Methods for Events Emission	Monitor	Category
JPaX	Temporal logic in Maude rewriting tool	Automated instrumentation by using JSpy (modified JVM)	Observer	O, pre/post, A
Java-MaC	Past-time interval temporal logic	Automated instrumentation (instrumentor)	Runtime Checker	O, pre/post, A
JMoP	ptLTL, ftLTL, EREs	Automated instrumentation by using AspectJ	Embedded in code or parallel process to the system on the same or different machine	OC, pre/post, S/A
Jassda	CSP_{Jassda}	API based (from JVMs by using the Java Debug Interface)	Trace checker	OC, pre/post, A
Temporal Rover	LTL/MTL assertions	Instrumentation	Embedded (using alternating finite automata)	OC, pre/post, S
JPF	User defined assertions, LTL (by using BANDERA)	BANDERA abstraction capability	JVM^{JPF}	OC, A
JNuke	-	API based (JNuke VM with RV API)	Runtime verification algorithm	O, post, A

2.3 Abductive reasoning

A key characteristic of the approach that has been undertaken for the generation of diagnostic information for S&D violations is the use of abductive reasoning. Thus, in this section we provide an overview of research in this area and highlight the basic aspects of this research to enable the reader understand the relationship of our approach to it.

Abduction in general is defined as the reasoning process of generating explanations for a set of observations and searching among them to find the best one. According to Peirce [127], abduction is an inference process from effect to cause. In the context of artificial intelligence, the standard formalization of abduction defines an explanation as a set of assumptions/hypotheses, which, along with the underlying knowledge, logically entails a set of observations [33]. Therefore, if φ explains ω, in connection with the underlying theory T in an abductive fashion, then ω must be derivable from $\varphi \cup T$.

Our overview of the related work related to abduction focuses on the logic based approaches to abduction, including temporal logic based abduction, and the selection criteria that have been proposed in the literature in order to make selections of explanations produced by abduction in cases where more than one such explanations are generated. This is because research in these areas is most closely related to the approach that has been described above.

2.3.1 Logic-Based Abduction

Abduction based on models expressed in some logic-based language is the most widely accepted approach by the researchers in the field of abduction [33, 59, 86, 87, 97, 122, 135]. In this approach, the knowledge, which is represented in any logical language for deductive reasoning purposes, is also used in abductive reasoning. A logic-based representation for abduction consists of a theory T, which is defined in some logic language. The set of predicates or sentences or symbols, which can be accepted as explanations to observations, are called abducibles [41, 135] and abducibles can only be members of the body of the underlying theory formulas.

If the abductive process results in a sentence φ as an explanation of observation ω, the following conditions must be satisfied:

- $T \cup \varphi \vdash \omega$
- $T \cup \varphi$ to be consistent, and
- φ contains an abducible predicate or a set of abducible predicates

In the context of abduction, the relationship between φ and ω is considered as some kind of causal relationship including for example interpretations of the form "φ is the reason for ω being true". However, as Levesque has shown [103], abduction is not concerned exclusively with relationships between causes and effects. Levesque [103] suggests the extension of the notion of explanation in order to grasp that φ is sufficient, and not only necessary, to sanction a belief in the proposition ω. Consequently, there must not be a direct causal relationship between both φ and ω, but, in connection with the domain theory, φ is enough for ω to be true.

Generally, a basic abductive reasoning procedure operates as follows according to [148]. The underlying domain knowledge is formulated in a set of clauses in some logic language (a theory representation). For a given conjunction of input literals/sentences, which have to be explained, the abductive inference procedure computes all the possible abducibles by backward chaining on the input literals/sentences, using the clauses of the underlying theory. This procedure is similar to the way that proofs are computed in Prolog. In case that there is no fact or consequent of a rule in the underlying theory, which can be unified with a sub-goal of the current proof, the proof does not fail. On the contrary, the abductive reasoning procedure provides the choice of flagging that sub-goal as an assumption/explanation, assuming that there are not any consistency implications. The abductive reasoning procedure gives a proof of the conjunction of the input literals using the rules and facts of the underlying theory, together with a set of assumptions/hypotheses. Briefly, an abductive proof is considered as an explanation of the input literals in connection with the logically encoded underlying knowledge.

In the context of logic programming with integrity constraints, Eshgi and Kowalski [59] have also considered abduction as an alternative framework to the principle of *negation as failure* (NAF). More specifically, these authors have shown that if negative conditions are considered as abducibles and appropriate denials and disjunctions are imposed as integrity constraints, negation as failure can be successfully simulated by abductive reasoning. In this approach, logic programs using negation as failure are converted into an abductive framework, where integrity constraints that are more general than denials can be defined. For this reason, an abductive formulation includes a Horn

clause theory (T) without denials (i.e., negated predicates), a set of integrity constraints (I) and a set of abducible predicates (A). In this context, an explanation set E is an abductive solution for the query q if and only if E consists of a set of variable free abducible predicates, $T \cup E \models q$ and $T \cup E \cup I$ is satisfiable. Note that in this approach, integrity constraints can be considered as a selection criterion for alternative explanations that are produced by abductive reasoning.According to this criterion, abduced hypotheses, which do not satisfy the integrity constraints, are ruled out.

The transformation of NAF-formulation into an abductive formulation can be done in three steps. Firstly, all negative atoms ¬n are replaced by new atoms n*. Subsequently integrity constraints of the form "$\leftarrow n^*(x) \wedge n(x)$" are added to the theory. The newly added integrity constraints ensure that both n(x) and n*(x) cannot be true simultaneously for each value of x, as n* has replaced ¬n. Finally, all n* have to be declared as abducibles. The above conversion procedure succeeds in eliminating all negative atoms by introducing new unambiguous abducible atoms for them. Also, instead of testing the provability of negated conditions by negation as failure, the consistency of abducible predicates is checked.

2.3.2 Temporal Abduction

Temporal abduction refers to cases where the observations, which should be explained, are associated with temporal information, as in the case of the diagnostic framework. Console et al. [41] describe temporal abduction as a type of reasoning for "generating explanations, which do not only account for the presence of observations, but also for temporal information on them, based on temporal knowledge in the domain theory". In temporal abduction problems, temporal knowledge can be expressed as temporal constraints [28, 41], which are associated to the rules of the underlying domain theory. Such temporal constraints must be satisfied by the temporal information associated to the generated explanation. On the other hand, in cases where the underlying theory is expressed in some temporal language, like *Event Calculus*, temporal knowledge can be represented as information embedded in the underlying theory formulas.

Console et al. [41] have proposed a temporal abduction approach, which makes use of temporal constraints associated with the observations and the formulation of the underlying domain theory. The temporal consistency check of each candidate explanatory formula, which could lead to a plausible explanation, is used as a pruning criterion for the

set of the accepted candidates, in every step of the abductive backward chaining procedure. Thus, only the temporal consistent candidates are used for building a plausible explanation. Since the temporal consistency checks required in each step of the abductive procedure can be computationally expensive, the Simple Temporal Problem framework (STP) [51] was used in this approach. The STP framework treats the temporal checks as a constraint satisfaction problem. More specifically, in the STP framework, each of the binary constraints, which represent time dependencies between the actual times of events, contains only one time interval. In particular Console et al. [41] used LATER [29], which is a general purpose temporal reasoning system dealing with special classes of temporal constraints as the aforementioned ones.

The main differences between the work in model based diagnosis and our abduction based explanation process are that our process is based on Event Calculus, treats the time constraint satisfaction problem as a linear programming problem, generates all the possible alternative explanations for the observations (whereas others' frameworks generate a single explanation), and provides overall beliefs in the validity of explanations by looking at the consequences of such explanations. These beliefs are also used in order to rank explanations and select some of them as the most plausible ones.

2.3.3 Selecting Abduced Explanations

Generally, the evaluation and selection of one or more explanations from the entire set of explanations, which an abductive reasoning process can generate, is one of the main problems in abductive reasoning. The problem is how to choose effectively among the alternative explanations, which might have been generated by abductive inference for a given observation. In the relevant literature, there are several criteria that have been proposed in order to assess and select the most preferred abductive explanation. Generally, these criteria fall into two categories. The first of these categories includes criteria, which require the *logical* and *syntactic simplicity* of the abductive explanations [87, 122]. The second category includes criteria which favour the explanation specificity in the selection process [8, 120]. In the following, we review each of these categories in more detail.

2.3.3.1 Logical and Syntactic Simplicity Criteria

In the context of abductive reasoning, "simplicity" is generally interpreted as logical simplicity. Logical simplicity means that the preferred assumptions/hypotheses are those that contain the fewest different abducibles. Therefore, the preferred explanations are those that require the fewest additional assumptions/hypotheses to what has been observed.

The basic criteria that underpin syntactic simplicity in the literature are the criteria of "non-triviality", "basicness", and "minimality". To appreciate the basic criteria, which underpin syntactic simplicity, suppose that T is a first-order theory (knowledge base) and E is a set of hypotheses/explanations for an observation Ω. According to the simplicity criteria, *e* can be considered as an accepted explanation, if it belongs to E and satisfies the following conditions:

- T ∪ e must be consistent (i.e. a "consistency" criterion that is present in most logic-based approaches to abduction)
- The observation Ω must not be a direct consequence of the hypothesis *e*. (this condition precludes that Ω itself can be considered as a feasible explanation of Ω ("non-triviality" criterion)
- Every consistent explanation *e* must be trivial, i.e., there must be no other explanation for the explanation itself. This criterion favours the most specific explanation ("basicness" criterion)
- There must not exist any more general explanations for Ω than e ("minimality" criterion)

Although the importance of logical simplicity is stressed other forms of simplicity have also been proposed in the literature. Peirce [127], for example, has proposed the use of "psychological simplicity". The criterion of "psychological simplicity" suggests the selection of an explanation hypothesis if this hypothesis is the one that would be intuitively preferred. Current research, though, favours logical simplicity selection criteria, due to the fact that "psychological simplicity" is hard to define and implement.

2.3.3.2 Specificity Selection Criteria

Following the logic simplicity criteria, someone may still end up with more than one alternative explanations of an observation. Thus, additional selection criteria are often required in order to reduce the set of alternative hypotheses. An additional criterion used for this purpose is a criterion that requires the explanation to have a certain level of specificity. Relevant approaches for selecting the preferred abductive explanation, which focus on a certain level of specificity, have been classified into two groups by Appelt and Pollack [8], namely the global and local criteria.

2.3.3.2.1 Global criteria

Global criteria are heuristics, which can be applied for directing the selection procedure to the most preferred abductive explanation, by considering entire sets of explanations. Appelt and Pollack [8] have distinguished some selection criteria, which are global criteria, namely: cardinality based criteria, least presumptive or least specific abduction and most specific abduction.

Cardinality comparisons were first introduced in diagnosis applications to ensure that the preferred explanations imply the failure of the smallest number of components of the under examination system. In diagnosis applications in particular, the system which is under examination is considered as a set of distinguishable components whose intended input and output behaviour is completely specified in terms of a base theory. The diagnostic task in this setting involves an abductive inference procedure reasoning for the faulty system behaviour, as it has been observed during the system's runtime. The output of the abductive procedure is an explanation set, which explains the captured faulty behaviour of the system. More specifically, the explanation set identifies groups of system components, which have failed, and whose failure could account for the faulty behaviour of the system. The accepted explanation set, under the evaluation criterion of this approach, should imply the failure of the smallest number of components. As Appelt and Pollack note in [8], the cardinality comparison criterion cannot be generally applied, due to the fact that it assumes that the system that is the subject of diagnosis always has a set of distinguishable and enumerable components. However, in other contexts, for example natural language understanding and plan recognition systems, the notion of distinguishable and enumerable components is difficult to define.

The "less presumptive explanation" criterion was suggested by Poole in [130], as an alternative to the cardinality criterion. According to it, given two alternative sets of explanation E1 and E2 and a logical theory T, E1 is less presumptive than E2 if and only if $T \cup E2 \models E1$. An abductive inference procedure, which is called "least specific abduction" and realizes the aforementioned selection criterion, has been proposed by Stickel in [156]. Note that the less presumptive or least specific explanations provide the most general explanation. Thus, least specific abduction is not necessarily an appropriate strategy for diagnostic tasks, which require very detailed knowledge about the origin of failures and there are diagnostic tasks, as the ones that we are dealing with, where the most specific explanation criterion is considered more adequate.

2.3.3.2.2 Local Criteria

Local criteria are considered as an alternative to global criteria for evaluating alternative abductive explanation sets [8]. Generally, this category of criteria assumes that weights are associated with the rules of the underlying theory. Each explanation set, then, is evaluated by combining the weights of the rules, which were used to derive the members of the set. Appelt and Pollack [8] distinguish further local criteria into: weighted abduction, cost-based abduction and Bayesian statistical methods.

2.3.3.2.2.1 Weighted Abduction

Appelt and Pollack [8] have proposed another approach, called "weighted abduction", for assessing alternative explanations. This approach introduces the concept of explanations costs during the abductive inference procedure.

Assuming that T is a theory used in abduction, an underlying preference order on the models of T is assumed. In weighted abduction, one set of hypotheses/explanations A_1 is better than an alternative A_2, if A_1 restricts the models of T to a more highly valued subset than A_2 does. The weights, which are assigned on the literals/atoms of the rules of T, impose implicitly constraints on the assumed preference order. For instance, a rule $p^{\alpha} \supset q$ with a weighted literal p in its body ($\alpha<1$) reflects that there is a preference order of the models, which satisfy q. The models, which are more preferred than every model satisfying q, are those satisfying p as well. By this means, if p is satisfied by a model then that model provides an explanation for q.

In general, in the context of weighted abduction, assuming that there is an abduction problem with goal φ, an underlying theory T and a set of explanations A, an abductive proof would be the search for a set of explanations A such that the preferred models of T ∪ {φ} to satisfy T ∪ A. Regarding the goal φ, the set of explanations A must be adequate (T ∪ A |- φ), consistent (T ∪ A |/- ¬φ) and syntactically minimal (if ψ∈A then T ∪ A – {ψ} |/- φ). The explanation set A, also, must satisfy a "semantic greatest lower bound" condition requiring that the model which entails T ∪ A must be the most preferable one. Finally, the A set must not entail the negation of its elements at a cost that is less than that of the A set itself (i.e., the "defeat" condition).

According to the algorithm proposed in [8], all the explanation sets for a goal φ are generated by using the theory rules, which are Horn clauses whose body literals are associated with a weighting factor (i.e. rules of the form $p_1^{w1} \wedge \ldots \wedge p_n^{wn} \Rightarrow q$). The goal φ can be either assumed at its predefined assumption cost or unified with a fact in the knowledge base (zero cost proof), or unified with another atom that has already been assumed, or proved by applying backward chaining using a rule $p_1^{w1} \wedge \ldots \wedge p_n^{wn} \Rightarrow q$. In the latter case, the proof cost of the goal matched with q is computed by multiplying the weights w_i of the conditions p_i in the body of the rule. In this process, the best solution of the abduction problem is the explanation set with the lowest cost proof. The algorithm also filters out explanation sets, which do not satisfy the consistency and the defeat conditions.

The weighted abduction approach has some computational difficulties as pointed out by Appelt and Pollack which arise due to the need to determine a candidate set at a minimal cost, as well as, guarantee that all candidate explanations are consistent with the underlying theory. In particular, the detection of the minimal cost explanation requires the computation all possible explanations with a direct effect in the computational cost of the overall process. Regarding the consistency of the abduced explanations with the base theory, it has been claimed that in some domains an incomplete inconsistency check, focusing only on specific points, could be applicable. For instance, in the context of TACITUS text understanding system [79], most of the inconsistencies result from the erroneous identification of two distinct individuals. Thus, most of the inconsistencies can be detected by checking variable typing constraints for the assumed literals. An incomplete consistency check can be computed relatively quickly and thus, where

applicable, it is more preferable than the exhaustive detection of all the inconsistencies that could arise.

Moreover, regarding the assignment of weighting factors on the literals/atoms of the rules of the underlying theory, two issues have been identified in [8]. The first issue is that, the encoding of preference as weighting factors assigned on the atoms of the theory rules is difficult, because the rules might not provide the right collection of atoms for attaching the weights. The second issue is that, the assignment of weight factors on a particular rule may have an effect on the set of preferences as a whole. Thus, the difficulty of the preference encoding process is that all the preferences introduced by each rule must be considered in connection with all the other rules in the underlying theory.

2.3.3.2.2.2 Cost-based Abduction

Ng and Mooney [120] have suggested a metric for assessing the quality of the abductive explanations, called "explanatory coherence metric". Briefly, explanatory coherence controls the choice of the hypotheses, in a fashion that, the preferred hypotheses have the more connections between any pair of observations in the proof graph. The explanations generated by the coherence criterions, as Ng and Mooney [120] point out, are usually syntactically simple explanations, due to the fact that tight connections between the observations and explanations are preferred. In contrast with the weighted abduction approach, the explanatory coherence is a more concrete criterion than rule weights, due to the fact that the rule weights can be chosen arbitrarily. On the other hand, in order to cope with incomplete information, the use of both coherence and likelihood knowledge has also been considered.

2.3.3.2.2.3 Bayesian Inference Methods

Due to the fact that uncertainty is an inherent feature of abductive reasoning, the likelihood of truthness of abducible explanations, can play significant role in the selection of the most preferable abductive explanation. Thus, probabilistic models and, in particular, Bayesian models [50, 83, 84, 85, 96, 123, 124, 125, 126, 131, 132] have been used to identify the "most probable" abductive explanation. The use of Bayesian models imposes some limitations in the generality of logic-based abductive reasoning. In particular, the set of possible hypotheses must be determined in advance. Moreover, an a

priori probability must be assigned to each of the possible hypothesis in advance, as well as, the conditional probabilities of consequences, given particular assumptions, must be predetermined. When these prerequisites are met, the Bayes' theorem can be applied in order to compute the conditional probabilities of the predefined possible hypotheses, given the observations to be explained. Based on the outputs of the Bayes's rule, the most possible combination of hypotheses, which jointly explain the observations, is selected.

An approach, which addresses the diagnostic problem under the notions of abduction, time and uncertainty, was proposed by Santos [140], Santos presents an extended model of Bayesian networks, which except for abductive and uncertain reasoning; temporal reasoning is considered as well. Also, Santos provides a description of a general computation method for the proposed model, which is based on linear constraint satisfaction that determines the least weighted explanation. The proposed model uses an interval representation of time based on Allen's interval algebra, while uncertainty is expressed as a probability (weight) assigned to each rule in the knowledge base like in Bayesian networks. In particular, the underlying domain knowledge is modelled as a directed acyclic graph whereby nodes represent propositions/events that can be true or false. Each node is associated with a time range in which the node is true or false due to its truth-value. The nodes are connected with directed arcs, which illustrate the logical/causal and temporal dependency between the connected nodes. Abductive explanations can typically considered nodes without parents.

Our approach also uses a probabilistic explanation assessment approach. However, our approach is not based on Bayesian abduction. The reason for this is to avoid the need to elicit the a-priori and conditional probability measures which are required by this approach. Furthermore, as also discussed in [162, 163, 164], the choice of the Dempster Shafer theory of evidence [146] as the framework for calculating the likelihoods of abduced explanations has been dictated by the need to represent the uncertainty regarding the confirmation of the consequences of these explanations as we are discussing in following section (see Section 5.4) and reason in the presence of this uncertainty.

Chapter 3: Preliminaries

3.1 Overview

The aim of this chapter is to provide the reader with an overview of the underpinning theoretical background of our approach. As already mentioned, our diagnostic approach has been structured upon an event reasoning monitoring framework [109, 153, 154] that is based on Event Caclulus [149]. Therefore, Section 3.2 provides a short overview on Event Calculus.

Section 3.3 provides an extended discussion of the monitoring framework highlighting on the formal specifications it uses. Finally, Section 3.4 covers the principles of the Dempster Shafer theory of evidence [146] that underpins our explanation validity assessment process.

3.2 Event Calculus

The event calculus (EC) [149] is a first-order temporal formal language that can be used to specify properties of dynamic systems, which change over time. Such properties are specified in terms of *events* and *fluents*.

An event in EC is something that occurs at a specific instance of time (e.g., invocation of an operation), has instantaneous duration and may change the state of a system. Fluents are conditions regarding the state of a system and are initiated and terminated by events. A fluent may, for example, signify that a specific system variable has a particular value at a specific instance of time. The occurrence of an event is represented by the predicate Happens(e,t). This predicate signifies that an instantaneous event e occurs at some time t.

The initiation of a fluent is signified by the EC predicate Initiates(e,f,t) whose meaning is that a fluent f starts to hold after the event e at time t. The termination of a fluent is signified by the EC predicate Terminates(e,f,t) whose meaning is that a fluent f ceases to hold after the event e occurs at time t. An EC formula may also use the predicates Initially(f) and HoldsAt(f,t) to signify that a fluent f holds at the start of the operation of a system and that f holds at time t, respectively.

Event calculus defines a set of axioms that can be used to determine when a fluent holds based on initiation and termination events which may have occurred regarding this fluent. These axioms are listed in Table 3-1.

Table 3-1 – Axioms of Event Calculus

(EC-A1) (∃e,t) Happens(e,t)∧ t1<t<t2 ∧Terminates(e,f,t) ⇒ Clipped(t1,f,t2)
(EC-A2) Initially(f) ∧ ¬Clipped(0,f,t) ⇒ HoldsAt(f,t)
(EC-A3) (∃e,t1) Happens(e,t1)∧ t1<t ∧ Initiates(e,f,t1) ∧¬Clipped(t1,f,t) ⇒ HoldsAt(f,t)

The axiom *EC-A1* in Table 3-1 states that a fluent *f* is clipped (i.e., ceases to hold) within the time range from t1 to t2, if an event *e* occurs at some time point *t* within this range and *e* terminates *f*. The axiom *EC-A2* states that a fluent *f* holds at time t, if it held at time 0 and has not been terminated between 0 and t. Finally, the axiom *EC-A3* states that a fluent *f* holds at time t, if an event *e* has occurred at some time point t1 before t which initiated f at t1 and *f* has not been clipped between t1 and t.

3.3 The EVEREST monitoring framework

In this section, we present the EVEREST (*EVE*nt *REaS*oning *T*oolkit) monitoring framework [153, 154].

3.3.1 Specification of monitoring rules and assumptions in EVEREST

In this section, we give an overview of the *EC-Assertion* language [109, 153, 155] that is used by EVEREST monitoring framework in order to express properties (a.k.a. formulas) to be checked at runtime.

As it has been discussed in [109, 153, 155], EVEREST uses two different types of formulas during monitoring, namely *monitoring rules* and *assumptions*. The formulas of the former of these types (i.e., monitoring rules) express the properties that need to be checked at runtime. The formulas of the latter type (i.e., assumptions) are used in order to derive information about the state of the system that is being monitored based on observations of its behaviour and/or the state of the monitoring process itself. Furthermore, as we will see later in the following chapters, in the context of diagnosis, assumptions may also be used in order to express basic causal relations that can help the identification of the possible causes of the violations of monitoring rules.

Despite their different roles in the monitoring process, both monitoring rules and assumptions are specified in *EC-Assertion* [109, 153, 155] a language that is based on *Event Calculus* (EC) [149]. EC is a first-order temporal logic language, which can be used for representing and reasoning about events and their effects over time.

As mentioned above, an *event* in EC is an occurrence that takes place at a specific instance of time (e.g., invocation of a system operation, receipt or dispatch of a message) and may have an effect. The effects of events in EC are represented by *fluents*, i.e., conditions which may change over time. A fluent may, for example, specify that a state indicating that a system has received a message has been reached or that following the receipt of a message a system variable is set to a specific value. In Event Calculus, *fluents* are initiated and/or terminated by event occurrences.

The occurrence of an event in *EC-Assertion* is represented by the predicate `Happens(e,t,ℜ(lb,ub))`. `Happens(e,t,ℜ(lb,ub))` denotes that an event *e* of instantaneous duration occurs at some time *t* within the time range *ℜ(lb,ub)* (i.e., lb ≤ t ≤ ub). Exending standard EC, *EC-Assertion Happens* predicate definition includes the time range *ℜ(lb,ub)* for the time variable *t* due to uniformity and compactness purposes. More specifically, by including the time range for the time variable of the *Happens* predicate within the predicate signature, we aim to have all the information (i.e., event *e*, time variable *t*, and time constraints expressed by *ℜ(lb,ub)*) that is relevant to the *Happens* predicate defined in a single predicate. It should be noted that the uniqueness of an event *e* is based on its occurrence time represented by the time variable *t* constrained by *ℜ(lb,ub)*. Therefore, EVEREST is designed to treat and reason for an event as a set of information that includes temporal constraints for its occurrence. EVEREST uses temporal reasoning as a significant part of its event reasoning strategy. Moreover, considering that *ℜ(lb,ub)* is equivalent to the inequality expression lb ≤ t ≤ ub that can be specified as relational predicates, by using the *ℜ(lb,ub)* notation, we succedd in reducing the number of predicates specified in *EC-Assertion* formulas. The boundaries *lb* and *ub* that define time ranges are specified as expressions over time variables of the other predicates in an *EC-Assertion* formula and time durations following the BNF grammar of Figure 3-1.

```
<boundaryExp>  ::=  <timeVar> | <constant> | <boundaryExp> ["+"|"-"]
                    <durationExp>

<durationExp>  ::=  <pDuration> | <pDuration> "+" <pDuration> |
                    <pDuration> "-" <pDuration> | <coeff> "*" <pDuration> |
                    <constant>

<pDuration>    ::=  <timeVar> "-" <timeVar>

<timeVar>      ::=  "t1" | "t2"  | … | "tn"

<constant>     ::=  <REAL>

<coeff>        ::=  <REAL>

Note: Numeric tokens are represented by <REAL>
```

Figure 3-1 – Grammar for specifying boundaries of time variables

Thus, the boundary expressions for *ub* and *lb* are linear expressions of the form:

$$lb = l_0 + l_1 t_1 + l_2 t_2 + \ldots + l_n t_n$$

$$ub = u_0 + u_1 t_1 + u_2 t_2 + \ldots + u_n t_n$$

where t_i (i=1, ..., n).

Whilst the standard Event Calculus supports the specification of arbitrary events and fluents, *EC-Assertion* that we use for the specification of monitoring rules and assumptions can use only specific types of events and fluents. More specifically, events represent invocations of system operations, responses from such operations, or exchanges of messages between different system components and are specified using the following form:

event(_id, _sender, _receiver, _status, _sig, _source)

In this form:

- *_id* is a unique identifier of the event
- *_sender* is the identifier of the system component that sends the message represented by the event
- *_receiver* is the identifier of the system component that receives the message represented by the event
- *_status* represents the processing status of an event. This status is: (i) REQ-B (REQuest-Before), if the event is a request for the invocation of an operation or a message that has been received but whose processing has not started yet; (ii)

REQ-A (REQuest-After), if the event is a request for the invocation of an operation or a message that has been received and whose processing has started; (iii) RES-B (RESult-Before), if the event is a response generated upon the completion of an operation that has not been dispatched yet; or (iv) RES-A (RESult-After), if the event is a response generated upon the completion of an operation that has already been dispatched.

- *_sig* is the signature of the dispatched message, or the operation invocation or response that is represented by the event.
- *_source* is the identifier of the component where the event was captured.

Fluents also have a restricted form in *EC-Assertion* and are defined as relations between objects of the following form:

$$relation(Object_1, \ldots, Object_n)$$

where *relation* is the name of a relation which associates *n* objects, namely $Object_1$, …, and $Object_n$.

The initiation or termination of a fluent *f* due to the occurrence of an event *e* at time *t* is denoted by the predicates `Initiates(e,f,t)` and `Terminates(e,f,t)`, respectively. A formula may also use the predicates `Initially(f,t0)` and `HoldsAt(f,t)` to denote that a fluent *f* holds at t_0 i.e. the start of the execution of a system, and at any time *t,* respectively.

The assumptions and monitoring rules are specified in terms of the aforementioned predicates and have the general form

$$body \Rightarrow head$$

If a formula of the above form expresses a rule, the meaning of a rule is that if its body evaluates to *True*, its head must eventually evaluate to *True*. A formula of the above form that represents an assumption has the meaning that if body of the formula evaluates to *True* then it can be deduced that its head evaluates to *True*.

Further to the above, it should be noted that the monitoring language requires that only one of the *Happens* predicates in a rule or assumption can have unconstrained lower and upper time boundaries. The predicate with the non constrained time variable in a formula is called “unconstrained” predicate. Unconstrained predicates should always

appear in the body of the formula. During the monitoring process, rules are activated by events that can be unified with the unconstrained *Happens* predicates in their bodies. When this unification is possible, the monitor generates a rule instance to represent the partially unified rule and keeps this instance active until all the other predicates in it have been successfully unified with events and fluents of appropriate types or it is deduced that no further unifications are impossible. In the latter case, the rule instance is deleted. When a rule instance is fully unified, the monitor checks if the particular instantiation that it expresses is satisfied.

An example of a rule that can be expressed in the monitoring language is given by the formula below:

```
∀ _eID1,_C1,_C3:String; t1:Time
  Happens(e(_eID1,_C3,_C1, REQ-A, op(), _C3),t1,ℜ(t1,t1)) ⇒
∃ _eID2:String ;t2:Time Happens(e(_eID2,_C3,_C1, RES-A, op(),
  _C1),t2,ℜ(t1+1,t1+k))
```

The property expressed by this rule is that an event *e(_eID1,_C3,_C1, REQ-A, op(), _C3)* representing the dispatch of a request for the invocation of the execution of the operation *op()* in the component *_C3* of a system must be followed by another event *e(_eID2,_C3,_C1, RES-A, op(), _C1)* representing the receipt of the result of the execution of the operation *op()* in the component *_C3* by *_C1* and that the latter event must be captured at the *_C1* component in no more than k time units after the dispatch of the result by *_C3*. Thus, this rule expresses a *bounded availability* property for the communication channel between the components *_C3* and *_C1* since it requires that results generated by *_C3* are transmitted within a bounded time period.

3.3.2 Standard EVEREST assumptions

Given the EC-based language that is used in EVEREST framework for specifying monitoring rules and assumptions, we should note that are some standard EVEREST assumptions with regards to the conditions that any given fluent holds. In the following, we provide the specifications of the standard EVEREST assumptions.

SA1.
```
Initiates(_e1,_f,t1,R(t1,t1)) ∧
¬∃_e2,t2. Terminates(_e2,_f,t2) ∧
t2 ≥ t1 ∧ t3 ≥ t1  ⇒
HoldsAt(_f,t3)
```

SA2. **Initially**(_f,t0) ∧

¬∃_e1,t1. **Terminates**(_e1,_f,t1) ∧

t1 ≥ t0 ⇒

HoldsAt(_f,t1)

The first standard EVEREST assumption (SA1) states that if there is an event e1 that initiates the fluent f at time point t1 and there is not an event e2 that terminates f at time point t2, where t2 is greater than or equal to t1, then f holds at any time point t3, where t3 is greater than or equal to t1. In other words, to check whether a fluent f holds at some time point t1, EVEREST is implemented to check whether an initiation of fluent f before or at t1 is stored in EVEREST's relevant data base. If there is such an initiation, EVEREST then checks whether there is a termination of f at or after t1. If such a termination does not exist, EVEREST considers that f holds at any time point after t1. Similarly, the second standard EVEREST assumption (SA2) states that if the fluent f holds since the start of the execution of the system being monitored and there is not an event e1 that terminates f at time point t1, then f holds at time point t1.

3.4 The Dempster – Shafer Theory of Evidence

The key characteristic of the Dempster-Shafer theory that underpins our approach is that provides a framework for handling ignorance when assessing the likelihood of a proposition and its negation on the basis of the available evidence [146]. More specifically, the Dempster-Shafer theory allows the assignment of likelihood measures *m*, called "basic probability assignments" or "mass", to a proposition *P* and its negation $\neg P$ for which it might hold that $m(P) + m(\neg P) \leq 1$[3]. A *basic probability assignment* or *mass function* in the Dempster-Shafer theory is a function from the powerset of a set of mutually exclusive propositions θ called "frame of discernment" to the range [0…1] or, equivalently, a function *m* of the following form:

Axiom 1. $m: \wp(\theta) \rightarrow [0...1]$

3 In contrast, the axioms of the classic probability theory require that condition Prob(P) + Prob(¬P) = 1 for all valid probability functions *Prob* and propositions *P*.

A function m of this form provides a measure of belief in the truth of the disjunction of the propositions in different subsets of θ (i.e., elements of its powerset $\wp(\theta)$) which cannot be attributed directly (split) to any of these propositions individually. Formally, a function m of the above form is a basic probability assignment only if it also satisfies the following two axioms:

Axiom 2. $m(\varnothing) = 0$, *and*

Axiom 3. $\sum_{P \subseteq \theta} m(P) = 1$

The first of these axioms prevents basic probability assignments from assigning a non-zero basic probability measure to an empty proposition set. The second axiom requires that the sum of the basic probability measures which are assigned by a function m to different subsets of a frame of discernment θ must be equal to 1. The subsets P of θ for which $m(P) > 0$ are called "focals" of m and the union of these subsets is called "core" of m. Each basic probability assignment function m in the Dempster-Shafer theory induces a unique "belief" function *Bel*, which is defined as:

Axiom 4. $Bel: \wp(\theta) \rightarrow [0...1]$, *and*

Axiom 5. $Bel(A) = \sum_{B \subseteq A} m(B)$

A belief function *Bel* measures the total belief that is committed to the set of propositions P by accumulating the basic probability measures which are committed to the different subsets of P. Belief functions must also satisfy the following axioms:

Axiom 6. $Bel(\varnothing) = 0$

Axiom 7. $Bel(\theta) = 1$

Axiom 8. $\sum_{I \subseteq \{1,...,n\} \text{ and } I \neq \theta} (-1)^{|I|+1} Bel(\cap_{i \in I} P_i) \leq Bel(\cup_{i=1,...,n} P_i)$

where $n = |\wp(\theta)|$, $P \subseteq \theta$ *and* $i = 1,...,n$

In the Dempster-Shafer theory, two basic probability assignments m_1 and m_2 can be combined according to the rule of the "orthogonal sum":

Axiom 9. $m_1 \oplus m_2 (P) = (\sum_{X \cap Y = P} m_1(X) \times m_2(Y)) / (1 - k_0)$

where $k_0 = \sum_{V \cap W = \varnothing \text{ and } V \subseteq \theta \text{ and } W \subseteq \theta} m_1(V) \times m_2(W)$

In this formula, k_0 is a normalising parameter used to increase the belief assigned to the non-empty intersections of the focals of m_1 and m_2 in proportion to the belief that would be assigned to the empty intersections of these focals.

The rule of the "orthogonal sum" can be applied as long as:

$$\sum_{A \cap B \neq \emptyset, A \subseteq \theta \text{ and } B \subseteq \theta} m_1(A) m_2(B) < 1$$

This condition precludes the combination of conflicting basic probability assignments(i.e. assignments, one of which provides a degree of support of 1 to some proposition, while the other provides an equal degree to the negation of this proposition). The total belief about a proposition P, Bel(P) does not reflect the extent to which some- body fails to doubt P. This is given by a third measure called "upper probability" (or "plausibility" [146]), defined as:

Axiom 10. $$Pl(P) = 1 - Bel(P) = \sum_{B \subseteq \theta} m(B) - \sum_{A \subseteq P} m(A) = \sum_{B \cap P \neq \emptyset} m(B)$$

Since $\sum_{B \subseteq P} m(B) \leq \sum_{B \cap P \neq \emptyset} m(B)$, it also holds that $Pl(P) \geq Bel(P)$ .Hence, for each proposition P, there is a range [Bel(P), Pl(P)] which its belief falls within. Essentially, Pl(P) reflects the total belief which has not been assigned to the negation of P.

To clarify the above definitions consider the following example. In a medical diagnosis problem, there are four mutually exclusive hypotheses:

C(cold), F(flu), M(meningitis) and NP(no problem).

Thus, a frame θ discerning the potential dignosis of the medical problem is as follows:

θ ={C, F, M, NP).

Let assume that based on studies of relevant medical cases it is known that fever is a symptom of C and F in the 40% of the examined cases, while fever occurs in the 30% of M examined cases. Therefore, let assume that there is a basic probability assignment (BPA) m_1 specified as follows:

$$m_1(P) = \begin{cases} 0.4, \text{ if } P = \{C, F\} \\ 0.3, \text{ if } P = \{M\} \\ 0.3, \text{ otherwise or if } P = \theta \end{cases}$$

Assume also that nausea is a symptom of C, F and M in the 80% of the examined cases. Therefore, there is a BPA m_2 specified as follows:

$$m_2(P) = \begin{cases} 0.8, \text{ if } P = \{C, F, M\} \\ 0.2, \text{ otherwise or if } P = \theta \end{cases}$$

Let assume that a diagnosis for a patient experiencing fever and nausea is requested. To compute the combination of m_1 and m_2 BPAs, the orthogonal sum given in Axiom 9 can be used .It should be noted that the only intersection of given sets that yields to empty set is $\{M\} \cap \{C,F,NP\}$. Therefore, for k_0 it holds that:

$k_0 = 0.3*0.8 = 0.24$

For instance, for {C, F}, it holds that:

$$m_1 \oplus m_2 (\{C,F\}) = (\sum_{X \cap Y = \{C,F\}} m_1(X) \times m_2(Y)) / (1 - k_0)$$
$$= 0.32 + 0.08 / 1 - 0.24$$
$$= 0.526$$

The degree of belief that the combination of m_1 and m_2 assigns to the rest of the sets are shown in

Table 3-2. Also, in

Table 3-2, the total belief and plausibility of the considered sets are shown. By taking into account the belief and plausibility measures of

Table 3-2, the patient of our case possibly experiences C or F or NP with a possibility that ranges within [0.842, 0.921], while there is a weak belief for the M cause that lies within [0.079, 0.158].

Table 3-2 – Medical problem DS belief measurements

Sets / BPAs	{C,F}	{M}	{C, F, NP}	θ
m_1	0.4	0.3	0	0.3
m_2	0	0	0.8	0.2
$m_{1 \oplus} m_2$	0.526	0.079	0.316	0.079
Bel	0.526	0.079	0.842	
Pl	0.921	0.158	0.921	

Chapter 4: Extending EVEREST Monitoring Framework for Diagnosis

4.1 Overview

This chapter discusses how the EVEREST monitoring framework extended to support the basic formalization of the diagnostic task. In particular, Section 4.2 provides the basic formulation of the diagnostic task and the relevant assumptions that should be taken into account. Essentially, Section 4.2 provides the definitions of predicate sets necessary for the formalization of the diagnostic task in the context of EC-Assertion.

Having given the basic formal characteristics of the diagnostic task in Section 4.2, Section 4.3 provides the reader with the EC-Assertion specifications of the motivating example of the ATMS discussed in the introductory chapter. More specifically, the

example in Section 4.3 highlights the categorization of predicates that are used to specify assumptions necessary for the undertaking of the diagnostic task according to our approach.

4.2 Basic formulation of the diagnostic problem and assumptions

The generation of explanations of individual events in the diagnosis process is based on abductive reasoning. As defined in [122], the purpose of abductive reasoning is to find a set of *atomic formulas* Φ, which in conjunction with a theory *TH* entail a set of *observations* Ω. Formally, Φ is a set of atomic formulas that satisfy the following conditions:

- $TH \cup \Phi \vdash \Omega$, (Condition 1)
- $\forall$ f in Φ: predicates(f) $\subseteq$ APreds (Condition 2)

In the above conditions, *predicates(f)* denotes the predicates of formula *f* (i.e., a singleton set as Φ is assumed to be an atomic formula) and *APreds* is a set of abducible predicates. Given that the above conditions are satisfied, the set of formulae Φ can be seen as a possible cause of Ω or, in other words, as a possible *explanation* (or hypothesis) of why Ω has happened.

The basis of abductive reasoning is the derivation of the precondition *a* of a logical implication

$$a \Rightarrow b$$

when the consequence *b* of the implication is known to be true. Obviously this derivation is only a conjecture, as the meaning of "$a \Rightarrow b$" is that *b* is true when *a* is true but not vice versa or, in other words, that *a* is a sufficient condition for the occurrence of *b* but not a necessary condition. Thus, *b* may have been the consequence of a different cause and the derivation of *a* from *b* may be incorrect even if *a* is consistent with a broader logical theory (i.e., it satisfies *Condition 1* above). However, despite this widely discussed logical fallacy of abductive reasoning, the use of it as a *heuristic* form of searching for possible causes of effects is useful in the absence of any other alternatives and when the likelihood of the derivations that can be generated by it can be assessed against further

evidence. To this end, abductive reasoning has been used as one of the main approaches for providing *fault diagnosis* [41].

In fault diagnosis, abduction is used to derive the faults that appear to be the likely cause of the problem, given a theory that relates the faults with their effects and a set of effects that have been observed. This idea also underpins the use of abductive reasoning for diagnosis but the exact use of this form of reasoning has some key differences from other approaches. In the following, we will discuss in more detail these differences and the measures taken to assess the plausibility of the derivations obtained by abductive reasoning in the diagnosis process. Before doing this, however, it is necessary to establish the correspondences between the sets of formulas in the abstract formulation of abductive reasoning and the key artefacts in the monitoring process.

More specifically, the sets of formulas in *Condition 1* and *Condition 2* above have the following meanings:

- Ω is the set of the runtime (also referred as recorded or logged) events and fluents which are involved in the violation of a monitoring rule. More specifically, runtime events are atomic formulas that contain fully instantiated *Happens* predicates. In the same manner, runtime fluents are atomic formulas that contain fully instantiated *HoldsAt* predicates. In EVEREST context, a predicate is considered as fully instantiated if all of the terms of the predicate are ground. For instance, considering the ATMS scenario, we have specified the following *Happens* predicate:

```
Happens(e(_id2,_r2,_receiver1,RES-A,signal(_r2,_a,_s),
        _source2),t2,R(t1, t1+5))
```

  A fully instantiated predicate of the above type is as follows:

```
Happens(e(E14,Radar112,HLTAirBase,RES-A,
        signal(Radar112,BA3768,HLT_East),
        HLTAirBaseSource), 13,R(12,17))
```

  while a partially instantiated one is the following:

```
Happens(e(_id,_r,_receiver,RES-A,signal(_r,BA3768,
        _s),_source),t2,R(12,17))
```

- *TH* is the set of the *assumptions* specified for the system that is being monitored and the events that have been recorded in the log of the monitoring framework at the time *t* when the generation of an abductive explanation is required excluding the runtime events, which belong to *Ω*. Since *TH* might have different elements depending on the time point when the generation of an abductive explanation is required in the rest of this report we will refer to it as *TH(t)* to signify the time *t* of enquiring for an explanation explicitly.
- *APreds* is a predefined set of the predicates that can *only* appear in the leaves of the different abductive trees, which can potentially be generated by the given assumptions of theory *TH*. Esentially, the members of *APreds* (called *abducibles* henceforth) can appear only as body predicates of the assumptions of the underlying theory *TH*.

We should also note some further assumptions that we make about the formulas (i.e., *assumptions* and *monitoring rules*) and the events used in the monitoring framework. More specifically, if the set of the assumptions for a system that is being monitored is denoted by *AS* and the set of all the events of this system that have been recorded in the log of the monitoring framework at some time point *t* is denoted by *RE(t)*, we assume that:

- $TH(t) = AS \cup (RE(t) - \Omega)$ (Condition 3)

We also assume that the predicates used in the assumptions and monitoring rules belong to one of the following three sets [162, 163, 164]:

- The set of observable predicates *OPreds*. A predicate is observable if it can be unified with an event, which is generated during the operation of the system being monitored, is captured by the captors of the EVEREST framework and is finally recorded in the event log of the EVEREST framework. The truth value of the observable predicates can be established by the successful unification process of the predicates themselves with runtime events or by deduction on the assumptions of the system being monitored based on recorded events of the event log of the EVEREST framework and other previously derived predicates.
- The set of derived predicates *DPreds*. A predicate is derived if it can be grounded only by applying unification and deductive reasoning on the

assumptions of the system being monitored based on recorded events of the event log of the EVEREST framework and other previously derived predicates. Regarding the evaluation of the truth value of the derived predicates, the truth value of this kind of predicates can be established by deductive reasoning.

- The set of abducible predicates *APreds* has been defined above. The truth value of abducibles can be established by abductive reasoning on the assumptions of the system being monitored based on recorded events of the event log of the EVEREST framework and other previously derived predicates.

Finally, we assume that:

- The standard Event Calculus predicate *Initially*, whose formal specification contain fluent, is considered as observable predicate or formally:

  `Initially` $\in$ OPreds (Condition 4)

- The remaining standard Event Calculus predicates *Initiates*, *Terminates* and *HoldsAt*, whose formal specifications contain fluents, are always derived predicates. Assuming that *FluentContainersPreds* = {`Initiates`, `Terminates`, `HoldsAt`}, we formally have:

  FluentContainersPreds $\subseteq$ DPreds (Condition 5)

- The set *DPreds* have no elements is common with *APreds*, while OPreds and may have common elements with DPreds and APreds or formally:

  $DPreds \cap APreds = \varnothing$ (Condition 6)

  $(DPreds \cap OPreds \neq \varnothing) \vee (DPreds \cap OPreds = \varnothing)$ (Condition 7)

  $(APreds \cap OPreds \neq \varnothing) \vee (APreds \cap OPreds = \varnothing)$ (Condition 8)

- The assumptions, which are used for generating abductive explanations, are formulated as Horn clauses [7].

- The set of assumptions, which is used for generating abductive explanations, is hierarchical. This means that the dependency graph of the theory, i.e. the graph connecting two assumptions A1 and A2 with an arc from A1 to A2 if a

predicate in the head of A1 appears also in the body of A2, is acyclic (see [33]).

4.3 EC Specifications of the Air Traffic Management System (ATMS) Motivating Example

As an example of cases where monitoring information needs to be enhanced by diagnostic explanations, consider an air traffic management system, referred to as "ATMS" in the following. ATMS uses different radars to monitor the trajectories of airplanes in different air spaces. It is also connected with a system that keeps a record of flight plans which are submitted by different planes ahead of flights to indicate the expected route of a flight and request flight permission.

The operations of ATMS may be monitored at runtime to ensure the integrity of its components and the information generated by them. Monitoring, for example, may focus on properties related to: (i) the liveness of the radars connected to ATMS, and (ii) the generation of mutually consistent information by them. An example of a property of this kind relates to cases where air spaces are covered by different radars or have overlapping areas covered by different radars. In such cases, to check the integrity of the information that is provided by the different radars which cover an airspace, we can monitor a rule requiring that if one of these radars sends a signal indicating that an airplane is in the airspace, every other radar that covers the same space should also send a signal indicating the presence of the plane in it within a certain period of time after the receipt of the initial signal. Such a rule can be specified in the monitoring language of EVEREST framework as follows:

ATMS.R1. ∀t1∈Time, ∃t2∈Time, ∀_r1∈Radars, ∀_receiver1,
∀_a∈Airplanes, ∀_s∈Airspaces, ∀_r2∈Radars, ∀_source1.
Happens(e(_id1,_r1,_receiver1,RES-A,signal(_r1, _a, _s),
_source1),t1,R(t1,t1)) ∧
HoldsAt(covers(_r1,_s),t1) ∧
HoldsAt(covers(_r2,_s), t1) ∧
_r2 ≠ _r1 ⇒
Happens(e(_id2,_r2,_receiver1,RES-A,signal(_r2, _a, _s),
_source1),t2,R(t1, t1+5))

Rule ATMS.R1 is violated in all cases where the monitor receives a *signal* event by one of the radars of ATMS that covers a specific airspace but not the other. Clearly, whilst in such cases, knowing that the rule has been violated is important for the operation of ATMS. However, the violation report on its own is not sufficient for establishing the reasons why the second expected signal was not received and taking appropriate action (if possible). In fact, the violation may have been due to several reasons, including the following:

- The radar that did not send the expected signal was malfunctioning (*Cause 1*).
- The communication link between the radar that did not send the expected signal and the monitor was malfunctioning or an intruder captured the signal and prevented it from reaching the monitor (*Cause 2*).
- The radar that sent the expected signal was malfunctioning or its identity was faked by an intruder that sent a fake signal to the monitor (*Cause 3*).

Identifying which of the above reasons has caused the violation is important for taking actions that would restore the integrity of the operation of ATMS.

The assumptions of ATMS are as follows:

ATMS.A1. **Initially**(covers(R1,S1),t0)

ATMS.A2. **Initially**(covers(R2,S1),t0)

The first two assumptions ATMS.A1 and ATMS.A2 state that radars R1 and R2 cover airspace S1 since the start of the execution of ATMS.

ATMS.A3. ∀t1∈Time, ∃t2∈Time, ∀_sender1, ∀_receiver2, ∀_source2, ∀_a∈Airplanes, ∀_s∈Airspaces, ∃_r∈Radars.
Happens(e(_id1,_sender1,_receiver2,RES-A,inspace(_a,_s),
_source2),t1,R(t1,t1)) ∧
HoldsAt(covers(_r,_s),t1) ⇒
Happens(e(_id2,_r,_receiver2,RES-A,signal(_r,_a,_s),
_source2),t2,R(t1,t1+5))

Assumption ATMS.A3 states that if there is an airplane _a moving in airspace _s at some time point t1 and it holds that radar _r covers _s at t1, then it is expected that there is a signal from _r notifying that _a moves in _s at some time point t2 within t1 and 5 time units after t1. Please note that the predicate Happens(e(_id1,_sender1, _receiver2, RES-A, inspace(_a,_s), _captor2), t1, R(t1,t1)) is an abducible

predicate, while the predicate `Happens(e(_id2, _r, _receiver2, RES-A, signal(_r,_a,_s), _captor2), t2, R(t1,t1+5))` is an observable predicate.

ATMS.A4. `∀t1∈Time, ∃t2∈Time, ∀_sender1, ∀_receiver2, ∀_source2, ∀_a∈Airplanes, ∀_s∈Airspaces.`

Happens`(e(_id1,_sender1,_receiver2,RES-A,inspace(_a,_s), _source2),t1,R(t1,t1)) ⇒`

Happens`(e(_id2,_a,_receiver2,RES-A,permissionRequest(_a, _s),_source2),t2,R(t1-20,t1-1))`

Assumption ATMS.A4 states that if there is an airplane _a moving in airspace _s at some time point t1, then it was expected that _a has requested permission for entering _s at some time point t2 within 20 and 1 time units before t1.

ATMS.A5. `∀t1∈Time, ∀_sender, ∀_receiver, ∀_source, ∀_a∈Airplanes, ∀_s∈Airspaces.`

Happens`(e(_id1,_sender,_receiver,RES-A,inspace(_a,_s), _source),t1,R(t1,t1)) ⇒`

Initiates`(e(_id1,_sender,_receiver,RES-A,inspace(_a, _s), _source), inairspace(_a,_s),t1)`

Assumption ATMS.A5 states that if there is an airplane _a moving in airspace _s at some time point t1, then the fluent *inairspace* is initiated at t1.

ATMS.A6. `∀t1∈Time, ∃t2∈Time, ∀_receiver, ∀_source, ∀_a∈Airplanes, ∀_s∈Airspaces, ∃_airportX∈Airports.`

Initiates`(e(_id1,_sender,_receiver,RES-A,inspace(_a, _s),_source),inairspace(_a,_s),t1) ∧`

HoldsAt`(landing_airspace_for(_s,_airportX),t1) ⇒`

Happens`(e(_id2,_a,_receiver,RES-A, landingRequest(_a,_airportX),_source),t2, R(t1-10,t1))`

Assumption ATMS.A6 states that the fluent *inairspace* is initiated at t1 by an airplane _a moving in airspace _s at t1 and it holds that the landing airspace for_airportX is _s at t1, then it was expected that _a has requested landing permission from the control base of _airportX at some time point t2 within 10 time units before t1 and t1.

ATMS.A7. `∀t1∈Time, ∀_sender, ∀_receiver, ∀_source, ∀_a∈Airplanes, ∀_airportX∈Airports, ∃_s∈Airspaces.`

Happens`(e(_id1,_sender,_receiver,RES-A,`

changeOfLandingApproach(_airportX,_s),

_source),t1,R(t1,t1)) ⇒

Initiates(e(_id1,_sender,_receiver,RES-A,
changeOfLandingApproach(_airportX,_s),_source),
landing_airspace_for(_s,_airportX),t1)

Assumption ATMS.A7 states that if there is an event that changes the landing approach of _airportX to _s at t1, then the fluent, which specifies that airspace _s is the landing airspace of _airportX is initiated at t1.

ATMS.A8. ∀t1∈Time, ∀_sender, ∀_receiver, ∀_source, ∀_a∈Airplanes, ∀_airportX∈Airports, ∃_s∈Airspaces.

Happens(e(_id1,_sender,_receiver,RES-A,

removeLandingApproach(_airportX,_s),

_source),t1,R(t1,t1)) ⇒

Terminates(e(_id1,_sender,_receiver,RES-A,
removeLandingApproach(_airportX,_s),_source),
landing_airspace_for(_s,_airportX),t1)

Assumption ATMS.A8 states that if there is an event that reconfigures the ATMS by setting airspace _s as a no longer valid landing approach of _airportX at t1, then the fluent, which specifies that airspace _s is the landing airspace of _airportX, is terminated at t1.

In terms of the predicate sets APreds, DPreds and OPreds, which are defined in Section 4.1, the membership of the predicates of the ATMS theory (i.e. the set of ATMS rule and assumptions) is as follows:

APreds = {**Happens**(e(_id,_r,_receiver,RES-A,inspace(_a,_s),

_source2),t,R(t,t))

}

Dpreds = { **HoldsAt**(covers(_r,_s),t),

Initiates(e(_id,_sender,_receiver,RES-A,inspace(_a,

_s), _source), inairspace(_a,_s),t),

HoldsAt(landing_airspace_for(_s,_airportX),t),

Initiates(e(_id,_sender,_receiver,RES-A,

```
                changeOfLandingApproach(_airportX,_s),_source),
                landing_airspace_for(_s,_airportX),t)
        }
OPreds = { Initially(covers(R1,S1),t0), Initially(covers(R2,S1),t0),
        Happens(e(_id,_r,_receiver,RES-A,signal(_r, _a, _s),
                _source),t,R(t,t)),
        Happens(e(id,_a,_receiver,RES-A,permissionRequest(_a,
                _s),_source),t,R(t,t)),
        Happens(e(_id,_a,_receiver,RES-A,
                landingRequest(_a,_airportX),_source),t,R(t,t)),
        Happens(e(_id,_sender,_receiver,RES-A,
                changeOfLandingApproach(_airportX,_s),
                _source),t,R(t,t))
    }
```

Chapter 5: The Diagnostic Approach

5.1 Overview

The aim of this chapter is to provide the reader with a detailed description of our diagnostic approach. As a roadmap for the content of this chapter, we initially provide a high level overview of the diagnostic process.

The overall process of diagnosing the causes of S&D monitoring rule violations has four main stages as discussed in [162, 163, 164]. As shown in Figure 5-1, these stages are:

1. Explanation generation
2. Explanation effect identification
3. Explanation plausibility assessment
4. Diagnosis generation

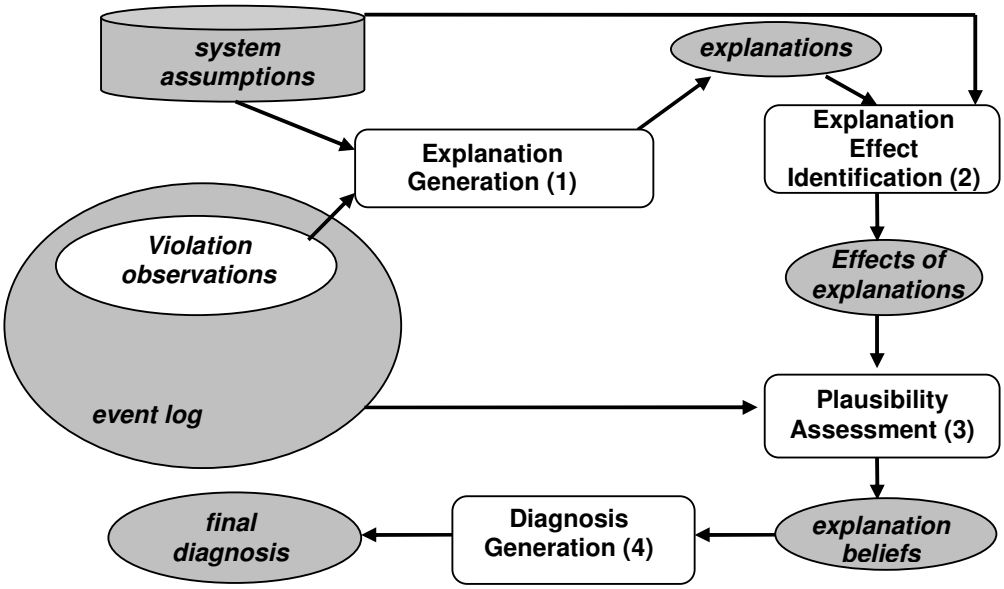

Figure 5-1 – The overall process of the diagnostic approach

In the first of these stages (i.e., explanation generation), the diagnosis process generates all the *possible explanations* for the individual events which have caused an S&D monitoring rule violation. These events are referred to as "violation observations" in the following. The possible explanations of violation related observations are generated from *assumptions* that have been given to the monitor regarding the operation of the system that is being monitored using abductive reasoning. The explanations generation step is discussed in details in Section 5.2.

After generating explanations for the individual violation observations, the diagnosis process enters its second stage, namely the stage of *explanation effect identification*, which is discussed in Section 5.3. This stage is concerned with the identification of all the possible consequences of the explanations of the individual violation observations if these explanations were valid. Whilst the generation of individual explanations from the observation violations are generated by abductive reasoning, the effects of individual explanations are derived by deduction using the assumptions specified in S&D patterns.

Following the identification of the effects of individual explanations, the diagnosis process enters its third stage. At this stage, the process assesses the likelihood of the

validity of the individual event explanations. To do so, the expected effects of the individual explanations are checked against the event log of the EVEREST monitoring framework to find if there are events in the log that match the expected effects. Every match that is found between an expected effect and an event in the log casts confirming evidence to the explanation associated with the effect. On the other hand, the absence of a matching event for an effect casts disfavouring evidence to the explanation. Based on the confirming and disconfirming elements of evidence which are identified during this stage, the diagnosis process estimates a belief and a plausibility measure for each individual explanation. The diagnosis process third step details are worked out in Section 5.4.

Finally, at the fourth stage of the diagnosis process, namely the stage of diagnosis generation which is discussed in Section 5.5, the diagnosis framework constructes alternative aggregated explanations for the S&D rule violation from the explanations of the individual violation observations and computes beliefs in the validity of these aggregate explanations. Using these beliefs the framework also identifies the most plausible aggregate explanation for the violation.

In the rest of this chapter, we discuss each of the above stages in detail presenting the reasoning mechanisms which are deployed in them and giving examples of applying these mechanisms.

5.2 Generation of Explanations

5.2.1 The process of generating explanations

Abductive reasoning is used only in the first stage of the diagnosis process, as a mechanism of trying to find possible causes of the runtime events that have caused a violation of an S&D monitoring rule. Establishing the possible causes of the events which are involved in a rule violation has a dual role in the diagnosis process: first it provides a possible explanation of the individual events and fluents which have caused the violation, and second it provides confirmatory evidence that these events have indeed taken place and have not been the result of some attack or malfunctioning in the monitoring framework and/or the system(s) being monitored by it. The latter role of individual event explanations is very significant as the possibility of attacks in the monitoring framework and the systems, which are being monitored by it, cannot be precluded. In the scenario

that we introduced in Section 0, for instance, received radar signals may be the result of a radar malfunctioning or an attack by an intruder who has faked the identity of a radar R and sends signals which appear to have been sent by R.

The explanations of the events, which are involved in the violation of an S&D monitoring rule, are generated by backward chaining. More specifically, when the diagnostic framework is given a specific event E to find an explanation for, it searches through the assumptions, which are known about the system that is being monitored to see if they have a predicate P in their head that can be unified with E. This check is performed in two steps. For every assumption A that has such a predicate, the framework checks if the unification between P and E covers all the non time variables of A (i.e., it provides bindings for all these variables) and, if it does, it tries to generate an explanation for E using A. More specifically, the framework checks if the time constraints which are imposed by the event E on the instantiated predicates (conditions) in the body of A, can lead to concrete and feasible time ranges for these predicates. To do this, the framework retrieves the constraints that relate the time variable of the predicate P in the head of A that was matched with E and the time variables of each of the predicates in the body of A, replaces the time variable t_p of P with the time stamp of the event E that needs to be explained and calculates the maximum and minimum possible values for the time variables of the predicates in the body of A.

The calculation of the minimum and maximum values of the time variables of the predicates in the body of A is treated as a linear programming problem. This is possible due to the way in which constraints for time variables are specified in the monitoring rule (note that the same language is used to specify both monitoring rules and assumptions. As we discussed in Section 3.3, according to this language, each Event Calculus formula that specifies a monitoring rule or assumption must define an upper and a lower boundary for the time variables of all the predicates in the formula.

Thus, the upper and lower boundaries *ub* and *lb* of a time variable *t* of a predicate in a formula become effectively linear expressions of the form:

$$lb = l_0 + l_1 t_1 + l_2 t_2 + \ldots + l_n t_n$$

$$ub = u_0 + u_1 t_1 + u_2 t_2 + \ldots + u_n t_n$$

where t_i (i=1, ..., n) are other time variables in the formula and the constraints related to t are of the form:

$$l_0 + l_1 t_1 + l_2 t_2 + \ldots + l_n t_n \leq T_E \qquad \text{(C1)}$$

$$T_E \leq u_0 + u_1 t_1 + u_2 t_2 + \ldots + u_n t_n \qquad \text{(C2)}$$

Then from the above formulas, it might be possible to compute the minimum and maximum possible values of any variable t_i (i=1, ..., n) in them by solving the linear optimization problems $max(1t_i + \sum_{j=1,\ldots,n,\, j\neq i} 0*t_j)$ and $min(1t_i + \sum_{j=1,\ldots,n,\, j\neq i} 0*t_j)$ subject to the constraints *C1* and *C2*. An evident candidate method for solving these problems is George Dantzig's classic *Simplex* method, which is revisited in [63]. By solving these problems for each of the time variables of the predicates in the body of an assumption A, it can be established if a concrete and feasible time range exists for these variables.

In cases that such ranges exist, the explanation generation procedure applies the most general unifier of E and P to the predicates in the body of A and checks if the instantiated predicates which result in the body of A are abducible predicates or can be matched with events already recorded in the event log of the monitor. In both sub-cases, the instantiated abducible predicates in the body of A are added to the ongoing explanation. In sub-cases where an instantiated predicate in the body of A is neither an abducible predicate nor does it correspond to a recorded event, backward chaining is applied again on it to try to find other assumptions which have predicates in their head that can be unified with it. Such predicates are retrieved and the process is repeated for the predicates in the bodies of the relevant assumptions. In the case of an assumption that has been retrieved for constructing an explanation for an event E but has some predicate P' in its body that does not correspond to abducible predicate or a recorded event and, furthermore, cannot be explained through other assumptions, the assumption is abandoned and no explanation is constructed from it E.

```
Explain(e, t_min(e), t_max(e), f_init)
1.  // let Φ_e be a list keeping the disjunction of possible explanations of atomic predicate e
2.  Φ_e = [ ]_OR
3.  // e is an abducible atom; add e to the current explanation
4.  If e ∈ ABD Then
5.     Φ_e = append(Φ_e , [(e, t_min(e), t_max(e))])
6.     //let AbductivePath_e[ ] be the list keeping the identifiers of f that were visited for abducing e
7.     append(AbductivePath_e[ ], f_init)
8.  // e is not an abducible atom; find explanations for it
9.  Else
10.    // try all alternative explanations by reasoning on all f that belong to the assumptions set AS
11.    For all f ∈ AS  Do
12.       // let function mgu return the most general unifier of e and a predicate p if this unifier exists
13.       u = mgu(head(f), e)
14.       If u ≠ ∅ and u covers all non time variables in body(f) Then
15.          // let CND_f be a list keeping the body predicates of formula f (conditions of f henceforth)
16.          Copy body(f) into CND_f
17.          FormulaFailed = False
18.          // let Φ_f be a list keeping a conjunction of elements explaining the conditions of f
19.          Φ_f = []_AND
20.          While FormulaFailed = False and CND_f ≠ ∅ DO
21.             Remove some condition C from CND_f
22.             Compute the min and max possible values t_min(C), t_max(C) of C based on t_min(e) and t_max(e)
23.             // t_min(C), t_max(C) are not undeterminable
24.             If t_min(C) ≠ NULL and t_max(C) ≠ NULL Then
25.                C_u = ApplyUnification(u, C)
26.                // C is an abducible atom; add it to current explanation
27.                If C ∈ ABD Then
28.                   Φ_f = append (Φ_f , [(C_u, t_min(C), t_max(C))]_ABD )
29.                   append(AbductivePath_Cu[ ], f)
30.                // C is not an abducible atom
31.                Else
32.                   find a derived or recorded predicate e_c such that: e_c can be unified with C_u and
33.                                                   t_min(e_c) ≥ t_min(C) and t(e_c) ≤ t_max(C)
34.                   // no recorded or derived predicate matching C has been found
35.                   If e_c = NULL Then
36.                      Φ_C = Explain(C, t_min(C), t_max(C), f)
37.                      If Φ_C is empty Then
38.                         FormulaFailed = True
39.                      Else
40.                         Φ_f = append(Φ_f , Φ_C)
41.                      End If
42.                   End If
43.                End If
44.             End If
45.          End While
46.          If FormulaFailed = False Then Φ_e = append (Φ_e,Φ_f) End if
47.       End if
48.    End For
49. End If
50. return(Φ_e)
END Explain
```

Figure 5-2 - Algorithm for generating explanations of atomic predicates

The algorithm for generating explanations for an atomic predicate E is called *Explain* and is listed in Figure 5-2. This algorithm generates a list representing the alternative explanations for a particular atomic predicate. In general there might be zero or more alternative explanations for an atomic predicate and each of these explanations consist of abduced atomic formulas.

The algorithm starts by getting as input an atomic predicate *e* for which an explanation is required and the boundaries of the time range of this predicate $t_{min}(e)$ and $t_{max}(e)$. It also has a fourth input parameter, called f_{init}, which represents the initial formula that is to be used for generating explanations. This parameter is not used in the initial invocation of *Explain* since the objective of the process is to find all the possible alternative explanations of the input predicate. In subsequent recursive invocations of the algorithm, however, it is used to indicate the identifier of the last formula that was used in the generation of an ongoing explanation since along with explanations the algorithm records the backward chaining path through which the abduced atomic predicates of each explanation were generated (see lines 7 and 29 in Figure 5-2). Given an input predicate *e* that is to be explained, if the predicate symbol of *e* is an abducible predicate, a pair of the predicate e and its time range (i.e., (e, t_{min}(e), t_{max}(e))) is added to the current list of explanations (i.e., list Φ_e) and the algorithm terminates by returning this list of explanations (see lines 4, 5 and 50 in Figure 5-2). If, however, *e* is not an abducible predicate, *Explain* checks through the known assumptions of the system that is being monitored (i.e., the elements of the set AS) to find those whose head could be unified with *e*. The unification test is performed by calling the function *mgu* that returns the most general unifier of two formulas [94]. In general, the general unifier can be considered as a list containing value bindings for all the variables of the two input formulas. However, the *mgu* function adaptation in EVEREST returns value bindings for all the non-time variables of the input formulas. Thus, if a non empty unifier is found between *e* and a predicate in the head of an assumption *f*, the algorithm checks if the unification covers all the predicates in the body of the assumption and, if it does, it creates a condition list (called CND_f) with the predicates of the body of the assumption and tries to explain each of these predicates (see lines 14-16 in Figure 5-2). More specifically, for each of these predicates, the algorithm computes initially the minimum and maximum possible values (t_{min}(C), t_{max}(C)). This computation is based on the *Simplex* algorithm [63] using as

constraints the constraints defined by all the particular assumption *f* between the time variable of the current body predicate and the time variable of the predicate *P* in the head of *f* that was unified with *e* and two more constraints to ensure that the time variable of *P* is within the time range $t_{min}(e)$ and $t_{max}(e)$ of *e* (see line 22 in Figure 5-2). If this optimisation problem has a solution and the boundaries for the time variable of the predicate in the body of *f* can be identified, the *Explain* algorithm applies the detected unification between *f* and *e* to the current predicate in the body of *f* in order to instantiate it (see line 25 in Figure 5-2), and checks if the instantiated predicate is an abducible predicate (see line 27 in Figure 5-2). If the current instantiated predicate in the body of *f* is an abducible predicate, it is added to a temporary explanation list for the assumption (Φ_f) along with its time range, while the list that keeps the abductive path of the predicate ($AbductivePath_{Cu}[\ ]$) is updated with the identifier *f* of the assumption from which it has been abduced (see lines 28 and 29 in Figure 5-2). Please note that the id of the event of each atomic formula, which is added to the explanation list, is set to *ABD*. Otherwise, if the current predicate is not an abducible predicate, the algorithm checks if there is a runtime event matching it and if it cannot find such an event it tries to generate an explanation of the predicate by abduction by calling itself recursively (see lines 31−36 in Figure 5-2). If this recursive call succeeds in generating an explanation of the current predicate by abduction, this explanation is added to the current explanation list and the algorithm proceeds by investigating the next condition of *f*.

The iteration over the conditions of *f* continues until either all the predicates in the body of *f* have been successfully explained or correspond to runtime events or until the algorithm encounters a predicate that cannot be explained. In this case, it terminates unsuccessfully for *f*.

The *Explain* algorithm is called for all the atomic predicates, which are involved in the violation of an S&D monitoring rule in order to generate all the possible explanations that can be found for these predicates. If an atomic predicate appears in a negated form in an S&D violation and the invocation of the algorithm *Explain* does not produce any explanations for the negated form of the predicate, the algorithm is invoked to produce explanations for the non-negated form of the predicate.

In the following section, we give an example of using the *Explain* algorithm to generate possible explanations for atomic predicates.

5.2.2 Examples of explanation generation

As an example of how the algorithm *Explains* works consider the generation of explanations for runtime events that cause a violation of Rule ATMS.R1, which was introduced in Sections 1.2 and 4.3 and is specified as follows:

ATMS.R1 ∀t1∈Time, ∃t2∈Time, ∀_r1∈Radars, ∀_receiver1,
∀_a∈Airplanes, ∀_s∈Airspaces, ∀_r2∈Radars, ∀_source1.
Happens(e(_id1,_r1,_receiver1,RES-A,signal(_r1, _a, _s),
_soure1),t1,R(t1,t1)) ∧
HoldsAt(covers(_r1,_s),t1) ∧
HoldsAt(covers(_r2,_s), t1) ∧
_r2 ≠ _r1 ⇒
Happens(e(_id2,_r2,_receiver1,RES-A,signal(_r2, _a, _s),
_source1),t2,R(t1, t1+5))

Assuming that the monitor has received the events shown in the log of Figure 5-3, Rule ATMS.R1 is violated by:

- the event (E4) (i.e., `Happens(e(E4,R1,AirBase,RES-A,signal(R1,A1,S1),AirBaseCaptor),7,R(7,7)))` in the event log of Figure 5-3
- the atomic formula `Happens(e(NF,R2,AirBase,signal(R2,A1,S1),AirBaseCaptor),t,R(7,12))`, which signifies the absence of a signal from radar R2 within the time period expected by Rule ATMS.R1 given the signal of radar R1, and
- the atomic formulas `HoldsAt(covers(R1,S1),7)` and `HoldsAt(covers(R2,S1),7)`

Event Log for ATMS:

```
(E1) Happens(e(E1,AirBase,AirBase,RES-A,changeOfLandingApproach(AR-
            a,S2),AirBaseCaptor),0,R(0,0))
(E2) Happens(e(E2,R2,AirBase,RES-A,signal(R2,A2,S2),AirBaseCaptor),1,
            R(1,1))
(E3) Happens(e(E3,AirBase,AirBase,RES-A,changeOfLandingApproach(AR-
            a,S1),AirBaseCaptor),2,R(2,2))
(E4) Happens(e(E4,R1,AirBase,RES-A,signal(R1,A1,S1),AirBaseCaptor),7,
            R(7,7))
(E5) Happens(e(E5,R2,AirBase,RES-A,signal(R2,A5,S1),AirBaseControl),13,
            R(13,13))
```

Figure 5-3 – Event log for ATMS

The truth value of the atomic formula *Happens(e(NF,R2,AirBase,signal(R2,A1,S1), AirBaseCaptor),t,R(7,12))* has been evaluated to *False* by virtue of the principle of *negation as failure* due to the fact that the monitor has received the events (E2) and (E5) from radar R2 at the time points T=1 and T=13 but no other event from the same radar between these two points. Also, the atomic formulas *HoldsAt(covers(R1,S1), 7)* and *HoldsAt(covers(R2,S1), 7)* are deduced by the monitor from:

- the standard EVEREST assumption SA2 that is specified as follows:

  Initially(_f,t0) ∧

  ¬∃_e1,t1. **Terminates**(_e1,_f,t1) ∧

  t1 ≥ t0 ⇒

  HoldsAt(_f,t1)

- the *Initially* ground predicates of assumptions ATMS.A1 and ATMS.A2, which signify that radars R1 and R2 cover the airspace S1 since the start of the execution of the ATMS, and

- the absence of any event that signifies the repositioning of any of the two radars until the time point T=7, when the monitor receives the signal for the presence of aircraft A1 in S1 from R1, and could essentially terminate the ground fluents `covers(R1,S1)` and `covers(R2,S1)` before time point T=7

Before describing the explanation generation process based on the given theory and event log, it would not be harmful to the understanding of the process itself to interpret the information, which is provided by the given theory and log event, about the ATMS. Firstly, as we have discussed above, the assumptions ATMS.A1 and ATMS.A2 along with the absence of any event that signifies the repositioning of any of the two radars until the time point T=7 signify that the airspace S1 is under the surveillance of the ATMS, and more specifically is covered by the radars R1 and R2.

By interpreting event (E1), it is signified that, besides airspace S1, airspace S2 and airport AR-a is under the surveillance of the ATMS. Additionally, given the assumption ATMS.A7 and the standard EVEREST assumption SA2, it holds that S2 is the airspace used for landing approach to AR-a at time point T=0 onwards, until the time point when an event, which can terminate this condition according to assumption ATMS.A8, occurs.

Similarly, event (E3) informs that airspace S1 can also be used for landing approach to AR-a at time point T=2 onwards, until again an event that can terminate this condition according to assumptions ATMS.A8 occurs. Moreover, event (E1) and (E3) introduce the system components *AirBase* and *AirBaseCaptor*, as the sender and receiver, and captor arguments of their signature respectively.

Events (E2), (E4) and (E5) signify the actual runtime information about the moving objects within the sections of airspace that is under surveillance of the ATMS. More specifically, by interpreting (E2), it is signified that there was a signal, which was generated by radar R2 and informs that airplane A2 was in airspace S2 at time point T=1. Also, from (E4), we understand that there was a signal, which was generated by radar R1 and informs that the airplane A1 was in airspace S1 at time point T=7. Finally, event (E5) signifies that there was a signal, which was generated by radar R2 and informs that the airplane A5 was in airspace S1 at time point T=13. It is notable that the three events do not introduce any new knowledge regarding the ATMS components.

The explanation generation process starts by trying to generate explanations for the formulae that signify the existence and absence of events involved in the violation of Rule ATMS.R1, namely *Happens(e(E4,R1,AirBase,RES-A,signal(R1,A1,S1),AirBaseCaptor),7,R(7,7))* and *Happens(e(NF,R2,AirBase,signal(R2,A1,S1),AirBaseCaptor),t,R(7,12))*. As we have discussed in Section 3.3.2 and 4.3, the following assumptions are also part of the formulas given to the diagnosis tool:

SA1 **Initiates**(_e1,_f,t1,R(t1,t1)) ∧
¬∃_e2,t2. **Terminates**(_e2,_f,t2) ∧
t2 ≥ t1 ⇒
HoldsAt(_f,t2)

SA2 **Initially**(_f,t0) ∧
¬∃_e1,t1. **Terminates**(_e1,_f,t1) ∧
t1 ≥ t0 ⇒
HoldsAt(_f,t1)

ATMS.A1 **Initially**(covers(R1,S1),t0)

ATMS.A2 **Initially**(covers(R2,S1),t0)

ATMS.A3 ∀t1∈Time, ∃t2∈Time, ∀_sender1, ∀_receiver2, ∀_source2,
∀_a∈Airplanes, ∀_s∈Airspaces, ∃_r∈Radars.
Happens(e(_id1,_sender1,_receiver2,RES-A,inspace(_a,_s),
_source2),t1,R(t1,t1)) ∧
HoldsAt(covers(_r,_s),t1) ⇒
Happens(e(_id2,_r,_receiver2,RES-A,signal(_r,_a,_s),
_source2),t2,R(t1,t1+5))

Given the assumptions above, when given the atomic formula *Happens(e(E4,R1,AirBase,RES-A,signal(R1,A1,S1),AirBaseCaptor),7,R(7,7))* to explain, the *Explain* algorithm detects that the formula can be unified with the predicate `Happens(e(_id2, _r, _receiver2, RES-A, signal(_r,_a,_s), _captor2), t2, R(t1,t1+5))` in the head of assumption ATMS.A3, and the most general unifier of the two formulae (i.e., {`_id2/E4, _r/R1, _receiver2/AirBase, _a/A1, _s/S1, _captor2/AirBaseCaptor`}) covers all the non time variables which appear in the body of this assumption. Furthermore, the linear constraint system that is generated from the definition of the boundaries of the time variable t2 in ATMS.A3 after substituting the timestamp T=7 of the event `Happens(e(E4,R1,AirBase,RES-A,signal(R1,A1,S1),AirBaseCaptor),7,R(7,7))` for this variable consists of the constraints $t_1 \leq 7$ and $7 \leq t_1 + 5$. From these two constraints, it is easy to see that the time range for the time variable t1 is [2, 7]. Thus, the conditions of *Explain* are satisfied and the algorithm generates the atomic formula `Happens(e(ABD,R1,AirBase,RES-A,inspace(A1,S1),AirBaseCaptor),t1,R(2,7))` as a possible explanation of `Happens(e(E4, R1, AirBase, RES-A, signal(R1,A1,S1), AirBaseCaptor), 7,R(7,7))`. Due to the fact that the predicate `Happens(e(_id,_r,_receiver,RES-A,inspace(_a,_s),_source2),t,R(t,t))` belongs to the set of the abducible predicates *APreds* (see Section 4.2) this intermediary explanation needs no further elaboration and can be used as an abduced explanation. Note, however, that as the explanation `Happens(e(ABD,R1,AirBase,RES-A,inspace(A1,S1),AirBaseCaptor),t1,R(2,7))` has been generated by the assumption ATMS.A3, the candidate explanation will be maintained only if the other instantiated predicate of the body of ATMS.A3, namely `HoldsAt(covers(R1,S1),t1)`, holds when t1 takes values in the range R(2,7). The validity of this predicate, however, can be deduced from the standard EVEREST assumption SA2, the assumption ATMS.A3 and the absence of an event that could reposition R1 and therefore needs no further exploration by backward chaining. Hence,

the *Explains* algorithm will return the atomic formula `Happens(e(ABD, R1,AirBase,RES-A,inspace(A1,S1),AirBaseCaptor),t1,R(2,7))` as an explanation of the event `Happens(e(E4, R1, AirBase, RES-A, signal(R1,A1,S1), AirBaseCaptor), 7,R(7,7))`. Figure 5-4 shows graphically the reasoning path through which the explanation of this event is generated and the list of explanations returned by the algorithm *Explain* in this case.

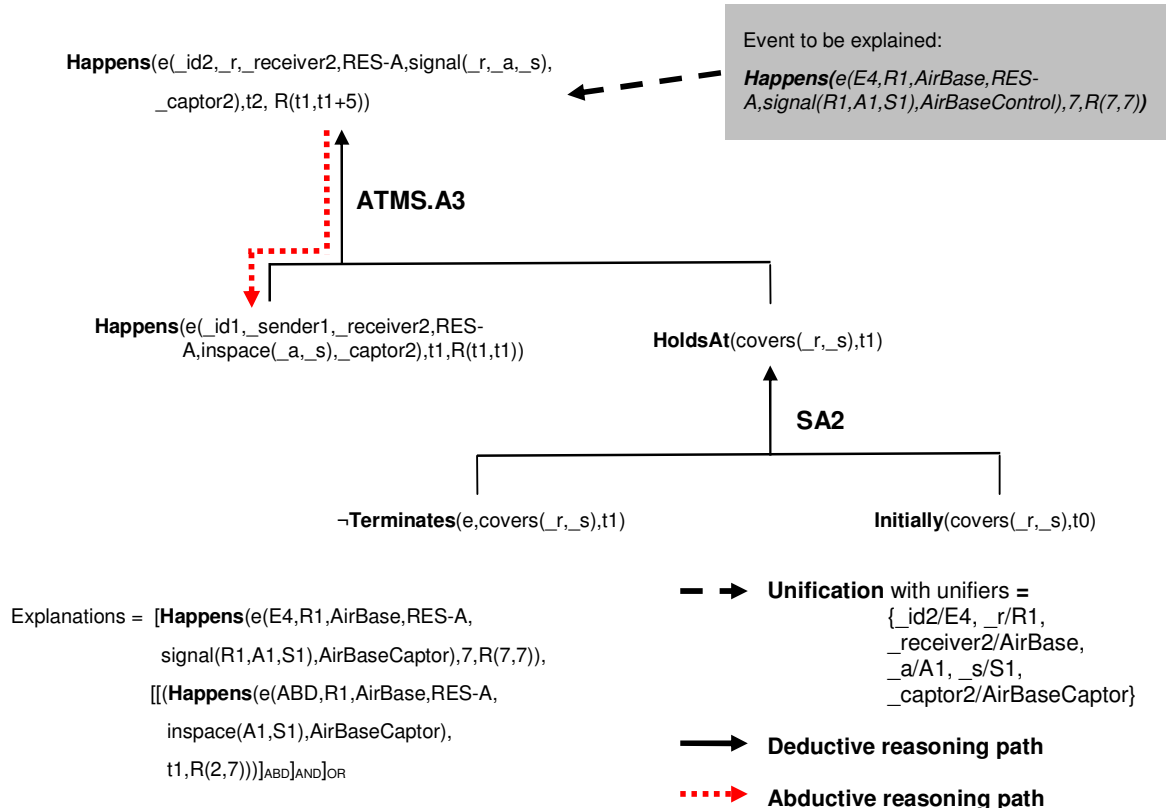

Figure 5-4 – Graphical view of explanation generation

5.3 Identification of Explanation Effects

5.3.1 The process of identifying explanation effects

After the generation of the possible explanations for the events involved in the violation of a rule, the diagnosis process identifies the expected effects of these explanations and uses them to assess the plausibility of the explanations. The assessment of explanation plausibility is based on the hypothesis that if the expected effects of an explanation match with events that have occurred (and recorded) during the operation of the system that is being monitored, then there is evidence about the validity of the explanation. This is

because the recorded events that match the expected effects of the explanation may have also been caused by the explanation itself. In case that any constituent abduced predicate of an explanation can be occurred or formally belongs to the OPreds set (see Section 4.2), it casts positive evidence to the plausibility of the explanation that is part of. It should be noted that under the same hypothesis the violation observation (event) that the explanation was generated for also casts positive evidence for the explanation. However, only the evidence that arises from this event is disregarded to avoid cycles in the reasoning process.

The identification of the expected effects of explanations is based on deductive reasoning. More specifically, given an explanation $Exp = P1 \wedge \ldots \wedge Pn$ that is expressed as a conjunction of abduced predicates, the diagnosis process iterates over its constituent predicates *Pi*. In case that any *Pi* is an observable predicate (i.e. belongs to the OPreds set (see Section 4.1)), then *Pi* itself is considered as expected consequence of explanation *Exp*. For each *Pi*, the diagnosis process finds the system assumptions $B1 \wedge \ldots \wedge Bn \Rightarrow H$ that have a predicate *Bj* in their body which can be unified with *Pi* and the rest of the predicates *Bu* ($u = 1,\ldots,n$ and $u \neq j$) in it are *True*. For such assumptions, if the predicate *H* in the head of the assumption is fully instantiated and its time range is determined, *H* is derived as a possible consequence of *Pi*. Then, if *H* is an observable predicate, i.e., a predicate that can be matched with recorded events, *H* is added to the expected effects of *Exp*. If *H*, however, is not an observable predicate, the effect identification process tries to generate the consequences of *H* recursively and, if it finds any such consequences that correspond to observable events, it adds them to the set of the expected effects of *Exp*. In this way, the diagnosis process computes the transitive closure of the effects of *Exp.*

To clarify the basis of this principle, assume that explanations of an event E1, which has been involved in the violation of a rule, need to be found based on the following set of assumptions:

(A1) $A \wedge B \Rightarrow E1$

(A2) $C \Rightarrow E1$

(A3) $C \Rightarrow E2$

(A4) $A \Rightarrow E3$

(A5) $B \Rightarrow E4$

Given the assumptions (A1) - (A5) and assuming that A, B and C are abducible atomic formulae, while C belongs to the OPreds set too, the algorithm *Explain* would generate the following two alternative explanations for E1:

$$[E1, [[[(A3:A)]_{ABD}, [(A3:B)]_{ABD}]_{AND}, [[(A4:C)]_{ABD}]]_{OR}$$

or, equivalently in a logical form, the explanations are as follows:

$$Exp_1(E1) \vee Exp_2(E1)$$

where:

$$Exp_1(E1) = A \wedge B, \text{ and}$$

$$Exp_2(E1) = C$$

To assess the plausibility of each of these explanations, we can identify the consequences that the individual atomic formulae that constitute them would have. Following this line of exploration, we can identify that

- if A had occurred it should have caused E3 due to assumption (A4)
- if B had occurred it should have caused E4 due to assumption (A5), and
- if C had occurred, there are two expected effects:

  ATMS.A1'. C should be included in the event log, due to the fact that C is observable predicate, and

  ATMS.A2'. C would have caused E2 due to assumption (A3)

Subsequently, if we assume that the event log of the monitoring infrastructure includes the events E1, E2, C, E3 and E5, the occurrence of C and E2 would cast supporting evidence for the hypothesis that C is true or, equivalently, that $Exp_2(E1)$ is valid. This evidence would be casted by the event E2. Similarly, E3 would cast some evidence that A is true and the absence of an event E4 in the log would provide some evidence that B is not true. Thus, there would be conflicting evidence for explanation $Exp_1(E1)$.

The assessment of the plausibility of alternative explanations in the diagnostic framework is based on this principle of collecting evidence about the truth of the abduced atomic formulae in explanations by searching for events in the log of the monitoring infrastructure that confirm or disconfirm the expected consequences of these abduced

formulae. This evidence is then used to compute belief measures in the existence of explanations using the Dempster Shafer theory of evidence [146] as we explain in Section 5.4.

Before, however, looking into the estimation of such beliefs, we present the process of generating the expected consequences of explanations. The generation of such consequences is based on the algorithm *Generate_AE_Consequences*, which is shown in Figure 5-5.

```
Generate_AE_consequences(AF: Set of Grounded Atomic Formulas, TLIST: List of Assumption Templates,
CNS: Set of Consequences)
1.  CNS = { }
2.  TLIST' = copy of TLIST
3.      For each atomic formula Pi ∈ AF Do
4.          If Pi is observable Then
5.              CNS = CNS ∪ { Pi }
6.          End If
7.          For each assumption template T in TLIST' Do
8.              For each predicate Q ∈ body(T) Do
9.                  If mgu(Pi,Q) ≠∅ and CompatibleTimeRange(Pi,Q) Then
10.                     T' := copy of T
11.                     Apply mgu(Pi,Q) onto T'
12.                     Set the truth value of Q in T' to True
13.                     Update time ranges of other predicates in T' based on the time range of Pi
14.                 If for all predicates R ∈ Body(T') such that R≠Q, R is true Then
15.                     If head(T') is fully instantiated Then
16.                         If head(T') is observable Then
17.                             CNS = CNS ∪ { (T.id, head(T')) }
18.                             delete T'
19.                         Else /*head(T') is a derived predicate */
20.                             CNS' = { }
21.                             Generate_consequences({head(T')}, TLIST', CNS')
22.                             CNS = CNS ∪ CNS'
23.                         End If
24.                     End If
25.                 Else /* there is a predicate R in Body(T') whose truth value is unknown */
26.                     If for all predicates R ∈ Body(T') such that R≠Q and
27.                                                         R is not true and
28.                                                         R is an abducible predicate Then
29.                         TLIST' = append (T', TLIST')
30.                     End If
31.                 End If
32.             End If
33.         End For
```

Figure 5-5 - Algorithm for computing the transitive closure of deductions from abduced predicates

The algorithm shown in Figure 5-5 takes as input the set of the abduced ground predicates P_i (i=1,…,n) of the conjunctive formula $P_1 \wedge P_2 \wedge \ldots \wedge P_n$ that constitutes an explanation (i.e. the AF input parameter) and finds all the grounded observable predicates

(i.e. the returned CNS input parameter) that could be derived from them. The computation of consequences is based on the assumptions specified for the system involved. The algorithm uses a set of templates of the given system assumptions (i.e., the input parameter *TLIST*). Let a template of any given assumption be a copy of the assumption formula. Please note that the aforementioned informal definition of a template implies implicitly that there is a process, which creates and deletes copies of any given assumption formula. During the reasoning process, a template can be partially or fully instantiated as result of the unification process that may take place.

To derive the possible consequences, the algorithm iterates over the input predicates P_i. For each *Pi*, in case that any *Pi* is an observable predicate (i.e. belongs to the OPreds set (see Section 4.2)) (see lines 3–6 in Figure 5-5), the algorithm adds *Pi* to the CNS set. Subsequently, the algorithm tries to find which of the partially instantiated assumption templates of the form *A: Body* $\Rightarrow$ *Head* have a predicate *P* in *Body* that can be unified with P_i and has a compatible time range with it (see lines 7-9 in Figure 5-5). For each assumption template that has such a predicate *Q*, the algorithm creates a new copy *T'* of it in order to represent the update (further instantiation) of the template with P_i and, in the new copy *T'*, it applies the unification found between *Q* and P_i to all the predicates of *T'*, sets the truth value of the predicate *Q* that was unified with P_i to *True* and updates the time ranges of all the predicates in *T'* based on the time range of P_i (see lines 10-13 in Figure 5-5). The creation of a new copy of the assumption template at this stage is necessary in order to ensure the completeness of the reasoning process. More specifically, by creating a new copy of the assumption template, there will be an opportunity to match the original assumption instance that is represented by the template *T* with some other predicate P_j in the conjunctive formula $P_1 \wedge P_2 \wedge \ldots \wedge P_n$, when the algorithm visits *Pj*.

Following the creation of the new template instance *T'* for P_i, the algorithm *Generate_AE_Consequences* checks whether the rest of the predicates in the body *T'* (if there are any) are also satisfied, i.e., they are grounded predicates and their truth value is *True* (see line 14 in Figure 5-5). For a *Happens*, *Initiates* or *Terminates* predicate R, this test is realized by checking the truth value of the predicate that is stored in the template since in cases where a grounded observable or derived predicate has already been unified with R, the truth value of R must have been set to *True*. For *HoldsAt* predicates, however, the test is a query to the fluent initiation and termination database of the monitoring framework that checks whether the condition expressed by the standard EVEREST

assumption SA1 and SA2 (see Section 3.3.2) is satisfied for the *HoldsAt* predicate at the required time point. Following this test, if the truth value of all the predicates R in the assumption template is *True* and, furthermore, the predicate head(T') in the *head* of *T'* is fully instantiated (i.e., all of head(T') variables have concrete values after the application of $mgu(P_i,Q)$ on *T'*) and the exact boundaries of its time range can be determined, the algorithm checks whether predicate head(T') that can been deduced from *T'* is an observable predicate (see line 16 in Figure 5-5). If head(T') is observable predicate, the algorithm adds it along with the identifier of the assumption that it used to derive it, to the possible consequences of the explanation $P_1 \wedge P_2 \wedge \ldots \wedge P_n$ (see line 17 in Figure 5-5). In this case, the algorithm also deletes *T'* as this fully instantiated predicate will be not useful to reasoning. If, however, the predicate head(T')is not an observable predicate, the algorithm still treats it as a derived predicate and recursively tries to identify the consequences of *head(T')* by invoking itself having *head(T')* as input (see line 21 in Figure 5-5). If the recursive invocation finds any consequences of *head(T')*, it adds them to the set of the expected consequences of the explanation (see line 22 in Figure 5-5). In this way, *Generate_AE_Consequences* computes the transitive closure of all the possible consequences of the abduced conjunctive explanation formula $P_1 \wedge P_2 \wedge \ldots \wedge P_n$, which could be matched with recorded events. These consequences are used in the next stage of the diagnosis process in order to find which of them indeed match with recorded events and which do not and calculate the likelihood of $P_1 \wedge P_2 \wedge \ldots \wedge P_n$.

Note that if, during the execution of the *Generate_AE_Consequences* algorithm, an input ground atomic formula can be unified with a predicate in the body of an assumption template *T'* but the rest of the body predicates of *T'* whose truth values cannot be established yet, the algorithm stops exploring *T'* further in the current iteration. However, *T'* is appended to the list of templates *TLIST'* in case that the predicates in the body of *T'* whose truth values cannot be established yet are abducible predicates (see lines 26-29 in Figure 5-5). By appending *T'* to *TLIST'*, we succeed in making *T'* available for consideration at a next iteration when another input atomic formula in AF (or, equivalently, in $P_1 \wedge P_2 \wedge \ldots \wedge P_n$) is considered. This is not necessary in the case of partially instantiated templates that have body predicates, which are not evaluated yet but correspond to derived or observable predicates. Such templates are not appended to *TLIST'* due to the fact that the truth value of derived or observable predicates is not

changing until the end of the execution of the current invocation of *Generate_AE_Consequences* (this point is discussed further below).

The specification of the algorithm *Generate_AE_Consequences* assumes that, when the algorithm is invoked, the set of the assumption templates *TLIST* encodes the following:

- any event (i.e ground observable predicate), which has been recorded and stored in the log of the EVEREST monitoring framework up to the invocation time of the algorithm, and
- any ground derived predicate, which can be generated from the recorded events up to the time of the invocation of the algorithm.

The above assumption is valid since as soon as a new recorded event arrives at the monitoring framework. Any new recorded event, e_n, is checked against the set of the assumption templates, which exist up to that point, to identify if there are body predicates of the existing templates, which e_n could be unified with. If there are such templates, e_n is unified with them, and all the head predicates of the templates, which can be derived following this unification, are generated. The algorithm, which processes recorded events in order to update the assumption template list and generate the transitive closure of the predicates that can be derived from recorded events, is shown in Figure 5-6. This algorithm is called *Generate_RE_Consequences* and operates based on the same forward reasoning process as *Generate_AE_Consequences*.

```
Generate_RE_consequences(AF: Set of Grounded Atomic Formulas, TLIST: List of Assumption Templates, CNS: Set of Consequences)
1.  CNS = { }
2.   For each atomic formula Pi ∈  AF Do
3.      For each assumption template T in TLIST Do
4.          For each predicate Q ∈ body(T) Do
5.              If mgu(Pi,Q) ≠∅ and CompatibleTimeRange(Pi,Q) Then
6.                  T' := copy of T
7.                  Apply mgu(Pi,Q) onto T'
8.                  Set the truth value of Q in T' to True
9.                  Update time ranges of other predicates in T' based on the time range of Pi
10.                 If for all predicates R ∈ Body(T') such that R≠Q, R is true Then
11.                     If head(T') is fully instantiated Then
12.                         If head(T') is observable Then
13.                             CNS = CNS ∪ { (T.id, head(T')) }
14.                             delete T'
15.                         Else /*head(T') is a derived predicate */
16.                             CNS' = { }
17.                             Generate_consequences({head(T')}, TLIST, CNS')
18.                             CNS = CNS ∪ CNS'
19.                         End If
20.                     End If
21.                 Else /* there is a predicate R in Body(T') whose truth value is unknown */
22.                     TLIST = append (T', TLIST)
23.                 End If
24.             End If
25.         End For
26.     End For
27.  Return (CNS)
END Generate_RE_consequences
```

Figure 5-6 – Algorithm for computing the transitive closure of deductions from recorded events

The differences between the *Generate_RE_Consequences* and *Generate_AE_Consequences* are that:

1. *Generate_RE_Consequences* is invoked to process ground observable predicates that represent recorded events whilst *Generate_AE_Consequences* is invoked to process ground abducible predicates representing abduced explanations, and

2. Instead of operating on a copy of the assumption template list (*TLIST*) as *Generate_AE_Consequences* does, *Generate_RE_Consequences* operates on this list directly and updates it when possible (see lines 21−23 in Figure 5-6).

Thus, at the end of an invocation of *Generate_RE_Consequences*, any templates, which remain partially instantiated after the unification process with a recorded or derived predicate, are made available for further updates. Further updates can be done either by the same algorithm, *Generate_RE_Consequences*, when a new recorded event occurs and the algorithm is invoked to process it or by *Generate_AE_Consequences* when the latter algorithm is called to generate consequences of abduced explanations.

It should be noted that the *Generate_RE_Consequences* algorithm is invoked every time that a new recorded event arrives to the monitoring framework and its results are required for the normal monitoring process since derived predicates are also necessary in detecting violations with respect to derived and recorded events. *Generate_AE_Consequences*, on the other hand, is invoked every time that a request for the diagnosis of a rule violation is made by the user of the monitoring framework. When any of the two algorithms is invoked, it obtains a lock over the current set of assumption templates (*TLIST*) to ensure that the other algorithm cannot process *TLIST*.

We should also note that the set of the consequences, which is produced by the algorithm *Generate_AE_Consequences*, does not include all the possible consequences, which could be produced from the abduced predicates $P_1 \wedge P_2 \wedge \ldots \wedge P_n$ constituting an explanation. It includes only the complete set of consequences that can be derived based on the knowledge of the system (i.e., the set of the ground observabe and derived predicates) at the time of the algorithm's invocation. This set of consequences is generally a subset of the set of all potential consequences, which could have been generated by using the available relevant knowledge. This can be explained, as we mentioned above, due to the fact that there might be assumption templates, whose body predicates that can be instantiated with recorded and derived events are not known yet at the time of the invocation of *Generate_AE_Consequences* algorithm. Such templates cannot be used for deriving any consequences

5.3.2 Examples of explanation effects identification

To elaborate on the process of generating explanation consequences, let us consider the ATMS example and more specifically the diagnosis of the violation of rule ATMS.R1, which was initially introduced in Section 4.3 and whose specifications is as follows:

ATMS.R1 ∀t1∈Time, ∃t2∈Time, ∀_r1∈Radars, ∀_receiver1,
∀_a∈Airplanes, ∀_s∈Airspaces, ∀_r2∈Radars, ∀_source1.
Happens(e(_id1,_r1,_receiver1,RES-A,signal(_r1, _a, _s),
_soure1),t1,R(t1,t1)) ∧
HoldsAt(covers(_r1,_s),t1) ∧
HoldsAt(covers(_r2,_s), t1) ∧
_r2 ≠ _r1 ⇒
Happens(e(_id2,_r2,_receiver1,RES-A,signal(_r2, _a, _s),
_source1),t2,R(t1, t1+5)

Assuming that the monitor has received the events shown in the log of Figure 5-3, the rule ATMS.R1 is violated by:

- the event (E4) (i.e., `Happens(e(E6, R1, AirBase, RES-A, signal(R1, A1, S1),AirBaseCaptor),7,R(7, 7)))` in the event log of Figure 5-3
- the atomic formula `Happens(e(NF,R2,AirBase,signal(R2,A1,S1), AirBaseCaptor),t,R(7,12))`, which signifies the absence of a signal from radar R2 within the time period expected by ATMS.R1 given the signal of radar R1, and
- the atomic formulas `HoldsAt(covers(R1,S1),7)` and `HoldsAt(covers(R2,S1),7)`

Also, as it is shown in Section 5.2.2, the computed explanation of event (E4) is as follows:

[**Happens**(e(E4,R1,AirBase,RES-A, signal(R1,A1,S1),AirBaseCaptor),7,R(7,7)),
[
[(**Happens**(e(ABD,R1,AirBase,RES-A,inspace(A1,S1),AirBaseCaptor),
t1,R(2,7)))]$_{ABD}$
]$_{AND}$
]$_{OR}$

Recall, also, that the following assumptions are considered for the ATMS example in Section 4.3:

ATMS.A5 ∀t1∈Time, ∀_sender, ∀_receiver, ∀_source, ∀_a∈Airplanes, ∀_s∈Airspaces.
Happens(e(_id1,_sender,_receiver,RES-A,inspace(_a,_s), _source),t1,R(t1,t1)) ⇒
Initiates(e(_id1,_sender,_receiver,RES-A,inspace(_a, _s), _source), inairspace(_a,_s),t1

ATMS.A6 ∀t1∈Time, ∃t2∈Time, ∀_receiver, ∀_source, ∀_a∈Airplanes, ∀_s∈Airspaces, ∃_airportX∈Airports.
Initiates(e(_id1,_a,_receiver,RES-A,inspace(_a, _s),_source),inairspace(_a,_s),t1) ∧
HoldsAt(landing_airspace_for(_s,_airportX),t1) ⇒
Happens(e(_id2,_a,_receiver,RES-A, landingRequest(_a,_airportX),_source),t2, R(t1-10,t1))

ATMS.A7 ∀t1∈Time, ∀_sender, ∀_receiver, ∀_source, ∀_a∈Airplanes, ∀_airportX∈Airports, ∃_s∈Airspaces.
Happens(e(_id1,_sender,_receiver,RES-A, changeOfLandingApproach(_airportX,_s), _source),t1,R(t1,t1)) ⇒
Initiates(e(_id1,_sender,_receiver,RES-A, changeOfLandingApproach(_airportX,_s),_source), landing_airspace_for(_s,_airportX),t1)

ATMS.A8 ∀t1∈Time, ∀_sender, ∀_receiver, ∀_source, ∀_a∈Airplanes, ∀_airportX∈Airports, ∃_s∈Airspaces.
Happens(e(_id1,_sender,_receiver,RES-A, removeLandingApproach(_airportX,_s), _source),t1,R(t1,t1)) ⇒
Terminates(e(_id1,_sender,_receiver,RES-A, removeLandingApproach(_airportX,_s),_source), landing_airspace_for(_s,_airportX),t1)

Recalling the meaning of the above assumptions, assumption ATMS.A5 states that when an event that signifies the entrance of an aircraft *_a* in an airspace *_s* becomes known, and at that timepoint a fluent called `inairspace(_a,_s)` should be initiated to signify the presence of the aircraft in the particular airspace. The second assumption (i.e., assumption ATMS.A6) states that when an aircraft *_a* enters an airspace *_s* that is used for approaching the final landing route to an airport then the aircraft *_s* must have made a landing request for the particular airport within the last 10 time units before entering *_s*. In this assumption, the airspace that is used as the landing approach for an airport is indicated by the fluent `landing_airspace_for(_s,_airportX)` and landing requests are expressed by operations of the form `landingRequest(_a, _airportX)`. Finally the third and fourth assumptions above, namely ATMS.A7 and ATMS.A8, are used for setting the airspace that is used as the landing approach for an airport. This is done by initiating and terminating respectively the fluent `landing_airspace_for(_s, _airportX)` every time operations of the form `changeOfLandingApproach( _airportX, _s)` and `removeLandingApproach(_airportX, _s)` are called.

Using the assumptions ATMS.A5 and ATMS.A6, we can derive certain expected consequences for the abduced formula `Happens(e(ABD,R1,AirBase,RES-A,inspace(A1,S1),AirBaseCaptor),t1,R(2,7)))` that was generated as a possible explanation of the event `Happens(e(E4, R1, AirBase, RES-A, signal(R1, A1, S1),AirBaseCaptor),7,R(7,7))` in the way we described in Section 5.2.2. In summary, if we assume that the airspace *S1* is the landing airspace of the airport *AR-a*, then the entrance of the aircraft *A1* into *S1* would make us expect that there should be some request from *A1* to land in *AR-a* or, equivalently, that a runtime event `Happens(e(DER,A1,AirBase,RES-A,landingRequest(A1,AR-a), AirBaseCaptor), t2, R(0,6))` should have occurred. This runtime event would, thus, be an expected consequence of the abduced explanation `Happens(e(ABD,R1,AirBase,RES-A,inspace(A1,S1),AirBaseCaptor),t1,R(2,7))).`

The reasoning path for deriving `Happens(e(DER,A1,AirBase,RES-A,landingRequest(A1,AR-a), AirBaseCaptor), t2, R(0,6))` is as follows:

1. *Step-1*: From the event (E3) in the log of Figure 5-3 (i.e., `Happens(e(E3,AirBase,AirBase,RES-A,changeOfLandingApproach(AR-a, S1),AirBaseCaptor),2,R(2,2))` and the assumption ATMS.A7 we can derive the fluent initiation formula:

```
Initiates(e(E5,AirBase,AirBase,RES-A,changeOfLandingApproach
(AR-a,S1),AirBaseCaptor),landing_airspace_for(AR-a,S2),2)
```
(DP1)

2. *Step-2*: From the atomic formula `Happens(e(ABD, R1, AirBase, RES-A, inspace(A1,S1),AirBaseCaptor),t1,R(2,7)))` that was abduced as an explanation of `Happens(e(E4, R1, AirBase, RES-A, signal(R1, A1, S1), AirBaseCaptor),7,R(7,7))` and assumption ATMS.A5, we can derive the fluent initiation formula:

```
Initiates(e(ABD,R1,AirBase,RES-A,inspace(A1,S1),
          AirBaseCaptor),t1,R(2,7)),inairspace(A1,S1),
          t1,R(2,7)))
```
(DP2)

- *Step-3*: From (DP1), the standard EVEREST assumption SA1, and the absence of any event e generated by the call of operation `removeLandingApproach(_airportX,_s)`, which could terminate the fluent `landing_airspace_for(AR-a,S2)` due to assumption ATMS.A8, within the time range [2,7], we can deduce the atomic formula DP3 that is given below. Please note that the check whether DP3 is *True* for *t* in [2, 7] is done by a query to the fluent initiation and termination database of EVEREST.

```
∀t∈[2,7].HoldsAt(landing_airspace_for(S1,AR-a),t)
```
(DP3)

3. *Step-4*: From (DP3), (DP2) and ATMS.A6, we can deduce the formula:

```
Happens(e(DER,A1,AirBase,RES-A,landingRequest(A1,AR-a),
        AirBaseCaptor),t2,R(0,6))
```
(DP4)

At this point we explain how the algorithms *Generate_RE_Consequences* and *Generate_AE_Consequences* will take the steps of the above reasoning path. *Step-1* (Figure 5-7) will be executed first by the algorithm *Generate_RE_Consequences* when the event `Happens(e(E3,AirBase,AirBase,RES-A,changeOfLandingApproach(AR-a, S1),AirBaseCaptor),2,R(2,2))` occurs at time *t=2* and *Generate_RE_Consequences* is invoked to process it. To generate the formula (DP1) that is derived in this step, *Generate_RE_Consequences* will:

i) create a new template of the formula ATMS.A7,

ii) unify the runtime event `Happens(e(E3,AirBase,AirBase,RES-A,changeOfLandingApproach(AR-a, S1), AirBaseCaptor),2,R(2,2))` with the only predicate in the body of the template, and

iii) create the derived predicate `Initiates(e(E3, AirBase, AirBase, RES-A, changeOfLandingApproach(AR-a, S1), AirBaseCaptor), 2, R(2, 2)), landing_airspace_for(S1, AR-a), 2)` as an instantiation of the head of the template.

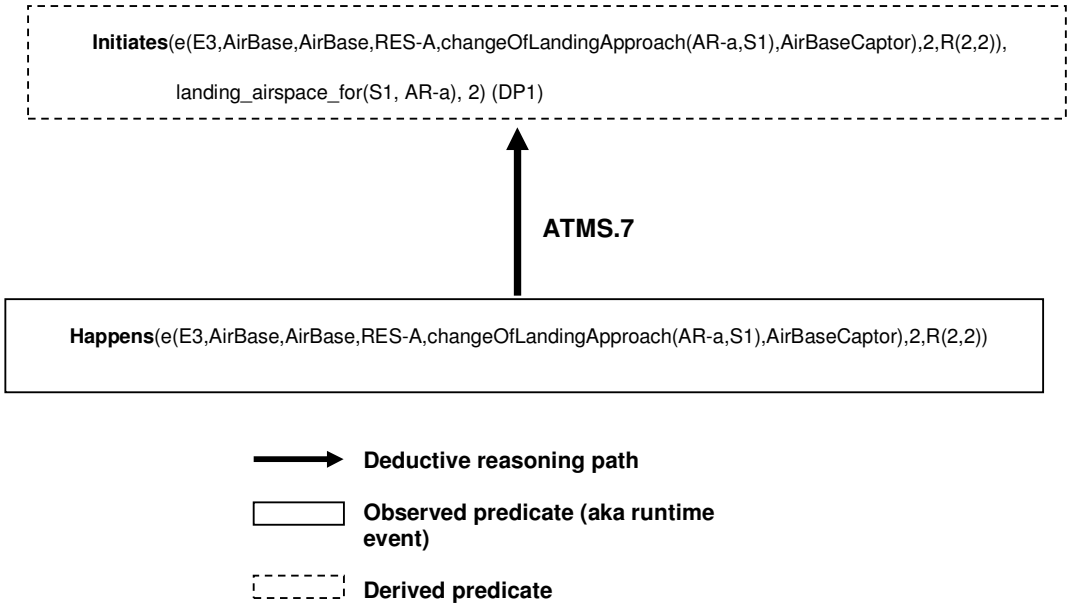

Figure 5-7 – Step1 executed by Generate_RE_Consequences

Subsequently, *Step-2* (Figure 5-8) will be executed by the algorithm *Generate_AE_Consequences* at some time point following the time point *t=10* when the violation of ATMS.R1 is detected and the generation of a diagnosis of this violation is requested. Suppose that *Generate_AE_Consequences* is invoked to generate the consequences of the abduced explanation `Happens(e(ABD, R1, AirBase, RES-A, inspace(A1, S1), AirBaseCaptor), t1, R(2,7)))` immediately of the detection of the violation of ATMS.R1 at *t=11*. Upon its invocation, *Generate_AE_Consequences* will:

i) create a new template of the formula ATMS.A5,

ii) unify `Happens(e(ABD, R1, AirBase, RES-A, inspace(A1, S1), AirBaseCaptor),t1,R(2,7)))` with the single predicate in the body of this template, and

iii) generate the derived predicate `Initiates(e(ABD, R1, AirBase, RES-A, inspace(A1, S1), AirBaseCaptor), t1, R(2,7)), inairspace(A1, S1), t1, R(2, 7)))` as an instantiation of the head of the template.

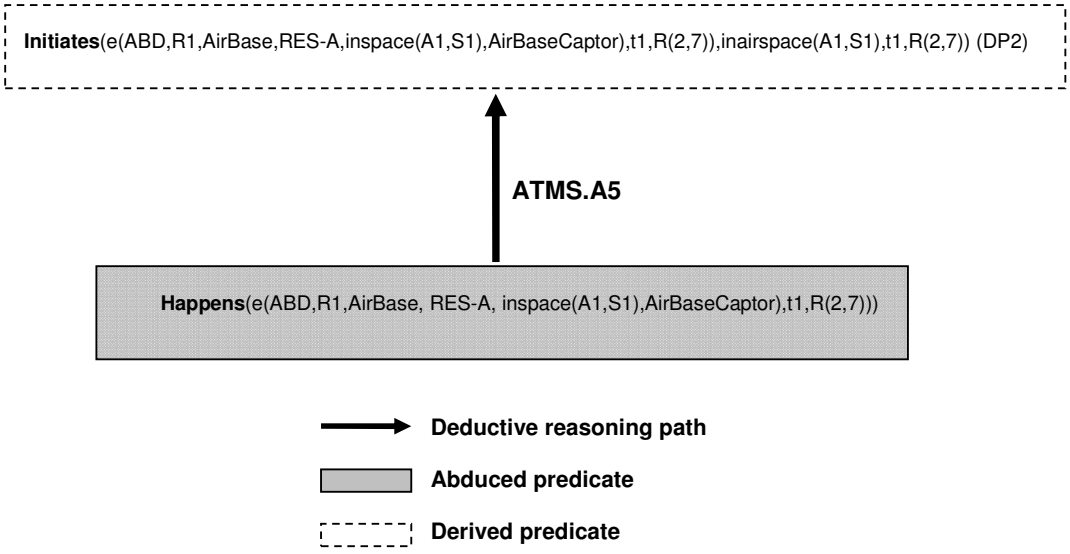

Figure 5-8 – Step2 executed by Generate_RE_Consequences

Following this, since `Initiates(e(ABD,R1,AirBase,RES-A,inspace(A1,S1), AirBaseCaptor),t1,R(2,7)),inairspace(A1,S1), t1,R(2,7))` is not an observable predicate, *Generate_AE_Consequences* will invoke itself recursively to generate further consequences from this predicate. Thus, at this point, *Generate_AE_Consequences* will execute *Step-4*.

In order to execute *Step-4*, *Generate_AE_Consequences* will identify that the derived predicate `Initiates(e(ABD, R1, AirBase, RES-A,inspace(A1, S1),AirBaseCaptor),t1,R(2,7)),inairspace(A1,S1),t1,R(2,7))` can be unified with a predicate in the body of the assumption ATMS.A6. Thus, it will create a new template of ATMS.A6 by unifying the derived predicate `Initiates(inspace(A1,S1),t1,R(2,7)), inairspace(A1,S1),t1,R(2,7)))` with this template. Following this, it will check if the remaining predicates in the body of the newly created assumption template (i.e., `HoldsAt(landing_airspace_for(S1,AR-`

a),t) for all *t* in [2,7]) is *True*. As this predicate is a HoldsAt predicate, *Generate_AE_Consequences* will query the fluent database of the monitoring framework to check if the HoldsAt predicate is true. During the execution of this query, the monitoring framework will execute *Step-3* (Figure 5-9), and to check the truthness of the HoldsAt predicate.

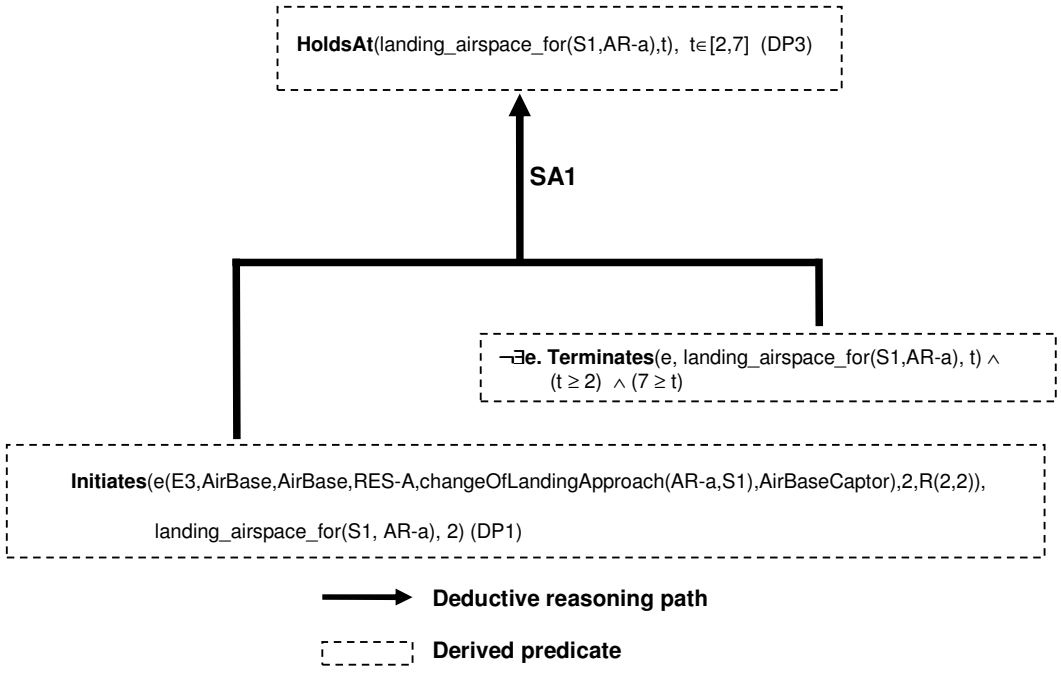

Figure 5-9 – Step3 executed by fluent maintenance mechanisms of EVEREST

When it is verified that the HoldsAt(landing_airspace_for(S1,AR-a),t) is *True* for *t* in [2,7], *Generate_AE_Consequences* will derive the consequence Happens(landingRequest(A1,AR-a), t2, R(0,6)) as an instantiation of the head of the formula ATMS.A6 (Figure 5-10).

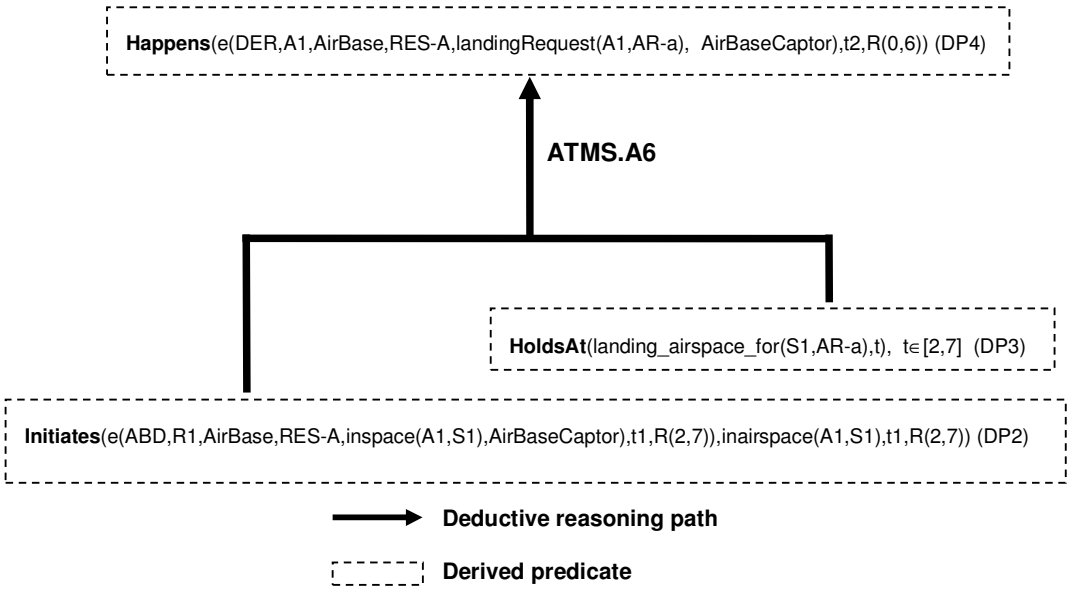

Figure 5-10 – Step4 executed by Generate_AE_Consequences

As we discussed earlier, the algorithm *Generate_AE_Consequences* might fail to generate all the possible consequences of a given explanation if certain runtime events required for them have not arrived yet. Assume, for instance, that in the above example the event `Happens(changeOfLandingApproach(AR-a,S1),2,R(2,2))` had not arrived at the monitoring framework until the time point *T=14* due to a delay in the communication channel between the event captor that captures events of these type and the monitoring framework. In this case, *Generate_RE_Consequences* would not have executed *Step-1*, when the algorithm *Generate_AE_Consequences* was invoked. Therefore, *Generate_AE_Consequences* algorithm would not be able to find that the predicate `HoldsAt(landing_airspace_for(S2,AR-a),t)` was *True* for all *t* in [2, 7]. Thus, in this case it would also be unable to deduce the formula `Happens(landingRequest(A1,AR-a), t2, R(0,6))` from ATMS.A6.

5.4 Plausibility Assessment

In this section, we describe the next step of the diagnostic process that is namely the assessment of the genuineness of the events involved in S&D rule violations. Goal of this step is to provide an assessment scheme for event genuineness based on the plausibility and correctness of the alternative explanations that are generated for an event in the previous steps of the diagnostic process.

5.4.1 Foundations of the assessment

The goal of the diagnostic process is to provide the most plausible cause of S&D violations. The causes of violations are provided in terms of genuineness of the violation observations and other correlated events. Let events be the set of the ground observables predicates. All the events are associated with a timestamp and a time range. In case events do not have a specific value for their timestamp, let them be called parametric events. On the other hand let the events that have a specific value for their timestamp be called fully specified events. Consequently, the events recorded in the log of the monitoring framework are fully specified. It should also be noted that the time range boundaries of any parametric event are different numbers, while the corresponding ones of any fully specified event are both equal to the timestamp of the event. Our initial hypothesis for event genuineness considers that:

- An event is genuine, if it has occurred and is a result of the monitored system's intended behaviour and not of an attack or fault (Hypothesis 1)

5.4.1.1 Event occurrence

Evaluating whether an event has occurred is based on the log of the monitoring framework. More specifically, if an event e_i, with timestamp te_i and time range [te_i^{LB}, te_i^{UB}], can be matched with an event recorded in the log of the monitoring framework, we assume that e_i has occurred. A match between a recorded event e^{log} in the log, which has been produced by an event captor $Captor(e^{log})$ and has a timestamp t_e^{log}, and an event e_i, whose occurrence is under question is detected only if the following three conditions are satisfied:

- e^{log} has been produced by the same event captor as the captor that e_i is expected to be produced from or, formally, if $Captor(e^{log}) = Captor(e_i)$ (Condition 9)
- e^{log} can be unified with e_i or formally $mgu(e^{log}, e_i) \neq \varnothing$ (Condition 10)
- The timestamp of the event e^{log} is equal to the timestamp of e_i in case that e_i is fully specified (i.e., $t_e^{log} = te_i$) or falls within the time range of e_i in case that e_i is parametric(i.e., $te_i^{LB} \leq t_e^{log}$ and $t_e^{log} \leq te_i^{UB}$) (Condition 11)

The absence of a matching recorded event for an event e_i at the time of the search does not necessarily mean that such an event has not occurred. The absence of a matching

recorded event could have been caused by delays in the "channel" between the event captors and the monitoring framework.

More specifically, there might be cases where, although a recorded event that satisfies the conditions 9 - 11 above may have occurred, this event might not have arrived at the event log of the monitoring framework yet, due to communication delays in the "channel" between the event captor, which captured the event, and the framework. To cope with this possibility, in cases where no matching recorded event is found at the time of the initial search, the search process should take into account the time of the last recorded event that has been received from the captor of e_i. If this time is less than the upper boundary te_i^{UB} of the time variable of e_i, the search process should wait until either a recorded event that matches e_i or a not matching recorded event with a timestamp that is greater than te_i^{UB} arrives from the relevant captor. The arrival of the first recorded event *e'* from the captor of e_i which does not match e_i and has a timestamp that is greater than te_i^{UB} would be sufficient to establish with certainty that no recorded event matching e_i has occurred so far. The reason that the arrival of such a recorded event would be sufficient for establishing the absence of any match for e_i is that we assume that the communication connections between event captors and the monitoring framework realise the TCP/IP protocol and, therefore, the events which are generated by a specific captor arrive at the monitoring framework in exactly the same order as the order in which they were captured and dispatched. Thus, it is valid to assume that if there was a recorded event matching e_i that had been captured and dispatched before *e'*, this event should have arrived at the monitoring framework before *e'*.

It should be appreciated, however, that although a "wait" search process would allow assessing with more certainty whether an expected consequence of an explanation has been confirmed or not by recorded events, the adoption of this approach could create two problems. The first problem is that it could delay the diagnosis generation process significantly, depending on the amount of traffic in the communication channels between the captors that generate the expected events and the monitoring framework and the speed of these channels. Such a delay might not be desirable in cases where a timely diagnosis is necessary in order to decide how to react to an S&D rule violation. The second potential problem of a "wait" search process is that if an event captor and/or the communication channel between it and the monitoring framework become unavailable, the diagnosis process could also be stalled.

To avoid these potential problems, the search for the occurrence of e_i in the log establishes the negation of e_i that is expected to have been produced by the captor *Captor*(e_i) if:

- there is no recorded event *e* satisfying *Conditions 9-11* above at the time of the search, and
- the last known value of the clock of *Captor*(e_i) (i.e., the timestamp of the last event in the log that has arrived from this captor) is greater than te_i^{UB}.

However, there is also a possibility that we would not be able to confirm or disconfirm the occurrence of an event or observable predicate at the time of the search. These would events e_i for which no matching recorded event satisfying the *Conditions 9-11* could be found at the time of the search and the last received event from the relevant captor had a timestamp that was less than or equal to their upper time boundary (i.e., $lastTimestamp(Captor(e_i)) \leq te_i^{UB}$). To cope with this uncertainty, we have decided to use the *Dempster Shafer theory of evidence* [146] (see Section 3.4 for a brief theory introduction) for assessing whether an event has been occurred based on the log of the monitoring framework, and therefore whether can be considered genuine.

5.4.1.2 Event as a result of intended system's behaviour

In order to evaluate whether an event e_i results from the system intended behaviour can be based on the co-occurrence of other events e_j ($j \neq i$) that can be related to e_i according to the specifications of the underlying monitoring theory. Let the events e_j ($j \neq i$) be the set of correlated events of e_i due to theory *T*. Particularly, in EVEREST, an underlying monitoring theory *T* includes a partial model of the intended behaviour of the monitored system (assumptions) that specifies causal relations between events. Thus, the occurrence of an event e_i can follow, as a consequence of, or be followed by, as a cause of the occurrence of its correlated events e_j ($j \neq i$). In the former case, backwards chaining can be applied on the assumptions of the underlying theory *T* by starting from e_i and reaching members of e_j that could potentially explain the occurrence of e_i. Similarly, regarding the latter case, forward chaining can be applied on the assumptions of the underlying theory *T* by starting from e_i and reaching members of e_j that, according to the assumptions of *T*, can be considered logical consequences of the occurrence of e_i. In order to traverse backward and forward through the assumptions of *T*, we devised algorithms based on abduction (see Section 2.3 for the underpinning theory of abductive

reasoning and 5.2.1 for the devised algorithm) and deduction (see Section 5.3 for the devised algorithm). For instance, assume the theory that includes the five formulas as follows:

F1. $E1 \Rightarrow E2$

F2. $E3 \wedge E4 \Rightarrow E2$

F3. $E1 \Rightarrow E5$

F4. $E5 \Rightarrow E6$

F5. $E3 \Rightarrow E7$

From F1 and F2, E2 is specified as a consequence of E1 and the conjunction of E3 and E4 respectively. Thus, in case that E2 occurs, then we can consider the hypothesis that the occurrence of either E1 or E3 and E4 could explain the occurrence of E2. In F3, E1 is specified as a possible explanation of E5, and thus if E1 occurs, E5 should occur as well. Similarly, formula F4 specifies that E6 is a (direct) consequence of E5. Please, also, note that E6 can be deductively inferred as an (indirect) consequence of E1 as well due to F3 and F4. Finally, formula F5 specifies that E7 is a (direct) consequence of E3.

By using abductive reasoning, we can only generate hypotheses for the occurrence of other events e_j ($j \neq i$) that could potentially explain the occurrence of an event e_i. Thus, the abductive explanations should be checked for their correctness and plausibility. The correctness and plausibility of abductive explanations could be validated by the occurrence of their expected consequences, which can be generated by deductive reasoning on the assumptions of the underlying theory *T*. Particularly, it is the occurrence of the consequences that can validate an explanation. Thus, the expected consequences of an explanation, which are generated by deduction, should match with events recorded in the log of EVEREST in order that the explanation could be considered plausible and correct, and therefore valid. Also, in cases that a generated explanation can be matched with an event recorded in the log of EVEREST, the explanation itself can be considered as a consequence of itself due to deduction (i.e., $A \Rightarrow A$ is true for any A). However, the recorded events that match to the generated consequences, as any other recorded event, should be questioned whether they result from the intended behaviour of the system, or equivalently whether they are genuine.

5.4.1.3 Underpinning principles for event genuineness

After taking into consideration the above remarks, our initial hypothesis (Hypothesis 1) was reconsidered, and finally the following hypotheses have been concluded:

- An event is genuine if it has occurred, i.e. is recorded in the log of the monitoring framework, and all of its explanations are valid (Hypothesis 2)
- An explanation is valid, i.e. plausible and correct, if all of its expected consequences match with events recorded in the log of the monitoring framework, which are genuine themselves (Hypothesis 3)

As a prerequisite for providing the formal framework of the assessment of event genuineness, the definition of the alternative explanations of an event and the search of supporting and refuting evidence for the generated alternative explanations are provided in the next section.

5.4.2 Alternative explanations, expected consequences and search for supporting and refuting evidence for alternative explanations

Following the identification of the expected consequences of alternative explanations of events, the diagnosis process searches the event log of the monitoring framework to find recorded events that can match these consequences, and therefore, cast evidence to the validity of the alternative explanations.

Let the set of alternative explanations of an event e_i is formally be defined as follows:

Definition 1: The explanation set of an event e_i is defined as follows:

$$EXP(e_i) = \{\Phi_{i1}, \ldots, \Phi_{iN}\}$$

where,

- Φ_{ij} (j=1,…,N) is a conjunction of abduced events of the form $P_{ij1} \wedge \ldots \wedge P_{ijK}$ that constitute a possible explanation of e_i and has been produced by the algorithm *Explain* (Figure 5-2).

At this point, it should be noted that in case that e_i belongs to APreds $\cap$ OPreds (i.e. e_i is an abducible and observable predicate), no explanation can be generated for e_i, thus, $EXP(e_i) = \varnothing$.

Definition 2: The set of the expected consequences of the exlanation Φ_{ij} of an event e_i is defined as follows:

$$CONS(e_i,\Phi_{ij}) = \{C_{ij1}, \ldots, C_{ijL}\}$$

where,

1. $C_{ij1},\ldots,C_{ijL}$ are atomic formulae, which can be derived from the conjunction of the abduced atomic formulae $P_{ij1} \wedge \ldots \wedge P_{ijK}$ that constitute the explanation Φ_{ij} and indicate parametric events that would have been caused by Φ_{ij}, if explanation Φ_{ij} had indeed occurred. The set $\{C_{ij1},\ldots,C_{ijL}\}$ is formally defined as:

   $$\{C_{ij1},\ldots,C_{ijL}\} = \text{Consequences}(\{P_{ij1},\ldots,P_{ijK}\})$$

   where Consequences(S) is the set of expected consequences that is generated from a set of atomic formulas S by the algorithm *Generate_AE_Consequences* (Figure 5-5).

   Please note that, given the algorithm *Generate_AE_Consequences* (Figure 5-5), Φ_{ij}^{C} includes the parametric events whose derivation path involves at least one of the abduced atomic formulae $P_{ij1},\ldots,P_{ijK}$. The consequences of each subset of the set of the abduced atomic formulas $P_{ij1},\ldots, P_{ijK}$ are considered consequences of the explanation Φ_{ij} due to the fact that

   $$P_{ij1} \wedge \ldots \wedge P_{ijK} \Rightarrow \bigwedge_{S \subseteq \Phi ij \text{ and } Px \in S} P_x$$

Based on the above definition, assume that we have the following three explanations of an event e_i that is involved in an S&D rule violation:

Φ_{i1} = **Happens**(AE1,t1,R(0,5))

Φ_{i2} = **Happens**(AE1,t1,R(0,5)) $\wedge$ **Happens**(AE2,t2,R(5,5))

Φ_{i3} = **Happens**(AE2,t1,R(12,12))

Also, assume that AE2 belongs to the observable predicates set OPreds and the following assumptions constitute the underlying theory:

Assumption A. **Happens**(AE1,t1,R(t1,t1)) $\Rightarrow$ **Happens**(E1,t1,R(t1+1,t1+1))

Assumption B. **Happens**(AE1,t1,R(t1,t1)) $\Rightarrow$ **Happens**(DE1,t2,R(t1,t1+1))

Assumption C. **Happens**(DE1,t1,R(t1,t1)) ∧ **Happens**(AE2,t2,R(t1,t1+1)) ⇒ **Happens**(E2,t3,R(t2,t2))

Assumption D. **Happens**(AE2,t1,R(t1,t1)) ⇒ **Happens**(E3,t1,R(t1,t1))

The set of the consequences of the abduced explanation Φ_{i2}, CONS(e_i,Φ_{i2}), will include the atomic formulae (i.e. events) Happens(AE2,t1,R(12,12)), Happens(E2, t3, R(6, 6)), Happens(E1, t1, R(1, 6)) and Happens(E3, t1, R(5, 5)). More specifically, CONS(e_i,Φ_{i2}) includes the event Happens(AE2,t1,R(12,12)) due to the fact that it is observable predicate. CONS(e_i,Φ_{i2}) also includes the consequences whose derivation path includes both atomic formulas of Φ_{i2}, Happens(AE1, t1, R(0, 5)) and Happens(AE2, t2, R(5, 5)). Thus, the event Happens(E2,t3,R(6,6)) is included in CONS(e_i,Φ_{i2}) as its derivation fits the criterion of Definition 1 since:

- from Assumption B and Happens(AE1,t1,R(0,5)) we derive Happens(DE1,t2,R(0,6)), and
- from Happens(DE1,t2,R(0,6)), Happens(AE2,t2,R(5,5)) and Assumption C we derive Happens(E2,t3,R(6,6)).

Moreover, the event Happens(E1,t1,R(0,6)) is included in CONS(e_i,Φ_{i2}) since:

- from Assumption A and Happens(AE1,t1,R(0,5)) we derive Happens(E1,t1,R(1,6)).

Finally, Happens(E3, t1, R(t1, t1)) is also included in CONS(e_i,Φ_{i2}) since:

- from Assumption D and Happens(AE2,t2,R(5,5)) we derive Happens(E3, t1, R(5, 5))

After deriving the expected consequences of an explanation Φ_{ij}, the diagnostic process searches the log of the monitoring framework to find recorded events that can match these consequences, as it will be shown in the folloing. It should be also noted that, as discussed in Section 5.4.1.1, a match between an event *e* in the log and a consequence C_{ijk} (k=1,…,K) is detected only if the *Conditions 9-11* are satisfied.

However, as discussed in Section 5.4.1.1, there is also a possibility that we may not be able to confirm or disconfirm some of the expected consequences of an explanation at the time of the search. These will be consequences C_{ijk}, with time range $[t_{ijk}^{lB}, t_{ijk}^{UB}]$, for which no matching event satisfying the *Conditions 9-11* (Section 5.4.1.1) could be found

at the time of the search and the last received event from the relevant captor had a timestamp that was less than or equal to their upper time boundary (i.e., $lastTimestamp(Captor(C_{ijk})) \leq t_{ijk}^{UB}$). As in the case of the uncertainty that appears in the occurrence of an event (discussed in Section 5.4.1.1), the use of *Dempster Shafer theory of evidence* [146] (see also Section 3.4) would be sufficient to cope with the uncertainty regarding the search for evidence for the validity of the generated explanations.

5.4.3 Event Genuineness

Based on Hypotheses 2 and 3, and Definition 1, the event genuineness is defined as follows:

Definition 2: The genuineness of an event *e* is defined as:

$$\text{Genuine}(e) = \bigwedge_{e_i \in U(e)} (\text{Explainable}(e_i))$$

where,

- *U(e)* is the set of the events that are recorded in the log of the monitoring framework and can be matched with *e* according to *Conditions 9-11* or formally:

  $U(e) = \{ e_i \mid e_i \in \text{EventLog}$ **and** $\text{Captor}(e) = \text{Captor}(e_i)$ **and** $\text{mgu}(e, e_i) \neq \varnothing$ **and**

  $te^{LB} \leq te_i$ **and** $te_i \leq te^{UB} \}$

  where,

  - *EventLog* is the set of the events in the log of the monitor
  - *mgu(X,Y)* is the most general unifier of the events X and Y [94], and
  - *Captor(X)* is the captor that produced X, in case that X is a fully specified event, or it is the expected captor to produce X, in case that X is a parametric event
  - te^{LB} and te^{UB} are the lower and upper boundary of the specified time range of *e* (or is expected to occur)[4]

4 t_e^{LB} and t_e^{UB} are both equal to the timestamp t_e of e, if e is a fully specified event stored in the log of the monitor.

- *Explainable(e_i)* is a proposition denoting that the all of the explanations of event e_i are valid and is formally defined as:

$$\text{Explainable}(e_i) = \bigwedge_{\Phi j \in EXP(e_i)} \text{Valid}(\Phi_j)$$

where,

- *EXP(e_i)* is the set of the alternative explanations that can be generated for the event e_i
- *Valid(Φ_j)* is a proposition denoting that the explanation Φ_j is valid and is defined as:

$$\text{Valid}(\Phi_j) = \bigwedge_{e_q \in CONS(e_i, \Phi_j)} \text{Genuine}(e_q)$$

where,

- *CONS(e_i, Φ_j)* is the set of the expected consequences of Φ_j (see Section 5.4.2)

5.4.4 Efficiency of the Event Genuineness Assessment

In this section, the factors that might impact the efficiency of an assessment process based on Definition 2 are discussed. More specifically, due to the fact that the event genuineness is recursively defined, the number of the recorded events that are stored in the log of the monitoring infrastructure and are taken into account by the assessment process, as well as, the circles that might occur during reasoning, are considered critical for the efficiency of the assessment process. In the following, more details and optimization suggestions are provided.

5.4.4.1 Diagnosis window

An assessment process for the genuineness of events based on Definition 2 takes into account all the events recorded in the log of the monitoring framework until the time of the assessment. Although such an assessment would be more precise with respect to the recorded behaviour of the system, the adoption of this approach could be problematic regarding the efficiency of the diagnosis generation process. More specifically, the recursive definition of event genuineness would lead to an exhaustive search of the entire log of the monitor, and therefore the completion time of the diagnosis generation process would increase significantly. As discussed in Section 5.4.1.1, such completion times

might not be desirable in cases where a timely diagnosis is necessary in order to decide how to react to an S&D rule violation.

To address this potential issue, we restrict the space where the search for matching events is done by imposing time boundaries for the accepted matching events and, therefore, for evidence for the event genuineness that is to be assessed [164]. More specifically, the time period over which the genuineness of an event is assessed is defined by the absolute time range $TR = [T_{min}, T_{max}]$. This range is determined by the constant w, which is called *diagnosis window* and required in any particular monitoring setting, and the timestamp of the original event e_i, whose occurrence was used as confirming evidence for the detection of a monitoring rule violation by EVEREST, by using the formulas:

$$T_{min} = t_i - w/2, \text{ and}$$

$$T_{max} = t_i + w/2$$

It should be noted that the *diagnosis window* is set by the user of EVEREST and determines the time boundaries within which the search for evidence for events genuineness is performed. For instance, consider the events of Figure 5-11. Let the *diagnosis window* be equal to 130 sec and assume that the timestamp of the event e_2, whose occurrence was used as confirming evidence for the detection of a monitoring rule violation that is under diagnosis by EVEREST, is equal to 4500 sec. Also, assume that the genuineness of e_2 is being assessed. According to above formulas, the defined time range within which the search of evidence should be performed is $TR = [4435, 4565]$ (because Tmin = 4500 – 130/2 = 4435 and Tmax = 4500 +130/2 = 4565). Also, assume that the time range of e_1, which is the expected consequence of the explanation Φ_{21} of e_2, is computed to be equal to $TR(e_1) = [4400, 4500]$ by the *Generate_AE_consequences* algorithm (see Section 5.3.1). However, by taking into account the diagnosis time range, the diagnostic process searches for recorded events that match e_1 within the time range $TR(e_1)' = TR(e_1) \cap TR = [4400, 4500] \cap [4435, 4565] = [4435, 4500]$. Thus, any recorded event e_x, which matches to e_1, and therefore, cast positive evidence to the validity of explanation of Φ_{21} of e_2 and its timestamp is within $[4435, 4500]$, is taken into account by the assessment process.

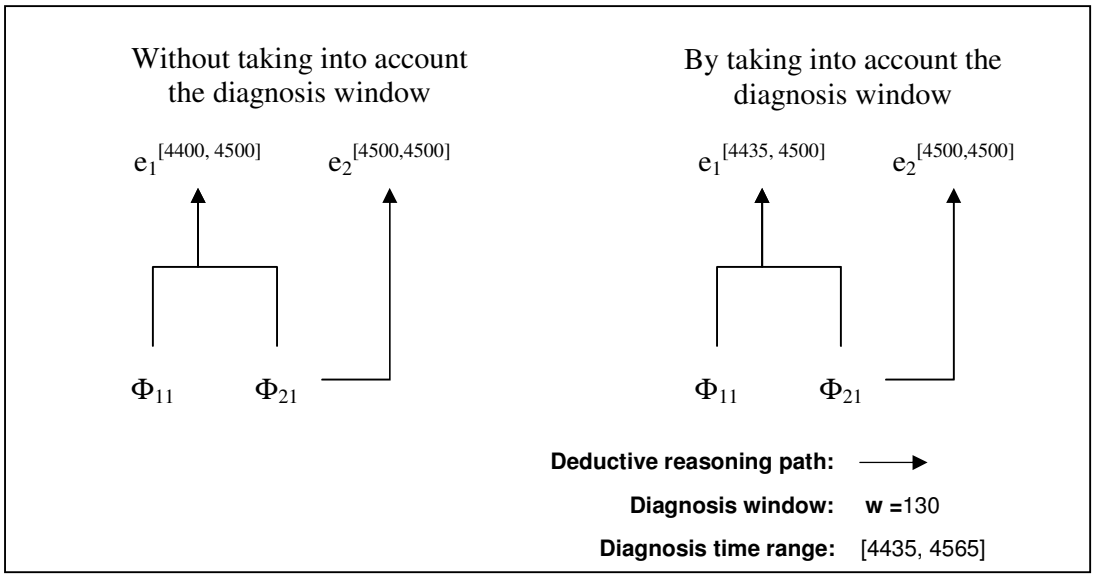

Figure 5-11 – Events and explanations

At this point, it should be also noted that nor do we impose time boundaries on neither do we exclude generated explanations due to diagnosis window. The reason we do not filter out generated explanations due to diagnosis window is that abduced events, which are members of a generated explanation and are specified partially or completely out of the diagnosis time range $TR = [T_{min}, T_{max}]$, can have consequences within the diagnosis time range according to the temporal constraints of the underlying theory, i.e., the temporal constraints of the assumptions that are used during the abductive and deductive stages of the diagnostic process. For instance, assume that we have the following assumptions:

A1: **Happens**(AE1,t1,R(t1,t1)) ⇒ **Happens**(E1,t1,R(t1+5,t1+7))

A2: **Happens**(AE1,t1,R(t1,t1)) ⇒ **Happens**(DE1,t2,R(t1,t1+8))

Also, assume that the genuineness of an event E1 occurred at t=25 is assessed, while the diagnosis window is set to w=2. Therefore the diagnosis time range is TR=[24,26]. From A1, the explanations set of E1 includes only the abduced event **Happens**(AE1,t1,R(18,20)), whose time range is specified completely out of TR as [18,20] ∩ [24,26] = ∅. From A2 the derived event **Happens**(DE1,t1,R(18,28)) can be identified as expected consequence of the above explanation. Thus, while the explanation of our example is specified out of TR, an expected consequence of this explanation is specified partially within TR, as [18,28] ∩ [24,26] = [24,26].

5.4.4.2 Circles occurrence

As any recursive process, the event genuineness assessment process of an event e_s is prone to circles occurrence. In particular, any recorded event that is reached and processed more than once, an infinite loop occurs in the assessment process of e_s. However, there is another factor that must be taken into account. In case that a recorded event e_r can cast evidence to the validity of n alternative explanations $\Phi_{i1},\ldots, \Phi_{in}$ of an event e_i, which is processed during the assessment of e_s, the evidence of e_r should be equally distributed to the aforementioned explanations. Thus, as loops must be avoided, it is also significant for the assessment process to distribute equally the evidence of any recorded event that is repeatedly reached as actual consequence of alternative explanations of the same event e_i.

To address the above issues, our diagnostic process assesses any recorded event only once, while it keeps the number of times that recorded event are reached as consequences of alternative explanations of the same event e_i. More specifically, any recorded event that is taken into account by our process is assessed only the first time that is reached, amd the result of the assessment is stored in the memory of diagnostic process. For understanding whether a recorded event has been already reached before, the diagnostic process keeps in its memory the following two lists, U_{Total} and U_o. The diagnostic process stores in the list U_{Total} the lists of matching recorded events for each event e_i, $U(e_i)$ (see Section 5.4.3), during all the recursive invocations for the assessment of e_s.

On the other hand, the list U_o contains lists for the recorded events that have been reached at least once during the assessment process of e_s. Except for the reached recorded event e_i, each sub-list of U_o contains the number of times ($occurenceTimes(e_i)$) that e_i was reached as a consequence of alternative explanations of the same event e_x, two Boolean variables to flag whether e_i has been already explained ($isExplained(e_i)$) and whether the process has assessed the explainability of event e_i ($isExplainabilityComputed(e_i)$), and two placeholders for the assessment result of the explanability of e_i ($assessmentOf(Explainable(e_i))$ and $assessmentOf(\neg Explainable(e_i))$). It should be noted that $isExplained(e_i)$ flag is updated with regards to, once the diagnostic process finds explanations for e_i, while the $isExplainabilityComputed(e_i)$ flag and the two placeholders $assessmentOf(Explainable(e_i))$ and $assessmentOf(\neg Explainable(e_i))$ are updated when the process assesses the explainability of e_i for the first time. Also, the reason we have two placeholders for the assessment result in the explainability of e_i, and how U_o is updated

with regards to these two placeholders is discussed below in Section 5.4.6. The U_o is formally specified as:

If $U_{Total} \neq \varnothing$, then $\forall$ U(e) $\in$ U_{Total} **and** $\forall$ $e_i \in$ U(e) $\exists$ $sublist_{Uo}(e_i) \in U_o$ such that:

$$sublist_{Uo}(e_i) = [e_i, occurrenceTimes(e_i), isExplained(e_i), isExplainabilityComputed(e_i),$$
$$assessmentOf(Explainable(e_i)), assessmentOf(\neg Explainable(e_i))]$$

Regarding the identification of an already considered recorded event, it should be reminded that the recorded events are fully specified events. Thus, if U_{Total} is not empty, a recorded event e_x is an already considered recorded event iff there is a recorded event e_y, which belongs to a sub-list U(e) of U_{Total}, it holds that e_y can be unified with e_x and the timestamps of e_y and e_x are equal, or formally e_x is an already considered recorded event iff:

$U_{Total} \neq \varnothing$ **and** $\exists$ $U(e_y) \in U_{Total}$ **and** $e_y \in$ EventLog **and** $mgu(e_x, e_y) \neq \varnothing$ **and** $t_x = t_y$

The means that the lists U_o and U_{Total} are populated with respect to the diagnosis window and used by the assessment process are dicussed below, in Sections 5.4.4.3 and 5.4.6, respectively. It should be noted that U_o and U_{Total} are set equal to the empty list in the beginning of the assessment of e_s.

It should be also noted that unless there was a repeated recorded event, it is unlikely that repeated generated explanations and expected consequences would occur. Before arguing on the reason why it is unlikely to have repeated explanations and consequences without processing an already processed event, it should be reminded that a generated explanation is a set of abduced parametric events, while an expected consequence is a derived parametric event. Also, as discussed in Section 5.4.4.1, the time ranges of the parametric events that consist the generated explanations or the expected consequenes are not restricted by the diagnosis window. Thus, the time ranges of generated explanations and consequences can be as wide as the time constraints of the underlying theory assumptions allow. Therefore, there are two cases that repeated explanations or consequences can occur. The first case presumes that an already considered recorded event is processed again, and therefore already considered explanations and consequences occur in the line of reasoning. However, this case is avoided by the means we have shown in the beginning of the section.

The second case presumes that two conditions should be satisfied for having repeated explanations and consequences. These two conditions are as follows:

i) The underlying theory includes at least two assumtpions A_i and A_j, whose body predicates and both body and head time constraints are the same, but their head predicates are different. Assume that p_i and p_j are the head predicates of A_i and A_j respectively.

ii) While there is an already considered recorded event e_i, i.e., belongs to U_{Total}, that can be unified with p_i, there is a recorded event e_j that respectively can be unified with p_j and has occurred at the same timepoint as e_i had occurred, i.e., their timestamps are equal $t_j=t_i$.

However, due to the fact that the recorded events timestamps are measured in msec, it is very unlikely that two different events can have equal timestamps. Thus, without harming the generality of our approach, assume that two different events cannot occur at the exact same point, and therefore their timestamps are not equal. By these means, the aforementioned condition can never be satisfied.

5.4.4.3 Handling efficiently explanations, conequences and matching recorded events

At this point, we describe how the diagnostic process handles explanations, consequences, and matching recored events by taking into account the diagnosis window and the already considered recorded events for avoiding loops. The algorithm for handling efficiently the generated explanations, the expected consequences and the matching recorded events that may appear during the genuineness assessment of an event e is a breadth-first search algorithm and is called *Preprocess*. The *Preprocess* algorithm is listed as follows:

Preprocess(toBeProcessed, TR, U_o, U_{Total}, EXP_{Total}, $CONS_{Total}$)

1. U(toBeProcessed,TR) = [ ]
2. **For** each e ∈ toBeProcessed **Do**
3. U(e,TR) = [ ]
4. **For** each e_l ∈ $EventLog^{TR(e)}$ **Do**
5. **If** mgu(e, e_l) ≠ Ø **Then**
6. **If** e_l ∉ U_{Total} **Then**
7. **If** e_l ∈ U_o **Then**
8. occurrenceTimes(e_l) = occurrenceTimes(e_l) +1
9. update(U_o, occurrenceTimes(e_l))
10. **Else**
11. occurrenceTimes(e_l) = 1
12. isExplained(e_l), isExplainabilityComputed(e_l) = False
13. append([e_l, occurrenceTimes(e_l), isExplained(e_l), isExplainabilityComputed(e_l), null, null], U_o)
14. **End If**
15. append(e_l, U(e,TR))
16. **End If**
17. **End If**
18. **End For**
19. append(U(e,TR), U(toBeProcessed,TR))
20. **End For**
21. appendAll(U(toBeProcessed,TR)), U_{Total})
22. EXP(toBeProcessed)=[ ]
23. **For** each U(e,TR) ∈ U(toBeProcessed,TR) **Do**
24. **If** U(e,TR) ≠ Ø **Then**
25. EXP_{Rel}(e)=[ ]
26. **For** each e_m ∈ U(e,TR) **Do**
27. **If** isExplained(e_m) == False **Then**
28. EXP(e_m) = Explain(e_m, t_{min}(e_m), t_{max}(e_m), f_{init})
29. append(EXP(e_m), EXP_{Rel}(e))
30. isExplained(e_m) = True
31. update(U_o, isExplained(e_m))
32. **End If**
33. **End For**
34. append(EXP_{Rel}(e), EXP(toBeProcessed))
35. append(EXP_{Rel}(e), EXP_{Total})
36. **End If**
37. **End For**
38. CONS(toBeProcessed, TR) = [ ]
39. **For** each EXP_{Rel}(e) ∈ EXP(toBeProcessed) **Do**
40. $CONS_{Rel}$(e, TR) = [ ]
41. **If** EXP_{Rel}(e) ≠ Ø **Then**
42. **For** each EXP(e_m) ∈ EXP_{Rel}(e) **Do**
43. **If** EXP(e_m) ≠ Ø **Then**
44. **For** each Φ_e ∈ EXP(e_m) **Do**
45. CONS(Φ_e) = Generate_AE_consequences(Φ_e, TLIST, CONS)
46. CONS(Φ_e, TR) = [ ]
47. **For** each C_j ∈ CONS(Φ_e) **Do**
48. TR' = TR(C_j) ∩ TR
49. **If** TR' ≠ Ø **Then**
50. TR(C_j) = TR'
51. append(C_j, CONS(Φ_e, TR))
52. **End If**
53. **End For**
54. **If** CONS(Φ_e, TR) = Ø **Then**
55. CONS(Φ_e, TR) = [C_{NULL}]
56. **End If**
57. append(CONS(Φ_e, TR), $CONS_{Rel}$(e, TR))
58. **End For**
59. **End If**
60. **End For**
61. **End If**
62. append($CONS_{Rel}$(e, TR), CONS(toBeProcessed, TR))

```
63.  append(CONS_Rel(e, TR), CONS_Total)
64.  End For
65.  toPreprocess = [ ]
66.  For each CONS_Rel(e, TR) ∈ CONS(toBeProcessed, TR) Do
67.   If CONS_Rel(e, TR) ≠ Ø Then
68.    For each CONS_e ∈ CONS_Rel(e, TR) Do
69.     If CONS_e ≠ [C_NULL] Then
70.      For each C_k ∈ CONS_e Do
71.       append (C_k, toPreprocess)
72.      End For
73.     End If
74.    End For
75.   If toPrerpocess ≠ Ø  Then
76.    Preprocess(toPreprocess, TR, U_o, U_Total, EXP_Total, CONS_Total)
77.   End If
78.  End For
79.  return(U_o)
END Preprocess
```

Figure 5-12 – Algorithm for handling efficiently explanations, consequences and matching recorded events

It should be noted that the *Preprocess* algorithm is invoked by the diagnostic process before the assessment of the genuineness belief of *e* starts. The primary objective of the algorithm is to generate a set of the recorded events, U_o, that are taken into account during the genuineness assessment of a given event *e*, as firstly introduced in Section 5.4.4.2. U_o also contains the number of times that a recorded event is reached as a consequence of the same event, as well as, a placeholder for the assessment result for each considered recorded event. On the other hand, the secondary objective is to compute all the necessary explanations, consequences and matching recorded events that may appear during the genuineness assessment of *e*. As soon as all the necessary explanations, consequences and matching recorded events are compiled by the *Preprocess* algorithm, the diagnostic process can use them for the actual genuineness assessment of e. For this reason, the algorithm is called *Preprocess*. Finally, as a convention, in the algorithmic specifications in Figure 5-12, it should be noted that the expression x ∈ y means that element x is a member of y, in case that y is a list of elements of type x, or x is a member of any sublist of elements of type x of y, in case that y is a complex construct of multilevel lists. Similarly, assume the corresponding interpretation for the expression x ∉ y.

The *Preprocess* algorithm starts by getting as input:

- a list of events. It should be noted that when the genuineness of an event e is questioned, thus the diagnostic process invokes the *Preprocess* algorithm for first time for the event e, the list *toBeProcessed* contains only the event e.
- the diagnostic time range TR, whose boundaries are computed as discussed in Section 5.4.4.1,
- the list of considered recorded events U_o, as discussed above, and
- the lists U_{Total}, EXP_{Total} and $CONS_{Total}$, which the diagnostic process uses to store the matching recorded events, the generated explanations, and the expected consequences, respectively, for each event e_i that may appear during the recursive invocations of the assessment process of e

For the given list *toBeProcessed*, the algorithm creates a new empty list *U(toBeProcessed,TR)* (see line 1 in Figure 5-12), which is used for storing the recorded events that match with the events in the list *toBeProcessed*. Thus, for each event e in list *toBeProcessed*, the algorithm creates a new empty list *U(e,TR)* (see lines 2-3 in Figure 5-12), which is used for storing the matching recorded events of e. For each event e_i that is recorded in the event log of EVEREST and is occurred within *TR(e)* (see line 4 in Figure 5-12), the algorithm checks whether e_i can be unified with e (see line 5 in Figure 5-12). If false, the algorithm disregards e_i. Otherwise, it is checked whether e_i has been already taken into account in previous recursive invocations of the algorithm, i.e., whether e_i belongs to U_{Total} list (see line 6 in Figure 5-12). If true, the algorithm again disregards e_i. Otherwise, the algorithm checks whether e_i has been already considered during the current invocaton of the algorithm, by checking whether U_o contains a sublist with regards to occeurrences of e_i. In case that e_i belongs to U_o, the algorithm updates U_o, by increasing by one the occurrence times of e_i (see lines 8-9 in Figure 5-12). On the other hand, in case that e_i has not been considered yet, the algorithm updates the already matching recorded events list U_o with the information regarding the first occurrence of e_i (see lines 11-13 in Figure 5-12). Once U_o has been updated for the current occurrence of e_i, the algorithm appends e_i in *U(e,TR)* (see line 15 in Figure 5-12). As soon as there is no other event occurred within *TR(e)* and recorded in the event log, thus, the algorithm has compiled the matching recorded event list *U(e,TR)* for e, the algorithm appends *U(e,TR)* to *U(toBeProcessed,TR)* (see line 19 in Figure 5-12). Also, when all events in

toBeProcessed have been considered, the algorithm appeands all elements of *toBeProcessed* to U_{Total} (see line 21 in Figure 5-12).

Having compiled the list *U(toBeProcessed,TR)*, the algorithm focuses on the explanations that are related to the events of *U(toBeProcessed,TR)*. Thus, the algorithm creates a new empty list, *EXP(toBeProcessed)*, for storing the aforementioned explanations (see line 22 in Figure 5-12). More specifically, for each non empty *U(e,TR)* in *U(toBeProcessed,TR)*, the algorithm creates a new empty list, $EXP_{Rel}(e)$, for storing relevant explanations of *e* (see lines 23-25 in Figure 5-12). As relevant explanations to *e*, the algorithm considers the explanations, which can be generated for each matching event of *e*, e_m, only if e_m has not been already explained during the current recursive invocation, i.e., *isExplained*(e_m) *is* currently *False* (see lines 26-27 in Figure 5-12). Consequently, the algorithm generates an explanation list for each e_m in *U(e,TR)*, $EXP(e_m)$, by invoking the *Explain* algorithm (discussed in Section 5.2.1) for each e_m, and it appends $EXP(e_m)$ to $EXP_{Rel}(e)$, while updates U_o that e_m has now been explained (see lines 28-31 in Figure 5-12). Finally, when all matching recorded events relevant to *e* have been considered, the algorithm appends $EXP_{Rel}(e)$ to *EXP(toBeProcessed)* and EXP_{Total} (see line 34-35 in Figure 5-12).

The algorithm resumes by compiling all the expected consequences that are related to any event *e* in the list *toBeProcessed*. Therefore, a new empty list *CONS(toBeProcessed, TR)* is created (see line 38 in Figure 5-12). Thus, for each $EXP_{Rel}(e)$ in *EXP(toBeProcessed)*, the algorithm continues by creating a new empty list $CONS_{Rel}(e, TR)$ for storing the consequences of the explanations contained in $EXP_{Rel}(e)$ (see lines 39-40 in Figure 5-12). As discussed below in this paragraph, the algorithm focuses only on consequences of the explanations of the matching events of *e*, which are identified within the diagnosis time range *TR*, while any other consequence is disregarded. Thus, if $EXP_{Rel}(e)$ is not empty, for each non empty explanation list $EXP(e_m)$ in $EXP_{Rel}(e)$, and for each explanation Φ_e in $EXP(e_m)$, the algorithm identifies consequences for Φ_e by invoking the *Generate_AE_consequences* algorithm and stores them in a consequences list $CONS(\Phi_e)$ (see lines 41-45 in Figure 5-12). However, due to the fact that our focus is only on consequences, which are identified within the diagnosis time range *TR*, and because the *Generate_AE_consequences* algorithm identifies consequences without taking into account *TR*, the *Preprocess* algorithm transforms the list $CONS(\Phi_e)$ into the list $CONS(\Phi_e, TR)$, which contains only consequences with respect to *TR* (see lines 46-52

in Figure 5-12). Each new $CONS(\Phi_e, TR)$ is checked whether is empty. If true, the algorithm sets $CONS(\Phi_e, TR)$ equal to a list $[C_{NULL}]$ that contains only the item C_{NULL} (see lines 54-56 in Figure 5-12). C_{NULL} is a special event, which denotes that no consequences can be identified. Once the last check is finished, the algorithm appends $CONS(\Phi_e, TR)$ to $CONS_{Rel}(e, TR)$ (see line 57 in Figure 5-12). As soon as all explanations of matching recorded events of *e* have been considered, the algorithm appends $CONS_{Rel}(e, TR)$ to $CONS(toBeProcessed, TR)$ and $CONS_{Total}$ (see lines 62-63 in Figure 5-12).

Having obtained the list $CONS(toBeProcessed, TR)$ by compiling the lists $CONS_{Rel}(e, TR)$, which contain the consequences of the explanations of the matching recorded events of any *e* in *toBeProcessed* list with respect to TR, the *Preprocess* algorithm invokes itself for a new list, called *toPreprocess* (see line 65 in Figure 5-12). The *toPreprocess* list contains all the identified consequences stored in the lists $CONS(e, TR)$. In particular, for each non empty $CONS_{Rel}(e, TR)$ (see lines 66-67 in Figure 5-12), the algorithm goes through each consequences lists $CONS_e$ of $CONS_{Rel}(e, TR)$ (see line 68 in Figure 5-12). Then, if $CONS_e$ is not equal to $[C_{NULL}]$, for each consequence of $CONS_e$, C_k, the algorithm appends C_k to *toPreprocess* (see lines 69-73 in Figure 5-12). When all individual consequences, related to *e,* have been taken into account, and if *toPreprocess* list is not empty, the algorithm invokes itself (see lines 75-77 in Figure 5-12). Finally, as soon as no other recursive invocations can be made, the algorithm returns U_o (see line 79 in Figure 5-12), by having also compiled the lists U_{Total}, EXP_{Total}, and $CONS_{Total}$.

5.4.4.3.1 Example of handling efficiently explanations, conequences and matching recorded events

As an example of handling efficiently explanations, consequences and matching recorded events by using the *Preprocess* algorithm (Figure 5-12) consider the violation of rule ATMS.R1 (see Section 4.3). For sake of compactness, the following indicative example of the Preprocess algorithm is not based on the set of ATMS assumptions, which was firstly introduced in Section 4.3 due to the extended number of assumtpions. Instead, the example is based on a more compact set of assumptions. The compact ATMS assumptions set consists of the following assumptions:

ATMS.A1'. **Initially**(covers(R1,S1),t0)

ATMS.A2'. **Initially**(covers(R2,S1),t0)

ATMS.A3'. **Initially**(landing_airspace_for(S1,AR-a),t0)

ATMS.A4'. ∀t1∈Time, ∃t2∈Time, ∀_sender1, ∀_receiver2, ∀_source2,
∀_a∈Airplanes, ∀_s∈Airspaces, ∃_r∈Radars.
Happens(e(_id1,_sender1,_receiver2,RES-A,inspace(_a,_s),
_source2),t1,R(t1,t1)) ∧
HoldsAt(covers(_r,_s),t1) ⇒
Happens(e(_id2,_r,_receiver2,RES-A,signal(_r,_a,_s),
_source2),t2,R(t1,t1+5))

ATMS.A5'. ∀t1∈Time, ∃t2∈Time, ∀_sender1, ∀_receiver2, ∀_source2,
∀_a∈Airplanes, ∀_s∈Airspaces.
Happens(e(_id1,_sender1,_receiver2,RES-A,inspace(_a,_s),
_source2),t1,R(t1,t1)) ⇒
Happens(e(_id2,_a,_receiver2,RES-A,permissionRequest(_a,
_s),_source2),t2,R(t1-20,t1-1))

ATMS.A6'. ∀t1∈Time, ∃t2∈Time, ∀_sender1, ∀_receiver2, ∀_source2,
∀_a∈Airplanes, ∀_s∈Airspaces, ∃_r∈Radars.
Happens(e(_id1,_a,_receiver2,RES-A,landingRequest(_a,
_airportX),_source2),t1,R(t1,t1)) ∧
HoldsAt(landing_airspace_for(_s,_airportX),t1) ∧
HoldsAt(covers(_r,_s),t1) ⇒
Happens(e(_id2,_r,_receiver2,RES-A,signal(_r,_a,_s),
_source2),t2,R(t1,t1+5))

ATMS.A7'. ∀t1∈Time, ∃t2∈Time, ∀_sender1, ∀_receiver2, ∀_source2,
∀_a∈Airplanes, ∀_s∈Airspaces, ∃_r∈Radars.
Happens(e(_id1,_a,_receiver2,RES-A,landingRequest(_a,
_airportX),_source2),t1,R(t1,t1)) ∧
HoldsAt(landing_airspace_for(_s,_airportX),t1) ⇒
Happens(e(_id2,_a,_receiver2,RES-A,permissionRequest(_a,
_s),_source2),t2,R(t1-10,t1-2))

In particular, the first four assumptions, ATMS.A1', ATMS.A2', ATMS.A4', and ATMS.A5', are the same as the assumptions, ATMS.A1, ATMS.A2, ATMS.A3, and ATMS.A4 respectively, which have been already discussed in Section 4.3. Assumption ATMS.A3' specifies that the landing airspace for airport AR_a is airspace S1 since the start of the execution of ATMS. Assumption ATMS.A6' states that if an airplane _a requests a permission to land at airport _airportX at some timepoint t1, and it holds that the landing airspace for_airportX is airspace _s and radar _r covers airspace _s at t1, then it is expected that there should be a signal from _r notifying that _a moves in _s at some

time point t2 within t1 and 5 time units after t1. Similarly, assumption ATMS.A7' specifies that if an airplane _a requests a permission to land at airport _airportX at some timepoint t1, and it holds that the landing airspace for_airportX is _s at t1, then it was expected that _a has requested permission for entering _s at some time point t2 within 10 and 2 time units before t1. In terms of the predicate sets APreds', DPreds' and OPreds', which are defined in Section 4.1, the membership of the predicates of the above compact ATMS theory is as follows:

APreds' = {**Happens**(e(_id,_r,_receiver,RES-A,inspace(_a,_s),
_source2),t,R(t,t)),
Happens(e(_id1,_a,_receiver2,RES-A,landingRequest(_a,
_airportX),_source2),t,R(t,t))
}

Dpreds' = { **HoldsAt**(covers(_r,_s),t),
HoldsAt(landing_airspace_for(_s,_airportX),t)
}

OPreds' = { **Initially**(covers(R1,S1),t0), **Initially**(covers(R2,S1),t0),
Initially(landing_airspace_for(S1,AR-a),t0),
Happens(e(_id,_r,_receiver,RES-A,signal(_r, _a, _s),
_source),t,R(t,t)),
Happens(e(id,_a,_receiver,RES-A,permissionRequest(_a,
_s),_source),t,R(t,t)),
Happens(e(_id,_a,_receiver,RES-A,
landingRequest(_a,_airportX),_source),t,R(t,t))
}

Suppose also that the following events have taken place and been received by EVEREST at time t=25 when a request for diagnosing the violation of ATMS.R1, which has been caused by the events *Happens(e(E7,R1,AirBase,RES-A,signal(R1,A1,S1),R1Captor),17,R(17,17))* (referred as E6 henceforth) and *Happens(e(NF,R2,AirBase,signal(R2,A1,S1),R2Captor),t,R(17,22))*, is requested:

Event Log for ATMS:

```
E1 Happens(e(E1,R2,AirBase,RES-A,signal(R2,A2,S2),R2Captor),1,
            R(1,1))
E2 Happens(e(E2,R2,AirBase,RES-A,signal(R2,A2,S2),R2Captor),5,
            R(5,5))
E3 Happens(e(E3,R2,AirBase,RES-A,signal(R2,A2,S2),R2Captor),8,
            R(8,8))
E4 Happens(e(E4,A1,AirBase,RES-A,permissionRequest(A1,S1),
             AirBaseCaptor),10,R(10,10))
E5 Happens(e(E5,A1,AirBase,RES-A,landingRequest(A1,AR-a),
             AirBaseCaptor),13,R(13,13))
E6 Happens(e(E7,R1,AirBase,RES-A,signal(R1,A1,S1),R1Captor),17,
            R(17,17))
E7 Happens(e(E8,R2,AirBase,RES-A,signal(R2,A5,S1),R2Captor),23,
            R(23,23))
```

Figure 5-13 – Event log for ATMS

Assuming that the genuineness of E6 is being assessed, and therefore the *Preprocess* algorithm is invoked for E6, let the *diagnosis window*, *w*, be equal to 26. Thus, due to the fact that the event E7 (Figure 5-13) was used as the confirming evidence for the detection of ATMS.R1 violation by EVEREST, the *diagnosis time range*, *TR*, is computed as follows:

$TR = [t_{E7} - w/2, t_{E7} + w/2] = [23\text{-}13, 13+13] = [10, 36]$

Initial invocation: Phase of seach for matching recorded events

For the initial invocation of the *Preprocess* algorithm, while the lists U_o, U_{Total}, EXP_{Total}, and $CONS_{Total}$ are empty, the algorithm processes the list $toBeProcessed_o$ = [E6]. Figure 5-14 illustrates the generated explanations/consequences tree that the initial invocation of *Preprocess* algorithm generated for event E6.

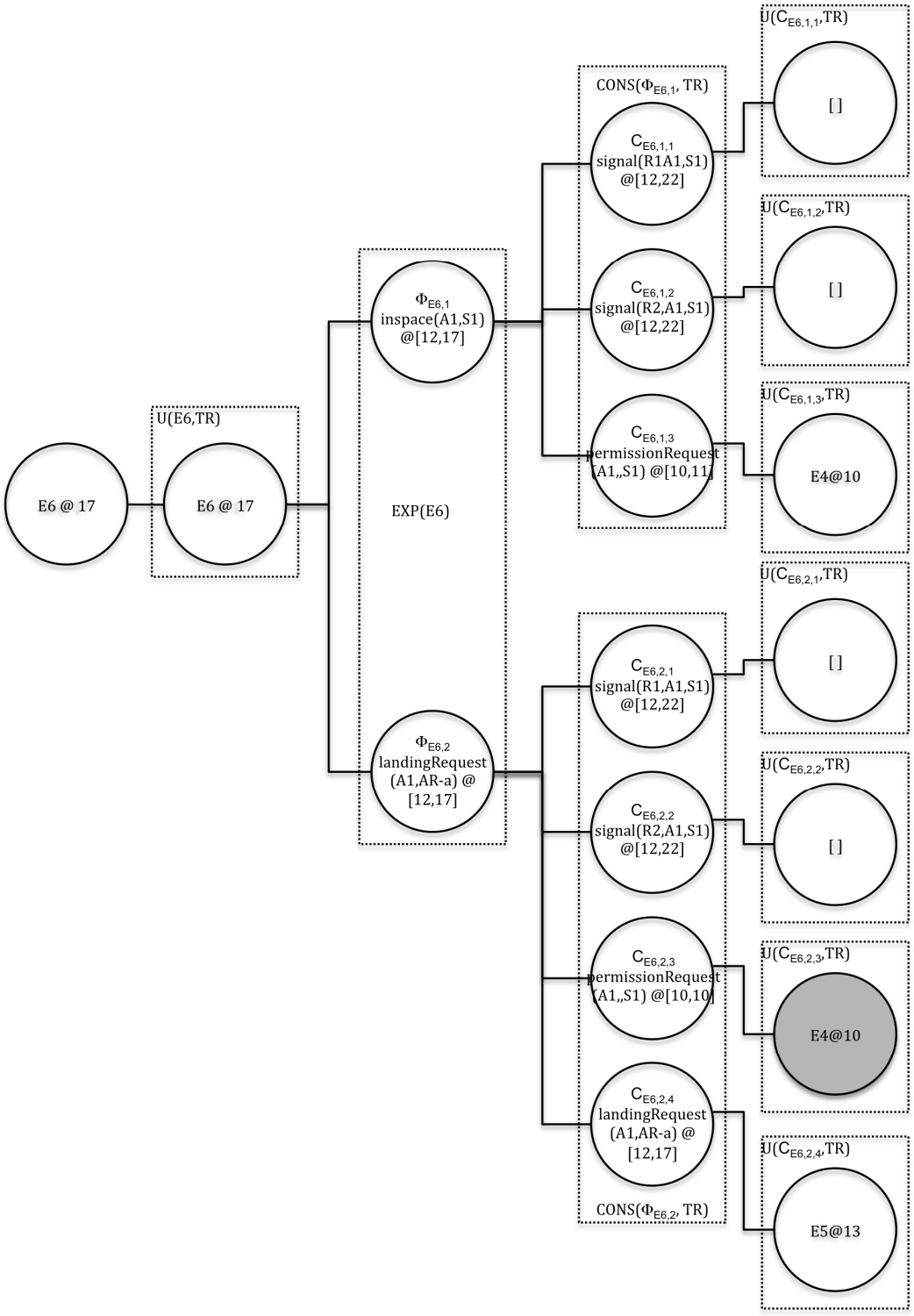

Figure 5-14 – Explanations/Consequences tree for event E6

The algorithm starts by compiling the list *U(toBeProcessed$_o$,TR)*. For populating the aforementioned list, the algorithm compiles the list *U(E6,TR)*. Once all matching recoded events for E6 are found by searching the event log, illustrated in Figure 5-13, with respect

to the diagnosis time range and to already considered recorded events in the current and previous recursive invocations, the algorithm updates the lists U_o, $U(toBeProcessed_o,TR)$, and U_{Total}. The aforementioned lists are currently compiled as follows:

U(E6,TR) = [E6]

U_o = [[E6,1,null]]

$U(toBeProcessed_o,TR)$ = [[U(E6,TR) : E6]]

U_{Total} = [[U(E6,TR) : E6]]

Initial invocation: Phase of explanation generation

The algorithm resumes by compling the $EXP(toBeProcessed_o)$. For compiling the aforementioned list, the algorithm searches for explanations for each $U(e,TR)$ in $U(toBeProcessed_o,TR)$. At this point, there is only $U(E6,TR)$ in $U(toBeProcessed_o,TR)$, as well as, $U(E6,TR)$ contains only E6. Thus, the algorithm compiles the lists $EXP(E6)$ and $EXP_{Rel}(E6)$ after the invocation of the *Explain* algorithm for each matching recorded event of the given event E6, and updates $EXP(toBeProcessed_o)$ and EXP_{Total} as illustrated in Figure 5-14 and as follows:

$EXP(E6) = [[\Phi_{E6,1}]_{AND}, [\Phi_{E6,2}]_{AND}]_{OR}]$,

where

$\Phi_{E6,1}$: `Happens(e(ABD,R1,AirBase,RES-A,inspace(A1,S1),AirBaseCaptor), t1,R(12,17))`

$\Phi_{E6,2}$: `Happens(e(ABD,R1,AirBase,RES-A,landingRequest(A1,AR-a), AirBaseCaptor),t1,R(12,17))`

$EXP_{Rel}(E6) = [EXP(E6): [[\Phi_{E6,1}]_{AND}, [\Phi_{E6,2}]_{AND}]_{OR}]]$

$EXP(toBeProcessed_o) = [\ EXP_{Rel}(E6) : [\ EXP(E6): [[\Phi_{E6,1}]_{AND}, [\Phi_{E6,2}]_{AND}]_{OR}]\ ]\ ]$

$EXP_{Total} = [\ EXP_{Rel}(E6) : [\ EXP(E6): [[\Phi_{E6,1}]_{AND}, [\Phi_{E6,2}]_{AND}]_{OR}]\ ]\ ]$

Initial invocation: Phase of consequence identification

After the $EXP(toBeProcessed_o)$ is populated, the algorithm resumes by compiling the list $CONS(toBeProcessed_o,TR)$, which should contain all the expected consequences of all the alternative explanations of E6 with respect to the diagnosis window. For populating the aforementioned list, the algorithm constructs the list $CONS_{Rel}(E6,TR)$. The algorithm populates the list $CONS_{Rel}(E6,TR)$ by compiling the list $CONS(E6,TR)$, which contains the the identified consequences of the only matching event for E6. The expected consequences $CONS(\Phi_{E6,1})$ and $CONS(\Phi_{E6,2})$ of the alternative explanations of E6, $\Phi_{E6,1}$ and $\Phi_{E6,2}$ respectively, are identified with two invocations of the *Generate_AE_consequences* algorithm for each of alternative explanations. The *Preprocess* algorithm, then, transforms the lists $CONS(\Phi_{E6,1})$ and $CONS(\Phi_{E6,2})$ to $CONS(\Phi_{E6,1},TR)$ and $CONS(\Phi_{E6,2},TR)$ respectively, by disregarding any individual consequence that is identified out of the diagnosis time range *TR*. All the aforementioned lists, along with the the lists $CONS(toBeProcessed_o,TR)$ and $CONS_{Total}$ are computed as follows (see also Figure 5-14):

$CONS(\Phi_{E6,1}) = [C_{E6,1,1}, C_{E6,1,2}, C_{E6,1,3}]$,

where

$C_{E6,1,1}$: `Happens(e(DER,R1,AirBase,RES-A,signal(R1,A1,S1),R1Captor),t1, R(12,22))`

$C_{E6,1,2}$: `Happens(e(DER,R2,AirBase,RES-A,signal(R2,A1,S1),R2Captor),t1, R(12,22))`

$C_{E6,1,3}$: `Happens(e(DER,A1,AirBase,RES-A,permissionRequest(A1,S1), AirBaseCaptor),t1,R(0,11))`

$CONS(\Phi_{E6,1},TR) = [C'_{E6,1,1}, C'_{E6,1,2}, C'_{E6,1,3}]$,

where

$C'_{E6,1,1}$: `Happens(e(DER,R1,AirBase,RES-A,signal(R1,A1,S1),R1Captor),t1, R(12,22))`

$C'_{E6,1,2}$: `Happens(e(DER,R2,AirBase,RES-A,signal(R2,A1,S1),R2Captor),t1, R(12,22))`

$C'_{E6,1,3}$: `Happens(e(DER,A1,AirBase,RES-A,permissionRequest(A1,S1),`

AirBaseCaptor),t1,R(10,11))

$CONS(\Phi_{E6,2}) = [C_{E6,2,1}, C_{E6,2,2}, C_{E6,2,3}, C_{E6,2,4}]$,

where

$C_{E6,2,1}$: Happens(e(DER,R1,AirBase,RES-A,signal(R1,A1,S1),R1Captor),t1,
R(12,22))

$C_{E6,2,2}$: Happens(e(DER,R2,AirBase,RES-A,signal(R2,A1,S1),R2Captor),t1,
R(12,22))

$C_{E6,2,3}$: Happens(e(DER,A1,AirBase,RES-A,permissionRequest(A1,S1),
AirBaseCaptor),t1,R(2,10))

$C_{E6,2,4}$: Happens(e(ABD,R1,AirBase,RES-A,landingRequest(A1,AR-a),
AirBaseCaptor),t1,R(12,17))[5]

$CONS(\Phi_{E6,2},TR) = [C'_{E6,2,1}, C'_{E6,2,2}, C'_{E6,2,3}, C'_{E6,2,4}]$,

where

$C'_{E6,2,1}$: Happens(e(DER,R1,AirBase,RES-A,signal(R1,A1,S1),R1Captor),t1,
R(12,22))

$C'_{E6,2,2}$: Happens(e(DER,R2,AirBase,RES-A,signal(R2,A1,S1),R2Captor),t1,
R(12,22))

$C'_{E6,2,3}$: Happens(e(DER,A1,AirBase,RES-A,permissionRequest(A1,S1),
AirBaseCaptor),t1,R(10,10))

$C'_{E6,2,4}$: Happens(e(ABD,R1,AirBase,RES-A,landingRequest(A1,AR-a),
AirBaseCaptor),t1,R(12,17))

$CONS(E6,TR) = [\ [CONS(\Phi_{E6,1},TR): C'_{E6,1,1}, C'_{E6,1,2}, C'_{E6,1,3}],$

$[CONS(\Phi_{E6,1},TR): C'_{E6,2,1}, C'_{E6,2,2}, C'_{E6,2,3}, C'_{E6,2,4}]\ ]$

$CONS_{Rel}(E6,TR) = [CONS(E6,TR)]$

$CONS(toBeProcessed_o,TR) = [CONS_{Rel}(E6,TR)\ ]$

---

5 It should be noted that $\Phi_{E6,2}$ is included as a consequence of itself ($C_{E6,2,4}$), as the landingRequest events are abducible and observable, i.e., they belong to the intersection of *APreds* and *OPreds*.

$CONS_{Total}$ = [$CONS_{Rel}$(E6,TR)]

Initial invocation: Phase of self-invocation preparation

Once the list *CONS(toBeProcessed$_o$,TR)* is compiled, the algorithm prepares the list *toPreprocess$_o$* for invoking itself. Recall that the *toPreprocess$_o$* list should contain all the expected consequences of the alternative explanations of all the matching events of E6, which are not equall to C_{NULL}. At this point, there is no C_{NULL} identified, thus the *toPreprocess$_o$* list is compiled as follows as follows:

$toPreprocess_o$ = [$C'_{E6,1,1}$, $C'_{E6,1,2}$, $C'_{E6,1,3}$, $C'_{E6,2,1}$, $C'_{E6,2,2}$, $C'_{E6,2,3}$, $C'_{E6,2,4}$]

1st self-invocation: Phase of seach for matching recorded events

The algorithm compiles the *U(toBeProcessed$_1$,TR)* list by processing each event in *toPreprocess$_o$* list with respect to the already considered events in the initial invocation. As illustrated in Figure 5-14, we have for:

$C'_{E6,1,1}$:

$U(C'_{E6,1,1},TR)$ = [], as E6 that is the only recorded event and can match with $C'_{E6,1,1}$ has been already considered in the previous invocation, i.e., E6 is already in U_{Total}. Thus, U_o is not updated, while *U(toBeProcessed$_o$,TR)* becomes:

$U(toBeProcessed_1,TR)$ = [[$U(C'_{E6,1,1},TR)$:]]

$C'_{E6,1,2}$:

$U(C'_{E6,1,2},TR)$ = []

Thus, U_o and *U(toBeProcessed$_o$,TR)* become:

U_o = [[E6,1,null]]

$U(toBeProcessed_1,TR)$ = [[$U(C'_{E6,1,1},TR)$:], [$U(C'_{E6,1,2},TR)$:]]

$C'_{E6,1,3}$:

$U(C'_{E6,1,3},TR)$ = [E4]

Thus, U_o and $U(toBeProcessed_o,TR)$ become:

U_o = [[E6,1,null], [E4,1,null]]

$U(toBeProcessed_1,TR)$ = [[$U(C'_{E6,1,1},TR)$:], [$U(C'_{E6,1,2},TR)$:],

[$U(C'_{E6,1,3},TR)$: E4]]

$C'_{E6,2,1}$:

$U(C'_{E6,2,1},TR)$ = [], as E6 that is the only recorded event and can match with $C'_{E6,2,1}$ has been already considered in the previous invocation, i.e., E6 is already in U_{Total}. Thus, U_o is not updated, while $U(toBeProcessed_o,TR)$ becomes:

$U(toBeProcessed_1,TR)$ = [[$U(C'_{E6,1,1},TR)$:], [$U(C'_{E6,1,2},TR)$:],

[$U(C'_{E6,1,3},TR)$: E4], [$U(C'_{E6,2,1},TR)$:]]

$C'_{E6,2,2}$:

$U(C'_{E6,2,2},TR)$ = [].

U_o = [[E6,1,null], [E4,1,null]]

$U(toBeProcessed_1,TR)$ = [[$U(C'_{E6,1,1},TR)$:], [$U(C'_{E6,1,2},TR)$:],

[$U(C'_{E6,1,3},TR)$: E4], [$U(C'_{E6,2,1},TR)$:],

[$U(C'_{E6,2,2},TR)$:]]

$C'_{E6,2,3}$:

$U(C'_{E7,1,3},TR)$ = [E4]

At this point, it should be noted that although E4 can match $C'_{E6,1,3}$, the algorithm takes into account E4 as a matching recorded event of $C'_{E6,2,3}$ as well, due to the fact that both consequences are processed in the same recursive invocation. In other words, the permission request from airplane A1 (denoted by E4) can cast evidence in the validity of both alternaive explanations that state that the airplane A1 moves in airspace S1 within 12 and 17 (denoted by $\Phi_{E6,1}$) or the airplane A1 has requested permission to land at

airport AR-a again within 12 and 17 (denoted by $\Phi_{E6,2}$). Thus, the algorithm updates the occurrence times variable of E4 in list U_o, while $U(toBeProcessed_o,TR)$ becomes:

U_o = [[E6,1,null], [E4,2,null]]

$U(toBeProcessed_1,TR)$ = [[$U(C'_{E6,1,1},TR)$:], [$U(C'_{E6,1,2},TR)$:],

[$U(C'_{E6,1,3},TR)$: E4], [$U(C'_{E6,2,1},TR)$:],

[$U(C'_{E6,2,2},TR)$:], [$U(C'_{E6,2,3},TR)$: E4]]

Finally, for $C'_{E6,2,4}$, we have:

$U(C'_{E6,2,4},TR)$ = [E5]

Thus, U_o and $U(toBeProcessed_o,TR)$ become:

U_o = [[E6,1,null], [E4,2,null], [E5,1,null]]

$U(toBeProcessed_1,TR)$ = [[$U(C'_{E6,1,1},TR)$:], [$U(C'_{E6,1,2},TR)$:],

[$U(C'_{E6,1,3},TR)$: E4], [$U(C'_{E6,2,1},TR)$:],

[$U(C'_{E6,2,2},TR)$:], [$U(C'_{E6,2,3},TR)$: E4],

[$U(C'_{E6,2,4},TR)$: E5]]

As soon as, matching recorded events for all in toBeProcessed1 are found, the algorithm appends all elements of $U(toBeProcessed_1,TR)$ to U_{Total}. Thus, U_{Total} becomes:

U_{Total} = [[U(E6,TR) : E6], [$U(C'_{E6,1,1},TR)$:], [$U(C'_{E6,1,2},TR)$:],

[$U(C'_{E6,1,3},TR)$: E4], [$U(C'_{E6,2,1},TR)$:], [$U(C'_{E6,2,2},TR)$:],

[$U(C'_{E6,2,3},TR)$: E4], [$U(C'_{E6,2,4},TR)$: E5]]

Invocating similarly the *Prerpocess* algorithm for event E4 which is considered as the only matching event of consequence $C'_{E6,2,3}$, the explanations/consequences tree pictured in Figure 5-15 is generated.

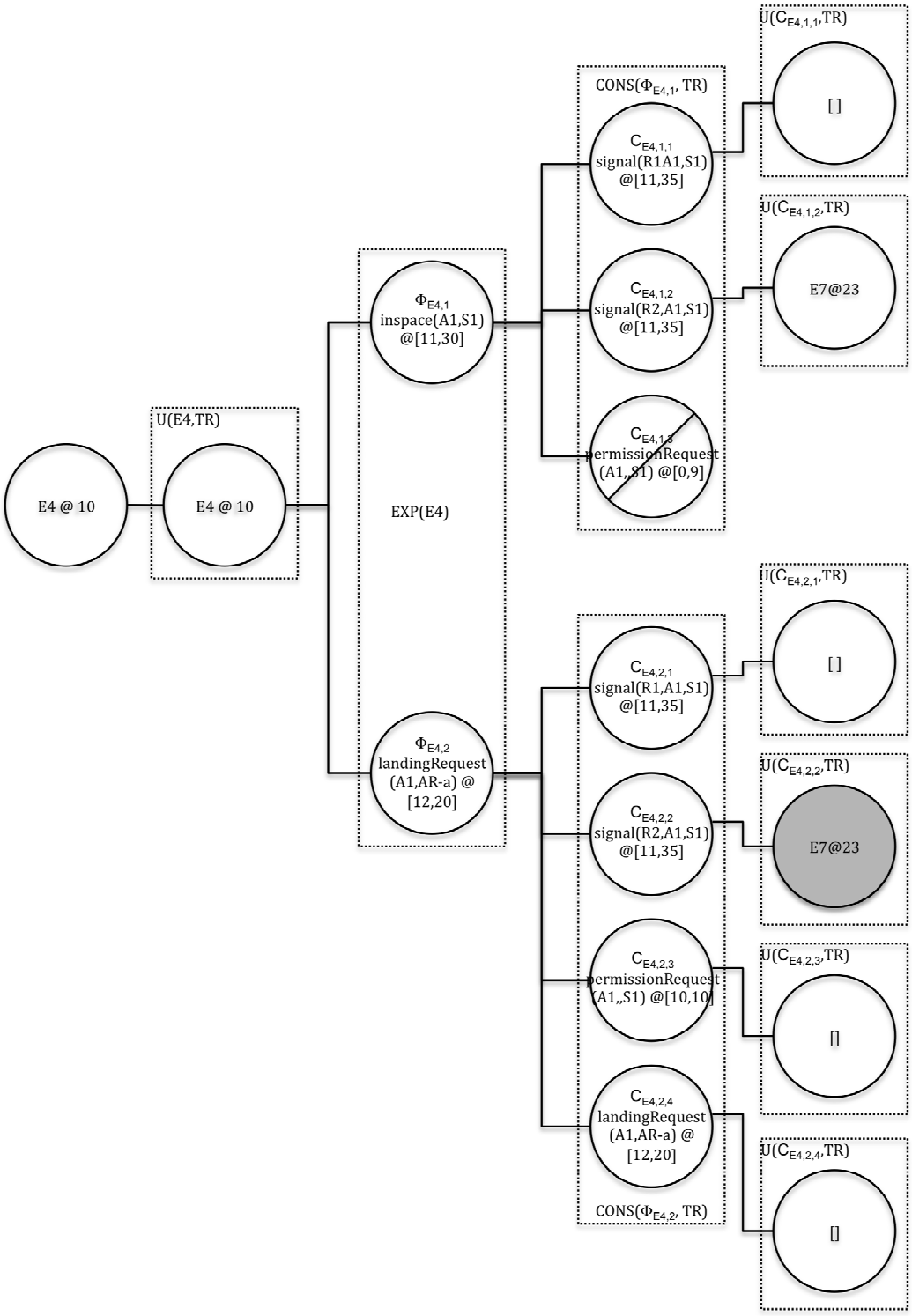

Figure 5-15 – Explanations/Consequences tree for event E4 considered as matching event of consequence $C'_{E6,2,3}$

Similarly, **Figure 5-16** illustrates the explanations/consequences tree for event E7 considered as matching event of consequence $C'_{E4,2,3}$.

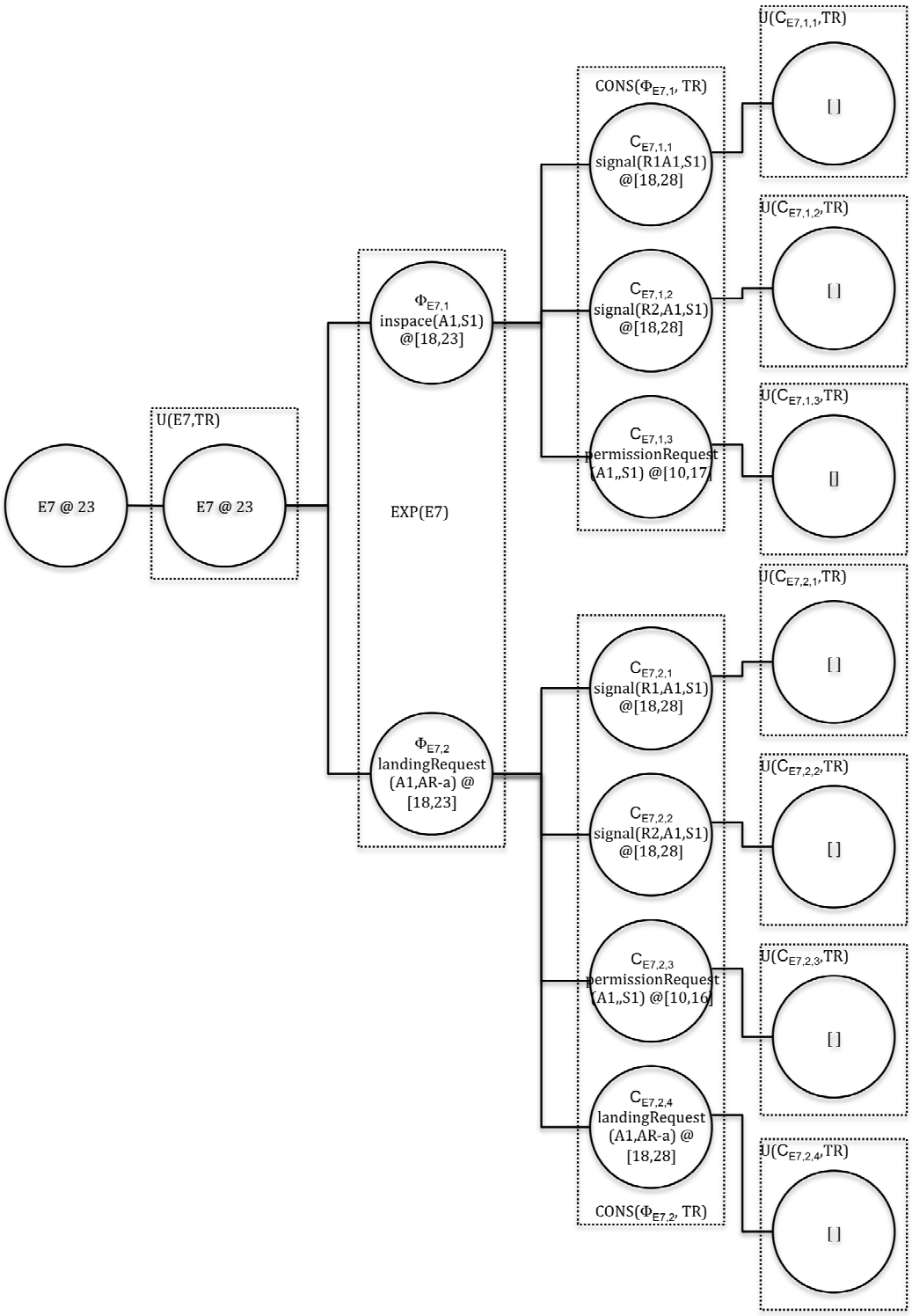

Figure 5-16 - Explanations/Consequences tree for event E7 considered as matching event of consequence $C'_{E4,2,3}$

By resuming the algorithm, we can observe that U_o does not change furthermore. For this reason, and due to the fact that, although we have taken into consideration a compact ATMS theory and the small event set of Figure 5-13, the example over-expands, we provide you the final list U_o, and the initial parts of the final lists U_{Total}, EXP_{Total}, and $CONS_{Total}$, as generated for E6 by the *Preprocess* algorithm. The output lists are as follows:

U_o = [[E7,1,null], [E6,2,null], [E4,2,null], [E5,1,null]]

U_{Total} = [[U(E7,TR) : E7], [U(C'$_{E7,1,1}$,TR) :], [U(C'$_{E7,1,2}$,TR) : E6],

[U(C'$_{E7,1,3}$,TR) : E4], [U(C'$_{E7,2,1}$,TR) :], [U(C'$_{E7,2,2}$,TR) : E6],

[U(C'$_{E7,2,3}$,TR) : E4], [U(C'$_{E7,2,4}$,TR) : E5],…]

EXP_{Total} = [[EXP_{Rel}(E7) : [EXP(E7): [[$\Phi_{E7,1}$]$_{AND}$, [$\Phi_{E7,2}$]$_{AND}$]$_{OR}$]],…]

$CONS_{Total}$ = [[[CONS($\Phi_{E7,1}$,TR): C'$_{E7,1,1}$, C'$_{E7,1,2}$, C'$_{E7,1,3}$],

[CONS($\Phi_{E7,1}$,TR): C'$_{E7,2,1}$, C'$_{E7,2,2}$, C'$_{E7,2,3}$, C'$_{E7,2,4}$]],…]

5.4.5 Reconsideration of Event Genuineness Formal Definition

Based on the remarks regarding the efficiency of the event genuineness assessment discussed in the previous section, the definition of event genuineness is reconsidered by taking into account the necessity of introducing a window to restrict the search space of the event genuineness assessment and excluding already considered recorded events. Therefore, the reconsidered version of the event genuineness definition is based on the set U_o, U_{Total}, EXP_{Total}, and $CONS_{Total}$ as they are introducesd in Section 5.4.4.3 and is given as follows:

Definition 3: The genuineness of an event *e* given the sets of already considered matching recorded events, U_o, and a diagnostic time range of interest $TR = [T_{min}, T_{max}]$ is defined as:

$$Genuine(e,U_o,TR) = \bigvee_{e_i \in U(e,TR)}(Explainable(e_i,U_o,TR))$$

where,

- *U(e,TR)* is an element of U_{Total}, and contains events that are recorded in the log of the monitoring framework with respect to *TR*, and can be unified with *e* (see also Sections 5.4.4.2 and 5.4.4.3), or formally:

$U(e,TR) \in U_{Total}$, and

$U(e,TR) = \{ e_i \mid e_i \in EventLog^{TR}$ **and** $Captor(e) = Captor(e_i)$ **and** $mgu(e, e_i) \neq \varnothing$

and $te^{LB} \leq te_i$ **and** $te_i \leq te^{UB} \}$

where,

- *$EventLog^{TR}$* is the part of monitor log that contains recorded events whose timestamps are within *TR*, or formally:

  $\forall x \in EventLog^{TR}, t_x \in TR$

- *Captor(X)* is the captor that produced X, in case that X is a fully specified event, or it is the expected captor to produce X, in case that X is a parametric event

- *mgu(X,Y)* is the most general unifier of the events X and Y [94]

- t_x is the timestamp of the event x

- t_x^{UB} and t_x^{LB} is the upper and lower boundary of the time range *TR(x)* that event x is specified within

- U_{Total} is the list that the diagnostic process uses to store the matching recorded events for each event *x* that may appear during the recursive invocations of the assessment process of *y* and is compiled by the *Preprocess* algorithm (as discussed in Section 5.4.4.3)

- *Explainable(e_i, U_o, TR)* is a proposition denoting that the event e_j has at least one valid explanation. The proposition is formally defined as:

  $Explainable(e_i, U_o, TR) = \bigvee_{\Phi j \in EXP(e_i)} Valid(e_i, \Phi_j, U_o, TR)$

  where,

  - *$EXP(e_i)$* is an element of *EXP_{Total}*, and contains the alternative explanations that can be generated for the event *e_i* (see also Sections 5.4.2 and 5.4.4.3)

  - *EXP_{Total}* is the list which the diagnostic process uses to store the generated explanations for each event *x* that may appear during the recursive

invocations of the assessment process of *y*, and is compiled by the *Preprocess* algorithm (as discussed in Section 5.4.4.3)

- *Valid(e_i,Φ_j, U_o, TR)* is a proposition denoting that explanation Φ_j of event e_i is valid within the time range of interest $TR = [T_{min},T_{max}]$. This proposition is defined as:

  $$Valid(e_i,\Phi_j,U_o,TR) = \bigvee_{e_q \in CONS(\Phi j,TR)} Genuine(e_q,U_o,TR)$$

  where,

  - *CONS(Φ_j,TR)* is is an element of $CONS_{Total}$, and contains the expected consequences of the explanation Φ_j, which occurred within $TR = [T_{min},T_{max}]$ (see also Section 5.4.4.3)
  - $CONS_{Total}$ is the list which the diagnostic process uses to store the expected consequences of all generated explanations for each event *x* that may appear during the recursive invocations of the assessment process of *y*, and is compiled by the *Preprocess* algorithm (as discussed in Section 5.4.4.3)

It should be noted that even though our hypotheses about event genuineness (*Hypotheses 2-3*) consider an event as genuine if all of its explanations are valid and an explanation as valid if all of its consequences are genuine events, Definition 3 specifies an event as genuine if at least one of its explanations is valid and an explanation as valid if at least one of its consequences is genuine event. This is a relaxation of our initial hypotheses regarding event genuineness, which is introduced due to the introduction of the diagnosis time window and the exclusion of already considered recorded events. By using the diagnosis time window and the concept of already considered recorded events, there is the possibility that no actual explanations, consequences or matching recorded events could be generated, identified and found, respectively, at any recursive invocation of the genuineness assessment of an event.

5.4.6 Belief Functions

Definition 3 establishes the logical criteria for the assessment of event genuineness. As discussed in Section 5.4.1.1, during the diagnosis process there might be uncertainty regarding the occurrence, and therefore the genuineness of the involved events. To deal

with this uncertainty, the diagnosis mechanism does not use strict logical values for the genuineness of an event. On the contrary, the diagnosis process advocates an approximate reasoning approach, which generates a degree of belief in the genuineness of an event by computing intermediate degrees of belief in the membership of an event in the log of the monitor and the existence of some valid explanation for the event. These degrees of belief are computed by combining partial beliefs to genuineness, explainability and validity that are assigned by *basic probability assignments or mass functions* founded in the axiomatic framework of the Dempster Shafer theory of evidence [146] (denoted as *DS Theory* in the following). The three basic probability assignemnts or mass functions that are used by the diagnostic process for computing the degree of belief in the genuineness of an event are as follows:

- the function m^{GN}, which measures the likelihood that an event e is genuine with respect to the diagnostic time range of interest TR and previously considered recorded events U_o, i.e., the likelihood of the proposition denoted by Genuine(e,U_o,TR) = $\bigvee_{e_i \in U(e,TR)}$(Explainable($e_i$,$U_o$,TR)),

- the function m_i^{EX}, which measures the likelihood that an event e_i is explainable with respect to the diagnostic time range of interest TR and previously considered recorded events U_o, i.e., the likelihood of the proposition denoted by Explainable(e_i,U_o,TR) =$\bigvee_{\Phi_j \in EXP(e_i)}$Valid($e_i$,$\Phi_j$,$U_o$,TR), and

- the function m_j^{VL}, which measures the likelihood that an explanation Φ_j of an event e_i is valid with respect to the diagnostic time range of interest TR and previously considered recorded events U_o, i.e., the likelihood of the proposition denoted by Valid(e_i,Φ_j,U_o,TR) =$\bigvee_{e_q \in CONS(\Phi,TR)}$Genuine($e_q$,$U_o$,TR)

It should be noted that the above mass functions are used in order to assess the genuineness of the events involved in runtime S&D violations. More specifically, the above functions viewed as an individual mechanism process other genuine events in EVEREST event log in order to check whether they can cast confirming or refuting evidence to the violations observations genuineness. The genuineness of events is assessed based on their explainability, the validity of their explanations, and the genuineness of the expected effects of their explanations. The way how the concepts of

event genuineness, event explainability and explanation validity have been evolved to reach the definitions we are presenting in this thesis can be tracked in the series of our earlier publications [162, 163, 164]. The motivation throughout this research line of work has been the limitation of existing runtime monitoring and diagnosis approaches, as presented in Chapter 2, to perform runtime checks based on runtime events that they receive and analyse from the monitored system without assessing the trustworthiness of these events. Therefore, our approach has been devised to provide a mechainsm that assesses the trustworthiness of streams of events used for runtime monitoring by considering and using the concept of event genuineness as a term to model the event trustworthiness.

5.4.6.1 Frames of discerment

In accordance to the *DS Theory*, a prerequisite for defining formally and combining the above functions is the introduction of frames of discernment, i.e., sets of mutually exclusive propositions representing exhaustively the properties that the functions assign belief to. For defining the frames of discernment for the above functions suppose that the belief in the genuineness of event e_s is questioned. Thus, we have:

Definition 5: The frame of discernment θ_{es} discerns the genuineness property of event e_s. Therefore, θ_{es} describes the propositions *Genuine*(e_s,U_o,*TR)* and ¬*Genuine*(e_s,U_o,*TR)*. Assuming for simplicity that these propositions are denoted as GN_s and $\neg GN_s$, respectively, then the frame of discernment θ_{es} can be defined as a set of vectors of Boolean variable of the form $[G_s]$ where, in each vector, the Boolean variable G_s denotes whether e_s is genuine or not by taking the values *True* or *False* respectively. The frame of discernment θ_{es} will contain 2 vectors. Given the above assumptions about the construction of the frame of discernment θ_{es}, the propositions G_s, $\neg G_s$ and $G_s \vee \neg G_s$ will correspond to the following subsets of θ_{es}:

- GN_s will correspond to {[G_s = True]} referred to as GN_s henceforth
- $\neg GN_s$ will correspond to {[G_s = False]} referred to as GN_s' henceforth
- $GN_s \vee \neg GN_s$ will correspond to {[G_s = True or False]} which is equal to θ_{es}

Definition 6: The frame of discernment θ_{es}^{EX} discerns the explainability property of the matching recorded events of event e_s that consist the set $U(e_s,TR)$. Therefore, θ_{es}^{EX} describes the propositions $Explainable(e_i,U_o,TR)$ and $\neg Explainable(e_i,U_o,TR)$, where $e_i \in U(e_s,TR)$. Assuming for simplicity that these propositions are denoted as EX_i and $\neg EX_i$, respectively, then the frame of discernment θ_{es}^{EX} can be defined as a set of vectors of Boolean variables of the form $[E_1, E_2, \ldots, E_n]$, where $n = |U(e_s,TR)|$ and the Boolean variable E_i in each vector denotes whether the recorded event e_i is explainable or not by taking the values *True* or *False* respectively. Furthermore, suppose that by convention a vector denotes the conjunction of the propositions expressed by its variables and a set of vectors denotes the disjunction of the propositions that are represented by its elements. The frame of discernment θ_{es}^{EX} will contain 2^n vectors to denote all the different combinations of values of $E_1, E_2, \ldots, E_n$. Given the above assumptions about the construction of the frame of discernment θ_{es}^{EX}, the propositions EX_i, $\neg EX_i$ and $EX_i \vee \neg EX_i$ will correspond to the following subsets of θ_{es}^{EX}:

- EX_i will correspond to $\{[E_1, \ldots, E_n] \mid E_i = True\}$ referred to as EX_i henceforth
- $\neg EX_i$ will correspond to $\{[E_1, \ldots, E_n] \mid E_i = False\}$ referred to as EX_i' henceforth
- $EX_i \vee \neg EX_i$ will correspond to $\{[E_1, \ldots, E_n] \mid E_i = True \text{ or } E_i = False\}$ which is equal to θ_{es}^{EX}

Definition 7: The frame of discernment θ_{es}^{VL} discerns the validity property of the alternative explanations of a matching recorded event e_i of event e_s that consist the set $EXP(e_i)$. Therefore, θ_{es}^{VL} describes the propositions $Valid(e_i,\Phi_j,U_o,TR)$ and $\neg Valid(e_i,\Phi_j,U_o,TR)$, where $\Phi_j \in EXP(e_i)$. Assuming for simplicity that these propositions are denoted as VL_j and $\neg VL_j$, respectively, then the frame of discernment θ_{es}^{VL} can be defined as a set of vectors of Boolean variables of the form $[V_1, V_2, \ldots, V_m]$, where $m = |EXP(e_i)|$ and the Boolean variable V_j in each vector denotes whether the alternative explanation Φ_j is valid or not by taking the values *True* or *False* respectively. Furthermore, suppose that by convention a vector denotes the conjunction of the propositions expressed by its variables and a set of vectors denotes the disjunction of the propositions that are represented by its elements. The frame of discernment θ_{es}^{VL} will contain 2^m vectors to denote all the different combinations of values of $V_1, V_2, \ldots, V_m$. Given the above assumptions about the construction of the frame of discernment θ_{es}^{VL},

the propositions VL_j, $\neg VL_j$ and $VL_j \vee \neg VL_j$ will correspond to the following subsets of θ_{es}^{VL}:

- VL_j will correspond to $\{[V_1,V_2,\ldots,V_m] \mid V_j = \text{True}\}$ referred to as VL_j henceforth
- $\neg VL_j$ will correspond to $\{[V_1,V_2,\ldots,V_m] \mid V_j=\text{False}\}$ referred to as VL_j' henceforth
- $VL_j \vee \neg VL_j$ will correspond to $\{[V_1,V_2,\ldots,V_m] \mid V_j = \text{True or } V_j =\text{False}\}$ which is equal to θ_{es}^{VL}

Definition 8: The frame of discernment θ_{es}^{GN} discerns the genuineness property of the expected consequences of an alternative explanation Φ_j of a matching recorded event e_i of event e_s that consist the set $CONS(\Phi_j,TR)$. Therefore, θ_{es}^{GN} describes the propositions $Genuine(e_q,U_o,TR)$ and $\neg Genuine(e_q,U_o,TR)$, where $e_q \in CONS(\Phi_j,TR)$. Assuming for simplicity that these propositions are denoted as GN_q and $\neg GN_q$, respectively, then the frame of discernment θ_{es}^{GN} can be defined as a set of vectors of Boolean variables of the form $[G_1, G_2,\ldots, G_r]$, where $r = |CONS(\Phi_j,TR)|$ and the Boolean variable G_q in each vector denotes whether the expected consequence e_q is valid or not by taking the values *True* or *False* respectively. Furthermore, suppose that by convention a vector denotes the conjunction of the propositions expressed by its variables and a set of vectors denotes the disjunction of the propositions that are represented by its elements. The frame of discernment θ_{es}^{GN} will contain 2^r vectors to denote all the different combinations of values of $G_1, G_2,\ldots, G_r$. Given the above assumptions about the construction of the frame of discernment θ_{es}^{GN}, the propositions GN_q, $\neg GN_q$ and $GN_q \vee \neg GN_q$ will correspond to the following subsets of θ_{es}^{GN}:

- GN_q will correspond to $\{[G_1,G_2,\ldots,G_r] \mid G_q = \text{True}\}$ referred to as GN_q henceforth
- $\neg GN_q$ will correspond to $\{[G_1,G_2,\ldots,G_r]|G_q=\text{False}\}$ referred to as GN_q' henceforth
- $GN_q \vee \neg GN_q$ will correspond to $\{[G_1,G_2,\ldots,G_r] |G_q = \text{True or } G_q =\text{False}\}$ which is equal to θ_{es}^{GN}

5.4.6.2 Definitions of basic probability assignments

In this section, having defined the frames θ_{es}, θ_{es}^{EX}, θ_{es}^{V}, that discerns the genuineness, explainability and validity properties of individual events and sets of events that may

appear during the assessment of the belief in genuineness of the event e_s, we provide the definitions of the functions m^{GN}, m_i^{EX}, and m_j^{VL} that assigns partial belief to subsets of the aforementioned frames. The proof of the theorems that are used in the definitions of the functions below can be found in Section 5.5.

Definition 9: m^{GN} is a function measuring the basic probability of the genuineness and non-genuineness of the event e, by assigning basic probability to the propositions *Genuine(e,U$_o$,TR)* and *¬Genuine(e,U$_o$,TR)* that are denoted as *GN* and *¬GN* in the following and are described by subsets of θ_{es} and θ_{es}^{GN} (see *Definitions 5* and *8* repectively in Section 5.4.6.1), and is defined as:

1. If $U(e,TR) \neq \varnothing$, i.e. according to *Conditions 9 -11*, there are matching recorded events for e in the event log with respect to the diagnosis time range *TR* and previously considered recorded events U_o, we have the following cases:

   i) If $e = C_{NULL}$, i.e., C_{NULL} is a special event introduced to denote that all of the identified consequences of an explanation are not accepted due to the diagnosis time range *TR* (see Section 5.4.4.3), we have for m^{GN}:

   $m^{GN}(GN) = \alpha_2$

   $m^{GN}(\neg GN) = 1 - \alpha_2$

   $m^{GN}(GN \vee \neg GN) = 1 - m^{GN}(GN) - m_s^{GN}(\neg GN) = 0$

   where,

   - α_2 is a belief value within 0 and 1 that is predetermined by the user of the diagnostic framework

   As the following theorem indicates for this case, m^{GN} is a basic probability assignment to the genuineness of an event e, according to the axiomatic definition of such assignments in the context of the Dempster-Shafer theory of evidence with respect to θ_{es} and θ_{es}^{GN} (see *Definitions 5* and *8* repectively in Section 5.4.6.1).

   Theorem 5.1: *The evidence measure m^{GN} defined as:*

$$m^{GN}(P) = \begin{cases} \alpha_2, \text{ if } P = Genuine(e,U_o,TR) \\ 1 - \alpha_2, \text{ if } P = \neg Genuine(e,U_o,TR) \\ 0, \text{ otherwise} \end{cases}$$

where α_2 is a value within 0 and 1, is a DS basic probability assignment with respect to frames of discerment θ_{es} and *θ_{es}^{GN} (see Definitions 5 and 8 repectively in Section 5.4.6.1).*

ii) Otherwise, we have for m^{GN}:

$m^{GN}(GN) = Bel(\bigvee_{i=1,\ldots,|U(e,TR)|} Explainable(e_i,U_o,TR))$

$m^{GN}(\neg GN) = Bel(\bigwedge_{i=1,\ldots,|U(e,TR)|} \neg Explainable(e_i,U_o,TR))$

$m^{GN}(GN \vee \neg GN) = 1 - m^{GN}(GN) - m^{GN}(\neg GN)$

where, as the following theorem indicates,

- $Bel(\bigvee_{i=1,\ldots,|U(e,TR)|} Explainable(e_i,U_o,TR)) =$

$$= \sum_{I \subseteq U(e_i,TR) \text{ and } I \neq \varnothing} (-1)^{|I|+1} \left\{ \prod_{i \in I} m_i^{EX}(Explainable(e_i,U_o,TR)) \right\}$$

- $Bel(\bigwedge_{i=1,\ldots,|U(e,TR)|} \neg Explainable(e_i,U_o,TR)) =$

$$= \prod_{e_i \in U(e,TR)} \left\{ m_i^{EX}(\neg Explainabe(e_i,U_o,TR)) \right\}$$

Theorem 5.2: *If e is an event and U(e,TR) is the set of the events that are recorded in the log of the monitoring framework and can be unified with e, and it holds that $U(e,TR) \neq \varnothing$ with $n = |U(e,TR)|$, i.e. the number of the members of U(e,TR), the belief in the explainability of at least one recorded event in U(e,TR), $Bel(\bigvee_{i=1,\ldots,n} Explainable(e_i,U_o,TR))$, and in the explainability of none of the events in U(e,TR), $Bel(\bigwedge_{i=1,\ldots,n} \neg Explainable(e_i,U_o,TR))$, are measured by the following functions:*

$$Bel(\bigvee_{i=1,\ldots,n} Explainable(e_i, U_o, TR)) =$$

$$= \sum_{I \subseteq \{1,\ldots,n\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} m_i^{EX}(Explainable(e_i, U_o, TR)) \right\}$$

$$Bel(\bigwedge_{i=1,\ldots,n} \neg Explainable(e_i, U_o, TR)) =$$

$$= \prod_{i=1,\ldots,n} \{ m_i^{EX}(\neg Explainabe(e_i, U_o, TR)) \}$$

where m_i^{EX} *(i=1,…, n) is the basic probability assignment associated with the event* e_i.

Furthermore, as the following theorem indicates for this case, m^{GN} is a basic probability assignment to the genuineness of an event *e*, according to the axiomatic definition of such assignments in the context of the Dempster-Shafer theory of evidence with respect to θ_{es} and θ_{es}^{GN} (see *Definitions 5* and *8* repectively in Section 5.4.6.1).

Theorem 5.3: *The evidence measure* m^{GN} *defined as:*

$$m^{GN}(P) = \begin{cases} Bel(\bigvee_{i=1,\ldots,n} Explainable(e_i, U_o, TR)), \text{ if } P = Genuine(e, U_o, TR) \\ Bel(\bigwedge_{i=1,\ldots,n} \neg Explainable(e_i, U_o, TR)), \text{ if } P = \neg Genuine(e, U_o, TR) \\ 1 - Bel(\bigvee_{i=1,\ldots,n} Explainable(e_i, U_o, TR)) - Bel(\bigwedge_{i=1,\ldots,n} \neg Explainable(e_i, U_o, TR)), \text{ otherwise} \end{cases}$$

where $n = |U(e,TR)|$*, i.e., the number of the matching recorded events of e, is a DS basic probability assignment with respect to frames of discerment* θ_{es} and θ_{es}^{GN} *(see Definitions 5 and 8 repectively in Section 5.4.6.1).*

2. Else, if $U(e,TR) = \varnothing$, i.e., no recorded events matching with *e* were found in the event log with respect to the diagnosis time range *TR* and previously considered recorded events either because the matcing recorded events for *e* are already considered during previous recursive invocations of the Preprocess algorithm (see

Section 5.4.4.3) or because no such events are stored in the event log, we have the following cases:

i) If the last known value of the clock of *Captor(e)*, i.e., the timestamp of the last event in the log that has produced by *Captor(e)* at the time of the search is greater than the upper boundary of the time range that is specified for *e*, or formally *lastTimestamp(Captor(e))* > te^{UB}, we have for m^{GN}:

$$m^{GN}(GN) = Bel(\bigwedge_{i=1,\ldots,|A(e)|} \neg Explainable(e_i,U_o,TR))$$

$$m^{GN}(\neg GN) = Bel(\bigvee_{i=1,\ldots,|A(e)|} Explainable(e_i,U_o,TR))$$

$$m^{GN}(GN \vee \neg GN) = 1 - m^{GN}(GN) - m^{GN}(\neg GN)$$

where,

- *A(e)* contains the recorded events, e_A, which have been produced by *Captor(e)*, cannot be unified with *e*, and their timestamps, te_A, are within a time range whose lower boundary is open and equal to the upper boundary of *e*, te^{UB}, while the upper boundary is close and equal to the sum of te^{UB} and *lastTimestamp(Captor(e))*, or formally:

$$A(e) = \{ e_A \mid e_A \in EventLog \textbf{ and } mgu(e, e_A) = \varnothing \textbf{ and}$$
$$Captor(e) = Captor(e_A) \textbf{ and } te_A > te^{UB} \textbf{ and}$$
$$te_A \leq lastTimestamp(Captor(e))\}$$

- and from Theorem 5.2 and by replacing U(e,TR) with A(e), it holds that:

$$\blacksquare\ Bel(\bigwedge_{i=1,\ldots,|A(e_i)|} \neg Explainable(e_i,U_o,TR)) =$$
$$= \prod_{e_i \in A(e_i)} \{ m_i^{EX}(\neg Explainabe(e_i,U_o,TR)) \}$$

$$\blacksquare\ Bel(\bigvee_{i=1,\ldots,|A(e_i)|} Explainable(e_i,U_o,TR))$$
$$= \sum_{I \subseteq A(e_i) \text{ and } I \neq \varnothing} (-1)^{|I|+1} \{ \prod_{i \in I} m_i^{EX}(Explainable(e_i,U_o,TR)) \}$$

Furthermore, from *Theorem 5.3* and by replacing again U(e,TR) with A(e), m^{GN} is a basic probability assignment to the genuineness of an event *e*, according to

the axiomatic definition of such assignments in the context of the Dempster-Shafer theory of evidence with respect to θ_{es} and θ_{es}^{GN} (see *Definitions 5* and *8* repectively in Section 5.4.6.1).

To give a picture of the above case, Figure 5-17 illustrates some time points on the timeline of *Captor(e_i)*, which are significant for the aforementioned case. These significant time points are the lower and upper boundaries, te_i^{LB} and te_i^{UB}, which e_i is specified within, and the upper boundary, *lastTimestamp(Captor(e_i))*, that the search for events for *A(e_i)* should respect. For being fair in cases as the above, the belief function m^{GN} defines that the belief in the not genuineness of e_i depends on the explainability of recorded events that have produced by *Captor(e_i)* and occurred at timepoints within te_i^{UB} and *lastTimestamp(Captor(e_i))*. On the other hand, m^{GN} defines that the belief in the genuineness of e_i depends on recorded events that again have produced by *Captor(e_i)* and occurred at timepoints within te_i^{UB} and *lastTimestamp(Captor(e_i))* but cannot be explained.

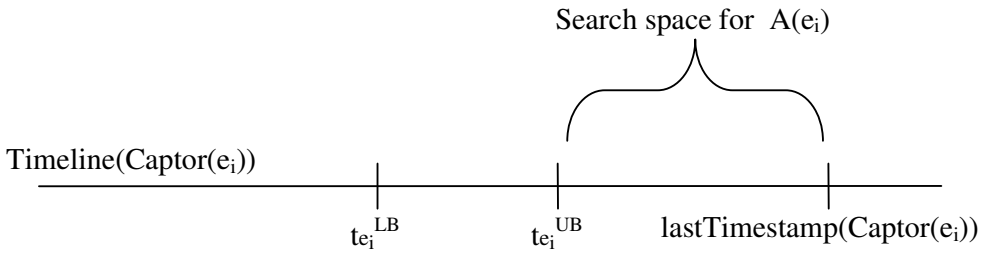

Figure 5-17 – Timeline of *Captor(e_i)*

ii) Else, if the last known value of the clock of *Captor(e_i)*, i.e., the timestamp of the last event in the log that has produced by *Captor(e_i)* at the time of the search is less than or equal to the upper boundary of the time range that is specified for e_i, or formally $lastTimestamp(Captor(e_i)) \leq te_i^{UB}$, we have for m^{GN}:

$m^{GN}(GN) = 0.5$

$m^{GN}(\neg GN) = 0.5$

$m^{GN}(GN \vee \neg GN) = 1 - m^{GN}(GN) - m^{GN}(\neg GN) = 0$

It should be noted that, from *Theorem 5.1* and by setting α_2 equal to 0.5, m^{GN} is a basic probability assignment to the genuineness of an event *e*, according to the axiomatic definition of such assignments in the context of the Dempster-Shafer theory of evidence with respect to θ_{es} and θ_{es}^{GN} (see *Definitions 5* and *8* repectively in Section 5.4.6.1).

iii) Else, the last known value of the clock of *Captor*(e_i) is *null*, i.e., there is no recorded event that is produced from *Captor*(e_i), we have for m^{GN}:

$m^{GN}(GN) = 0$

$m^{GN}(\neg GN) = 0$

$m^{GN}(GN \vee \neg GN) = 1 - m^{GN}(GN) - m^{GN}(\neg GN) = 1$

Similarly, from *Theorem 5.1* and by setting α_2 equal to 0 for this case, m^{GN} is a basic probability assignment to the genuineness of an event *e*, according to the axiomatic definition of such assignments in the context of the Dempster-Shafer theory of evidence with respect to θ_{es} and θ_{es}^{GN} (see *Definitions 5* and *8* repectively in Section 5.4.6.1).

It should be noted that such case could occur in the beginning of the monitoring session for the underlying system. More specifically, the fact that no recorded events produced from *Captor*(e_i) can be found, it might not mean necessarily that there is a system behaviour that deviates from the intended system behaviour. On the contray, it might mean that *Captor*(e_i) has not correctly produced yet any event according to systems specifications.

Definition 10: m_i^{EX} is a function measuring the degree of belief in the existence of a valid explanation for an event e_i by assigning basic probability to the propositions *Explainable*(e_i,U_o,*TR*) and $\neg$ *Explainable*(e_i,U_o,*TR*) that are denoted as EX_i and $\neg EX_i$ in the following and are described by subsets of θ_{es}^{EX} (see *Definition 6* in Section 5.4.6.1), and is defined as:

- If *isExplainabilityComputed*(e_i) =*False*, where *isExplainabilityComputed*(e_i) is a Boolean flag stored in the sublist for e_i in U_o and denotes that the process has not

assessed the explainability of event e_i in previous recursive invocation, and therefore, the variables *assessmentOf(Explainable(e_i))* and *assessmentOf(¬Explainable(e_i))* have still null values (see also in Sections 5.4.4.2 and 5.4.4.3), we have the following cases:

i) If $EXP(e_i) = \varnothing$, i.e., no explanations can be generated for e_i due to the fact that e_i is abducible and observable predicate (see also *Definition 1* in Section 5.4.2), we have that:

$m_i^{EX}(EX_i) = \alpha_1$

$m_i^{EX}(\neg EX_i) = 1 - \alpha_1$

$m_i^{EX}(EX_i \vee \neg EX_i) = 1 - m_i^{EX}(EX_i) - m_i^{EX}(\neg EX_i) = 0$

where,

α_1 is a belief value within 0 and 1 that is predetermined by the user of the diagnostic framework

As the following theorem indicates for this case, m_i^{EX} is a basic probability assignment to the explanablity of an event e_i, according to the axiomatic definition of such assignments in the context of the Dempster-Shafer theory of evidence with respect to θ_{es}^{EX} (see *Definition 6* in Section 5.4.6.1).

Theorem 5.4: *The evidence measure m_i^{EX} defined as:*

$$m_i^{EX}(P) = \begin{cases} \alpha_1, \text{ if } P = Explainable(e_i, U_o, TR) \\ 1 - \alpha_1, \text{ if } P = \neg Explainable(e_i, U_o, TR) \\ 0, \text{ otherwise} \end{cases}$$

where α_1 is a value within 0 and 1, is a DS basic probability assignment with respect to frame of discerment θ_{es}^{EX} (see Definitions 6 in Section 5.4.6.1).

ii) Else if $EXP(e_i) \neq \varnothing$, we have that:

$m_i^{EX}(EX_i) = Bel(\bigvee_{j=1,\ldots,|EXP(e_i)|} Valid(e_i, \Phi_j, U_o, TR))$

$m_i^{EX}(\neg EX_i) = Bel(\bigwedge_{j=1,\ldots,|EXP(e_i)|} \neg Valid(e_i,\Phi_j,U_o,TR)))$

$m_i^{EX}(EX_i \vee \neg EX_i) = 1 - m_i^{EX}(EX_i) - m_i^{EX}(\neg EX_i)$

where, as the following theorem indicates,

- $Bel(\bigvee_{j=1,\ldots,|EXP(e_i)|} Valid(e_i,\Phi_j,U_o,TR)) =$

$$=\sum_{J\subseteq EXP(e_i) \text{ and } J\neq\varnothing}(-1)^{|J|+1}\left\{\prod_{j\in J} m_j^{VL}(Valid(e_i,\Phi_j,U_o,TR))\right\}$$

- $Bel(\bigwedge_{j=1,\ldots,|EXP(e_i)|} \neg Valid(e_i,\Phi_j,U_o,TR))) =$

$$=\prod_{\Phi j\in EXP(e_i)}\left\{ m_j^{VL}(\neg Valid(e_i,\Phi_j,U_o,TR))\right\}$$

Theorem 5.5: *If e_i is an event and $EXP(e_i)$ is the set of the alternative explanations that are generated for e_i, and it holds that $EXP(e_i)\neq\varnothing$ with $m = |EXP(e_i)|$, i.e. the number of the members of $EXP(e_i)$, the belief in the validity of at least one alternative explanation in $EXP(e_i)$, $Bel(\bigvee_{j=1,\ldots,m} Valid(e_i,\Phi_j,U_o,TR))$, and in the validity of none of the alternative explanations in $EXP(e_i)$, $Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i,\Phi_j,U_o,TR))$, are measured by the following functions:*

$Bel(\bigvee_{j=1,\ldots,m} Valid(e_i,\Phi_j,U_o,TR)) =$

$$= \sum_{J\subseteq\{1,\ldots,m\} \text{ and } J\neq\varnothing}(-1)^{|J|+1}\left\{\prod_{j\in J} m_j^{VL}(Valid(e_i,\Phi_j,U_o,TR))\right\}$$

$Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i,\Phi_j,U_o,TR)) =$

$$= \prod_{j=1,\ldots,m}\{m_j^{VL}(\neg Valid(e_i,\Phi_j,U_o,TR))\}$$

where m_j^{VL} $(j=1,\ldots, n)$ is the basic probability assignment associated with the alternative explanation Φ_j of e_i.

Furthermore, as the following theorem indicates for this case, m_i^{EX} is a basic probability assignment to the explainability of an event e_i, according to the

axiomatic definition of such assignments in the context of the Dempster-Shafer theory of evidence with respect to θ_{es}^{EX} (see *Definition 6* in Section 5.4.6.1).

Theorem 5.6: *The evidence measure m_i^{EX} defined as:*

$$m_i^{EX}(P) = \begin{cases} Bel(\bigvee_{j=1,\ldots,m} Valid(e_i, \Phi_j, U_o, TR)), \text{ if } P = Explainable(e_i, U_o, TR) \\ Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i, \Phi_j, U_o, TR)), \text{ if } P = \neg Explainable(e_i, U_o, TR) \\ 1\text{-}Bel(\bigvee_{j=1,\ldots,m} Valid(e_i, \Phi_j, U_o, TR))\text{-}Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i, \Phi_j, U_o, TR)), \\ otherwise \end{cases}$$

where $m = |EXP(e_i)|$, i.e., the number of alternative explanations of e_i, is a DS basic probability assignment with respect to frame of discerment θ_{es}^{EX} (see Definition 6 in Section 5.4.6.1).

As soon as the assessment process finishes the computation of the belief in the explainability of e_i for first time, the process updates the sublist of e_i in U_o (see Section 5.4.4.2). In particular, the process sets the Boolean flag *isExplainabilityComputed(e_i)* to *True*, and sets values to the two placeholders for the assessment result of the explanability of e_i, *assessmentOf(Explainable(e_i))* and *assessmentOf(¬Explainable(e_i))*, as follows:

isExplainabilityComputed(e_i) = True

assessmentOf(Explainable(e_i)) = $m_i^{EX}(EX_i)$

assessmentOf(¬Explainable(e_i)) = $m_i^{EX}(\neg EX_i)$

2. If *isExplainabilityComputed(e_i) =True*, we have for m_i^{EX}:

$m_i^{EX}(EX_i)$ = assessmentOf(Explainable(e_i)) / occurrenceTimes(e_i)

$m_i^{EX}(\neg EX_i)$ = assessmentOf(¬Explainable(e_i)) / occurrenceTimes(e_i)

$m^{EX}(EX_i \vee \neg EX_i) = 1 - m_i^{EX}(EX_i) - m_i^{EX}(\neg EX_i)$

It should be noted that by these means the assessment process distributes equally the evidence of any recorded event that is repeatedly reached as actual

consequence of alternative explanations of the same event e_x. Moreover, we observe that *occurrenceTimes*(e_i) is always equal to or greater than 1. Also, the values *assessmentOf(Explainable*(e_i)*)* and *assessmentOf(Explainable*($\neg e_i$)*)*, which are respectively equal to $m_i^{EX}(EX_i)$ and $m_i^{EX}(\neg EX_i)$, are within 0 and 1 as m_i^{EX} satisfies Axiom 1 of *DS Theory* (see Section 3.4) shown in *Theorem 5.6*. Based on the aforementioned observations, the following theorem indicates that m_i^{EX} is a basic probability assignment to the explainability of an event e_i, according to the axiomatic definition of such assignments in the context of the Dempster-Shafer theory of evidence with respect to $\theta_{e_s}^{EX}$ (see *Definition 6* in Section 5.4.6.1).

Theorem 5.7: *The evidence measure* m_i^{EX} *defined as:*

$$m_i^{EX}(P) = \begin{cases} assessmentOf(Explainable(e_i))/occurrenceTimes(e_i), \\ if\ P = Explainable(e_i, U_o, TR) \\ \\ assessmentOf(\neg Explainable(e_i))/occurrenceTimes(e_i), \\ if\ P = \neg Explainable(e_i, U_o, TR) \\ \\ 1-[assessmentOf(Explainable(e_i))+assessmentOf(\neg Explainable(e_i)) \\ /occurrenceTimes(e_i))],\quad otherwise \end{cases}$$

where occurrenceTimes(e_i)≥ 1 *, i.e., the number of times that* e_i *was reached as a consequence of alternative explanations of the same event* e_s*, and the values assessmentOf(Explainable*(e_i)*) and assessmentOf(Explainable*($\neg e_i$)*) are within 0 and 1, is a DS basic probability assignment with respect to frame of discerment* $\theta_{e_s}^{EX}$ *(see Definition 6 in Section 5.4.6.1).*

Definition 11: m_j^{VL} is a function measuring the degree of belief in the existence of a genuine consequence of an explanation Φ_j generated for an event e_i by assigning basic probability to the propositions $Valid(e_i, \Phi_j, U_o, TR)$ and $\neg Valid(e_i, \Phi_j, U_o, TR)$ that are

denoted as VL_j and $\neg VL_j$ in the following and are described by subsets of θ_{es}^{VL} (see *Definition 7* in Section 5.4.6.1), and is defined as:

$$m_j^{VL}(VL_j) = Bel(\bigvee_{q=1,\ldots,|CONS(\Phi_j,TR)|} Genuine(e_q,U_o,TR))$$

$$m_j^{VL}(\neg VL_j) = Bel(\bigwedge_{q=1,\ldots,|CONS(\Phi_j,TR)|} \neg Genuine(e_q,U_o,TR))$$

$$m_j^{VL}(VL_j \vee \neg VL_j) = 1 - m_j^{VL}(VL_j) - m_j^{VL}(\neg VL_j)$$

where, as the following theorem indicates,

$$Bel(\bigvee_{q=1,\ldots,|CONS(\Phi_j,TR)|} Genuine(e_q,U_o,TR))$$

$$= \sum_{Q \subseteq CONS(\Phi_j,TR) \text{ and } Q \neq \varnothing} (-1)^{|Q|+1} \left\{ \prod_{q \in Q} m_q^{GN}(Genuine(e_q,U_o,TR)) \right\}$$

$$Bel(\bigwedge_{q=1,\ldots,|CONS(\Phi_j,TR)|} \neg Genuine(e_q,U_o,TR))$$

$$= \prod_{eq \in CONS(\Phi_j,TR)} \left\{ m_q^{GN}(\neg Genuine(e_q,U_o,TR)) \right\}$$

Theorem 5.8: *If Φ_j is an alternative explanation of e_i and $CONS(\Phi_j,TR)$ is the set of the expected consequences that are identified for Φ_j, and it holds that $CONS(\Phi_j,TR) \neq \varnothing$ with $r = |CONS(\Phi_j,TR)|$, i.e. the number of the members of $CONS(\Phi_j,TR)$, the belief in the genuineness of at least one expected consequence in $CONS(\Phi_j,TR)$, $Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q,U_o,TR))$, and in the genuineness of none of the expected consequences in $CONS(\Phi_j,TR)$, $Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q,U_o,TR))$, are measured by the following functions:*

$$Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q,U_o,TR)) =$$

$$= \sum_{Q \subseteq CONS(\Phi_j,TR) \text{ and } Q \neq \varnothing} (-1)^{|Q|+1} \left\{ \prod_{q \in Q} m_q^{GN}(Genuine(e_q,U_o,TR)) \right\}$$

$$Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q,U_o,TR)) =$$

$$= \prod_{eq \in CONS(\Phi_j,TR)} \left\{ m_q^{GN}(\neg Genuine(e_q,U_o,TR)) \right\}$$

where m_q^{GN} *(q=1,…, r) is the basic probability assignment associated with the expected consequence* e_q *of the alternative explanation* Φ_j *of* e_i*.*

Furthermore, as the following theorem indicates for this case, m_j^{VL} is a basic probability assignment to the validity of an alternative explanation Φ_j, according to the axiomatic definition of such assignments in the context of the Dempster-Shafer theory of evidence with respect to θ_{es}^{VL} (see *Definition 7* in Section 5.4.6.1).

Theorem 5.9: *The evidence measure* m_j^{VL} *defined as:*

$$m_j^{VL}(P) = \begin{cases} Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q, U_o, TR)), \text{ if } P = Valid(e_i, \Phi_j, U_o, TR) \\ Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q, U_o, TR)), \text{ if } P = \neg Valid(e_i, \Phi_j, U_o, TR) \\ 1\text{-}Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q, U_o, TR)) - Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q, U_o, TR)), \\ \text{otherwise} \end{cases}$$

where $r = |CONS(\Phi_j, TR)|$*, i.e., the number of expected consequences of the* Φ_j*, is a DS basic probability assignment with respect to frame of discerment* θ_{es}^{VL} *(see Definition 7 in Section 5.4.6.1).*

As indicated in case 1.i) of *Definition 9,* m^{GN} assigns a predetermined belief value α_2 to null consequences. Whilst the reasoning principle underpinning the diagnosis framework favours explanations, which are confirmed by the fact that they have consequences matched by genuine events other than the event that they were generated for, it would be unfair to disregard entirely explanations that have no other such consequences. Cases of such explanations are more likely to arise when the diagnosis window is narrow and, therefore, it may be possible to end up with explanations with no further consequences falling within the given diagnosis window. For such explanations, it is important to assign some belief measure in their validity but at the same time keep this measure low to reflect the absence of any evidence of runtime event in the given diagnosis interval. The definition of the belief function m^{GN} introduces the parameter α_2 to define the belief measure that should be used in such cases and leaves the choice of its exact value to the user of the framework. The expectation, however, is that this value will

be a number close to zero to ensure that explanations with no consequences cannot affect significantly the overall belief in the genuineness of events.

Similarly, as indicated in case □.i) of *Definition 10*, m^{EX} assigns some belief α_1 in the genuineness of events which have no explanations. This is a relaxation of the logical definition of event genuineness in *Definition 3* that is introduced for the following reason. An event e_i with no explanations of its own may be required to provide confirmatory evidence for a consequence of an explanation of another event e_j. If this were the case, the assignment of a zero belief in the explainability of e_i (due to the absence of an explanation for it) would reduce or even make equal to zero the basic probability of the genuineness of the event e_j whose explanation had to be confirmed by e_i. The stance reflected by *Definition 5* in this case is that the very presence of e_i in the log of the monitoring infrastructure should provide some evidence for the validity of the explanation of e_j even though e_i is not explainable itself and that the belief in the validity of this explanation should be higher than in cases where none of its consequences were matching with events in the log of the monitoring infrastructure. Thus, m^{EX} assigns a small belief in the genuineness of events with no explanation that is determined by the parameter α_1, which exact value is chosen by the user of the framework. The value of this parameter should be set very close to zero, in order to provide a close approximation of the logical definition of explainability (*Definition 3*) in cases where an event does not have any explanation. It should be noted that the predetermined measures, α_1 and α_2, must respect an order, which ensures that the effect of these values on the final assessment result is fair and reasonable. In particular, α_1 must be less than α_2 to ensure that null explanations affect less the assessment result by being compared to null consequences that may occur due to diagnosis window.

As a final remark regading the sets of belief values we have selected for m^{GN} in the cases 1.ii) and 1.iii) of *Definition 9*, both sets represent uncertainty. In 1.ii), m^{GN} is a *Bayesian function*, i.e., the sum of $m^{GN}(Genuine(e_i,U_o,TR))$ and $m^{GN}(\neg Genuine(e_i,U_o,TR))$ equals to 1 [146], that provides a model regarding the uncertainty for the genuineness of e_i, which restricts e_i to be either genuine or not, given the evidence that $Captor(e_i)$ is operable and produces events according to the monitored system specifications. On the other hand, in 1.iii), m^{GN} represents a model of total uncertainty for the genuineness of e_i, and therefore the occurrence of e_i, as no confirming or refuting evidence have been produced by $Captor(e_i)$. Thus, in 1.iii), the correct behaviour of $Captor(e_i)$ could be

questioned as well. Of course, as a future line of work, it would be interesting to explore uncertainty models in terms of belief functions that could be more appropriate for answering plausibly the different questions that may appear in both cases.

5.5 Diagnosis Generation

5.5.1 The diagnosis generation process

The last phase of the diagnosis process is concerned with the generation of a final diagnosis of a violation based on the beliefs computed for the genuineness of the individual events involved in it. This final diagnosis is a report of the confirmed and unconfirmed predicates, which are involved in the violation that is generated as shown in the algorithm of Figure 5-18.

```
Generate_Violation_Explanation(R: Instance of Violated Rule)
1.  For each predicate P in R Do
2.     If P is negated Then
3.        Explanations(P) = explain(¬P, t_min(P), t_max(P), NULL)
4.        Generate_AE_Consequences(Explanations(P), Assumptions, P_Consequences)
5.     Else
6.        Explanations(P) = explain(P, t_min(P), t_max(P), NULL)
7.        Generate_AE_Consequences(Explanations(P), Assumptions, P_Consequences)
8.     End If
9.     [Bel(P), Pls(P)] = ComputeBeliefRange(P, Explanations(P), P_Consequences)
10.    If 1-Pls(P) < Bel(P) Then
11.       If P is negated Then
12.          UnconfirmedPredicates = UnconfirmedPredicates ∪ {P}
13.       Else
14.          ConfirmedPredicates = ConfirmedPredicates ∪ {P}
15.       End if
16.    End if
17. End For
18. For all P in ConfirmedPredicates Do report P as a confirmed predicate End for
19. For all P in UnconfirmedPredicares Do report P as unconfirmed predicate End for
END
```

Figure 5-18 – Final diagnosis generation algorithm

More specifically, this algorithm takes as input a template that represents an instantiation of an S&D monitoring rule that has been violated and generates explanations for the individual predicates which are involved in the violation by calling the *Explain* algorithm initially (see lines 3 and 6 in Figure Figure 5-18). In the case of negated predicates, the explanations are generated for the positive form of the predicate. This is because negated predicates cannot appear in the head of assumptions and, therefore, it is not possible to generate explanations for them directly. By virtue, however, of attempting to generate an explanation for the positive form of a negated predicate, the diagnosis process can still establish beliefs in the genuineness of the event represented by the predicate as we discussed above. It should also be noted that, as they do not appear in assumption heads, negated predicates cannot have been generated by deduction from assumptions during the monitoring process. Thus, their presence in violated rule instances is established by the principle of negation as failure when the expected predicate has not been seen in the event log of the monitoring system within the time range that it is expected to occur. Thus, an attempt to generate an explanation for the positive form of the predicate during the diagnosis process provides a means of confirming or not whether the application of the principle of negation as failure was reasonable given evidence from other events in the event log.

Having generated explanations for the individual predicates, the *Generate_Violation_Explanation* algorithm computes a belief range for the event repre-sented by each predicate and classifies the predicate as confirmed or unconfirmed depending on whether the belief in the genuineness of the event represented by it exceeds the belief in the non genuineness of this event. More specifically, a non negated predicate P will be classified as a confirmed predicate if $Bel(P) \geq Bel(\neg P)$[6]. A negated predicate ¬P, will be classified as a unconfirmed predicate if $Bel(P) \leq Bel(\neg P)$. Finally, the algorithm reports the classifications of individual predicates as confirmed or unconfirmed to the user (see lines 18-19 in Figure 5-18).

6 Bel(P) and Bel(¬P) represent the proposition $Bel(Genuine(e,U_o,TR))$ and $Bel(\neg Genuine(e,U_o,TR))$ respectively.

5.5.2 Examples of diagnosis generation

In the case of the example regarding the violation of Rule ATMS.R1, the algorithm will report *P1:* `Happens(e(E4,R1,AirBase,RES-A,signal(R1,A1,S1), AirBaseCaptor),7,R(7,7))` as a confirmed predicate and *P2:* `Happens(e(NF,R2,AirBase,signal(R2,A1,S1), AirBaseCaptor),t,R(7,12))` as an unconfirmed predicate. This will be due to the beliefs in the genuineness and non genuineness of the events unified with these predicates which are shown in Table 5-1.

Table 5-1 - Beliefs in genuineness of violation observations of Rule ATMS.R1

Predicate (P)	Bel(Genuine(P,U_o,TR))	Bel(¬Genuine(P,U_o,TR))	Confirmed
P1	$2\alpha_1 - \alpha_1^2$	0	YES
P2	0	$2\alpha_1 - \alpha_1^2$	NO

It should be noted that in order to calculate the belief and disbelief in the genuineness of *P2*, the algorithm calculates the belief and disbelief in the genuineness of ¬*P2* assuming that there is an event of signal sent from the radar R2 at some time point from t=7 to t=12 in the event log.

5.6 Mathematical Appendix: Proofs of Theorems in Chapter 5

Theorem 5.1: *The evidence measure m^{GN} defined as:*

$$m^{GN}(P) = \begin{cases} \alpha_2, \text{ if } P = Genuine(e,U_o,TR) \\ 1 - \alpha_2, \text{ if } P = \neg Genuine(e,U_o,TR) \\ 0, \text{ otherwise} \end{cases}$$

where α_2 is a value within 0 and 1, is a DS basic probability assignment with respect to frame of discerment θ_{es} and *θ_{es}^{GN} (see Definitions 5 nad 8 repectively in Section 5.4.6.1).*

Proof: To prove that m^{GN} is a DS basic probability assignment it is sufficient that m^{GN} satisfies the axioms Axiom 1, Axiom 2, and Axiom 3 of *DS Theory* (see Section 3.4).

(Axiom 1): Regarding θ_{es}, Axiom 1 is satisfied since:

If $P = GN_s$, $m_s^{GN}(P) = \alpha_2$ where $0 < \alpha_2 < 1$.

If $P = \neg GN_s$, $m_s^{GN}(P) = 1 - \alpha_2$ with $0 < 1 - \alpha_2 < 1$ since $0 < \alpha_2 < 1$.

If $P \neq GN_s$ or $P \neq \neg GN_s$, $m_s^{GN}(P) = 0$ by definition.

Similarly, Axiom 1 is satisfied with respect to θ_{es}^{GN} since:

If $P = GN_q$, $m_q^{GN}(P) = \alpha_2$ where $0 < \alpha_2 < 1$.

If $P = \neg GN_q$, $m_q^{GN}(P) = 1 - \alpha_2$ with $0 < 1 - \alpha_2 < 1$ since $0 < \alpha_2 < 1$.

If $P \neq GN_q$ or $P \neq \neg GN_q$, $m_q^{GN}(P) = 0$ by definition.

(Axiom 2): Regarding θ_{es}, Axiom 2 is satisfied by m_s^{GN} since its focals GN_s and GN_s' are non empty sets by definition of θ_{es}:

$GN_s = \{[G_s = True]\} \neq \varnothing$ and

$GN_s' = \{[G_s = False]\} \neq \varnothing$.

Thus, the basic probability assigned to the empty set by m_s^{GN} is $m_s^{GN}(\varnothing) = 0$

Similarly, Axiom 2 is satisfied by m_s^{GN} with respect to θ_{es}^{GN} since its focals GN_q and GN_q' are non empty sets by definition of θ_{es}^{GN}:

$GN_q = \{[G_1, G_2, \ldots, G_r] \mid G_q = True\} \neq \varnothing$ and

$GN_q' = \{[G_1, G_2, \ldots, G_r] \mid G_q = False\} \neq \varnothing$.

Thus, the basic probability assigned to the empty set by m_q^{GN} is $m_q^{GN}(\varnothing) = 0$

(Axiom 3): Regarding θ_{es}, Axiom 3 is satisfied since:

$$\sum_{P \subseteq \theta_{es}} m_s^{GN}(P) = \sum_{P \subseteq \theta_{es} \text{ and } P \neq GN_s \text{ and } P \neq \neg GN_s} m_s^{GN}(P) + m_s^{GN}(GN_s) + m_s^{GN}(\neg GN_s)$$

$$= 0 + a_2 + 1 - a_2 = 1$$

Similarly, Axiom 3 is satisfied with respect to θ_{es}^{GN} since:

$$\sum_{P \subseteq \theta_{es}^{GN}} m_q^{GN}(P) = \sum_{P \subseteq \theta_{es}^{GN} \text{ and } P \neq GN_q \text{ and } P \neq \neg GN_q} m_q^{GN}(P) + m_q^{GN}(GN_q) + m_q^{GN}(\neg GN_q)$$

$$= 0 + a_2 + 1 - a_2 = 1$$

♦

Lemma 5.1: *If P_i and $\neg P_i$ (i=1,…,n) are propositions, which denote whether some property P holds for some element L_i of set S, with n=|S|, and are described by subsets of the frame of discernment θ, and according to Dempster-Shafer Theory there are m_1,…, m_n functions, which assign basic probability to the property of elements L_1,…,L_n, and therefore to the subsets of θ that describe P_i,…, P_n respectively, and can be combined using the rule of the orthogonal sum with $k_0 = 0$, the total belief in the disjunction of P_i, i.e., in the truth of one at least P_i, Bel($\vee_{i=1,\ldots,n} P_i$), and in the conjunction of P_i, i.e., in the non truth of all P_i, Bel($\wedge_{i=1,\ldots,n} \neg P_i$), are measured by the following functions:*

$$Bel(\vee_{i=1,\ldots,n} P_i) = \sum_{I \subseteq \{1,\ldots,n\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} m_i(P_i) \right\}$$

$$Bel(\wedge_{i=1,\ldots,n} \neg P_i) = \prod_{i=1,\ldots,n} \{ m_i(\neg P_i) \}$$

Proof: The belief function Bel in the lemma must be obtained by combining the BPAs m_1, …, m_n which are associated with P_1, …, P_n. The combination of the basic probability assignments m_i (i=1,…,n) requires their mapping on a common frame of discernment. Assume that the common frame of discernment for combining m_1, …, m_n is θ. Suppose also that θ is defined as a set of vectors of Boolean variables of the form [p_1, p_2, …, p_n], where the Boolean variable p_i in each vector denotes whether property P_i holds for element e_i or does not by taking the values *True* or *False* respectively. Furthermore, suppose that by convention a vector denotes the conjunction of the propositions expressed by its variables and a set of vectors denotes the disjunction of the propositions that are represented by its elements. The frame of discernment θ will contain 2^n vectors to denote all the different combinations of values of p_1, p_2, …, p_n.

Given the assumptions about the construction of the frame of discernment θ, the *focals* P_i, $\neg P_i$ and $P_i \vee \neg P_i$ of each of the basic probability assignments m_i will correspond to the following subsets of θ:

- P_i will correspond to $\{[p_1, \ldots, p_n] \mid p_i = \text{True}\}$ referred to as P_i henceforth
- $\neg P_i$ will correspond to $\{[p_1, \ldots, p_n] \mid p_i = \text{False}\}$ referred to as P_i' henceforth
- $P_i \vee \neg P_i$ will correspond to $\{[p_1, \ldots, p_n] \mid p_i = \text{True or } p_i = \text{False}\}$ which is equal to θ

Having established the common frame of discernment θ for combining $m_1, \ldots, m_n$, and assuming that $m_1, \ldots, m_n$ can be combined by using the simplified version of the rule of the orthogonal sum with $k_0 = 0$:

$m(P) = m_i \oplus m_j(P) = \Sigma_{X \cap Y = P} m_i(X) \times m_j(Y)$

(T5.1.1)

we can now prove this lemma by induction on n or, equivalently, by proving first the case for n=2, assuming that the lemma holds for n=k and proving finally that the lemma also holds for n=k+1.

For n=2, we have that:

Given the frame of discernement θ that we introduced above, the focals of the BPAs m_1 and m_2 and the basic probability measures that m_1 and m_2 will assign to them will be:

m_1:

$P_1 = \{[p_1, p_2] \mid p_1 = \text{True}\}$ with basic probability measure $m_1(P_1)$

$P_1' = \{[p_1, p_2] \mid p_1 = \text{False}\}$ with basic probability measure $m_1(P_1')$

$P_1 \cup P_1' = \{[p_1, p_2] \mid p_1 = \text{True or } p_1 = \text{False}\}$ with basic probability measure

$$m_1(P_1 \cup P_1') = (1 - m_1(P_1) - m_1(P_1'))$$

m_2:

$P_2 = \{[p_1, p_2] \mid p_2 = \text{True}\}$ with basic probability measure $m_2(P_2)$

$P_2' = \{[p_1, p_2] \mid p_2 = \text{False}\}$ with basic probability measure $m_2(P_2')$

$P_2 \cup P_2' = \{[p_1, p_2] \mid p_2 = \text{True or } p_2 = \text{False}\}$ with basic probability measure

$$m_2(P_2 \cup P_2') = (1 - m_2(P_2) - m_2(P_2'))$$

Thus from (T5.1.1), we have that the combination of m_1 and m_2 will provide basic probability assignments to the following subsets of θ:

$P_1 \cap P_2$: $m_1(P_1) \times m_2(P_2)$

$P_1 \cap P_2'$: $m_1(P_1) \times m_2(P_2')$

$P_1 \cap (P_2 \cup P_2') = P_1$: $m_1(P_1) \times (1 - m_2(P_2) - m_2(P_2'))$

$P_1' \cap P_2$: $m_1(P_1') \times m_2(P_2)$

$P_1' \cap P_2'$: $m_1(P_1') \times m_2(P_2')$

$P_1' \cap (P_2 \cup P_2') = P_1'$: $m_1(P_1') \times (1 - m_2(P_2) - m_2(P_2'))$

$(P_1 \cup P_1') \cap P_2 = P_2$: $(1 - m_1(P_1) - m_1(P_1')) \times m_2(P_2)$

$(P_1 \cup P_1') \cap P_2' = P_2'$: $(1 - m_1(P_1) - m_1(P_1')) \times m_2(P_2')$

$(P_1 \cup P_1') \cap (P_2 \cup P_2') = \theta$: $(1 - m_1(P_1) - m_1(P_1')) \times (1 - m_2(P_2) - m_2(P_2'))$

Having obtained the focals of $m_1 \oplus m_2$, we then obtain $Bel(EX_1 \vee EX_2)$ or, equivalently, $Bel(EX_1 \cup EX_2)$ from axiom Axiom 5 of the Dempster Shafer theory (Section 3.4), i.e. the formula $Bel(A) = \Sigma_{B \subseteq A} m(B)$. By applying this formula, we will have (assuming that $m = m_1 \oplus m_2$):

$$\begin{aligned}
Bel(P_1 \cup P_2) &= \Sigma_{B \text{ where } B \subseteq (P_1 \cup P_2)} m_1 \oplus m_2 (B) \\
&= m(P_1 \cap P_2) + m(P_1 \cap P_2') + m(P_1) + m(P_1' \cap P_2) + m(P_2) \\
&= m_1(P_1) \times m_2(P_2) + \\
&\quad m_1(P_1) \times m_2(P_2') + \\
&\quad m_1(P_1) \times (1 - m_2(P_2) - m_2(P_2')) + \\
&\quad m_1(P_1') \times m_2(P_2) + \\
&\quad ((1 - m_1(P_1) - m_1(P_1')) \times m_2(P_2) \\
&= m_1(P_1) \times m_2(P_2) + \\
&\quad m_1(P_1) \times m_2(P_2') + \\
&\quad m_1(P_1) - m_1(P_1) \times m_2(P_2) - m_1(P_1) \times m_2(P_2')) +
\end{aligned}$$

$$m_1(P_1') \times m_2(P_2) +$$

$$m_2(P_2) - m_1(P_1) \times m_2(P_2) - m_1(P_1') \times m_2(P_2)$$

$$= m_1(P_1) + m_2(P_2) - m_1(P_1) \times m_2(P_2)$$

Also,

$$Bel(P_1' \cap P_2') = \Sigma_{B \text{ where } B \subseteq (P1' \cap P2')} m_1 \oplus m_2(B)$$

$$= m(P_1' \cap P_2')$$

$$= m_1(P_1') \times m_2(P_2')$$

Thus the lemma holds for n=2.

For n=k, we assume that the lemma holds or, equivalently, that

$$Bel\left(\bigvee_{i=1,\ldots,k} P_i\right) = Bel\left(\bigcup_{i=1,\ldots,k} P_i\right)$$

$$= \sum_{I \subseteq \{1,\ldots,k\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} \left\{ m_i(P_i) \right\} \right\}$$

and

$$Bel\left(\bigwedge_{i=1,\ldots,k} \neg P_i\right) = Bel\left(\bigcap_{i=1,\ldots,k} P_i'\right)$$

$$= \prod_{i \in \{1,\ldots,k\}} \left\{ m_i(P_i') \right\}$$

Then, for n=k+1, the lemma can be proven as follows.

From Theorem 3.4 in [146] (p. 63), we have that the combination of BPAs $m_1 \oplus m_2 \oplus \ldots \oplus m_k \oplus m_{k+1} = (m_1 \oplus m_2 \oplus \ldots \oplus m_k) \oplus m_{k+1}$. Thus, if we assume that $m_T^k = m_1 \oplus m_2 \oplus \ldots \oplus m_k$ there will be that $m_1 \oplus m_2 \oplus \ldots \oplus m_k \oplus m_{k+1} = m_T^k \oplus m_{k+1}$. To combine m_T^k and m_{k+1}, we can consider the intersections of the focal elements of interest of the two functions. Let also $P_T^k = \left\{ \bigcup_{i=1,\ldots,k} (P_i) \right\}$ and $P_T^{k'} = \left\{ \bigcap_{i=1,\ldots,k} (P_i') \right\}$. Then the combinations of the focals of interest of m_T^k and m_{k+1}^{EX} will be:

$P_T^k \cap P_{k+1}$: $m_T^k(P_T^k) \times m_{k+1}(P_{k+1})$

$P_T^k \cap P_{k+1}'$: $m_T^k(P_T^k) \times m_{k+1}(P_{k+1}')$

$P_T^k \cap (P_{k+1} \cup P_{k+1}') = P_T^k$: $m_T^k(P_T^k) \times (1 - m_{k+1}(P_{k+1}) - m_{ijk+1}(P_{ijk+1}'))$

$P_T^{k,} \cap P_{k+1}$: $m_T^k(P_T^{k,}) \times m_{k+1}(P_{k+1})$

$P_T^{k,} \cap P_{k+1}'$: $m_T^k(P_T^{k,}) \times m_{k+1}(P_{k+1}')$

$P_T^{k,} \cap (P_{k+1} \cup P_{k+1}') = P_T^{k,}$: $m_T^k(P_T^{k,}) \times (1 - m_{k+1}(P_{k+1}) - m_{k+1}(P_{k+1}'))$

$(P_T^k \cup P_T^{k,}) \cap P_{k+1} = P_{k+1}$: $(1 - m_T^k(P_T^k) - m_T^k(P_T^{k,})) \times m_{k+1}(P_{k+1})$

$(P_T^k \cup P_T^{k,}) \cap P_{k+1}' = P_{k+1}'$: $(1 - m_T^k(P_T^k) - m_T^k(P_T^{k,})) \times m_{k+1}(P_{k+1}')$

$(P_T^k \cup P_T^{k,}) \cap (P_{k+1} \cup P_{k+1}') = \theta$: $(1 - m_T^k(P_T^k) - m_T^k(P_T^{k,})) \times$
$(1 - m_{k+1}(P_{k+1}) - m_{k+1}(P_{k+1}'))$

Thus, for $Bel(\bigvee_{i=1,\ldots,k+1} EX_i)$ we will have that:

$$
\begin{aligned}
Bel(\bigvee_{i=1,\ldots,k+1} P_i) &= Bel(\cup_{i=1,\ldots,k+1} P_i) \\
&= Bel(\{\cup_{i=1,\ldots,k} P_i\} \cup P_{k+1}) \\
&= Bel(P_T^k \cup P_{k+1}) \\
&= \sum_{B \text{ where } B \subseteq (P_T^k \cup Pk+1)} m_T^k \oplus m_{k+1}(B) \\
&= m_T^k(P_T^k) \times m_{k+1}(P_{k+1}) + \\
&\quad m_T^k(P_T^k) \times m_{k+1}(P_{k+1}') + \\
&\quad m_T^k(P_T^k) \times (1 - m_{k+1}(P_{k+1}) - m_{k+1}(P_{k+1}')) + \\
&\quad m_T^k(P_T^{k,}) \times m_{k+1}(P_{k+1}) + \\
&\quad ((1 - m_T^k(P_T^k) - m_T^k(P_T^{k,})) \times m_{k+1}(P_{k+1}) \\
&= m_T^k(P_T^k) + m_{k+1}(P_{k+1}) - m_T^k(P_T^k) \times m_{k+1}(P_{k+1})
\end{aligned}
$$

Then, since

$$
\begin{aligned}
m_T^k(P_T^k) &= m_T^k(\{\bigcup_{i=1,\ldots,k}(P_i)\}) \\
&= \sum_{I \subseteq \{1,\ldots,k\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \{\prod_{i \in I} \{ m_i(P_i) \}\}
\end{aligned}
$$

we will have that,

$Bel(\bigvee_{i=1,\ldots,k+1} P_i) =$

$$= \sum_{I\subseteq\{1,\ldots,k\} \text{ and } I\neq\emptyset} (-1)^{|I|+1}\left\{\prod_{i\in I}\left\{ m_i(P_i)\right\}\right\} + m_{k+1}(P_{k+1}) -$$

$$\sum_{I\subseteq\{1,\ldots,k\} \text{ and } I\neq\emptyset} (-1)^{|I|+1}\left\{\prod_{i\in I}\left\{ m_i(P_i)\right\}\right\}\times m_{k+1}(P_{k+1})$$

$$= \sum_{I\subseteq\{1,\ldots,k\} \text{ and } I\neq\emptyset} (-1)^{|I|+1}\left\{\prod_{i\in I}\left\{ m_i(P_i)\right\}\right\} +$$

$$\sum_{I=\{k+1\}}(-1)^{|I|+1}\left\{\prod_{i\in I}\left\{ m_i(P_i)\right\}\right\} -$$

$$\sum_{I\subseteq\{1,\ldots,k\} \text{ and } I\neq\emptyset} (-1)^{|I|+1}\left\{\prod_{i\in I}\left\{ m_{k+1}(P_{k+1})\times m_i(P_i)\right\}\right\}$$

$$= \sum_{I\subseteq\{1,\ldots,k\} \text{ and } I\neq\emptyset} (-1)^{|I|+1}\left\{\prod_{i\in I}\left\{ m_i(P_i)\right\}\right\} +$$

$$\sum_{I=\{k+1\}}(-1)^{|I|+1}\left\{\prod_{i\in I}\left\{ m_i(P_i)\right\}\right\} -$$

$$\sum_{I'=I\cup\{k+1\} \text{ and } I\subseteq\{1,\ldots,k\} \text{ and } I\neq\emptyset} (-1)^{|I\cup\{k+1\}|+1}\left\{\prod_{i\in I'}\left\{ m_i(P_i)\right\}\right\}$$

In the above sum however,

- the item $\sum_{I\subseteq\{1,\ldots,k\} \text{ and } I\neq\emptyset} (-1)^{|I|+1}\left\{\prod_{i\in I}\left\{ m_i(P_i)\right\}\right\}$ covers all the subsets of $\{1,\ldots,k+1\}$ that do not include the element k+1
- the item $\sum_{I=\{k+1\}}(-1)^{|I|+1}\left\{\prod_{i\in I}\left\{ m_i(P_i)\right\}\right\}$ covers only the singleton subset $\{k+1\}$ of $\{1,\ldots,k+1\}$
- finally, the item $\sum_{I'=I\cup\{k+1\} \text{ and } I\subseteq\{1,\ldots,k\} \text{ and } I\neq\emptyset} (-1)^{|I\cup\{k+1\}|+1}\left\{\prod_{i\in I'}\left\{ m_i(P_i)\right\}\right\}$ covers all the subsets of $\{1,\ldots,k+1\}$ that include the element k+1 except from the singleton set $\{k+1\}$

Thus,

$$\text{Bel}(\bigvee_{i=1,\ldots,k+1}P_i) = \text{Bel}(\cup_{i=1,\ldots,k+1}P_i)$$

$$= \sum_{I\subseteq\{1,\ldots,k+1\}\text{and } I\neq\emptyset}(-1)^{|I|+1}\left\{\prod_{i\in I}\left\{ m_i(P_i)\right\}\right\}$$

Also for $\text{Bel}(\bigwedge_{i=1,\ldots,k+1}P_i')$ we will have that

$$\text{Bel}(\bigwedge_{i=1,\ldots,k+1} P_i') = \text{Bel}(\bigcap_{i=1,\ldots,k+1} P_i')$$

$$= \text{Bel}(P_T^{k'} \cap P_{k+1}')$$

$$= \sum_{B \text{ where } B \subseteq (P_T^{k'} \cap Pk+1')} m_T^k \oplus m_{k+1}(B)$$

$$= m_T^k (P_T^{k'}) \times m_{k+1}(P_{k+1}')$$

Then since,

$$m_T^k(P_T^{k'}) = m_T^k\left(\left\{\bigcap_{i=1,\ldots,k} P_i'\right\}\right)$$

$$= \prod_{i \in \{1,\ldots,k\}} \left\{ m_i(P_i') \right\}$$

we will have that,

$$\text{Bel}(\bigwedge_{i=1,\ldots,k+1} P_i') = \prod_{i \in \{1,\ldots,k\}} \left\{ m_i(P_i') \right\} \times m_{(+1}(P_{k+1}')$$

$$= \prod_{i \in \{1,\ldots,k+1\}} \left\{ m_i(P_i') \right\}$$

Thus,

$$\text{Bel}(\bigwedge_{i=1,\ldots,k+1} P_i') = \text{Bel}(\bigcap_{i=1,\ldots,k+1} P_i')$$

$$= \prod_{i \in \{1,\ldots,k+1\}} \left\{ m_i(\neg P_i') \right\}$$

♦

Theorem 5.2: *If e is an event and U(e,TR) is the set of the events that are recorded in the log of the monitoring framework and can be unified with e, and it holds that U(e,TR)≠∅ with n = |U(e,TR)|, i.e. the number of the members of U(e,TR), the belief in the explainability of at least one recorded event in U(e,TR),* $Bel(\bigvee_{i=1,\ldots,n} Explainable(e_i,U_o,TR))$*, and in the explainability of none of the events in U(e,TR),* $Bel(\bigwedge_{i=1,\ldots,n} \neg Explainable(e_i,U_o,TR))$*, are measured by the following functions:*

$$Bel(\bigvee_{i=1,\ldots,n} Explainable(e_i,U_o,TR)) =$$

$$= \sum_{I \subseteq \{1,\ldots,n\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} m_i^{EX}(Explainable(e_i, U_o, TR)) \right\}$$

$$Bel(\bigwedge_{i=1,\ldots,n} \neg Explainable(e_i, U_o, TR)) =$$

$$= \prod_{i=1,\ldots,n} \{ m_i^{EX}(\neg Explainabe(e_i, U_o, TR)) \}$$

where m_i^{EX} (i=1,…, n) is the basic probability assignment associated with the event e_i.

Proof: The belief function Bel in the theorem must be obtained by combining the BPAs m_1^{EX}, …, m_n^{EX} which are associated with e_1, …, e_n. The combination of the basic probability assignments m_i^{EX} (i=1,…,n) requires their mapping on a common frame of discernment, i.e., a set of mutually exclusive propositions representing exhaustively the properties that m_i^{EX} assign belief to. This common frame of discernment has been defined as θ_{es}^{EX} (see *Definition 6*) in Section 5.4.6.1.

Given the assumptions about the construction of the frame of discernment θ_{es}^{EX}, the *focals* EX_i, $\neg EX_i$ and $EX_i \vee \neg EX_i$ of each of the basic probability assignments m_i^{EX} will correspond to the following subsets of θ_{es}^{EX}:

- EX_i will correspond to $\{[E_1, \ldots, E_n] \mid E_i = True\}$ referred to as EX_i henceforth
- $\neg EX_i$ will correspond to $\{[E_1, \ldots, E_n] \mid E_i = False\}$ referred to as EX_i' henceforth
- $EX_i \vee \neg EX_i$ will correspond to $\{[E_1, \ldots, E_n] \mid E_i = True \text{ or } E_i = False\}$ which is equal to θ_{es}^{EX}

Thus, the *core* (see page 40 in [146]) of each m_i^{EX} will be $C_i = EX_i \cup EX_i' \cup \theta_{es}^{EX} = \theta_{es}^{EX}$ for all *i* (i=1, …, n) and:

$$\cap_{i=1,\ldots,n} C_i = \theta_{es}^{EX} \neq \emptyset \qquad \text{(T5.2.1)}$$

Due to (T5.2.1) and Theorem 3.2 (see page 40 in [146]), it follows that the basic probability assignments m_i^{EX} (i=1,…,n) can be combined.

Furthermore assuming that S_i^+ represents the focal set EX_i of the basic probability assignment m_i^{EX}, we observe that

$$\forall\, I \subseteq \{1,2,\ldots,n\} \cap_{i \varepsilon I} S_i^+ = \{[E_1, \ldots, E_{|I|}] \mid \forall\, i \in I\ E_i = True\} \neq \emptyset$$

Similarly, assuming that S_i^- represents the focal set EX_i' of the basic probability assignment m_i^{EX}, we observe that

$\forall I \subseteq \{1,2,\ldots,n\} \cap_{i\varepsilon I} S_i^- = \{[E_1, \ldots, E_{|I|}] \mid \forall i \in I\ E_i=\text{False}\} \neq \varnothing$

Finally, assuming that S_i^{θ} represents the focal set $EX_i \vee \neg EX_i$ of the basic probability assignment m_i^{EX}, we observe that

$\forall I \subseteq \{1,2,\ldots,n\} \cap_{i\varepsilon I} S_i^{\theta} = \theta_{es}{}^{EX} \neq \varnothing$

Thus, in general, assuming that S_i represents one of the three possible focal sets of the basic probability assignment m_i^{EX} we will also have that

$\forall I \subseteq \{1,2,\ldots,n\} \cap_{i\varepsilon I} S_i \neq \varnothing$ (T5.2.2)

Subsequently from (T5.2.2), we have that:

$\Sigma_{i\varepsilon I,\, j\varepsilon I,\, i \neq j} \Sigma_{Si \cap Sj=\varnothing}\, m_i(S_i) \times m_j(S_j) = 0$ (T5.2.3)

Therefore, according to Theorem 3.1 (see page 60 in [146]), the basic probability assignments m_i^{EX} can be combined using the rule of the orthogonal sum (defined by Axiom 9 in Section 3.4) which, since $k_0 = 0$ due to (T5.2.3) above, is simplified to the following formula:

$m^{EX}(P) = m_i^{EX} \oplus m_j^{EX}(P) = \Sigma_{X \cap Y = P}\, m_i^{EX}(X) \times m_j^{EX}(Y)$ (T5.2.4)

Having established the common frame of discernment for combining $m_1^{EX}, \ldots, m_n^{EX}$, and the simplified version of the rule of the orthogonal sum for obtaining the combination $m_1^{EX} \oplus m_2^{EX} \oplus \ldots \oplus m_n^{EX}$, the theorem holds due to *Lemma 5.1*, by using the following substitutions in *Lemma 5.*1: S := U(e,TR), $L_i := e_i$, P_i := Explainable(e_i,U_o,TR), $\neg P_i$:= ¬Explainable(e_i,U_o,TR), $\theta := \theta_{es}{}^{EX}$, $p_i := E_i$, and $m_i := m_i^{EX}$.

♦

Lemma 5.2: *If* $\forall i \in \{1,2,\ldots,n\}$, $0 \leq x_i \leq 1$, *it holds that*

$$0 \leq \sum_{I \subseteq \{1,\ldots,n\} \text{ and } I \neq \varnothing} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} \leq 1$$

Proof: This lemma can be proven by induction on n.

For n=2 we have

i) $x_1 + x_2 - x_1x_2 \geq 0 \Rightarrow x_1(1- x_2) \geq - x_2$ (L5.2.1)

However

$x_2 \geq 0 \Rightarrow - x_2 \leq 0$ (L5.2.2)

and

$x_2 \leq 1 \Rightarrow 0 \leq 1 - x_2$ (L5.2.3)

From (L5.2.2) and (L5.2.3), (L5.2.1) holds due to the fact that x_1 and $(1- x_2)$ are individually equal to or greater than 0, and thus their product is equal to or greater than 0 and consequently greater than $- x_2$, which is less than or equal to 0.

ii) $x_1 + x_2 - x_1x_2 \leq 1 \Rightarrow x_1(1- x_2) \leq 1 - x_2 \Rightarrow x_1(1- x_2) – (1 - x_2) \leq 0 \Rightarrow$

$(x_1 - 1)(1 - x_2) \leq 0$ (L5.2.4)

However

$x_1 \leq 1 \Rightarrow x_1 - 1 \leq 0$ (L5.2.5)

From (L5.2.5) and (L5.2.2), (L5.2.4) holds due to the fact that $(x_1 - 1)$ is less than or equal to 0 while $(1- x_2)$ is equal to or greater than 0, and thus their product is less than or equal to 0.

Thus the lemma holds for n=2.

For n=k, we assume that the lemma holds or, equivalently, that

$$0 \leq \sum_{I \subseteq \{1,\ldots,k\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} \leq 1 \quad \text{(L5.2.6)}$$

Then, for n=k+1, the lemma can be proven as follows.

$$\sum_{I \subseteq \{1,\ldots,k+1\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} =$$

$$= \sum_{I \subseteq \{1,\ldots,k\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} + x_{k+1} -$$

$$\left[\sum_{I \subseteq \{1,\ldots,k\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} \right] \times x_{k+1} \quad \text{(L5.2.7)}$$

Assuming that $X = \sum_{I \subseteq \{1,\ldots,k\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\}$, with $0 \leq X \leq 1$from (L5.2.6), we have for (L5.2.7)

$$\Sigma_{I \subseteq \{1,\ldots,k+1\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} = X + x_{k+1} - X \times x_{k+1}$$

(L5.2.8)

However, due to the fact that $0 \leq X \leq 1$ and $0 \leq x_{k+1} \leq 1$, as we have shown in the case n=2, it holds that $0 \leq X + x_{k+1} - X \times x_{k+1} \leq 1$.

Thus, it holds that

$$0 \leq \Sigma_{I \subseteq \{1,\ldots,k+1\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} \leq 1$$

♦

Lemma 5.3: *If $\forall i \in \{1,2,\ldots,n\}$, $0 \leq x_i \leq 1$, it holds that*

$$0 \leq \Pi_{i \in \{1,\ldots,n\}} x_i \leq 1$$

Proof: This lemma can be proven by induction on n.

<u>For n=2</u> we have

i) The inequality $x_1 x_2 \geq 0$ holds due to the fact that x_1 and x_2 are individually equal to or greater than 0, and thus their product is equal to or greater than 0 as well.

ii) $x_1 x_2 \leq 1 \Rightarrow x_1 x_2 \leq x_2 + 1 - x_2 \Rightarrow x_2(x_1 - 1) \leq 1 - x_2$ (L5.3.1)

However

$x_1 \leq 1 \Rightarrow x_1 - 1 \leq 0$ (L5.3.2)

and

$x_2 \leq 1 \Rightarrow 1 - x_2 \geq 0$ (L5.3.3)

Due to the fact that x_2 is equal to or greater than 0, and we have from (L5.3.2) that $x_1 - 1$ is less than or equal to 0, the product $x_2(x_1 - 1)$ is less than or equal to 0, and consequently less than $1 - x_2$, which is equal to or greater than 0 due to (L5.3.3). Thus, (L5.3.1) holds.

Thus the lemma holds for n=2.

<u>For n=k</u>, we assume that the lemma holds or, equivalently, that

$$0 \leq \prod_{i \in \{1,\ldots,k\}} x_i \leq 1$$ (L5.3.4)

Then, for n=k+1, the lemma can be proven as follows.

$$\prod_{i \in \{1,\ldots,k+1\}} x_i = (x_1 x_2 \ldots x_k) x_{k+1}$$

$$= \left[\prod_{i \in \{1,\ldots,k\}} x_i\right] \times x_{k+1} \qquad \text{(L5.3.5)}$$

However, due to the fact that we have from (L2.4) that $0 \leq \prod_{i \in \{1,\ldots,k\}} x_i \leq 1$ and $0 \leq x_{k+1} \leq 1$, as we have shown in the case n=2, it holds that $0 \leq \left[\prod_{i \in \{1,\ldots,k\}} x_i\right] \times x_{k+1} \leq 1$.

Thus, it holds that

$$0 \leq \prod_{i \in \{1,\ldots,k+1\}} x_i \leq 1$$

♦

Lemma 5.4: *If $\forall i \in \{1,2,\ldots,n\}$, $0 \leq x_i \leq 1$, $0 \leq x_i' \leq 1$, and $0 \leq x_i + x_i' \leq 1$, it holds that*

$$0 \leq \sum_{I \subseteq \{1,\ldots,n\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{\prod_{i \in I} x_i\right\} + \prod_{i \in \{1,\ldots,n\}} x_i' \leq 1$$

Proof: This lemma can be proven by induction on n.

For n=2 we have

i) $x_1 + x_2 - x_1 x_2 + x_1' x_2' \geq 0$ (L5.4.1)

However, as we have shown in *Lemma 1*, we observe that

$0 \leq x_1 + x_2 - x_1 x_2 \leq 1$ (L5.4.2)

and as we have shown in *Lemma 2*, we observe that

$0 \leq x_1' x_2' \leq 1$ (L5.4.3)

From (L5.4.2) and (L5.4.3), (L5.4.1) holds due to the fact that $x_1 + x_2 - x_1 x_2$ and $x_1' x_2'$ are individually equal to or greater than 0 and less than or equal to 1, and thus their sum is equal to or greater than 0.

ii) Before proving the inequality

$x_1 + x_2 - x_1 x_2 + x_1' x_2' \leq 1$ (L5.4.4)

we know that

$$x_1 + x_1' \leq 1 \Rightarrow x_1' \leq 1 - x_1 \qquad \text{(L5.4.5)}$$

and also

$$x_2 + x_2' \leq 1 \Rightarrow x_2' \leq 1 - x_2 \qquad \text{(L5.4.6)}$$

However, we observe from (L5.4.5) and (L5.4.6) that

$$x_1 + x_2 - x_1x_2 + x_1'x_2' \leq x_1 + x_2 - x_1x_2 + (1 - x_1)(1 - x_2) \qquad \text{(L5.4.7)}$$

Thus, we can prove (L5.4.4) by proving that

$$x_1 + x_2 - x_1x_2 + (1 - x_1)(1 - x_2) \leq 1 \qquad \text{(L5.4.8)}$$

Indeed, we have for (L5.4.8) that

$$x_1 + x_2 - x_1x_2 + (1 - x_1)(1 - x_2) \leq 1 \Rightarrow x_1 + x_2 - x_1x_2 + 1 - x_2 - x_1 + x_1x_2 \leq 1 \Rightarrow$$

$$\Rightarrow 1 \leq 1 \qquad \text{(L5.4.9)}$$

The inequality (L5.4.9) holds, consequently (L5.4.4) holds.

Thus the lemma holds for n=2.

For n=k, we assume that the lemma holds or, equivalently, that

$$0 \leq \sum_{I \subseteq \{1,\ldots,k\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} + \prod_{i \in \{1,\ldots,k\}} x_i' \leq 1 \qquad \text{(L5.4.10)}$$

Then, for n=k+1, the lemma can be proven as follows.

$$\sum_{I \subseteq \{1,\ldots,k+1\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} + \prod_{i \in \{1,\ldots,k+1\}} x_i' =$$

$$= \left[\sum_{I \subseteq \{1,\ldots,k\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} + x_{k+1} - \right.$$

$$\left. \left[\sum_{I \subseteq \{1,\ldots,k\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} \right] \times x_{k+1} \right] +$$

$$\left[\left\{ \prod_{i \in \{1,\ldots,k\}} x_i' \right\} \times x_{k+1}' \right] \qquad \text{(L5.4.11)}$$

Assuming that $X = \sum_{I \subseteq \{1,\ldots,k\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\}$, with $0 \leq X \leq 1$from *Lemma 5.2*, and $X' = \prod_{i \in \{1,\ldots,k\}} x_i'$, with $0 \leq X' \leq 1$ from *Lemma 5.3*, we have for (L5.4.11) that

$$\sum_{I \subseteq \{1,\ldots,k+1\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} + \prod_{i \in \{1,\ldots,k+1\}} x_i' =$$

$$= X + x_{k+1} - X \times x_{k+1} + X' \times x_{k+1}' \qquad \text{(L5.4.12)}$$

However, due to the fact that the following inequalities hold

$0 \leq X \leq 1$

$0 \leq X' \leq 1$

$0 \leq x_{k+1} \leq 1$

$0 \leq x_{k+1}' \leq 1$

$0 \leq x_{k+1} + x_{k+1}' \leq 1$

and also we have from (L5.4.10) that

$0 \leq X + X' \leq 1$

Thus, as we have shown in the case n=2, it holds that

$0 \leq X + x_{k+1} - X \times x_{k+1} + X' \times x_{k+1}' \leq 1$

Consequently, it holds that

$$0 \leq \sum_{I \subseteq \{1,\ldots,k+1\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} x_i \right\} + \prod_{i \in \{1,\ldots,k+1\}} x_i' \leq 1$$

♦

Theorem 5.3: *The evidence measure m^{GN} defined as:*

$$m^{GN}(P) = \begin{cases} Bel(\bigvee_{i=1,\ldots,n} Explainable(e_i,U_o,TR)), \text{ if } P = Genuine(e,U_o,TR) \\ Bel(\bigwedge_{i=1,\ldots,n} \neg Explainable(e_i,U_o,TR)), \text{ if } P = \neg Genuine(e,U_o,TR) \\ 1 - Bel(\bigvee_{i=1,\ldots,n} Explainable(e_i,U_o,TR)) - Bel(\bigwedge_{i=1,\ldots,n} \neg Explainable(e_i,U_o,TR)), \\ \text{otherwise} \end{cases}$$

where n = |U(e,TR)|, i.e., the number of the the matching recorded events of e, is a DS basic probability assignment with respect to frames of discerment θ_{es} and θ_{es}^{GN} *(see Definitions 5 and 8 repectively in Section 5.4.6.1).*

Proof: To prove that m^{GN} is a DS basic probability assignment it is sufficient that m^{GN} satisfies the axioms Axiom 1, Axiom 2, and Axiom 3 of *DS Theory* (see Section 3.4).

(Axiom 1): Regarding θ_{es}, Axiom 1 is satisfied when:

i) If $P = GN_s$, then it must hold that $0 \leq m_s^{GN}(P) \leq 1$, or equivalently:

$$0 \leq Bel(\bigvee_{i=1,\ldots,n} Explainable(e_i,U_o,TR)) \leq 1 \quad (T.5.3.1)$$

From *Theorem 5.2*, we have that:

$$Bel(\bigvee_{i=1,\ldots,n} Explainable(e_i,U_o,TR)) =$$

$$= \sum_{I \subseteq \{1,\ldots,n\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} m_i^{EX}(Explainable(e_i,U_o,TR)) \right\} \quad (T.5.3.2)$$

Thus, by using (T.5.3.2), we have equivalently for (T.5.3.1):

$$0 \leq \sum_{I \subseteq \{1,\ldots,n\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} m_i^{EX}(Explainable(e_i,U_o,TR)) \right\} \leq 1$$

(T.5.3.3) However, from *Theorems 5.4, and 5.6*, we have that m_i^{EX} is *DS basic probability assignment*, and therefore, m_i^{EX} satisfies Axiom 1 of *DS Theory*, or equivalently:

$$\forall i \in \{1,\ldots,n\}\ , 0 \leq m_i^{EX} \leq 1 \quad (T.5.3.4)$$

Therefore, from (T.5.3.4) and by substituting x_i with $m_i^{EX}(Explainable(e_i,U_o,TR))$ in *Lemma 5.2*, the inequality (T.5.3.3) holds. Thus, (T.5.3.1) holds.

ii) If $P = \neg GN_s$, then it must hold that $0 \leq m_s^{GN}(P) \leq 1$, or equivalently:

$$0 \leq Bel(\bigwedge_{i=1,\ldots,n} \neg Explainable(e_i,U_o,TR)) \leq 1 \quad (T.5.3.5)$$

From *Theorem 5.2*, we have that:

$$Bel(\bigwedge_{i=1,\ldots,n} \neg Explainable(e_i,U_o,TR)) =$$

$$= \prod_{i=1,\ldots,n} \{m_i^{EX}(\neg Explainabe(e_i,U_o,TR))\} \quad (T.5.3.6)$$

Thus, by using (T.5.3.6), we have equivalently for (T.5.3.5):

$$0 \leq \prod_{i=1,\ldots,n} \{m_i^{EX}(\neg Explainabe(e_i,U_o,TR))\} \leq 1 \quad (T.5.3.7)$$

However, from (T.5.3.4) and by substituting x_i' with $m_i^{EX}(\neg Explainabe(e_i,U_o,TR))$ in *Lemma 5.3*, the inequality (T.5.3.7) holds. Thus, (T.5.3.5) holds.

iii) If $P \neq GN_s$ or $P \neq \neg GN_s$, then it must hold that $0 \leq m_s^{GN}(P) \leq 1$, or equivalently:

$$0 \leq Bel(\bigvee_{i=1,\ldots,n} Explainable(e_i,U_o,TR)) + Bel(\bigwedge_{i=1,\ldots,n} \neg Explainable(e_i,U_o,TR)) \leq 1 \quad \text{(T.5.3.8)}$$

From (T.5.3.2) and (T.5.3.6), we have equivalently for (T.5.3.8):

$$0 \leq \sum_{I \subseteq \{1,\ldots,n\} \text{ and } I \neq \emptyset} (-1)^{|I|+1} \left\{ \prod_{i \in I} m_i^{EX}(Explainable(e_i,U_o,TR)) \right\} + \prod_{i=1,\ldots,n} \{ m_i^{EX}(\neg Explainabe(e_i,U_o,TR)) \} \leq 1 \quad \text{(T.5.3.9)}$$

However, from (T.5.3.4) and by substituting x_i with $m_i^{EX}(Explainable(e_i,U_o,TR))$ and x_i' with $m_i^{EX}(\neg Explainable(e_i,U_o,TR))$ in *Lemma 5.3*, the inequality (T.5.3.9) holds. Thus, (T.5.3.8) holds.

Similarly, Axiom 1 is satisfied with respect to θ_{es}^{GN}, for the cases that $P = GN_q$, $P = \neg GN_q$, and $P \neq GN_q$ or $P \neq \neg GN_q$.

(Axiom 2): Regarding θ_{es}, Axiom 2 is satisfied by m_s^{GN} since its focals GN_s, GN_s' and $GN_s \cup GN_s'$ are non empty sets by definition of θ_{es}:

$GN_s = \{[G_s = True]\} \neq \emptyset$,

$GN_s' = \{[G_s = False]\} \neq \emptyset$ and

$GN_s \cup GN_s' = \{[G_s = True \text{ or } False]\} \neq \emptyset$.

Thus, the basic probability assigned to the empty set by m_s^{GN} is $m_s^{GN}(\emptyset) = 0$

Similarly, Axiom 2 is satisfied by m_s^{GN} with respect to θ_{es}^{GN} since its focals GN_q, GN_q' and $GN_q \cup GN_q'$ are non empty sets by definition of θ_{es}^{GN}:

$GN_q = \{[G_1,G_2,\ldots,G_r] \mid G_q = True\} \neq \emptyset$,

$GN_q' = \{[G_1,G_2,\ldots,G_r] \mid G_q = False\} \neq \emptyset$ and

$GN_q \cup GN_q' = \{[G_q = True \text{ or } False]\} \neq \emptyset$.

Thus, the basic probability assigned to the empty set by m_q^{GN} is $m_q^{GN}(\emptyset) = 0$

(Axiom 3): Regarding θ_{es}, Axiom 3 is satisfied since:

$$\sum_{P \subseteq \theta_{e_s}} m_s^{GN}(P) = \sum_{P \subseteq \theta_{e_s} \text{ and } P \neq GN_s \text{ and } P \neq \neg GN_s} m_s^{GN}(P) + m_s^{GN}(GN_s) + m_s^{GN}(\neg GN_s)$$

= 1-Bel($\bigvee_{i=1,\ldots,n}$Explainable(e_i,U_o,TR))-Bel($\bigwedge_{i=1,\ldots,n}\neg$Explainable($e_i$,$U_o$,TR)) +

Bel($\bigvee_{i=1,\ldots,n}$ Explainable(e_i,U_o,TR)) + Bel($\bigwedge_{i=1,\ldots,n}\neg$Explainable($e_i$,$U_o$,TR)) =

=1

Similarly, Axiom 3 is satisfied with respect to θ_{es}^{GN} since:

$$\sum_{P \subseteq \theta_{e_s}^{GN}} m_q^{GN}(P) = \sum_{P \subseteq \theta_{e_s}^{GN} \text{ and } P \neq GN_q \text{ and } P \neq \neg GN_q} m_q^{GN}(P) + m_q^{GN}(GN_q) + m_q^{GN}(\neg GN_q)$$

= 1-Bel($\bigvee_{i=1,\ldots,n}$Explainable(e_i,U_o,TR))-Bel($\bigwedge_{i=1,\ldots,n}\neg$Explainable($e_i$,$U_o$,TR)) +

Bel($\bigvee_{i=1,\ldots,n}$ Explainable(e_i,U_o,TR)) + Bel($\bigwedge_{i=1,\ldots,n}\neg$Explainable($e_i$,$U_o$,TR)) =

=1

♦

Theorem 5.4: *The evidence measure m_i^{EX} defined as:*

$$m_i^{EX}(P) = \begin{cases} a_1, \text{ if } P = Explainable(e_i,U_o,TR) \\ 1 - \alpha_1, \text{ if } P = \neg Explainable(e_i,U_o,TR) \\ 0, \text{ otherwise} \end{cases}$$

where α_1 is a value within 0 and 1, is a DS basic probability assignment with respect to frame of discerment θ_{es}^{EX} (see Definitions 6 in Section 5.4.6.1).

Proof: To prove that m_i^{EX} is a DS basic probability assignment it is sufficient to show that m_i^{EX} satisfies the axioms Axiom 1, Axiom 2, and Axiom 3 of *DS Theory* (see Section 3.4).

(Axiom 1): Axiom 1 is satisfied since:

If P = EX_i, $m_i^{EX}(P) = \alpha_1$ where $0 < \alpha_1 < 1$.

If $P = \neg EX_i$, $m_i^{EX}(P) = 1 - \alpha_1$ with $0 < 1 - \alpha_1 < 1$ since $0 < \alpha_1 < 1$.

If $P \neq EX_i$ or $P \neq \neg EX_i$, $m_i^{EX}(P) = 0$ by definition.

(Axiom 2): Axiom 2 is satisfied by m_i^{EX} since its focals EX_i and EX_i' are non empty sets by definition of θ_{es}:

$EX_i = \{[E_i = True]\} \neq \varnothing$ and

$EX_i' = \{[E_i = False]\} \neq \varnothing$.

Thus, the basic probability assigned to the empty set by m_i^{EX} is $m_i^{EX}(\varnothing) = 0$

(Axiom 3): Axiom 3 is satisfied since:

$$\sum_{P \subseteq \theta_{e_s}^{EX}} m_i^{EX}(P) = \sum_{P \subseteq \theta_{e_s}^{EX} \text{ and } P \neq EX_i \text{ and } P \neq \neg EX_i} m_i^{EX}(P) + m_i^{EX}(EX_i) + m_i^{EX}(\neg EX_i)$$

$$= 0 + a_1 + 1 - a_1 = 1$$

♦

Theorem 5.5: *If e_i is an event and $EXP(e_i)$ is the set of the alternative explanations that are generated for e_i, and it holds that $EXP(e_i) \neq \varnothing$ with $m = |EXP(e_i)|$, i.e. the number of the members of $EXP(e_i)$, the belief in the validity of at least one alternative explanation in $EXP(e_i)$, $Bel(\bigvee_{j=1,\ldots,m} Valid(e_i, \Phi_j, U_o, TR))$, and in the validity of none of the alternative explanations in $EXP(e_i)$, $Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i, \Phi_j, U_o, TR))$, are measured by the following functions:*

$$Bel(\bigvee_{j=1,\ldots,m} Valid(e_i, \Phi_j, U_o, TR)) =$$

$$= \sum_{J \subseteq \{1,\ldots,m\} \text{ and } J \neq \emptyset} (-1)^{|J|+1} \left\{ \prod_{j \in J} m_j^{VL}(Valid(e_i, \Phi_j, U_o, TR)) \right\}$$

$$Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i, \Phi_j, U_o, TR)) =$$

$$= \prod_{j=1,\ldots,m} \{m_j^{VL}(\neg Valid(e_i, \Phi_j, U_o, TR))\}$$

where m_j^{VL} $(j=1,\ldots, n)$ is the basic probability assignment associated with the alternative explanation Φ_j of e_i.

Proof: The belief function Bel in the theorem must be obtained by combining the BPAs $m_1^{VL},\ldots, m_m^{VL}$ which are associated with $\Phi_1, \ldots, \Phi_m$. The combination of the basic probability assignments m_j^{VL} (j=1,…,m) requires their mapping on a common frame of discernment, i.e., a set of mutually exclusive propositions representing exhaustively the properties that m_j^{VL} assign belief to. This common frame of discernment has been defined as θ_{es}^{VL} (see *Definition 7*) in Section 5.4.6.1.

Given the assumptions about the construction of the frame of discernment θ_{es}^{VL}, the *focals* VL_j, $\neg VL_j$ and $VL_j \vee \neg VL_j$ of each of the basic probability assignments m_j^{VL} will correspond to the following subsets of θ_{es}^{VL}:

- VL_j will correspond to $\{[V_1, \ldots, V_m] \mid V_j = \text{True}\}$ referred to as VL_j henceforth
- $\neg VL_j$ will correspond to $\{[V_1, \ldots, V_m] \mid V_i = \text{False}\}$ referred to as VL' henceforth
- $VL_j \vee \neg VL_j$ will correspond to $\{[V_1, \ldots, V_m] \mid V_j = \text{True or } V_j = \text{False}\}$ which is equal to θ_{es}^{VL}

Thus, the *core* (see page 40 in [146]) of each m_j^{VL} will be $C_j = VL_j \cup VL_j' \cup \theta_{es}^{VL} = \theta_{es}^{VL}$ for all *j* (j=1,…, m) and:

$$\cap_{j=1,\ldots,m} C_j = \theta_{es}^{VL} \neq \varnothing \qquad \text{(T5.5.1)}$$

Due to (T5.5.1) and Theorem 3.2 (see page 40 in [146]), it follows that the basic probability assignments m_j^{VL} (j=1,…, m) can be combined.

Furthermore assuming that S_j^{+} represents the focal set VL_j of the basic probability assignment m_j^{VL}, we observe that

$$\forall J \subseteq \{1,2,\ldots, m\} \cap_{j \varepsilon J} S_j^{+} = \{[V_1, \ldots, V_{|J|}] \mid \forall j \in J\ V_j = \text{True}\} \neq \varnothing$$

Similarly, assuming that S_j^{-} represents the focal set VL_j' of the basic probability assignment m_j^{VL}, we observe that

$$\forall J \subseteq \{1,2,\ldots, m\} \cap_{j \varepsilon J} S_j^{-} = \{[V_1, \ldots, V_{|J|}] \mid \forall j \in J\ V_j = \text{False}\} \neq \varnothing$$

Finally, assuming that S_j^{θ} represents the focal set $VL_j \vee \neg VL_j$ of the basic probability assignment m_j^{VL}, we observe that

$\forall J \subseteq \{1,2,\ldots, m\} \cap_{j \in J} S_j^{\theta} = \theta_{es}^{VL} \neq \varnothing$

Thus, in general, assuming that S_j represents one of the three possible focal sets of the basic probability assignment m_j^{VL} we will also have that

$\forall J \subseteq \{1,2,\ldots, m\} \cap_{j \in J} S_j \neq \varnothing$ (T5.5.2)

Subsequently from (T5.5.2), we have that:

$\Sigma_{i \in I,\, j \in I,\, i \neq j} \Sigma_{Si \cap Sj = \varnothing} m_i(S_i) \times m_j(S_j) = 0$ (T5.5.3)

Therefore, according to Theorem 3.1 (see page 60 in [146]), the basic probability assignments m_j^{VL} can be combined using the rule of the orthogonal sum (defined by Axiom 9 in Section 3.4) which, since $k_0 = 0$ due to (T5.5.3) above, is simplified to the following formula:

$m^{VL}(P) = m_i^{VL} \oplus m_j^{VL}(P) = \Sigma_{X \cap Y = P} m_i^{VL}(X) \times m_j^{VL}(Y)$ (T5.5.4)

Having established the common frame of discernment for combining $m_1^{VL}, \ldots, m_m^{VL}$, and the simplified version of the rule of the orthogonal sum for obtaining the combination $m_1^{VL} \oplus m_2^{VL} \oplus \ldots \oplus m_m^{VL}$, the theorem holds due to *Lemma 5.1*, by using the following substitutions in *Lemma 5.*1: S := EXP(e_i), $L_i := \Phi_j$, P_i := Valid(e_i,Φ_j,U_o,TR), $\neg P_i = \neg$ Valid(e_i,Φ_j,U_o,TR), $\theta = \theta_{es}^{VL}$, $p_i = V_j$, and $m_i = m_j^{VL}$.

♦

Theorem 5.6: *The evidence measure m_i^{EX} defined as:*

$$m_i^{EX}(P) = \begin{cases} Bel(\bigvee_{j=1,\ldots,m} Valid(e_i, \Phi_j, U_o, TR)), & \text{if } P = Explainable(e_i, U_o, TR) \\ Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i, \Phi_j, U_o, TR)), & \text{if } P = \neg Explainable(e_i, U_o, TR) \\ 1\text{-}Bel(\bigvee_{j=1,\ldots,m} Valid(e_i, \Phi_j, U_o, TR)) - Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i, \Phi_j, U_o, TR)), & \text{otherwise} \end{cases}$$

where m = |EXP(e_i)|, i.e., the number of alternative explanations of e_i, is a DS basic probability assignment with respect to frame of discerment θ_{es}^{EX} (see Definition 6 in Section 5.4.6.1).

Proof: To prove that m_i^{EX} is a DS basic probability assignment it is sufficient to show that m_i^{EX} satisfies the axioms Axiom 1, Axiom 2, and Axiom 3 of *DS Theory* (see Section 3.4).

(Axiom 1): Axiom 1 is satisfied when:

i) If $P = EX_i$, then it must hold that $0 \le m_i^{EX}(P) \le 1$, or equivalently:

$$0 \le Bel(\bigvee_{j=1,\ldots,m} Valid(e_i,\Phi_j,U_o,TR)) \le 1 \qquad (T.5.6.1)$$

From *Theorem 5.5*, we have that:

$$Bel(\bigvee_{j=1,\ldots,m} Valid(e_i,\Phi_j,U_o,TR)) =$$

$$= \sum_{J \subseteq \{1,\ldots,m\} \text{ and } J \neq \emptyset} (-1)^{|J|+1} \left\{ \prod_{j \in J} m_j^{VL}(Valid(e_i,\Phi_j,U_o,TR)) \right\} \qquad (T.5.6.2)$$

Thus, by using (T.5.6.2), we have equivalently for (T.5.6.1):

$$0 \le \sum_{J \subseteq \{1,\ldots,m\} \text{ and } J \neq \emptyset} (-1)^{|J|+1} \left\{ \prod_{j \in J} m_j^{VL}(Valid(e_i,\Phi_j,U_o,TR)) \right\} \le 1 \qquad (T.5.6.3)$$

However, from *Theorem 5.9*, we have that m_j^{VL} is *DS basic probability assignment*, and therefore, m_j^{VL} satisfies Axiom 1 of *DS Theory*, or equivalently:

$$\forall j \in \{1,\ldots,m\}, \; 0 \le m_j^{VL} \le 1 \qquad (T.5.6.4)$$

Therefore, from (T.5.6.4) and by substituting x_i with $m_j^{VL}(Valid(e_i,\Phi_j,U_o,TR))$ in *Lemma 5.2*, the inequality (T.5.6.3) holds. Thus, (T.5.6.1) holds.

ii) If $P = \neg EX_i$, then it must hold that $0 \le m_i^{EX}(P) \le 1$, or equivalently:

$$0 \le Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i,\Phi_j,U_o,TR)) \le 1 \qquad (T.5.6.5)$$

From *Theorem 5.5*, we have that:

$$Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i,\Phi_j,U_o,TR)) =$$

$$= \prod_{j=1,\ldots,m} \{ m_j^{VL}(\neg Valid(e_i,\Phi_j,U_o,TR)) \} \qquad (T.5.6.6)$$

Thus, by using (T.5.6.6), we have equivalently for (T.5.6.5):

$$0 \leq \prod_{j=1,\ldots,m} \{m_j^{VL}(\neg Valid(e_i,\Phi_j,U_o,TR))\} \leq 1 \qquad \text{(T.5.6.7)}$$

However, from (T.5.6.4) and by substituting x_i' with $m_j^{VL}(\neg Valid(e_i,\Phi_j,U_o,TR))$ in *Lemma 5.3*, the inequality (T.5.6.7) holds. Thus, (T.5.6.5) holds.

iii) If $P \neq EX_i$ or $P \neq \neg EX_i$, then it must hold that $0 \leq m_i^{EX}(P) \leq 1$, or equivalently:

$$0 \leq Bel(\bigvee_{j=1,\ldots,m} Valid(e_i,\Phi_j,U_o,TR)) + Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i,\Phi_j,U_o,TR)) \leq 1 \qquad \text{(T.5.6.8)}$$

From (T.5.6.2) and (T.5.6.6), we have equivalently for (T.5.6.8):

$$0 \leq \sum_{J \subseteq \{1,\ldots,m\} \text{ and } J \neq \emptyset} (-1)^{|J|+1} \left\{ \prod_{j \in J} m_j^{VL}(Valid(e_i,\Phi_j,U_o,TR)) \right\} + \prod_{j=1,\ldots,m} \{m_j^{VL}(\neg Valid(e_i,\Phi_j,U_o,TR))\} \leq 1 \qquad \text{(T.5.6.9)}$$

However, from (T.5.6.4) and by substituting x_i with $m_j^{VL}(Valid(e_i,\Phi_j,U_o,TR))$ and x_i' with $m_j^{VL}(\neg Valid(e_i,\Phi_j,U_o,TR))$ in *Lemma 5.3*, the inequality (T.5.6.9) holds. Thus, (T.5.6.8) holds.

(Axiom 2): Axiom 2 is satisfied by m_i^{EX} since its focals EX_i, EX_i' and $EX_i \cup EX_i$' are non empty sets by definition of θ_{es}^{EX}:

$EX_i = \{[E_i = True]\} \neq \emptyset$

EX_i' $= \{[E_i = False]\} \neq \emptyset$ and

$EX_i \cup EX_i' = \{[E_i = True \text{ or } False]\} \neq \emptyset$.

Thus, the basic probability assigned to the empty set by m_i^{EX} is $m_i^{EX}(\emptyset) = 0$

(Axiom 3): Axiom 3 is satisfied since:

$$\sum_{P \subseteq \theta_{es}^{EX}} m_i^{EX}(P) = \sum_{P \subseteq \theta_{es}^{EX} \text{ and } P \neq EX_i \text{ and } P \neq \neg EX_i} m_i^{EX}(P) + m_i^{EX}(EX_i) + m_i^{EX}(\neg EX_i)$$

$$= 1 - Bel(\bigvee_{j=1,\ldots,m} Valid(e_i,\Phi_j,U_o,TR)) - Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i,\Phi_j,U_o,TR)) +$$

$$Bel(\bigvee_{j=1,\ldots,m} Valid(e_i,\Phi_j,U_o,TR)) + Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i,\Phi_j,U_o,TR)) =$$

$$= 1$$

♦

Theorem 5.7: *The evidence measure m_i^{EX} defined as:*

$$m_i^{EX}(P) = \begin{cases} assessmentOf(Explainable(e_i))/occurrenceTimes(e_i), \\ \text{if } P = Explainable(e_i, U_o, TR) \\ \\ assessmentOf(\neg Explainable(e_i))/occurrenceTimes(e_i), \\ \text{if } P = \neg Explainable(e_i, U_o, TR) \\ \\ 1-[assessmentOf(Explainable(e_i))+assessmentOf(\neg Explainable(e_i)) \\ /occurrenceTimes(e_i))], \quad otherwise \end{cases}$$

where $occurrenceTimes(e_i) \geq 1$, i.e., the number of times that e_i was reached as a consequence of alternative explanations of the same event e_s, and the values $assessmentOf(Explainable(e_i))$ and $assessmentOf(Explainable(\neg e_i))$ are within 0 and 1, is a DS basic probability assignment with respect to frame of discerment θ_{es}^{EX} (see Definition 6 in Section 5.4.6.1).

Proof: To prove that m_i^{EX} is a DS basic probability assignment it is sufficient to show that m_i^{EX} satisfies the axioms Axiom 1, Axiom 2, and Axiom 3 of *DS Theory* (see Section 3.4). For this purpose, suppose that:

β = assessmentOf(Explainable(e_i))/occurrenceTimes(e_i), and

γ = assessmentOf($\neg$Explainable(e_i))/occurrenceTimes(e_i),

where $0 < \beta < 1$, $0 < \gamma < 1$, and $0 < 1 - \beta - \gamma < 1$

(Axiom 1): Axiom 1 is satisfied since:

If $P = EX_i$, $m_i^{EX}(P) = \beta$ with $0 < \beta < 1$.

If $P = \neg EX_i$, $m_i^{EX}(P) = \gamma$ with $0 < \gamma < 1$ since $0 < \alpha_1 < 1$.

If $P \neq EX_i$ or $P \neq \neg EX_i$, $m_i^{EX}(P) = 1 - \beta - \gamma$ with $0 < 1 - \beta - \gamma < 1$.

(Axiom 2): Axiom 2 is satisfied by m_i^{EX} since its focals EX_i, EX_i' and $EX_i \cup EX_i'$ are non empty sets by definition of θ_{es}:

$EX_i = \{[E_i = True]\} \neq \varnothing$

$EX_i' = \{[E_i = False]\} \neq \varnothing$ and

$EX_i \cup EX_i' = \{[E_i = True \text{ or } False]\} \neq \varnothing$.

Thus, the basic probability assigned to the empty set by m_i^{EX} is $m_i^{EX}(\varnothing) = 0$

(Axiom 3): Axiom 3 is satisfied since:

$$\sum_{P \subseteq \theta_{e_s}^{EX}} m_i^{EX}(P) = \sum_{P \subseteq \theta_{e_s}^{EX} \text{ and } P \neq EX_i \text{ and } P \neq \neg EX_i} m_i^{EX}(P) + m_i^{EX}(EX_i) + m_i^{EX}(\neg EX_i)$$

$$= 1 - \beta - \gamma + \beta + \gamma = 1$$

♦

Theorem 5.8: *If Φ_j is an alternative explanation of e_i and $CONS(\Phi_j,TR)$ is the set of the expected consequences that are identified for Φ_j, and it holds that $CONS(\Phi_j,TR) \neq \varnothing$ with $r = |CONS(\Phi_j,TR)|$, i.e. the number of the members of $CONS(\Phi_j,TR)$, the belief in the genuineness of at least one expected consequence in $CONS(\Phi_j,TR)$, $Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q,U_o,TR))$, and in the genuineness of none of the expected consequences in $CONS(\Phi_j,TR)$, $Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q,U_o,TR))$, are measured by the following functions:*

$$Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q,U_o,TR)) =$$

$$= \sum_{Q \subseteq CONS(\Phi_j,TR) \text{ and } Q \neq \varnothing} (-1)^{|Q|+1} \left\{ \prod_{q \in Q} m_q^{GN}(Genuine(e_q,U_o,TR)) \right\}$$

$$Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q,U_o,TR)) =$$

$$= \prod_{eq \in CONS(\Phi_j,TR)} \left\{ m_q^{GN}(\neg Genuine(e_q,U_o,TR)) \right\}$$

where m_q^{GN} $(q=1,\ldots, r)$ is the basic probability assignment associated with the expected consequence e_q of the alternative explanation Φ_j of e_i.

Proof: The belief function Bel in the theorem must be obtained by combining the BPAs $m_1^{GN},\ldots, m_r^{GN}$ which are associated with $e_1, \ldots, e_q$. The combination of the basic probability assignments m_q^{GN} (q=1,…, r) requires their mapping on a common frame of discernment, i.e., a set of mutually exclusive propositions representing exhaustively the

properties that m_q^{GN} assign belief to. This common frame of discernment has been defined as θ_{es}^{GN} (see *Definition 8*) in Section 5.4.6.1.

Given the assumptions about the construction of the frame of discernment θ_{es}^{GN}, the *focals* GN_q, $\neg GN_q$ and $GN_q \vee \neg GN_q$ of each of the basic probability assignments m_q^{GN} will correspond to the following subsets of θ_{es}^{GN}:

- GN_q will correspond to $\{[G_1, \ldots, G_r] \mid G_q = \text{True}\}$ referred to as GN_q henceforth
- $\neg GN_q$ will correspond to $\{[G_1, \ldots, G_r] \mid G_q = \text{False}\}$ referred to as GN_q' henceforth
- $GN_q \vee \neg GN_q$ will correspond to $\{[G_1, \ldots, G_r] \mid G_q = \text{True or } G_q = \text{False}\}$ which is equal to θ_{es}^{GN}

Thus, the *core* (see page 40 in [146]) of each m_q^{GN} will be $C_q = GN_q \cup GN_q' \cup \theta_{es}^{GN} = \theta_{es}^{GN}$ for all *q* (q=1,…, r) and:

$$\cap_{q=1,\ldots,r} C_q = \theta_{es}^{GN} \neq \varnothing \qquad \text{(T5.8.1)}$$

Due to (T5.8.1) and Theorem 3.2 (see page 40 in [146]), it follows that the basic probability assignments m_q^{GN} (q=1,…, r) can be combined.

Furthermore assuming that S_q^{+} represents the focal set GN_q of the basic probability assignment m_q^{GN}, we observe that

$$\forall\, Q \subseteq \{1,2,\ldots, r\}\ \cap_{q \varepsilon Q} S_q^{+} = \{[G_1, \ldots, G_{|r|}] \mid \forall\, q \in Q\ G_q = \text{True}\} \neq \varnothing$$

Similarly, assuming that S_q^{-} represents the focal set GN_q' of the basic probability assignment m_q^{GN}, we observe that

$$\forall\, Q \subseteq \{1,2,\ldots, r\}\ \cap_{q \varepsilon Q} S_q^{-} = \{[G_1, \ldots, G_{|r|}] \mid \forall\, q \in Q\ G_q = \text{False}\} \neq \varnothing$$

Finally, assuming that S_q^{θ} represents the focal set $GN_q \vee \neg GN_q$ of the basic probability assignment m_q^{GN}, we observe that

$$\forall\, Q \subseteq \{1,2,\ldots, r\}\ \cap_{q \varepsilon Q} S_q^{\theta} = \theta_{es}^{GN} \neq \varnothing$$

Thus, in general, assuming that S_q represents one of the three possible focal sets of the basic probability assignment m_q^{GN} we will also have that

$\forall$ Q $\subseteq$ {1,2,…, r} $\cap_{q\varepsilon Q} S_q \neq \varnothing$

(T5.8.2)

Subsequently from (T5.8.2), we have that:

$$\Sigma_{i\varepsilon I,\, j\varepsilon I,\, i \neq j} \Sigma_{Si \cap Sj=\varnothing}\, m_i(S_i) \times m_j(S_j) = 0 \qquad \text{(T5.8.3)}$$

Therefore, according to Theorem 3.1 (see page 60 in [146]), the basic probability assignments m_q^{GN} can be combined using the rule of the orthogonal sum (defined by Axiom 9 in Section 3.4) which, since $k_0 = 0$ due to (T5.8.3) above, is simplified to the following formula:

$$m^{GN}(P) = m_i^{GN} \oplus m_j^{GN}(P) = \Sigma_{X \cap Y = P}\, m_i^{GN}(X) \times m_j^{GN}(Y) \qquad \text{(T5.8.4)}$$

Having established the common frame of discernment for combining m_1^{GN},…, m_r^{GN} , and the simplified version of the rule of the orthogonal sum for obtaining the combination $m_1^{GN} \oplus m_2^{GN} \oplus \ldots \oplus m_r^{GN}$, the theorem holds due to *Lemma 5.1*, by using the following substitutions in *Lemma 5.*1: S := CONS(Φ_j,TR), $L_i := e_q$, P_i := Genuine(e_q,U_o,TR), $\neg P_i$ = $\neg$Genuine(e_q,U_o,TR), $\theta = \theta_{es}^{GN}$, $p_i = G_q$, and $m_i = m_q^{GN}$.

♦

Theorem 5.9: *The evidence measure m_j^{VL} defined as:*

$$m_j^{VL}(P) = \begin{cases} Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q, U_o, TR)), \text{ if } P = Valid(e_i, \Phi_j, U_o, TR) \\ Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q, U_o, TR)), \text{ if } P = \neg Valid(e_i, \Phi_j, U_o, TR) \\ 1\text{-}Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q, U_o, TR)) - Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q, U_o, TR)), \text{ otherwise} \end{cases}$$

where r = |CONS(Φ_j,TR)|, i.e., the number of expected consequences of the Φ_j, is a DS basic probability assignment with respect to frame of discerment θ_{es}^{VL} (see Definition 7 in Section 5.4.6.1).

Proof: To prove that m_j^{VL} is a DS basic probability assignment it is sufficient to show that m_j^{VL} satisfies the axioms Axiom 1, Axiom 2, and Axiom 3 of *DS Theory* (see Section 3.4).

(Axiom 1): Axiom 1 is satisfied when:

i) If $P = VL_j$, then it must hold that $0 \leq m_j^{VL}(P) \leq 1$, or equivalently:

$$0 \leq Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q,U_o,TR)) \leq 1 \qquad (T.5.9.1)$$

From *Theorem 5.8*, we have that:

$$Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q,U_o,TR)) =$$

$$=\sum_{Q\subseteq CONS(\Phi_j,TR) and Q\neq\varnothing}(-1)^{|Q|+1}\left\{\prod_{q\in Q} m_q^{GN}(Genuine(e_q,U_o,TR))\right\}$$

(T.5.9.2)

Thus, by using (T.5.9.2), we have equivalently for (T.5.9.1):

$$0 \leq \sum_{Q\subseteq CONS(\Phi_j,TR) and Q\neq\varnothing}(-1)^{|Q|+1}\left\{\prod_{q\in Q} m_q^{GN}(Genuine(e_q,U_o,TR))\right\} \leq 1$$

(T.5.9.3) However, from *Theorem 5.1* and *5.3*, we have that m_q^{GN} is *DS basic probability assignment*, and therefore, m_q^{GN} satisfies Axiom 1 of *DS Theory*, or equivalently:

$$\forall q\in\{1,\ldots,r\},\ 0 \leq m_q^{GN} \leq 1 \qquad (T.5.9.4)$$

Therefore, from (T.5.9.4) and by substituting x_i with $m_q^{GN}(Genuine(e_q,U_o,TR))$ in *Lemma 5.2*, the inequality (T.5.9.3) holds. Thus, (T.5.9.1) holds.

ii) If $P = \neg VL_j$, then it must hold that $0 \leq m_j^{VL}(P) \leq 1$, or equivalently:

$$0 \leq Bel(\bigwedge_{j=1,\ldots,m} \neg Valid(e_i,\Phi_j,U_o,TR)) \leq 1 \qquad (T.5.9.5)$$

From *Theorem 5.8*, we have that:

$$Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q,U_o,TR)) =$$

$$=\prod_{eq\in CONS(\Phi_j,TR)}\left\{m_q^{GN}(\neg Genuine(e_q,U_o,TR))\right\}$$

(T.5.9.6)

Thus, by using (T.5.9.6), we have equivalently for (T.5.9.5):

$$0 \le \prod_{eq \in CONS(\Phi_j,TR)} \{ m_q^{GN}(\neg Genuine(e_q,U_o,TR)) \} \le 1 \qquad (T.5.9.7)$$

However, from (T.5.9.4) and by substituting x_i' with $m_q^{GN}(\neg Genuine(e_q,U_o,TR))$ in *Lemma 5.3*, the inequality (T.5.9.7) holds. Thus, (T.5.9.5) holds.

iii) If $P \neq VL_j$ or $P \neq \neg VL_j$, then it must hold that $0 \le m_j^{VL}(P) \le 1$, or equivalently:

$$0 \le Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q,U_o,TR)) + Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q,U_o,TR)) \le 1 \qquad (T.5.9.8)$$

From (T.5.9.2) and (T.5.9.6), we have equivalently for (T.5.9.8):

$$0 \le \sum_{Q \subseteq CONS(\Phi_j,TR) and Q \neq \varnothing} (-1)^{|Q|+1} \{ \prod_{q \in Q} m_q^{GN}(Genuine(e_q,U_o,TR)) \} +$$

$$\prod_{eq \in CONS(\Phi_j,TR)} \{ m_q^{GN}(\neg Genuine(e_q,U_o,TR)) \} \le 1 \qquad (T.5.9.9)$$

However, from (T.5.9.4) and by substituting x_i with $m_q^{GN}(Genuine(e_q,U_o,TR))$ and x_i' with $m_q^{GN}(\neg Genuine(e_q,U_o,TR))$ in *Lemma 5.3*, the inequality (T.5.9.9) holds. Thus, (T.5.9.8) holds.

(Axiom 2): Axiom 2 is satisfied by m_j^{VL} since its focals VL_j, VL_j' and $VL_j \cup VL_j$' are non empty sets by definition of θ_{es}^{VL}:

$VL_j = \{[V_j = True]\} \neq \varnothing$

$VL_j' = \{[V_j = False]\} \neq \varnothing$ and

$VL_j \cup VL_j' = \{[V_j = True \text{ or } False]\} \neq \varnothing$.

Thus, the basic probability assigned to the empty set by m_j^{VL} is $m_j^{VL}(\varnothing) = 0$

(Axiom 3): Axiom 3 is satisfied since:

$$\sum_{P \subseteq \theta_{es}^{VL}} m_j^{VL}(P) = \sum_{P \subseteq \theta_{es}^{VL} and\ P \neq VL_j\ and\ P \neq \neg VL_j} m_j^{VL}(P) + m_j^{VL}(VL_j) + m_j^{VL}(\neg VL_j)$$

$$= 1 - Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q,U_o,TR)) - Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q,U_o,TR)) +$$

$$Bel(\bigvee_{q=1,\ldots,r} Genuine(e_q,U_o,TR)) + Bel(\bigwedge_{q=1,\ldots,r} \neg Genuine(e_q,U_o,TR)) =$$

$$= 1$$

♦

Chapter 6: Experimental Evaluation of the Diagnostic Prototype

6.1 Overview

In this chapter, the experimental evaluation of the diagnostic prototype is discussed. In Section 6.2, we initially provide the experimental set up of laboratory simulations taken place for the evaluation of the diagnostic prototype. Particularly, an architectural overview of the EVEREST prototype and the diagnostic prototype that were used are provided. Having provided the designs of the prototypes used in our experiments, the target application of the evaluation is described with focus on its *EC-Assertion* formal monitoring specifications. It should be noted that the *EC-Assertion* formal monitoring specifications of the target application are a key point for our experimentation as they specify the monitoring theory (i.e., assumptions and rules formulas) as well as the events and fluents that are processed by the monitoring and diagnostic prototypes. Regarding the events that were used, Section 6.2 concludes with a discussion on the simulator deployed for the generation of events that could simulate the operation of the target application.

In Section 6.3, a discussion about the diagnostic framework performance characteristics that the evaluation is focused on is provided. After introducing the performance characteristics, the metrics that were used for the analysis and the measurement of our diagnosis framework performance are presented.

Having specified the significant performance characteristics for our evaluation, Section 6.4 introduces the factors that we selected to experiment with. The selected factors were selected by taking into account hypotheses regarding their impact on the key performance characteristics of the diagnostic prototype, and therefore our diagnostic approach. Provided the overview on the factors that may affect the performance of the diagnostic prototype, Section 6.4 presents the set of selected experiments through an accumulative experiments table. Finally, in Section 6.5, we provide the experimental results through relevant tables and charts, and try to give plausible explanations for their occurrence. (Perhaps, we could have used our or another diagnostic prototype for the results explanation!).

6.2 Experimental Set Up of Laboratory Simulations

6.2.1 Architecture of the EVEREST diagnostic prototype

This section describes the design of the diagnosis prototype and its integration with the other components of the EVEREST framework. In particular, the diagnosis prototype consists of two separate tools, namely the *event genuineness belief tool* (referred as *EGBT* henceforth) and the *violations diagnosis tool* (referred as *VDT* henceforth). The task that *EGBT* performs is to compute the genuineness belief range of events. On the other hand, *VDT* is responsible for generating diagnoses for any violation that the monitor of *EVEREST* detects. It should be noted that the *VDT* diagnosis results are generated by use of the genuineness belief ranges that *EGBT* computes for each *violation observation*.

In the following, Section 6.2.1.1 gives the overall *EVEREST* design focusing on the diagnosis prototype components, while Sections 6.2.1.2 and 6.2.1.3 detail the *EGBT* and *VDT* architectures respectively.

6.2.1.1 Overall EVEREST design

EGBT and *VDT* that consist the diagnostic prototype have been implemented as component of *EVEREST* framework. The framework uses *EGBT* to compute genuineness belief ranges of runtime events, and *VDT* to diagnose detected violations. Figure 6-1 illustrates the overall design of *EVEREST* framework and the most relevant interactions between the event receiver component, monitor, *EGBT* and *VDT*.

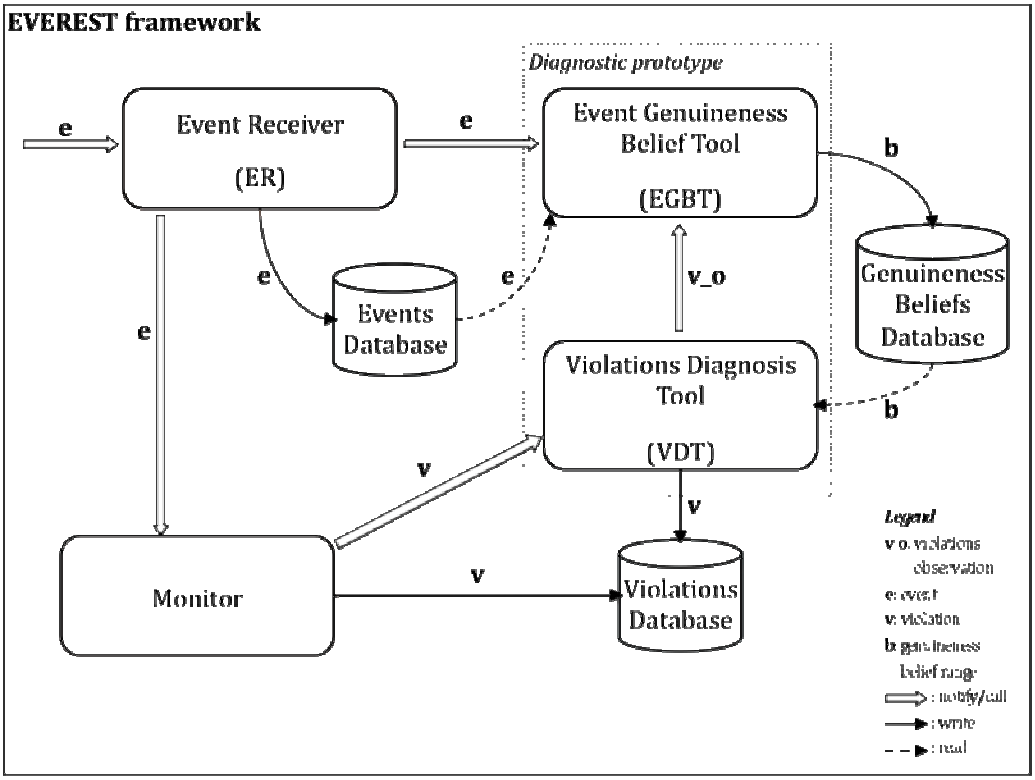

Figure 6-1 – Overall EVEREST design with respect to diagnostic prototype

When the framework receives an event *e*, the Event Receiver (ER) collects the event and stores it into the *events database*. Then, it notifies the monitor and *EGBT* of the new event. Upon this notification, *EGBT* computes a belief range in the genuineness of the event and stores it into the *genuineness beliefs database* whilst the monitor detects violations of all the monitoring rules that the event *e* can be unified with. Once a violation *v* is detected, the monitor stores it into the *violations database* and notifies *VDT* of violation *v*. Once notified, *VDT* analyzes violations *v* by extracting the *violation observations*, i.e., the events that were taken into account for the detection of violation *v*. For each *violation observation*, *VDT* requests their genuineness belief range from *EGBT*. Once the genuineness belief ranges *b* of the given *violation observations* are computed, *EGBT* stores *b* into the *genuineness beliefs database* and therefore *VDT* is able to read *b* from the aforementioned database. Finally, *VDT* generates a diagnosis for violation *v* by reasoning on *b* and updates the record of violation *v* in the *violations database*.

The components of the framework have been designed based on the objective of having a loose coupling between them. To achieve this, most of the data exchanges among them are not realized through direct method calls, but through a shared database. Thus, *EGBT* and *VDT* have been implemented as threads, and have been designed in such

a way to store their results into relevant databases. This implies, for instance, that when *VDT* invokes *EGBT*, it does not need to wait for the result of *EGBT*, but continues to operate until notified with *EGBT* results. Once notified, *VDT* can retrieve the computed result directly from the shared database, i.e., the *genuineness beliefs database* in .

It should also be noted that *EGBT* uses the *events database* for computing the genuineness belief ranges. *EGBT* needs to access the *events database* in read mode only as it only extract events from it in order to compute the belief range in the genuineness of an event. *EGBT* accesses the *genuineness beliefs database* in write mode because it needs to store in it the results of its computations. *VDT* accesses the genuineness database in read mode only as its computations do not require the modification of information stored in the *genuineness beliefs database*. In particular, when *VDT* needs an event genuineness belief range, *VDT* checks the *genuineness beliefs database* and if the required belief range is not stored in it, it calls directly *EGBT* to get the required information.

6.2.1.2 Event Genuineness Belief Tool

EGBT is composed by two main components: the *Event Genuineness Belief Interface* and the *Event Belief Handler* (EBH). The former component realizes and exposes the *EGBT* interface, whilst the latter computes event genuineness belief ranges.

EGBT architecture has been designed to support the case in which the *Event Receiver* (ER) sends events to the *EGBT* with a rate that is faster than the rate at which *EGBT* can consume these events given the time that it needs to compute genuineness belief ranges for previous events. For instance, ER sends an event to *EGBT* every one second and *EGBT* takes two seconds for computing the corresponding genuineness belief range.

The implementation is based on the *Consumer/Producer design pattern*, as shown in Figure 6-2. In this pattern, two processes, the producer and the consumer, share a common fixed-size buffer. The producer's task is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming data elements and removes them from the buffer (one element at a time). In *EGBT*, the producers are ER and *VDT*, whilst *EBH* plays the role of consumer. Once *EBH* computes the genuineness belief range for an event that it has removed from the buffer, it takes the next event from the head of the buffer and computes its own genuineness belief range. Thus, the event queue operates in FIFO mode.

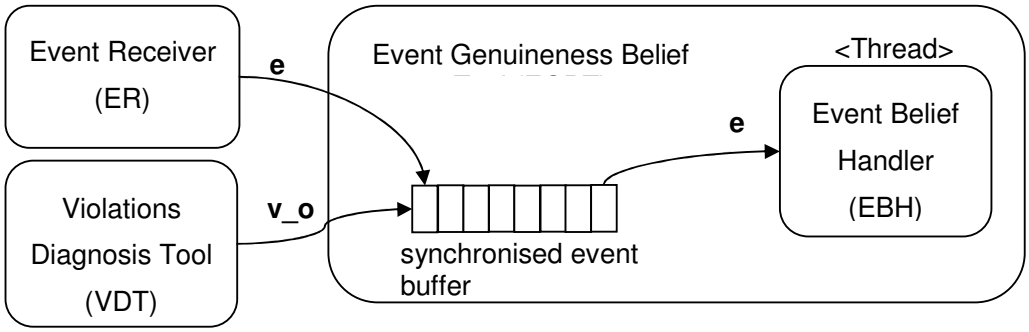

Figure 6-2 – EGBT architecture

6.2.1.3 Violations Diagnosis Tool

Similarly, *VDT* is composed by two main components: the *Violation Diagnosis Interface* and the *Violation Handler* (VH). The former component realises and exposes the *VDT* interface, whilst the latter generates the diagnosis for given violations. *VDT* architecture supports the case in which the *monitor* sends violations to the *VDT* with a rate that is faster than the rate at which *VDT* can consume these violations given the time that it needs to generate diagnoses for previous violations.

As in the case of *EGBT*, *VDT* implementation is based on the *Consumer/Producer design pattern*, as shown in **3**. In this pattern, two processes, the producer and the consumer, share a common fixed-size buffer. In *VDT*, the producer is the *monitor*, whilst *VH* plays the role of consumer. Again, the violation queue operates in FIFO mode as once *VH* generates the diagnosis for a violation that it has removed from the violation buffer, it takes the next violation from the head of the buffer and generates a diagnosis for it.

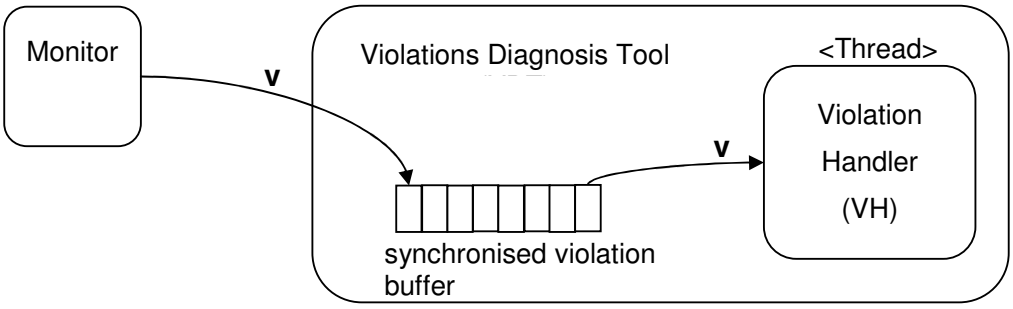

Figure 6-3 – VDT architecture

6.2.2 The monitored system

To evaluate the diagnosis prototype, we used a *Location Based Access Control System* (referred to *LBACS* in the following) as the system to be monitored. The *LBACS* manages access to different resources of an organisation, through a combination of user authentication, device identification and device location detection capabilities. In *LBACS*, users entering and moving within the premises of an organisation, using mobile computing devices (e.g., a notebook or smart phone) may be given access to different resources, such as the enterprise intranet, printers or the Internet, depending on the credentials of these devices and their exact location within the physical space of the organization. Resource access is granted depending on policies, which determine when access to a particular type of resource is considered to be harmful or not. For instance, a policy may determine that an authenticated employee of the organization who is trying to access a printer via the local wireless network, whilst being in an area of the premises that is accessible to the public, should be granted access, whilst authenticated visitors should only be given access to printers when they are in one of the organization's meeting rooms. The general architecture of *LBACS* is shown in **Figure 6-4**.

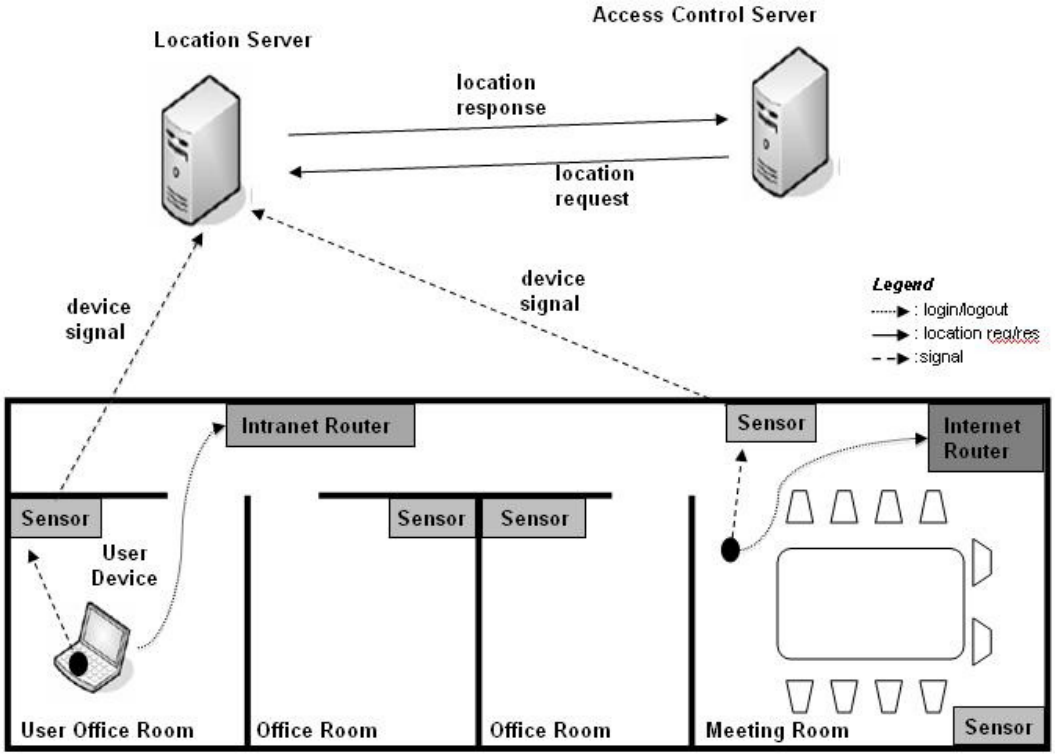

Figure 6-4 – LBACS architecture

As shown in the figure, *LBACS* is based on two servers: a location and an access control server. The control server polls the location server at regular intervals, in order to obtain the position of the devices of all the users who are currently authenticated in the system. To ensure the availability of accurate information about the location of mobile devices in *LBACS*, each device is expected to send signals to the location detection server periodically. The location of a device in *LBACS* is determined by the strength of signals sent from the device to the location server. In particular, a daemon in mobile devices sends signals to location server via location sensors. Based on the signals received from different sensors, the location server can calculate the position of a device with some accuracy measure.

The effectiveness of the access control solution of *LBACS* depends on several conditions regarding the operation of the different components that constitute it at runtime including, for example:

- The continuous *availability* of the location server (C1). The availability of these components is a prerequisite for the availability of device position, which is necessary for the access control system at runtime.
- The *liveness* of signal daemons in mobile devices (C2). Each device that is known to the access control server should send signals to the location server periodically and the maximum period of not receiving a signal should not be less than *m* time units

For the undertaken evaluation, the following operational scenario for *LBACS* was considered. A mobile device d is operable in the premises that are controlled by *LBACS*. The daemon of the device d broadcasts periodically signal to the location sensors of *LBACS*. As long as the location sensors receive signals from d, they forward the signals to the location server. While the device d is operable in the premises, the user of d may need to access to a resource r of the premises (e.g. a printer in some room). Thus, a request for access to the resource r is sent to the access control server by device d. In order to decide whether device d can access to resource r, the access control server needs information with regards to the location of device d. Therefore, the access control server interacts with the location server to get the location information of device d. The location server calculates the location of device d based on the forwarded signals from the location sensors and sends the location to the access control server. Since, the access

control server has received the location of the device d, it makes a decision on whether device d can have access to resource r and let device d know of the generated decision. In case that the decision of the access control server allows device d to access to resource r, device d can make use of resource r but should release r as long as device d tasks are over. On the other hand, in case that device d is not granted the access privilege, it can request again access to resource r.

Moreover, as shown in **Figure 6-4**, *LBACS* includes two wireless network controllers, the intranet and Internet routers namely. The operational scenario considers that intranet router provides access to the local wireless network for authenticated employees of the organization, whilst Internet router provides access to the Internet only for authenticated employees and visitors. However, the access control policy adopted in the scenario specifies a condition (C3) regarding the connection of authenticated devices to the routers that provide access to the organization intranet and Internet wireless networks. In particular, C3 specifies that no user should be allowed to login onto intranet and Internet routers simultaneously to reduce scope for masquerading attacks.

6.2.2.1 Monitoring specifications

In this section, the rules and assumptions that are used for monitoring *LBACS* are given. Condition C1 can be specified in the monitoring language of *EVEREST* framework as follows:

LBACS.R1.
```
∀t1∈Time, ∃t2∈Time, ∀_LServerId ∈ LocationServers,
∀_ACServerId ∈ AccessControlServers, ∀_deviceId∈Devices,
∀_source.
    Happens(e(_Id1,_ACServerId,_LServerId,REQ-B,locationRequest
            (_deviceId),_source), t1, R(t1,t1)) ⇒
    Happens(e(_Id2,_LServerId,_ACServerId,REQ-A,locationResponse
            (_deviceId),_source), t2, R(t1+1,t1+3000))
```

The monitoring rule LBACS.R1 is violated in all cases where, povided that the access control server of LBACS requests location information for a device from the location server of LBACS, the location server does not provide such information within the next 3 seconds after the corresponding request occurrence.

Condition C2 can be checked by two rules that are specified as:

LBACS.R2. ∀t1∈Time, ∃t2∈Time, ∀_LServerId ∈ LocationServers,
∀_ACServerId ∈ AccessControlServers, ∀_deviceId∈Devices,
∀_receiver1∈Sensors, ∀_source1, ∀_source2.

Happens(e(_Id1,_ACServerId,_LServerId,REQ-A,locationRequest
(_deviceId),_source1), t1, R(t1,t1)) ∧
¬**Happens**(e(_Id1,_ACServerId,_LServerId,REQ-A,locationRequest
(_deviceId),_source1), t2, R(0,t1-1)) ⇒
Happens(e(_Id2,_deviceId,_receiver1,REQ-A,signal(_deviceId),
_source2), t3, R(t1-2000,t1-1))

LBACS.R3. ∀t1∈Time, ∃t2∈Time, ∀_deviceId∈Devices, ∀_receiver1∈Sensors,
∀_source1.

Happens(e(_Id1,_deviceId,_receiver1,REQ-A, signal(_deviceId),
_source1), t1, R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId,_receiver1,REQ-A,signal(_deviceId),
_source1), t2, R(t1+1,t1+2000))

Rule LBACS.R2 checks when the first signal from a device should occur. In particular, the first signal from a given device is expected within the last two seconds before the first request for the device location made by the LBACS access control server. Once, a device sends its first signal, rule LBACS.R3 checks the periodical receipt of signals from the device, with maximum period of two seconds.

Finally, condition C3 can be specified as:

LBACS.R4. ∀t1 ∈ Time, ∃t2 ∈ Time, ∀t3 ∈ Time, ∀_deviceId ∈ Devices,
∀_userId ∈ Users, ∀_source1, ∀_source2.

Happens(e(_Id1,intranetRouter,_deviceId,REQ-B,
loginAcknowledgment(_userId),_source1),
t1,R(t1,t1)) ∧
Happens(e(_Id2,internetRouter,_deviceId,REQ-B,
loginAcknowledgment(_userId),_source2),
t2,R(t1+1,t2)) ⇒
Happens(e(_Id3,intranetRouter,_deviceId,REQ-B,
logoutAcknowledgment(_userId)
_source1),t3,R(t1+1,t2-1))

The monitoring rule LBACS.R4 is violated in all cases where a user known to LBACS logs into the Internet router of the system, while he is still logged into the LBACS intranet router by using in both cases the same device.

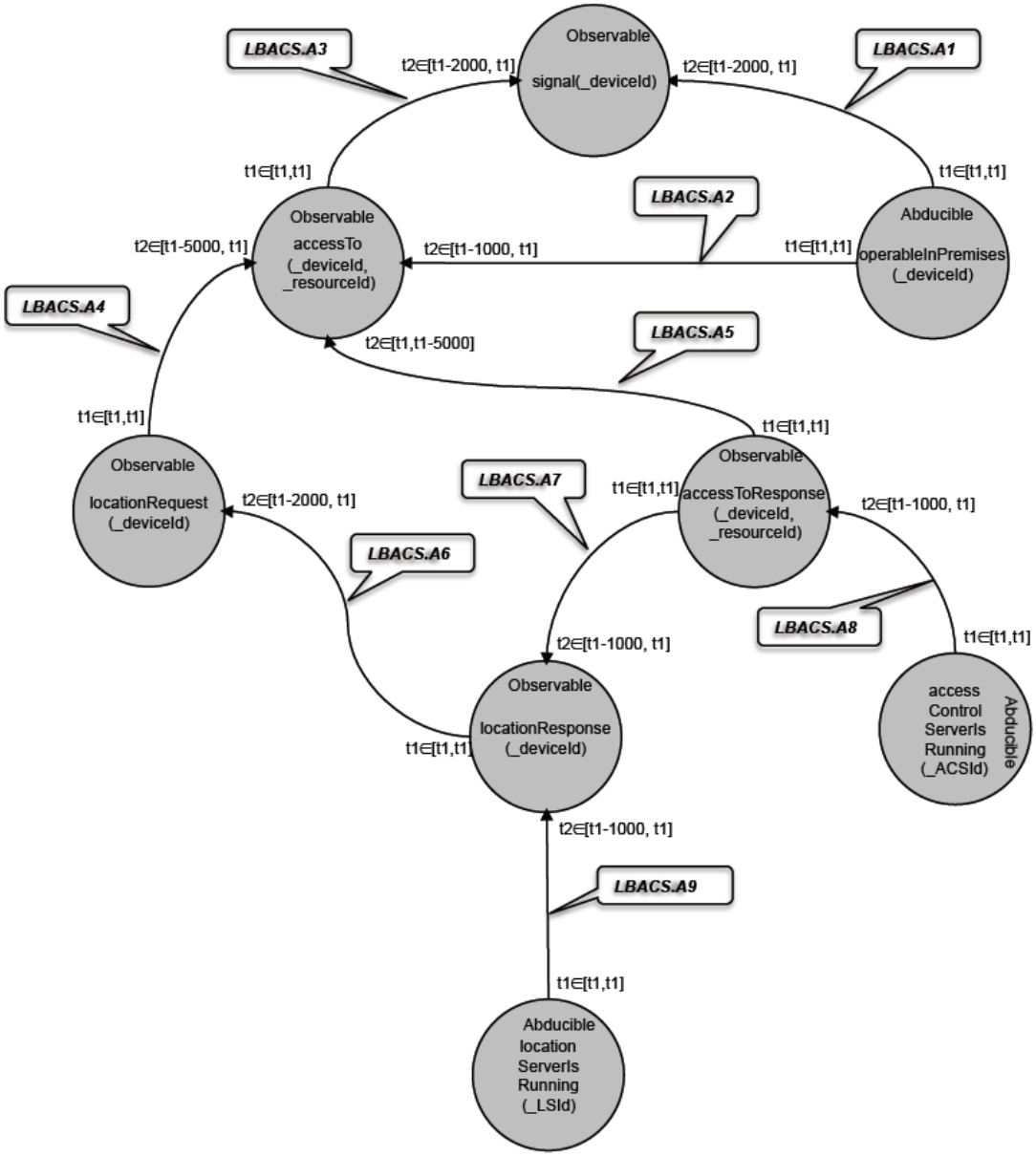

Figure 6-5 – LBACS theory graph part I

In the following, the *LBACS* assumptions are provided in the form of a directed graph, as shown in Figure 6-5 and Figure 6-6. More specifically, the graph in Figure 6-5 represents assumptions, which model *LBACS* intended behaviour with respect to device signalling,

resource access authorization and device location identification, as discussed in the operational scenario above. On the other hand, the graph in Figure 6-6 represents assumptions regarding the login and logout operations that can take place between *LBACS* devices and routers.

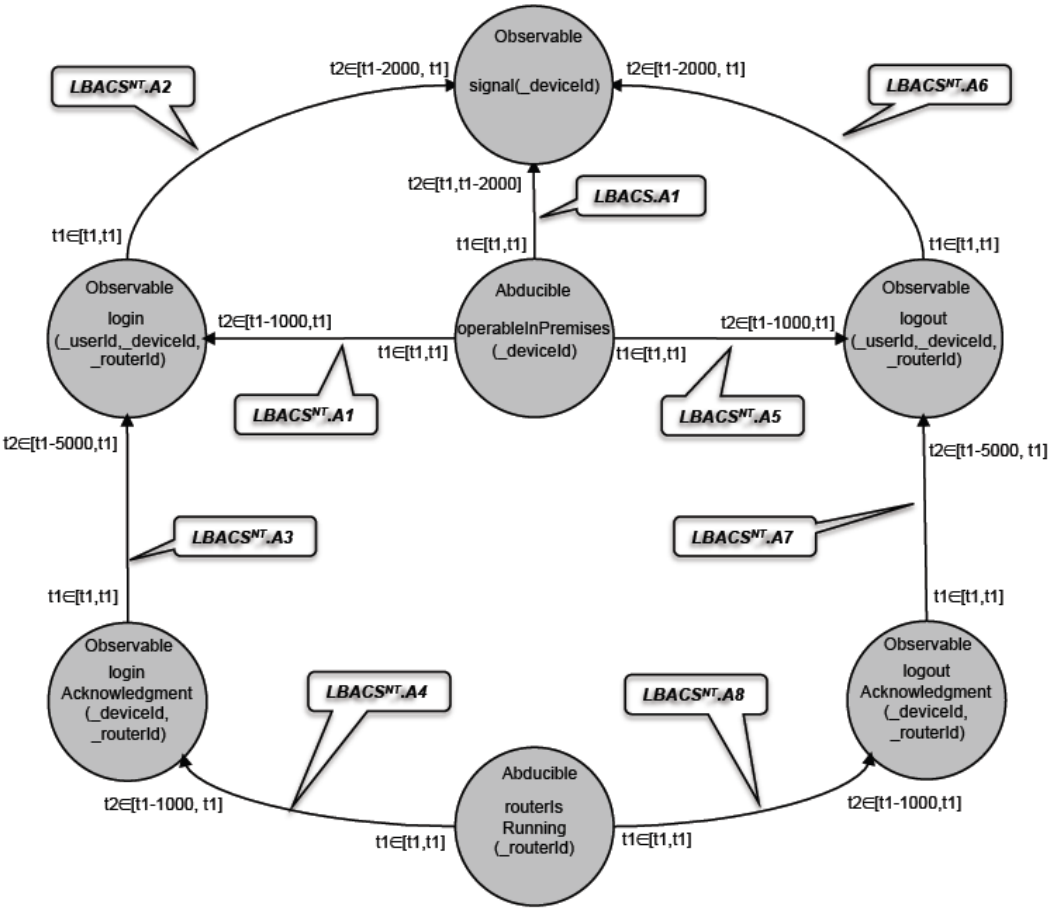

Figure 6-6 - LBACS theory graph part II

In the above graphs, a node represents observable and abducible events, i.e., events that can be captured by *EVEREST* captors at *LBACS* runtime and can be abduced by *EVEREST* diagnostic process, respectively. A directed edge (arrow) that connects two nodes represents the existence of an assumption that correlates the two events the connected nodes model. The direction of the edge shows the implication direction of the assumption, while labels on both edge ends show the relevant time ranges of the correlated events. For instance, LBACS.A1 in Figure 6-6 represents the following assumption expressed in *EVEREST* specification language:

LBACS.A1. ∀t1∈Time, ∃t2∈Time, ∀_deviceId∈Devices, ∀_receiver, ∀_sender, ∀_source.

```
Happens(e(_Id1,_sender,_receiver,REQ-A,operableInPremises
          (_deviceId),_source),t1,R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId,_receiver,REQ-A,signal(_deviceId),
          _source),t2,R(t1-2000,t1)
```

In particular, assumption LBACS.A1 states that if a device is operable in premises at some time point t1, it is expected that a signal should have sent from the device within the last two seconds before t1. As shown in Figure 6-6, the *operableInPremises* event is abducible, while the *signal* event is observable. The *EVEREST* specifications of all the assumptions that the graphs in Figure 6-5 and Figure 6-6 represent are provided in Appendix A.

6.2.3 The deployed simulator

For the undertaken evaluation, a generic simulator, which was developed by members of software engineering group of City University, was used to simulate the operational *LBACS* scenario discussed in Section 6.2.2 and generate corresponding event sequences. The simulator is based on a common channel architecture, which connects all simulated components and is used for message exchange between the components. The output of a simulation is the messages exchanged between the simulated components, referred to as *simulated events* in the following. Each *simulated event* specifies the sender component (*senderId*), the receiver component (*receiverId*), the actual payload of the message, which is the operation that sender component calls and its parameters (*operationName(parameters)*), and a timestamp indicating the event generation time (*t*). Thus, the signature of a simulated event is as follows:

e(senderId, receiverId, operationName(parameters), t)

The simulator should be aware of the specifications of each simulated component. A simulated component specification includes the *simulated events* that the component can generate and the trigger conditions upon which the *simulated event* generation should be done. Such trigger conditions, for instance, can be the receipt of a message or the end a specific time period. For capturing the incoming and outgoing *simulated events* of a simulated component, an event captor might be specified and attached to the simulated component.

Moreover, the simulator should be fed with the initial simulated events types - called *seed events* henceforth - that can trigger other components to generate events, for starting the simulation. The simulator should also be notified with the total number of *seed events* that must be generated, as well as, a time range per *seed event* type, whose boundaries restrict the occurrence period of the generated *seed events*. In particular, assuming that $[t^s_{max}, t^s_{min}]$ is the time range that restricts the generation period of *seed events* of e^s type, and e^s_n is an e^s type event generated at t_n, then e^s_{n+1} is generated at t_n which is result of adding to t_n a random number t within $[t^s_{max}, t^s_{min}]$. It should be noted that the simulator has a feature that enables the storage of *seed events*, which are generated during a simulation. By these means, a seed of each simulation is stored and, therefore, each simulation can be repeated. Once the *seed events* are generated, the simulator channel manager takes over the responsibility to dispatch the *seed events* to their specified recipient components. The recipient components, then, generate *simulated events* acting upon their specifications.

However, to be able to evaluate the diagnostic process and, especially, assess whether the diagnostic process results i.e., the genuineness belief ranges and final diagnosis reports, are correct, the above simulator is extended to take into account components, which are not legitimate and authorised components of the simulated system, referred to as *adversaries* henceforth. *Adversaries* objective is to create conditions according to which the simulated system deviates from its intended behaviour. To do so, the simulator should be notified with the number and exact capabilities of *adversaries*. Therefore, the specifications of an adversary should detail which types of *simulated events* the adversary can intercept, i.e. specify the adversary position in the simulated components topology, and how can affect the events, i.e., the types of attacks the adversary can carry out. The types of attack that the simulator supports at the moment are as follows:

- Delay of *simulated events*: The dispatch of the legitimate *simulated events* is carried out with some delay.
- Block of *simulated events*: The *simulated events* do not reach their initial and legitimate recipient.

Figure 6-7 pictures a UML model for a controlled simulation of *LBACS* based on the generic simulator discussed above.

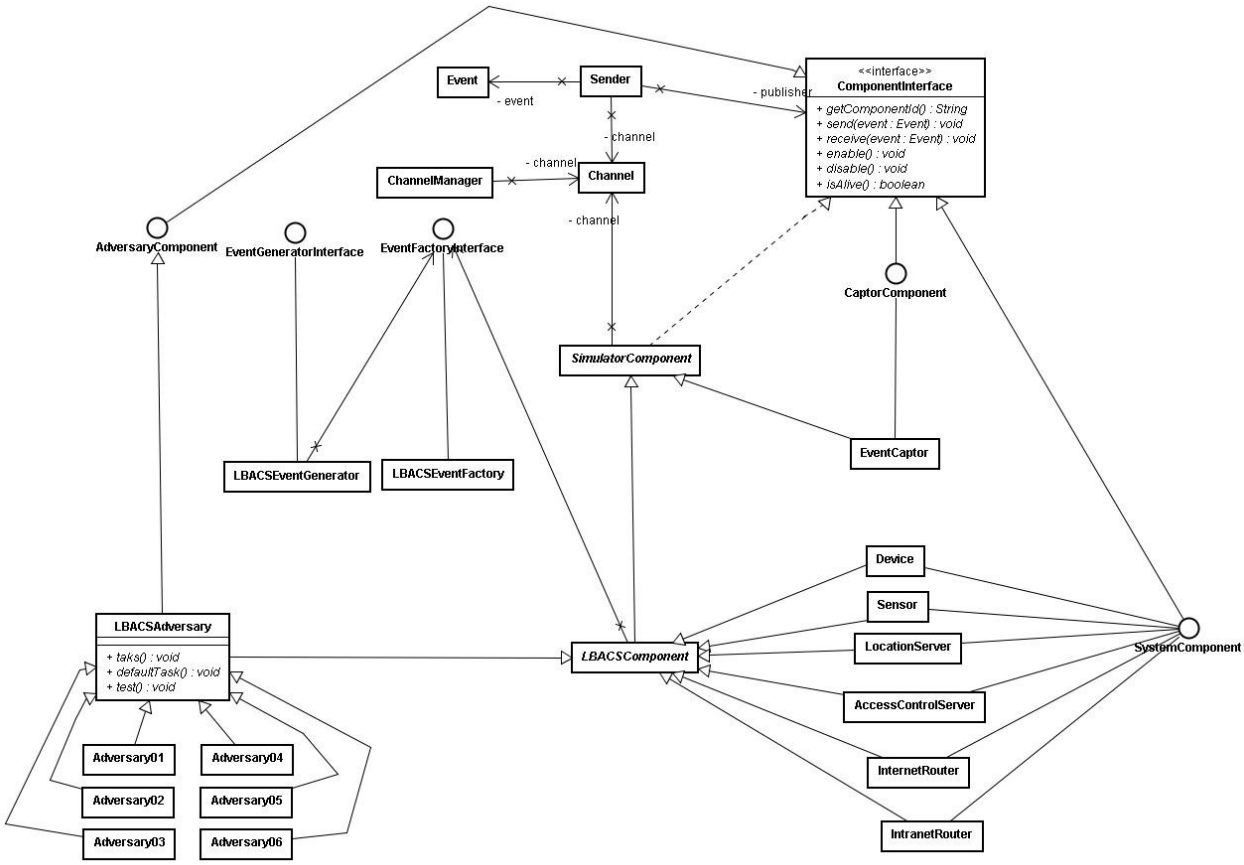

Figure 6-7 – LBACS simulator UML model

The above UML model can generate the *LBACS* model represented in Figure 6-8. More specifically, the model in Figure 6-8 represents the topology of LBACS simulated components and *adversaries*.

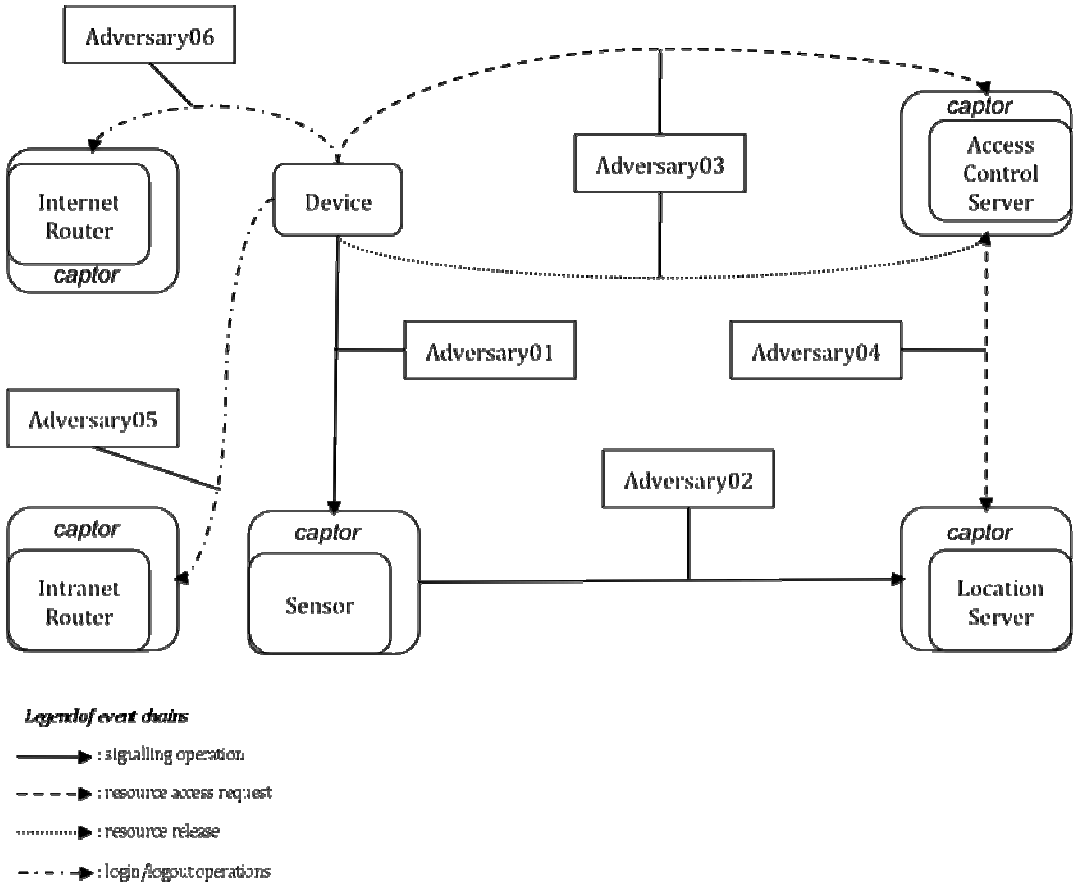

Figure 6-8 – LBACS simulated components topology

As shown in Figure 6-7 and Figure 6-8, the simulated *LBACS* components that have been taken into account are: a Device, a Sensor, a Location Server, an Access Control Server, an Intranet Router and an Internet Router. The *LBACS* simulator considers also an event captor for each one of the aforementioned simulated components except for Device. This decision was taken due to the generic case that a device is not equipped with an event captor. In fact, a device may be owned by a visitor of the organization premises protected by *LBACS*. A visitor's device might not be equipped with an event captor that could intercept the incoming and outgoing message traffic.

The models in above figures include also six types of *adversaries*. In particular, Adverary01 is able to intercept and affect events exchanged between Device and Sensor. Adversary02 can intervene and affect the message traffic between the Sensor and Location Server, whilst Adversary03 and Adversary04 are able to act similarly between Device and Access Control Server, and between Location Server and Access Control Server, respectively. Adversary05 and Adversary06, finally, intercept and affect the event

traffic between Device and Intranet Router, and Device and Internet Router, respectively. In a simulation, an adversary configuration specifies the number of instances of each *adversary* type, as well as, the details of the attack that each *adversary* instance can carry out.

6.3 Evaluation criteria and metrics

The aim of the diagnostic framework evaluation is the experimental assessment of the diagnostic prototype performance. In particular, the evaluation takes into account two performance characteristics of the diagnostic process. The first characteristic is the *correctness* of the belief ranges that the diagnostic process generates for the genuineness of events, as well as, the final diagnosis that the process generates for any given violation. This characteristic is measured by using the metrics *precision* and *recall* with respect to the genuineness belief ranges that *event genuineness belief tool* (*EGBT*) computes for any given event, and the final diagnosis that *violations diagnosis tool* (*VDT*) generates for any given violation.

The second characteristic of interest is the diagnostic process *responsiveness* that is strongly related to two computational time periods of interest, namely the *belief computational time* and the *diagnosis generation time*. More specifically, the former refers to the time that elapses while the genuineness belief range of an event is computed by *EGBT*, whilst the latter is specified as the time that *VDT* takes to generate the final diagnosis for a given violation. Furthermore, due to *EGBT* and *VDT* architecture, which is based on the *Consumer/Producer design pattern*, as discussed in Section 6.2.1.2, the *queue delay* of both components is measured as a supportive metric for the assessment of the diagnostic prototype *responsiveness*. In particular, the *EGBT queue delay* is defined as the time interval between an event insertion to and withdrawal from the *EGBT* local events queue (see Figure 6-2). Similarly, *VDT queue delay* refers to the time interval that a violation remains stored in the *VDT* local violations queue (see **3**).

In Section 6.3.1, the metrics *precision* and *recall* that are used for evaluating the diagnostic prototype *correctness* are introduced, whilst the metrics regarding *belief computational time*, *diagnosis generation time*, and *queue delays*, which are used for the diagnostic prototype *responsiveness* assessment are given in Section 6.3.2.

6.3.1 Correctness metrics

The evaluation of the diagnostic framework *correctness* is covered by the measurement of the *recall* and *precision* with respect to genuineness belief ranges that *EGBT* computes for any event, as well as, the final diagnosis that *VDT* generates for any violation. For being able to reason upon and eventually evaluate the diagnostic process *correctness*, the a priori knowledge of the genuineness of events used in the experimental evaluation is required.

6.3.1.1 Genuine and fake event sets

More specifically, the events used in the experiments are classified into two event sets, namely the *genuine* and *fake* events sets. Thus, assume that *genuine* events are events generated by legitimate components of the monitored system in accordance to monitored system intended behaviour. On the other hand, *fake* events are *genuine* events that have been captured and affected by not authorized components, *adversaries*, whose aim is to make the monitored system to deviate from its intended behaviour. As discussed in Section 6.2.3, for the undertaken evaluation, *adversaries* can delay or block intercepted *genuine* events.

Upon the capabilities of *adversaries*, the evaluation of the diagnostic process takes into account three categories of *fake* events. The first category includes *genuine* events that are captured and delayed by *adversaries*. Particularly, the delay impact on the *genuine* events is the alteration of their occurrence time, i.e., their timestamp, as discussed in Section 6.2.3.

Finally, the second category considers as *fake* events the events that are generated by legitimate components of the system being monitored, but are blocked by *adversaries*, and therefore *EVEREST* framework never receives them. However, in case that the specifications of monitoring rules that *EVEREST* uses to monitor the system include predicates that can be unified with such blocked events, their presence in violated rules instances is established by the principle of negation as failure when the expected event has not been seen in the *EVEREST* event log within the time range that it is expected to occur.

6.3.1.2 EGBT recall and precision

The *EGBT recall* and *precision* are measured with respect to definite *fake* and *genuine* events. In particular, given a belief range [B_{min} B_{max}), *EGBT recall* with respect to *fake* events, referred to as $EGBT\_Recall_F$ henceforth, expresses the ratio of definite *fake* events whose genuineness belief is bounded by [B_{min} B_{max}). Given again the belief range [B_{min} B_{max}), *EGBT precision* with respect to *fake* events, referred to as $EGBT\_Precision_F$ in the following, is defined as the ratio of events, which correspond to eventual *fake* events, and their genuineness belief is within [B_{min} B_{max}).

Similarly, given a belief range [B_{min} B_{max}), $EGBT\_Recall_G$, which refers to *EGBT recall* with respect to *genuine* events, is equal to the ratio of definite *genuine* events whose genuineness belief is within [B_{min} B_{max}). Given again the belief range [B_{min} B_{max}), *EGBT precision* with respect to *genuine* events, referred to as $EGBT\_Precision_G$ henceforth, expresses the ratio of events, which correspond to eventual *genuine* events , and their genuineness belief is within [B_{min} B_{max}).

In the undertaken expernimental evaluation, given a particular range of belief values [B_{min} B_{max}), $EGBT\_Recall_F$, $EGBT\_Precision_F$, $EGBT\_Recall_G$, and $EGBT\_Precision_G$ are measured according to the following formulas:

$$EGBT\_\mathrm{Re}call_F = \frac{|WR \cap F|}{|F|} \tag{6.2.1.2.1}$$

$$EGBT\_\mathrm{Pr}ecision_F = \frac{|WR \cap F|}{|WR|} \tag{6.2.1.2.2}$$

$$EGBT\_\mathrm{Re}call_G = \frac{|WR \cap G|}{|G|} \tag{6.2.1.2.3}$$

$$EGBT\_\mathrm{Pr}ecision_G = \frac{|WR \cap G|}{|WR|} \tag{6.2.1.2.4}$$

where in a given experiment:

- *WR* is the set of events whose genuineness belief computed by EGBT is within [B_{min} B_{max})
- *F* is the set of *fake* events
- *G* is the set of *genuine* events, and
- |*X*| is the cardinality of set *X*

EGBT recall and *precision* with respect to *fake* and *genuine* events were measured for ten different belief ranges. More specifically, the experimental evaluation took into account belief ranges from 0 to 0.1, 0.1 to 0.2, …, and 0.9 to 1. The use of different levels spanning the entire range of possible belief values, i.e., [0,1], enabled the evaluation of *EGBT recall* and *precision* when considering results at different belief ranges.

6.3.1.3 VDT recall and precision

Regarding the *correctness* of violations final diagnoses that are generated by *VDT*, it might be useful to recall that a final diagnosis of a violation is a report of the *confirmed* and *unconfirmed violation observations* i.e. events involved in the violation. More specifically, a *violation observation* P will be classified as a *confirmed* event if the belief in the genuineness of P is greater than or equal to the corresponding disbelief, i.e., Bel(Genuine(P)) ≥ Bel(¬Genuine(P)). It should be noted again that *VDT recall* and *precision* are measured with respect to *fake* and *genuine* events.

Thus, VDT_Recall_F, which refers to *VDT recall* with respect to *fake* events set *F* in the following, represents the ratio of definite *fake* events that are flagged as *uncofirmed* in corresponding final diagnosis reports. *VDT precision* with respect to *F*, referred to as $VDT\_Precision_F$ henceforth, is defined as the ratio of events that are flagged as *uncofirmed* in final diagnosis reports generated by *VDT* and correspond to eventual *fake* events.

Similarly, VDT_Recall_G, which refers to *VDT recall* with respect to *genuine* events set *G* in the following, expresses the ratio of definite *genuine* events that are flagged as *cofirmed* in corresponding final diagnosis reports. *VDT precision* with respect to *G*, referred to as $VDT\_Precision_G$ henceforth, is defined as the ratio of events that are flagged as *cofirmed* in final diagnosis reports generated by *VDT* and correspond to eventual *genuine* events.

In the undertaken evaluation, the formulas used for measuring VDT_Recall_F, $VDT\_Precision_F$, VDT_Recall_G, and $VDT\_Precision_G$ are as follows:

$$VDT\_\mathrm{Re}call_F = \frac{|UC \cap F|}{|F|} \quad (6.2.1.3.1)$$

$$VDT\_\mathrm{Pr}ecision_F = \frac{|UC \cap F|}{|UC|} \quad (6.2.1.3.2)$$

$$VDT\_\mathrm{Re}call_G = \frac{|CN \cap G|}{|G|} \quad (6.2.1.3.3)$$

$$VDT\_\mathrm{Pr}ecision_G = \frac{|CN \cap G|}{|CN|} \quad (6.2.1.3.4)$$

where in a given experiment:

- *UC* is the set of events that were flagged as *unconfirmed* in final diagnosis reports generated by VDT
- *CN* is the set of events that were flagged as *confirmed* in final diagnosis reports generated by VDT
- *F* is the set of *fake* events
- *G* is the set of *genuine* events, and
- |*X*| is the cardinality of set *X*

6.3.2 Responsiveness metrics

The evaluation of the diagnostic prototype *responsiveness* is based on statistic analysis of the *EGBT belief computational time* and the *VDT diagnosis generation time*. In particular, the following statistic measures are used:

- The mean, standard deviation/variance, and minimum and maximum *EGBT belief computational time*
- The mean time, standard deviation, variance, minimum and maximum *VDT diagnosis generation time*

To understand better the above measures, it may be useful to recall briefly the overall architecture of *EVEREST* framework (see Figure 6-1) and the interactions that take place between the event receiver (ER), the monitor, and the *EGBT* and *VDT* components. In *EVEREST* framework architecture, ER notifies the monitor and *EGBT* of new events sequentially. Also, once the monitor detects a violation, it notifies *VDT* of the new violation. Finally, *VDT* notifies *EGBT* of violation observations whose genuineness belief ranges are significant for generating violations final diagnoses. Since *EGBT* and *VDT* have been implemented as threads, and the monitor does not wait for any result from any of them, the computation time of the monitor is affected neither by *EGBT belief computational time* nor by *VDT diagnosis generation time*.

Regarding the first interaction, Figure 6-9 pictures the timelines of ER, monitor and *EGBT* components. As shown in the figure, ER notifies monitor and *EGBT* at times $T^n_{monitor}$ and T^n_{EGBT} respectively of event e_i. The notification of event e_i that causes the initiation of monitor computations at $T^s_{monitor}$ will subsequently trigger computations in *EGBT* at time T^s_{EGTB}. It should be noted that, as shown in the figure, *EGBT* may not process event e_i immediately after notification, and therefore the overhead introduced by *EGBT* event queue is the time distance between T^n_{EGBT} and T^s_{EGBT}. Finally, even if the monitor terminates with the processing of the event e_i, *EGBT* might still be performing computations upon the event. Thus, as shown in Figure 6-9, for instance, *EGBT* computations may continue after the end time of the monitor T_e, with *EGBT* computational time for the event e_i being equal to the distance between T^s_{EGBT} and T^e_{EGBT}. Analogously, *DVT* and *EGBT* components behave much the same with respect to their interaction.

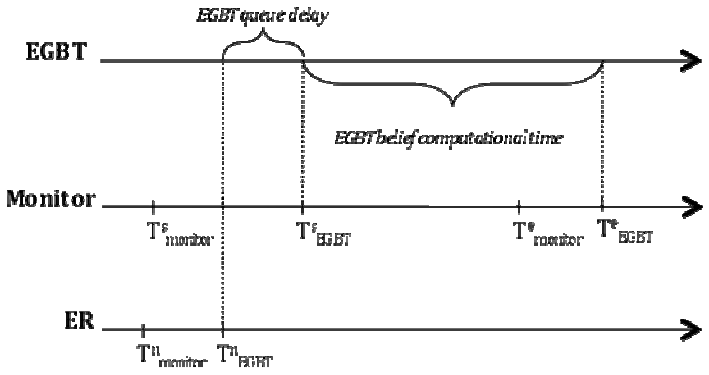

Figure 6-9 – ER, Monitor and EGBT timelines

Regarding the interaction between the monitor and *VDT* component, Figure 6-10 demonstrates the timelines of the monitor and *VDT* components. Once the monitor detects a violation v_i at time T^d_v, it notifies *VDT* of v_i at time T^n_{VDT}. As shown in the figure, *VDT* may not process violation v_i immediately after notification, and therefore an overhead equal to the distance between T^n_{DVT} and T^s_{DVT} is introduced by *VDT* violation queue.

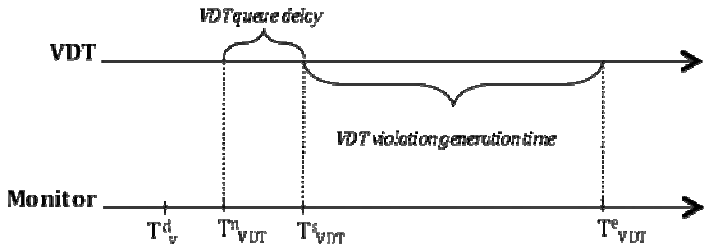

Figure 6-10 - Monitor and VDT timelines

6.4 Evaluation experiments design

To evaluate the diagnostic framework, the experiments were designed to investigate and analyse the impact of factors – referred to as *sensitivity factors* henceforth - on the performance of the diagnostic process. A non-exhaustive list of sensitivity factors that we have identified is as follows:

- Underlying monitoring theory characteristics like the number of assumptions being used during the abductive and deductive phases of our approach and the coverage of theory against the set of observed runtime events.
- Diagnosis window
- Constants α_1 and α_2 that are used in belief functions (see Section 5.4.6.2)
- Characteristics of the event set

To restrict the boundaries of the experimental evaluation presented in this thesis, we have decided to experiment with a specific set of experimental configurations regarding the *sensitivity factors* mentioned above.

Therefore, the experimental evaluation of our diagnosis approach was undertaken by using two experimental configurations that generated two experiments sets. The objective of the first experimental configuration – referred to as *experimentalConfiguration1* henceforth - focuses on examining the sensitivity of our approach with respect to the relation that can be observed between the intended behaviour of the system and the diagnosis window. More specifically, *experimentalConfiguration1* specifies experiments that expose how our prototype reacts on the utilization of different diagnosis windows, provided that the selected diagnosis windows have a mathematical relation with the time ranges of a common set of assumptions.

Having defined the objective of *experimentalConfiguration1* above, we found equally interesting to examine the sensitivity of our approach against the relation that could be identified between different runtime behaviours of the simulated system and the intended behaviour of the system. Therefore, the objective of the second experimental configuration – referred to as *experimentalConfiguration2* henceforth -highlights the sensitivity of our approach against different events sets provided that the generation of diagnosis results is based on a common monitoring theory.

In the following, we provide a detailed description of the experimental configurations used and an accumulative table for all the conducted experiments.

6.4.1 The LBACS simulations

Having introduced the sensitivity factors that we experimented with, the details of the *LBACS* simulations that have been performed for sake of the undertaken experimental evaluation are given. In Section 6.4.1.1, details for the generations of the *seed events* set that was used for the performed simulations are discussed, whilst the specifications for the simulation of the rest of *LBACS* events are provided in Section 6.4.1.2. Finally Section 6.4.1.3 discusses the attacks that have been simulated and examined in the experiments.

6.4.1.1 The LBACS simulations seed

In all *LBACS* simulations conducted for the evaluation of the diagnosis process, a common set of *seed events* have been used. The decision of using a common set of *seed events* was taken for ensuring that the results of the different performed simulations are independent of the initial experimental input that in our case is the *seed events* set.

The common set of *seed events* contain only events that the Device component of the *LBACS* simulator can generate. More specifically, the type of *seed events* that were generated for the undertaken evaluation are the ones corresponding to signalling operation, resource access request and resource release operations, login and logout operations. The following table summarizes the type of events that are included in the common set of *seed events* by displaying the operation name, the receiver component and the time range that restricts the generation period of each event type.

Table 6-1 – Types of seed events generated by LBACS Device

Operation name	Receiver component	Generation period time range (sec)
signal	Sensor	[2,2]
accessTo	Access Control Server	[3, 3.5]
resourceRelease	Access Control Server	[6, 7]
login	Intranet Router	[10, 15]
logout	Intranet Router	[10, 15]
login	Internet Router	[10, 15]
logout	Internet Router	[10, 15]

It should be noted that the max generation period of *signal* events is 2 sec for complying with monitoring rule LBACS.R1. The generation periods of the rest *seed events* types were selected randomly. By feeding the *LBACS* simulator with a *seed event* set complying with the type of events specified in *LBACS* monitoring theory (see Appendix A), *LBACS* simulator is able to generate all the event chains that are shown in Figure 6-8.

6.4.1.2 The LBACS simulation inter-event delays

Regarding the rest *LBACS* events that are generated by the simulator triggered by the *seed events* we discussed above, the following table provides the delay - referred to as *inter-event delay* henceforth - that simulator introduces. Upon an incoming event, a simulated component generates the predefined response by introducing a delay that simulates the execution of the requested operation. Table 6-2 shows the inter-event delays for each pair of simulated and triggering seed events. For instance, once the *LBACS* sensor component receives a *signal* event at t, the sensor generates a *forwardSignal* event at t', where t' is equal to t plus a random delay value that was chosen from the range [0.5, 1.5].

Table 6-2 – Inter-event delay ranges for simulated events

Simulated event operation name	Triggerring seed event	Inter - delay range (sec)
forwardSignal	signal	[0.5, 1.5]
accessToResponse	accessTo	[1, 3.6]
loginAcknowledgment	login	[1, 3.6]
logoutAcknowledgment	logout	[1, 3.6]

One could observe that the above *inter-event delay* ranges are specified with significantly small time distance between their boundaries. There are two reasons to have such ranges. Firstly, we wanted to have *inter-event delay* ranges that could map to the operation of an *LBACS* system, as presented in Figure 6-8, in a realistic time frame. For instance, a *forwardSignal* event will be realistically generated within [0.5, 1.5] after the occurrence of a *signal seed event*.

Moreover, it should be noted that for all simulated event types except *forwardSignal*, the chosen *inter-event delay* range is [1, 3.6]. One could observe that this repeated range has a median equal to 2.3 that was not randomly selected. On the contrary, by examining the time ranges specified for the *LBACS* assumptions used for the presented experiments (see Appendix A), the average of the time ranges length is computed equal to 2.3 again. Therefore, the above time range was selected in order that the simulated events are generated according to the intended behaviour of *LBACS*, as specified by the *LBACS* assumptions.

6.4.1.3 The LBACS simulated attack

The *LBACS* simulated attack that was analyzed during the undertaken evaluation is described by one *adversary* configuration. As mentioned above (Section 6.2.3), an *adversary* configuration specifies the number of instances of each *adversary* type, as well as, the details of the attack that each *adversary* instance can carry out in a simulation

The delay attack configuration specifies that there is one instance per each *adversary* type. Each *adversary* instance is randomly activated during the simulation, and randomly selects the event to affect from the events that can intercept. Also, all *adverasies* carry out

delay attacks. More specifically, the specifications of all *adversaries* include a predefined delay time range for each event type the *adversaries* can intercept and eventually affect. This predefined delay time range for a given event can guarantee a low genuineness belief for the event. During the simulation, the actual delay length for each affected event is selected randomly from the aforementioned predefined delay time range.

6.4.2 Experimental configurations and evaluation experiments sets

As discussed in the opening paragraphs of Section 6.4, the experimental evaluation of our diagnosis approach was undertaken by using two experimental configurations that generated two experiments sets. The objective of the first experimental configuration, *experimentalConfiguration1*, focuses on examining the sensitivity of our approach with respect to the relation that can be observed between the intended behaviour of the system and the diagnosis window. More specifically, *experimentalConfiguration1* specifies experiments that expose how our prototype reacts on the utilization of different diagnosis windows, provided that the selected diagnosis windows have a mathematical relation with the time ranges of a common set of assumptions.

On the other hand, with *experimentalConfiguration2* we aim to examine the sensitivity of our approach against the relation that could be identified between different runtime behaviours of the simulated system and the intended behaviour of the system. Therefore, the objective of *experimentalConfiguration2* highlights the sensitivity of our approach against different events sets provided that the generation of diagnosis results is based on a common monitoring theory.

Both experimental configurations specify experiments sets with some common input. This common input refers to the underlying monitoring theory and the selected values of belief functions constants. Regarding the underlying monitoring theory used in all conducted experiments, we used a subset of *LBACS* assumptions discussed in Section 6.2.2.1 and fully deployed in Appendix A. More specifically, we used assumptions LBACS.A1 to LBACS.A9 that model *LBACS* intended behaviour with respect to device signalling, resource access authorization and device location identification processes of the *LBACS* operational scenario. Regarding the belief functions constants α_1 and α_2, we have used common values for each conducted experiment. The constants α_1 and α_2 were set to 0.3 and 0.4 respectively. It should be noted that by setting the aforementioned values to the belief function constants, the diagnostic prototype is expected to assign a

higher degree of belief to cases that no consequences can be identified within the given diagnosis window than to cases that empty explanations sets are generated.

6.4.2.1 First experimental configuration

Besides the common assumptions set and the common couple of belief functions constants described above, we have used a set of events and a set of different diagnosis windows to achieve the objective of *experimentalConfiguration1*. More specifically, to conduct the evaluation experiments specified by *experimentalConfiguration1*, we generated a common event set with the *LBACS* simulator. The cardinality of the set is 5000 and was generated by using the *seeds events* set and the *inter-event delays* presented in Sections 6.4.1.1 and 6.4.1.2 respectively. As the delay attack configuration specifies in Section 6.4.1.3, *experimentalConfiguration1* experiments event set includes *fake* events that were generated by six *adversaries*. Each *adversary* was intercepting randomly and delaying the 20% of its incoming events by introducing a delay randomly selected from 2 sec to 6 sec. The delay attack range was specified with the aforementioned boundaries in order to assure that indeed *fake* events could trigger violations for the monitoring rules LBACS.R1 to LBACS.R4 (see Appendix A). It should be noted that it is the violations generated for the aforementioned rules that are given as input to *VDT*, in order to check the performance characteristics of both *VDT* and *EGBT*. As discussed in Section 6.2.1.3, given a monitoring rule violation, *VDT* extracts the *violation observations*, i.e., events involved in the violation, and requests *EGBT* to compute their genuineness belief and plausibility values.

Regarding the different values of diagnosis windows, we selected the following values: 1.5 sec, 2.3 sec, 2.5 sec, 5 sec, 7.5 sec, and 10 sec. The diagnosis windows were selected by taking into account the *inter-event delays* discussed in Section 6.4.1.2, and the time ranges of the underlying monitoring theory assumptions that were used. As mentioned in Section 6.4.1.2, for each type of simulated events, except for the *forwardSignal* event type, the *inter-event delays* (2.3 sec) were set equal to the average of the median of the time ranges specified in the assumptions. Therefore, the first diagnosis window (1.5 sec) is less than, the second diagnosis window (2.3 sec) is equal to, while the rest of windows as set greater than the given common *inter-event delay*. Our intuition is that our diagnosis performance would be optimal for diagnosis windows around the

common *inter-event delay*, whilst it would be decreasing as the diagnosis window increases and gets greater than the common *inter-event delay*.

6.4.2.2 Second experimental configuration

To achieve the objective of *experimentalConfiguration2*, three different event sets that have been generated by using a common *seed events* set and *inter-event delays* were used. More specifically, each set contains 5000 events generated by using the *seeds event*s set and the *inter-event delays* presented in Sections 6.4.1.1 and 6.4.1.2 respectively.

The sets differ in the number of the contained *fake* and *genuine* events. For the generation of all three events sets, the *delay attack configuration* was used again. As discussed in Section 6.4.1.3, a simulation that runs according to *delay attack configuration* results in event sets containing *fake* events that are generated by six *adversaries*. Regarding the first event set – referred to as *eventSet1* henceforth - each *adversary* was specified to intercept randomly and delay the 10% of its incoming events. Similarly, regarding the second event – referred to as *eventSet2* henceforth – the 20% of the incoming events of each adversary was intercepted randomly and delayed, whilst the third event set was generated with *adversaries* intercepting and delaying the 30% of their incoming events. For all three events sets generation, *adversaries* were introducing a delay randomly selected within [2, 6] seconds. The delay attack range was specified as such in order to assure that indeed *fake* events could trigger violations for the monitoring rules LBACS.R1 to LBACS.R4 (see Appendix A).

Regarding the diagnosis window, we have selected to set it equal to the average of the median of the time ranges specified in the assumptions, as discussed in Section 6.4.2.1. Therefore, the time window we used to conduct the experiments specified by *experimentalConfiguration2* was set equal to 2.3 sec.

6.4.2.3 Evaluation experiments sets

Having given the exact specifications of *experimentalConfiguration1* and *experimentalConfiguration1*, Table 6-3 presents the set of experiments we conducted for the evaluation of our approach. It should be noted that for each conducted experiment, we have given a unique id hoping that this will help the reader to associate experiments of different experimental configuration with its results given in the following section.

Table 6-3 – LBACS conducted experiments

		ExperimentalConfiguration2 Adversaries Capabilities		
		Delay in 10% of incoming events	**Delay in 20% of incoming events**	**Delay in 30% of incoming events**
Experimental Configuration1 Diagnosis Window (sec)	**1.5**		expConf1_1.5	
	2.3	expConf2_10%	expConf1_2.3 / expConf2_20%	expConf2_30%
	2.5		expConf1_2.5	
	5		expConf1_5	
	7.5		expConf1_7.5	
	10		expConf1_10	

6.5 Evaluation Experiments Results

The results of the *experimentalConfiguration1* and *experimentalConfiguration2* are presented and discussed in Sections 6.5.1 and 6.5.2 respectively. The presentation structure specifies that for each explanation configuration and for each individual experiment, we provide tables and discuss the experimental observations regarding the *EGBT* and *VDT correctness* and *responsiveness* metrics as specified in Sections 6.3.1.2, 6.3.1.3, and 6.3.2. For each experimental configuration, Sections 6.5.1 and 6.5.2 concludes with charts presenting an overall view of the individual experiments per configuration and a comparative discussion on the experimental observations against the objective of the relevant experimental configuration.

Regarding the *EGBT* and *VDT correctness* metrics especially, to try to help the reader to read and understand the relevant given tables and charts, brief examples are given in the following. Assume that Table 6-4 presents experimental results for *EGBT* correctness metrics according to the formulas discussed in Section 6.3.1.2.

Table 6-4 – Example table of EGBT correctness results

Belief Range	$EGBT\_Recall_F$	$EGBT\_Precision_F$	$EGBT\_Recall_G$	$EGBT\_Precision_G$
[0, 0.1)	0.10	1.00	0.00	0.00
[0.1, 0.2)	0.00	N/A	0.00	N/A
[0.2, 0.3)	0.00	N/A	0.00	N/A
[0.3, 0.4)	0.00	0.00	0.24	1.00
[0.4, 0.5)	0.25	0.26	0.17	0.74
[0.5, 0.6)	0.00	N/A	0.00	N/A
[0.6, 0.7)	0.25	0.38	0.10	0.62
[0.7, 0.8)	0.25	0.22	0.22	0.78
[0.8, 0.9)	0.05	0.08	0.14	0.92
[0.9, 1]	0.10	0.15	0.13	0.85

A brief guide of how Table 6-4 can be read and interpreted is as follows:

- $EGBT\_Recall_F$ for a given belief range equals to the percentage of *fake* events whose belief values lie within the given belief range. For instance, the $EGBT\_Recall_F$ for belief range [0, 0.1) that equals to 0.10 shows that the belief values of the 10% of the *fake* events lie within [0, 0.1). $EGBT\_Recall_F$ values with respect to lower belief ranges should ideally be greater than 0.5, whilst the values with respect to higher belief ranges should be ideally less than 0.5.

- $EGBT\_Precision_F$ for a given belief range equals to the percentage of events that their belief values lie within the given belief range and happen to be *fake*. For instance, the $EGBT\_Precision_F$ for belief range [0, 0.1) that equals to 1 shows that the 100% of the events whose belief values lie within [0, 0.1) are *fake*. $EGBT\_Precision_F$ values with respect to lower belief ranges should ideally be greater than 0.5, whilst the values with respect to higher belief ranges should be ideally less than 0.5.

- $EGBT\_Recall_G$ for a given belief range equals to the percentage of *genuine* events whose belief values lie within the given belief range. For instance, the $EGBT\_Recall_G$ for belief range [0.3, 0.4) that equals to 0.24 shows that the belief values of the 24% of the *genuine* events lie within [0.3, 0.4). $EGBT\_Recall_G$

values with respect to lower belief ranges should ideally be less than 0.5, whilst the values with respect to higher belief ranges should be ideally greater than 0.5.

- *EGBT_Precision*$_G$ for a given belief range equals to the percentage of events that their belief values lie within the given belief range and happen to be *genuine*. For instance, the *EGBT_Precision*$_G$ for belief range [0.3, 0.4) that equals to 1 shows that the 100% of the events whose belief values lie within [0.3, 0.4) are *genuine*. *EGBT_Recall*$_G$ values with respect to lower belief ranges should ideally be less than 0.5, whilst the values with respect to higher belief ranges should be ideally greater than 0.5.

- *N/A* cell value means that the corresponding value could not be computed as it maps to a fraction with zero denominator. For instance, the *N/A* value of *EGBT_Precision*$_G$ for belief range [0.2, 0.3) means that there no events whose belief values lie within [0.2, 0.3). It should be noted that rows with *N/A* cell values are not taken into account in our discussion, as they provide no experimental observations.

- *Light grey highlighted rows* include the *EGBT* correctness results for belief ranges within [0.3, 0.7). According to our intuition, the values in *light grey highlighted rows* are excluded from the results analysis, as they might not be useful and enough indicative information to be taken into account by a recovery decision making process.

Similarly, assume that the following table (Table 6-5) present experimental results of *VDT* correctness as specified in Section 6.3.1.3. Regarding the *correctness* of violations final diagnoses that are generated by *VDT*, it might be useful to recall that a final diagnosis of a violation is a report based on confirmation criterion discussed in Section 5.5.1. Therefore, the final diagnosis of a violation reports the *confirmed* and *unconfirmed violation observations* i.e. events involved in the violation. More specifically, a *violation observation* P is classified as a *confirmed* event if the belief in the genuineness of P is greater than or equal to the corresponding disbelief, i.e., Bel(Genuine(P)) $\geq$ Bel(¬Genuine(P)).

Table 6-5 Example table of VDT correctness results

Confirmation criterion	VDT_Recall_F	$VDT\_Precision_F$	VDT_Recall_G	$VDT\_Precision_G$
Bel(Genuine(P)) ≥ Bel(¬Genuine(P))	0.35	0.17	0.59	0.79

A brief guide of how the reader should read and interpret Table 6-5 is as follows:

- VDT_Recall_F equals to the percentage of *fake violation observations* that classified as *unconfirmed*. For instance, the VDT_Recall_F that equals to 0.35 shows that the 35% of the total *fake violation observations* have been classified as *unconfirmed*. Ideally, we would like to have VDT_Recall_F values greater than 0.5.
- $VDT\_Precision_F$ equals to the percentage of the *violation observations* that classified as *unconfirmed* and happen to be *fake* events. For instance, the $VDT\_Precision_F$ that equals to 0.17 shows that the 17% of the total *violation observations* that have been classified as *unconfirmed* are *fake* events. Ideally, we would like to have $VDT\_Precision_F$ values greater than 0.5.
- VDT_Recall_G equals to the percentage of *genuine violation observations* that classified as *confirmed*. For instance, the VDT_Recall_G that equals to 0.59 shows that the 59% of the total *genuine violation observations* have been classified as *confirmed*. Ideally, we would like to have VDT_Recall_G values greater than 0.5.
- $VDT\_Precision_G$ equals to the percentage of the *violation observations* that classified as *confirmed* and happen to be *genuine* events. For instance, the $VDT\_Precision_G$ that equals to 0.79 shows that the 79% of the total *violation observations* that have been classified as *confirmed* are *genuine* events. Ideally, we would like to have $VDT\_Precision_G$ values greater than 0.5.

6.5.1 ExplanationConfiguration1 Experiments Results

In this section, the results of experiments expConf1_1.5, expConf1_2.3, expConf1_2.5, expConf1_5, expConf1_7.5, and expConf1_10 specified in Section 6.4.2.1 are presented. As a reminder, the results of the above experiments have been generated by running the monitoring and diagnosis prototype with the following common inputs:

- a set of 5000 events that have been generated as discussed in Section 6.4.2.1

- the *LBACS* monitoring theory as described in Section 6.4.2, and
- the belief functions a1 and a2 set to 0.3 and 0.4 respectively (see also Section 6.4.2)

6.5.1.1 expConf1_1.5 results

The following results have been generated by setting the diagnosis window equal to 1.5 sec.

6.5.1.1.1 EGBT correctness results

Table 6-6 contains the results for the *EGBT correctness* metrics for experiment expConf1_1.5, whilst Figure 6-11 illustrates a representative chart of the given experimental results.

Table 6-6 – EGBT correctness results for experiment expConf1_1.5

Belief Range	$EGBT\_Recall_F$	$EGBT\_Precision_F$	$EGBT\_Recall_G$	$EGBT\_Precision_G$
[0, 0.1)	0.10	1.00	0.00	0.00
[0.1, 0.2)	0.00	N/A	0.00	N/A
[0.2, 0.3)	0.00	N/A	0.00	N/A
[0.3, 0.4)	0.00	0.00	0.24	1.00
[0.4, 0.5)	0.25	0.26	0.17	0.74
[0.5, 0.6)	0.00	N/A	0.00	N/A
[0.6, 0.7)	0.25	0.38	0.10	0.62
[0.7, 0.8)	0.25	0.22	0.22	0.78
[0.8, 0.9)	0.05	0.08	0.14	0.92
[0.9, 1]	0.10	0.15	0.13	0.85

Observing the $EGBT\_Recall_F$ results in above table, there is an undesired low percentage (10%) of *fake* events whose belief values lie within [0, 0.1), whilst the percentages of *fake* events in higher ranges are low and quite satisfying. $EGBT\_Precision_F$ results are quite satisfying as all events, whose belief value computed within [0, 0.1), happen to be *fake*. Also, the percentages of events having belief values within belief ranges higher than 0.7 and being *fake*, are as low as ideally expected.

Regarding $EGBT\_Recall_G$, the results in low belief ranges are as expected due to none *genuine* event with belief value within [0, 0.3) found. However, the $EGBT\_Recall_G$ results in higher belief ranges are not satisfying due to the fact that there are low percentages of *genuine* events with belief values within [0.7, 1]. Finally, the $EGBT\_Precision_G$ results are quite satisfying because none low belief valued events happened to be *genuine*, and the percentages of events having belief values greater than 0.7 and being *genuine* are quite high.

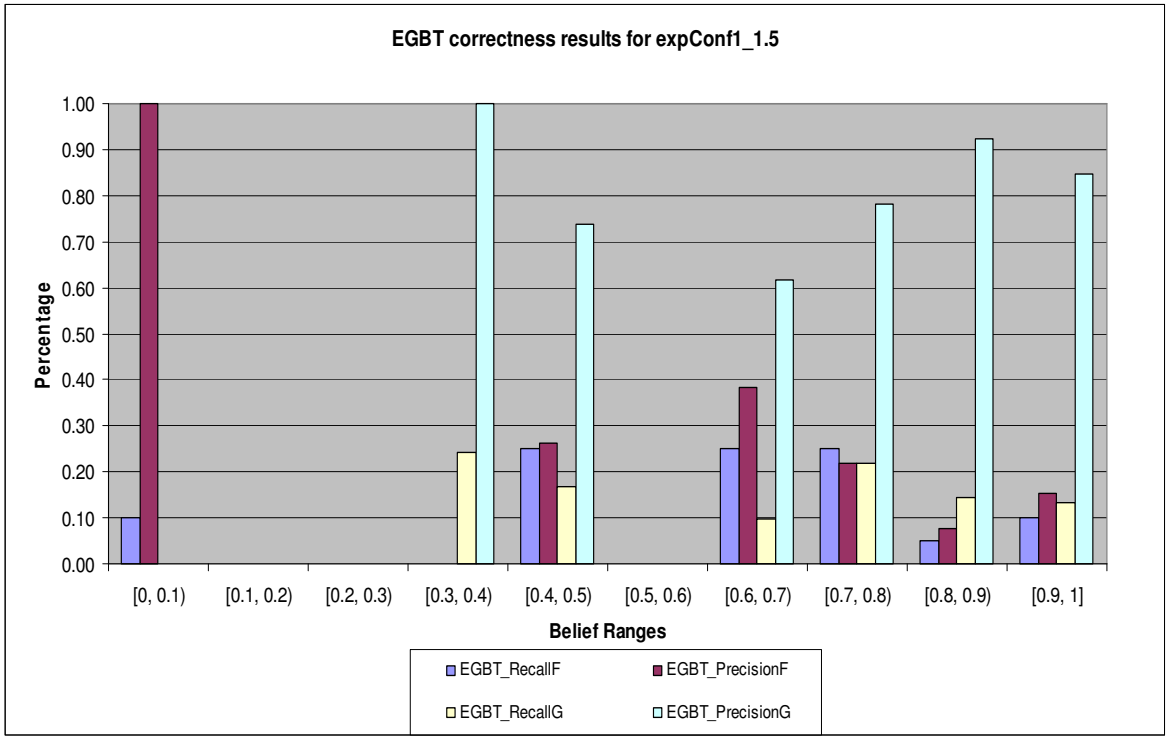

Figure 6-11 – EGBT correctness results for expConf1_1.5

6.5.1.1.2 VDT correctness results

The following table (Table 6-7) accumulates the results of *VDT correctness* metrics for experiment expConf1_1.5, whilst Figure 6-12 illustrates a representative chart of the given experimental results.

Table 6-7 - VDT correctness results for experiment expConf1_1.5

Confirmation criterion	VDT_Recall_F	$VDT\_Precision_F$	VDT_Recall_G	$VDT\_Precision_G$
Bel(Genuine(P)) ≥ Bel(¬Genuine(P))	0.35	0.17	0.59	0.79

Table 6-7 presents rather undesired VDT_Recall_F and $VDT\_Precision_F$ results. In particular, *VDT* has classified as *unconfirmed* events only the 35% of the total *fake violation observations*. Also, only the 17% of the total *unconfirmed violation observations* happened to be *fake* events.

On the other hand, VDT_Recall_G and $VDT\_Precision_G$ results are quite satisfying. More specifically, the 59% of the total *genuine violation observations* have been flagged as *confirmed* events by *VDT*. Finally, the 79% of the total *confirmed violation observations* happened to be *genuine* events.

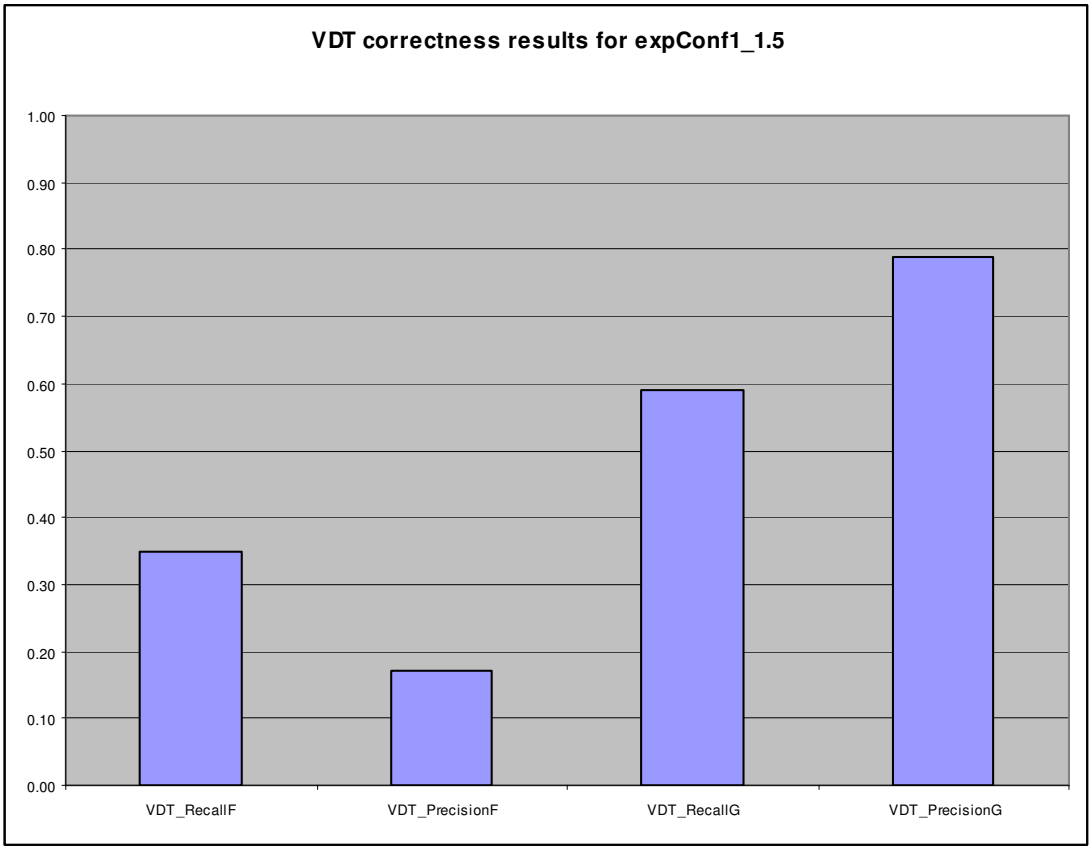

Figure 6-12 – VDT correctness results for expConf1_1.5

6.5.1.1.3 Responsiveness results

The following table (Table 6-8) presents the results of responsiveness metrics as specified in Section 6.3.2.

Table 6-8 - EGBT and VDT responsiveness results for experiment expConf1_1.5

Computational time types	Mean (sec)	Standard deviation (sec)	Max (sec)	Min (sec)
EGBT belief computational time	20.57	11.51	44.64	0.74
VDT diagnosis generation time	41.16	22.74	88.98	0.00

According to the results presented in above table, the time *EGBT* needed to compute the belief and plausibility of an event was 20.57 sec in average, with a standard deviation of 11.51 sec. The max and min computational times occurred are 44.64 sec and 0.74 sec respectively.

The time *VDT* consumed to generate a final diagnosis for a violation was is 41.16 sec in average, with a standard deviation of 22.74 sec. The max and min final diagnosis generation times occurred were 88.98 sec and 0 sec respectively.

6.5.1.2 expConf1_2.3 results

The following results have been generated by setting the diagnosis window equal to 2.3 sec.

6.5.1.2.1 EGBT correctness results

Table 6-9 contains the results for the *EGBT correctness* metrics for experiment expConf1_2.3, whilst Figure 6-13 illustrates a representative chart of the given experimental results.

Table 6-9 - EGBT correctness results for experiment expConf1_2.3

Belief Range	$EGBT\_Recall_F$	$EGBT\_Precision_F$	$EGBT\_Recall_G$	$EGBT\_Precision_G$
[0, 0.1)	0.15	1.00	0.00	0.00
[0.1, 0.2)	0.00	N/A	0.00	N/A
[0.2, 0.3)	0.00	N/A	0.00	N/A
[0.3, 0.4)	0.00	0.00	0.24	1.00
[0.4, 0.5)	0.45	0.31	0.24	0.69
[0.5, 0.6)	0.00	N/A	0.00	N/A
[0.6, 0.7)	0.20	0.22	0.17	0.78
[0.7, 0.8)	0.15	0.20	0.14	0.80
[0.8, 0.9)	0.05	0.08	0.14	0.92
[0.9, 1]	0.00	0.00	0.06	1.00

Observing the $EGBT\_Recall_F$ results in above table, there are undesired low percentages (15%, 0% and 0%) of *fake* events whose belief values lie within [0, 0.3), whilst the percentages of *fake* events in higher ranges are low as ideally expected. $EGBT\_Precision_F$ results are quite satisfying. This is because all events, whose belief value computed within [0, 0.1), happen to be *fake*. Also, the percentages of events having belief values within ranges higher than 0.6 (20%, 8%, and 0%) and being *fake*, are as low as expected.

Regarding $EGBT\_Recall_G$, the results in low belief ranges are quite satisfying as none *genuine* event with belief value within [0, 0.3) found. However, the $EGBT\_Recall_G$ results in higher belief ranges are not satisfying as there are low percentages (14%, 14%, and 6%) of *genuine* events with belief values within [0.7, 1]. Finally, the $EGBT\_Precision_G$ results are quite satisfying as none event with low belief value happened to be *genuine*, and the percentages of events, whose belief values were computed greater than 0.7 and which happened to be *genuine*, are quite high (80%, 92%, and 100%).

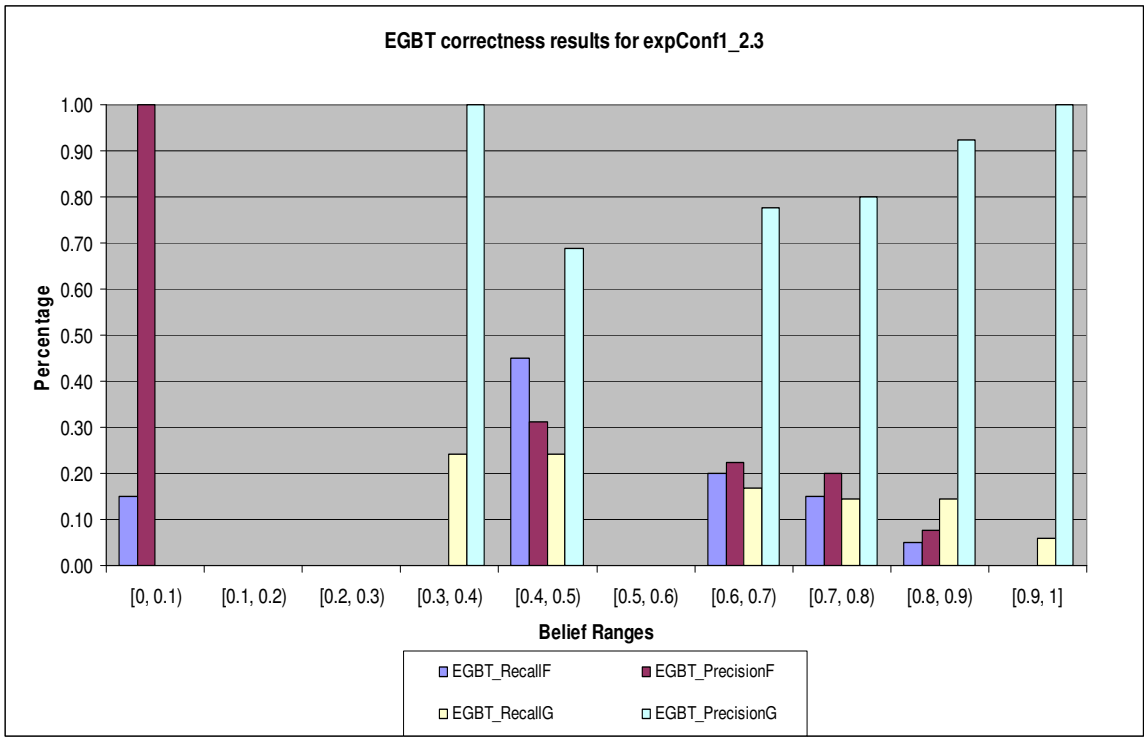

Figure 6-13 – EGBT correctness results for expConf1_2.3

6.5.1.2.2 VDT correctness results

The following table (Table 6-10) accumulates the results of *VDT correctness* metrics for experiment expConf1_2.3, whilst Figure 6-14 illustrates a representative chart of the given experimental results.

Table 6-10 - VDT correctness results for experiment expConf1_2.3

Confirmation criterion	VDT_Recall_F	$VDT\_Precision_F$	VDT_Recall_G	$VDT\_Precision_G$
Bel(Genuine(P)) ≥ Bel(¬Genuine(P))	0.60	0.23	0.52	0.84

Table 6-10 presents a quite satisfying VDT_Recall_F result. In particular, *VDT* has classified as *unconfirmed* events the 60% of the total *fake violation observations*. On the other hand, the $VDT\_Precision_F$ result is rather undesired due to the fact that only the 23% of the total *unconfirmed violation observations* happened to be *fake* events.

VDT_Recall_G and $VDT\_Precision_G$ results are quite satisfying. More specifically, the 52% of the total *genuine violation observations* have been flagged as *confirmed* events by

VDT. Finally, the 84% of the total *confirmed violation observations* happened to be *genuine* events.

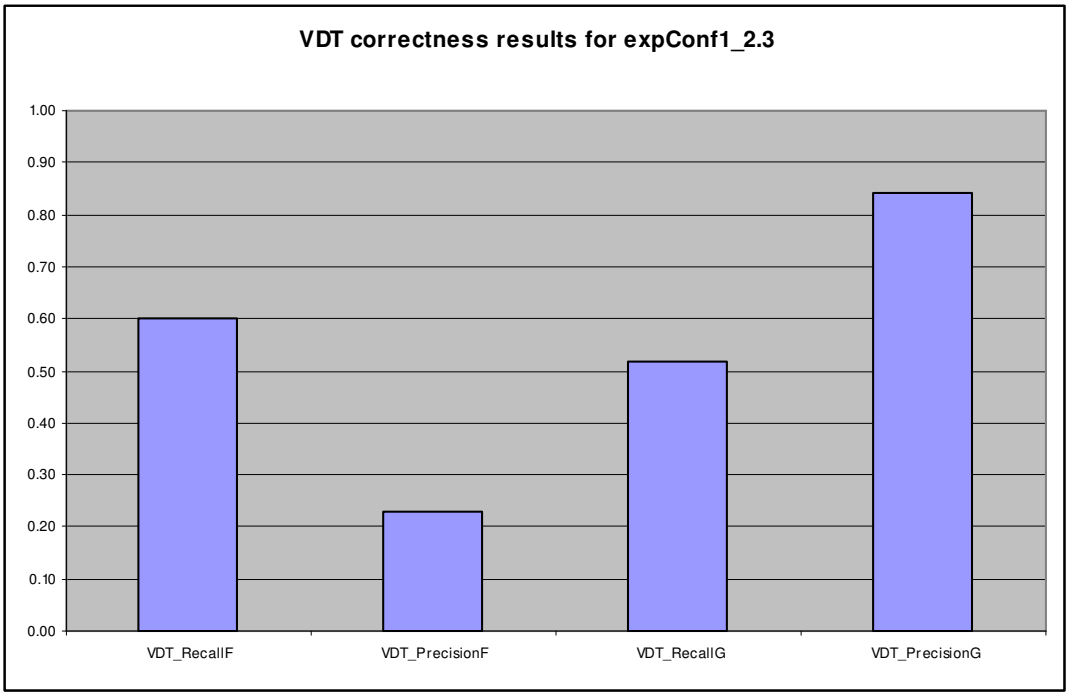

Figure 6-14 – VDT correctness results for expConf1_2.3

6.5.1.2.3 Responsiveness results

The following table (Table 6-11) presents the results of responsiveness metrics as specified in Section 6.3.2.

Table 6-11 - EGBT and VDT responsiveness results for experiment expConf1_2.3

Computational time types	Mean (sec)	Standard deviation (sec)	Max (sec)	Min (sec)
EGBT belief computational time	26.42	14.34	55.08	0.19
VDT diagnosis generation time	52.86	28.42	109.66	0.00

According to the results presented in above table, the time *EGBT* needed to compute the belief and plausibility of an event was 26.42 sec in average, with a standard deviation of 14.34 sec. The max and min computational times occurred are 55.08 sec and 0.19 sec respectively.

The time *VDT* consumed to generate a final diagnosis for a violation was is 52.86 sec in average, with a standard deviation of 28.42 sec. The max and min final diagnosis generation times occurred were 109.66 sec and 0 sec respectively.

6.5.1.3 expConf1_2.5 results

The following results have been generated by setting the diagnosis window equal to 2.5 sec.

6.5.1.3.1 EGBT correctness results

Table 6-12 contains the results for the *EGBT correctness* metrics for experiment expConf1_2.5, whilst Figure 6-15 illustrates a representative chart of the given experimental results.

Table 6-12 - EGBT correctness results for experiment expConf1_2.5

Belief Range	$EGBT\_Recall_F$	$EGBT\_Precision_F$	$EGBT\_Recall_G$	$EGBT\_Precision_G$
[0, 0.1)	0.25	0.71	0.02	0.29
[0.1, 0.2)	0.00	N/A	0.00	N/A
[0.2, 0.3)	0.00	N/A	0.00	N/A
[0.3, 0.4)	0.00	0.00	0.24	1.00
[0.4, 0.5)	0.30	0.21	0.27	0.79
[0.5, 0.6)	0.00	N/A	0.00	N/A
[0.6, 0.7)	0.25	0.28	0.16	0.72
[0.7, 0.8)	0.15	0.20	0.14	0.80
[0.8, 0.9)	0.05	0.11	0.10	0.89
[0.9, 1]	0.00	0.00	0.07	1.00

Observing the $EGBT\_Recall_F$ results in above table, there are undesired low percentages (25%, 0%, and 0%) of *fake* events whose belief values lie within [0, 0.3), whilst the percentages of *fake* events in higher ranges are low as ideally expected (15%, 5%, and 0%). $EGBT\_Precision_F$ results are quite satisfying. This is because the 71% of the events, whose belief value computed within [0, 0.1), happen to be *fake*. Also, the percentages of events having belief values within ranges greater than 0.7 and being *fake*, are as low as expected (20%, 11%, and 0%).

Regarding *EGBT_Recall*$_G$, the results in low belief ranges are quite satisfying as only the 2% of *genuine* events found with a belief value within [0, 0.1), and none genuine event found with a belief value within [0.1, 0.3). However, the *EGBT_Recall*$_G$ results in higher belief ranges are not satisfying as there are low percentages of *genuine* events with belief values within [0.7, 1] (14%, 10%, and 7%). Finally, the *EGBT_Precision*$_G$ results are almost satisfying as only the 29% of events with low belief value (within [0, 0.1)) happened to be *genuine*, and the percentages of events, whose belief values were computed greater than 0.7 and which happened to be *genuine*, are quite high (80%, 89%, and 100%).

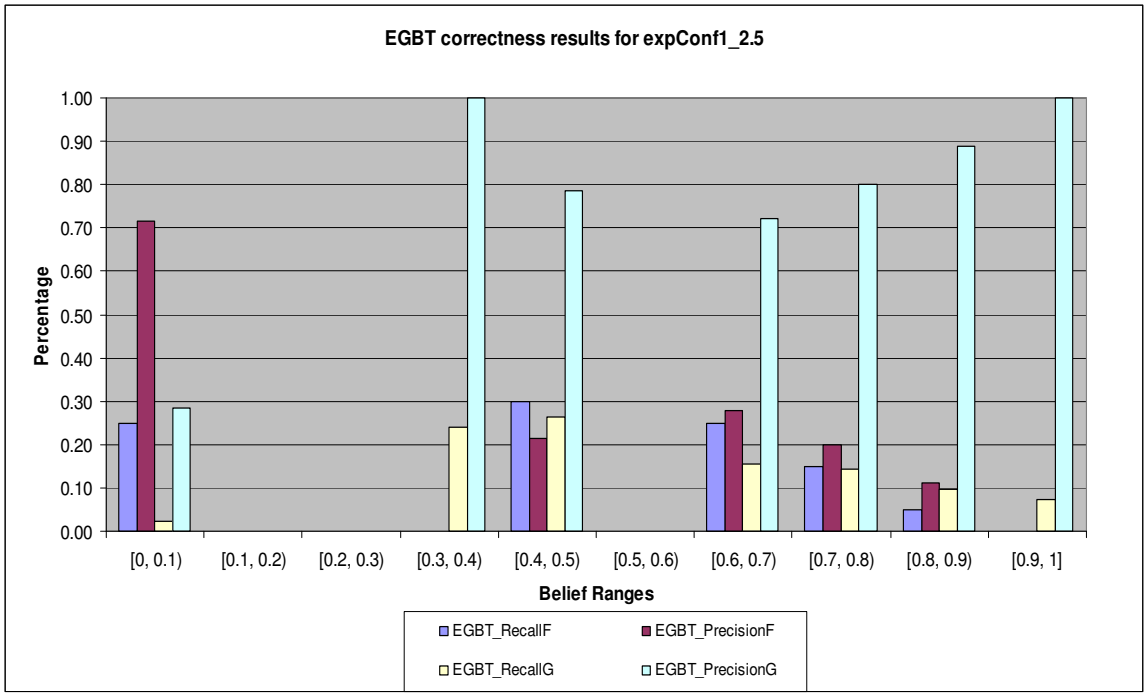

Figure 6-15 – EGBT correctness results for expConf1_2.5

6.5.1.3.2 VDT correctness results

The following table (Table 6-13) accumulates the results of *VDT correctness* metrics for experiment expConf1_2.5, whilst Figure 6-16 illustrates a representative chart of the given experimental results.

Table 6-13 - VDT correctness results for experiment expConf1_2.5

Confirmation criterion	VDT_Recall_F	$VDT\_Precision_F$	VDT_Recall_G	$VDT\_Precision_G$
Bel(Genuine(P)) ≥ Bel(¬Genuine(P))	0.55	0.20	0.47	0.81

Table 6-13 presents an almost satisfying VDT_Recall_F result. In particular, VDT has classified as *unconfirmed* events the 55% of the total *fake violation observations*. On the other hand, the $VDT\_Precision_F$ result is rather undesired due to the fact that only the 20% of the total *unconfirmed violation observations* happened to be *fake* events.

VDT_Recall_G result is rather undesired due to fact that only the 47% of the total *genuine violation observations* have been flagged as *confirmed* events by *VDT*. On the contrary $VDT\_Precision_G$ result is quite satisfying, as the 81% of the total *confirmed violation observations* happened to be *genuine* events.

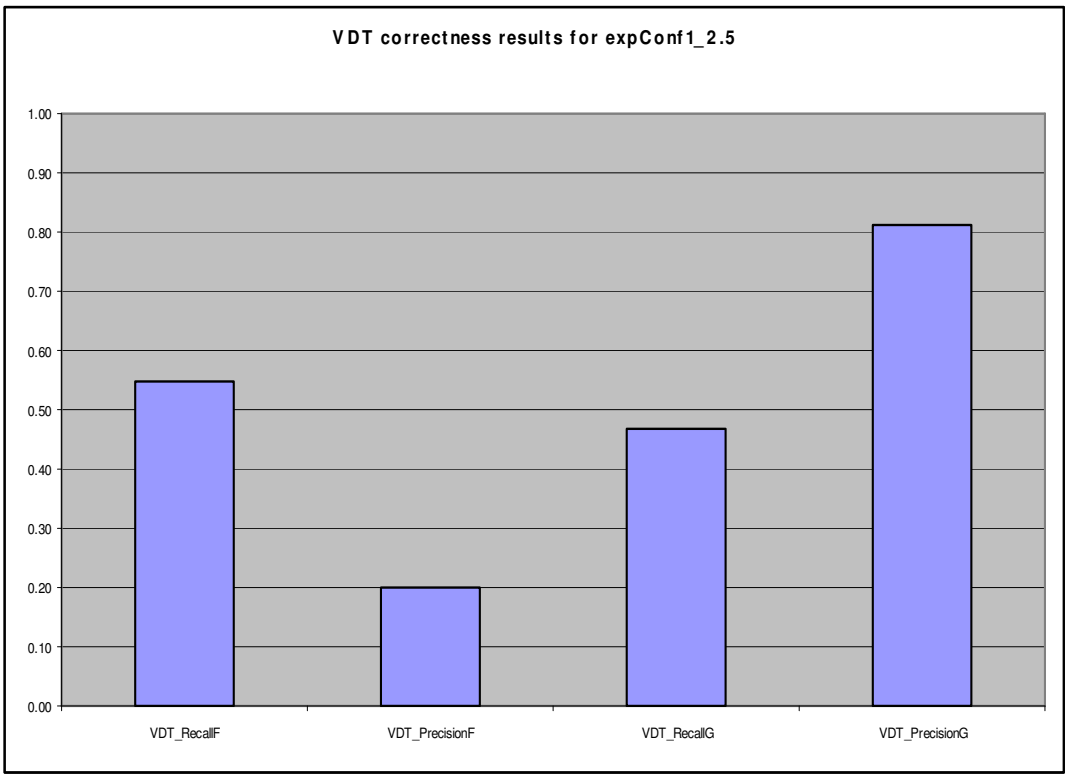

Figure 6-16 – VDT correctness results for expConf1_2.5

6.5.1.3.3 Responsiveness results

The following table (Table 6-14) presents the results of responsiveness metrics as specified in Section 6.3.2.

Table 6-14 - EGBT and VDT responsiveness results for experiment expConf1_2.5

Computational time types	Mean (sec)	Standard deviation (sec)	Max (sec)	Min (sec)
EGBT belief computational time	27.62	14.35	55.31	0.19
VDT diagnosis generation time	55.26	28.46	110.08	0.00

According to the results presented in above table, the time *EGBT* needed to compute the belief and plausibility of an event was 27.62 sec in average, with a standard deviation of 14.35 sec. The max and min computational times occurred are 55.31 sec and 0.19 sec respectively.

The time *VDT* consumed to generate a final diagnosis for a violation was is 55.26 sec in average, with a standard deviation of 28.46 sec. The max and min final diagnosis generation times occurred were 110.08 sec and 0 sec respectively.

6.5.1.4 expConf1_5 results

The following results have been generated by setting the diagnosis window equal to 5 sec.

6.5.1.4.1 EGBT correctness results

Table 6-15 contains the results for the *EGBT correctness* metrics for experiment expConf1_5, whilst Figure 6-17 illustrates a representative chart of the given experimental results.

Table 6-15 - EGBT correctness results for experiment expConf1_5

Belief Range	$EGBT\_Recall_F$	$EGBT\_Precision_F$	$EGBT\_Recall_G$	$EGBT\_Precision_G$
[0, 0.1)	0.60	0.29	0.35	0.71
[0.1, 0.2)	0.00	N/A	0.00	N/A
[0.2, 0.3)	0.00	N/A	0.00	N/A
[0.3, 0.4)	0.00	0.00	0.24	1.00
[0.4, 0.5)	0.05	0.17	0.06	0.83
[0.5, 0.6)	0.00	N/A	0.00	N/A
[0.6, 0.7)	0.25	0.17	0.30	0.83
[0.7, 0.8)	0.05	1.00	0.00	0.00
[0.8, 0.9)	0.05	0.20	0.05	0.80
[0.9, 1]	0.00	N/A	0.00	N/A

The $EGBT\_Recall_F$ results in above table are quite satisfying. This is due to facts that the 60% of *fake* events found with belief values lying within [0, 0.1), and the percentages of *fake* events in higher ranges are low as ideally expected (5%, 5% and 0%). On the other hand, $EGBT\_Precision_F$ results are rather undesired. Only the 29% of the events, whose belief value computed within [0, 0.1), happen to be *fake*. Also, another undesired result indicated that all events having belief values within [0.7, 0.8), were *fake*. Finally, only for the belief range [0.8, 0.9), $EGBT\_Precision_F$ is quite low (20%) as expected.

Regarding $EGBT\_Recall_G$, the results in low belief ranges are almost satisfying as the 35% of *genuine* events found with a belief value within [0, 0.1). However, the $EGBT\_Recall_G$ results in higher belief ranges are undesired as there are low percentages of *genuine* events with belief values within [0.7, 1] (0%, 5%, and 0%). Similarly, the $EGBT\_Precision_G$ result for the low range [0, 0.1) is quite undesired as the 71% of events with such low belief values happened to be *genuine*. Also, there were no *genuine* events among the events, whose belief values within [0.7, 0.8). On the contrary, there is a satisfying observation with regards to events having belief values within [0.8, 0.9). In particular, the 80% of such events happened to be *genuine*.

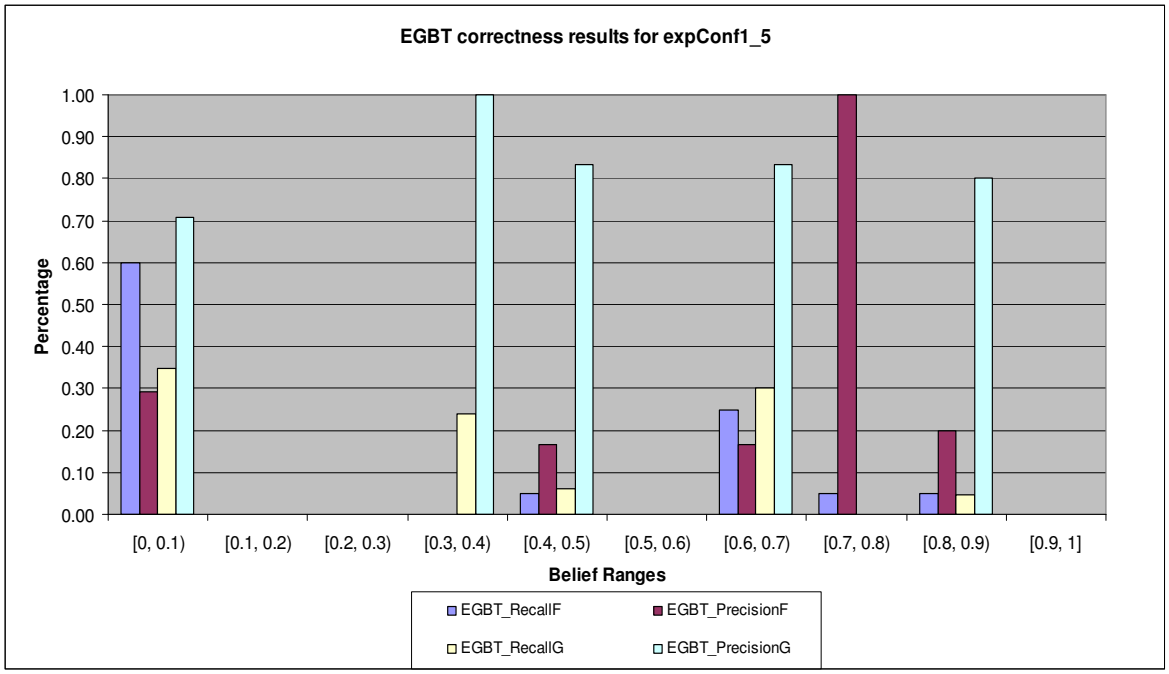

Figure 6-17 – EGBT correctness results for expConf1_5

6.5.1.4.2 VDT correctness results

The following table (Table 6-16) accumulates the results of *VDT correctness* metrics for experiment expConf1_5, whilst Figure 6-18 illustrates a representative chart of the given experimental results.

Table 6-16 - VDT correctness results for experiment expConf1_5

Confirmation criterion	VDT_Recall_F	$VDT\_Precision_F$	VDT_Recall_G	$VDT\_Precision_G$
Bel(Genuine(P)) ≥ Bel(¬Genuine(P))	0.65	0.19	0.35	0.81

Table 6-16 presents a satisfying VDT_Recall_F result. In particular, *VDT* has classified as *unconfirmed* events the 65% of the total *fake violation observations*. On the other hand, the $VDT\_Precision_F$ result is rather undesired due to the fact that only the 19% of the total *unconfirmed violation observations* happened to be *fake* events.

VDT_Recall_G result is rather undesired due to fact that only the 35% of the total *genuine violation observations* have been flagged as *confirmed* events by *VDT*. On the contrary $VDT\_Precision_G$ result is quite satisfying, as the 81% of the total *confirmed violation observations* happened to be *genuine* events.

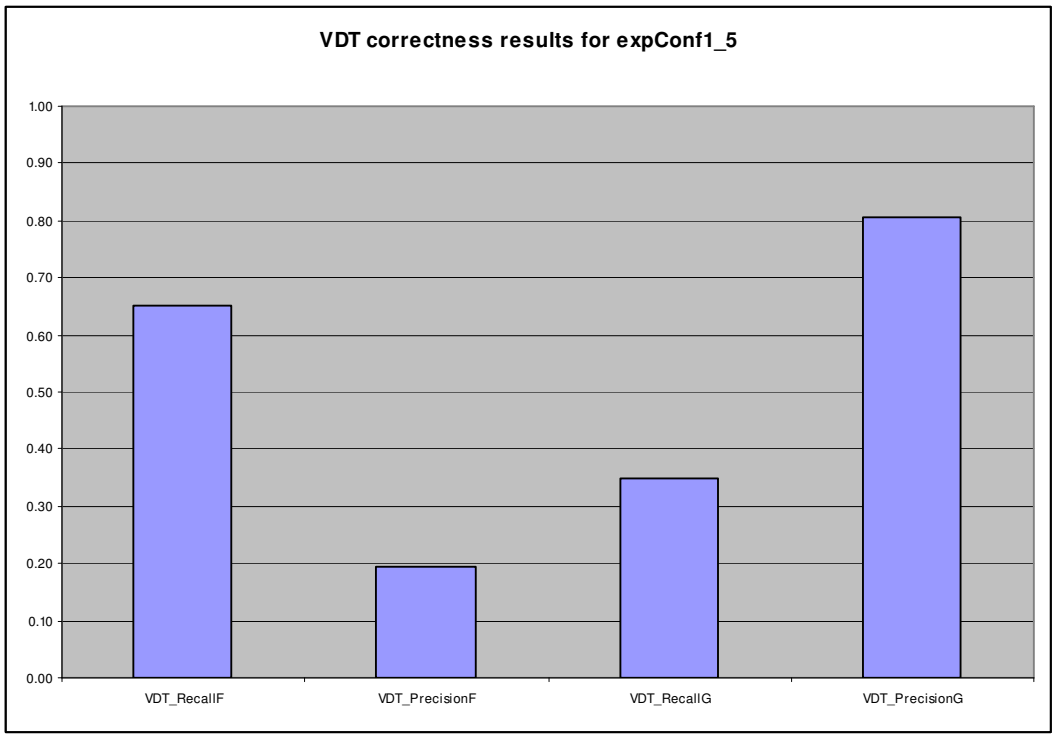

Figure 6-18 – VDT correctness results for expConf1_5

6.5.1.4.3 Responsiveness results

The following table (Table 6-17) presents the results of responsiveness metrics as specified in Section 6.3.2.

Table 6-17 - EGBT and VDT responsiveness results for experiment expConf1_5

Computational time types	Mean (sec)	Standard deviation (sec)	Max (sec)	Min (sec)
EGBT belief computational time	69.53	51.66	263.52	0.19
VDT diagnosis generation time	139.07	103.46	526.88	0.00

According to the results presented in above table, the time *EGBT* needed to compute the belief and plausibility of an event was 69.53 sec in average, with a standard deviation of 51.66 sec. The max and min computational times occurred are 263.52 sec and 0.19 sec respectively.

The time *VDT* consumed to generate a final diagnosis for a violation was is 139.07 sec in average, with a standard deviation of 103.46 sec. The max and min final diagnosis generation times occurred were 526.88 sec and 0 sec respectively.

6.5.1.5 expConf1_7.5 results

The following results have been generated by setting the diagnosis window equal to 7.5 sec.

6.5.1.5.1 EGBT correctness results

Table 6-18 contains the results for the *EGBT correctness* metrics for experiment expConf1_7.5, whilst Figure 6-19 illustrates a representative chart of the given experimental results.

Table 6-18 - EGBT correctness results for experiment expConf1_7.5

Belief Range	$EGBT\_Recall_F$	$EGBT\_Precision_F$	$EGBT\_Recall_G$	$EGBT\_Precision_G$
[0, 0.1)	0.70	0.25	0.51	0.75
[0.1, 0.2)	0.00	N/A	0.00	N/A
[0.2, 0.3)	0.00	N/A	0.00	N/A
[0.3, 0.4)	0.00	0.00	0.24	1.00
[0.4, 0.5)	0.25	0.23	0.20	0.77
[0.5, 0.6)	0.00	N/A	0.00	N/A
[0.6, 0.7)	0.05	0.25	0.04	0.75
[0.7, 0.8)	0.00	0.00	0.01	1.00
[0.8, 0.9)	0.00	N/A	0.00	N/A
[0.9, 1]	0.00	N/A	0.00	N/A

The $EGBT\_Recall_F$ results in above table are quite satisfying. This is due to the fact that the 70% of *fake* events found with belief values lying within [0, 0.1), and the percentages of *fake* events with respect to range [0.7, 1] are zero as ideally expected. On the other hand, $EGBT\_Precision_F$ results are rather complicated. Only the 25% of the events, whose belief value computed within [0, 0.1), happen to be *fake*. On the contrary, a rather satisfying result indicated that all events having belief values within [0.7, 0.8), were no *fake*.

Regarding *EGBT_Recall*$_G$, the results in the low belief ranges are rather undesired as the 51% of *genuine* events found with a belief value within [0, 0.1). Moreover, the *EGBT_Recall*$_G$ results in higher belief ranges are undesired as well. In particular there are very low (1%) and zero percentages of *genuine* events with belief values within [0.7, 1]. Similarly, the *EGBT_Precision*$_G$ result for the low range [0, 0.1) is quite undesired as the 75% of events with such low belief values happened to be *genuine*. On the contrary, there is a satisfying experimental observation with regards to events having belief values within [0.7, 0.8). In particular, the 100% of such events happened to be *genuine*.

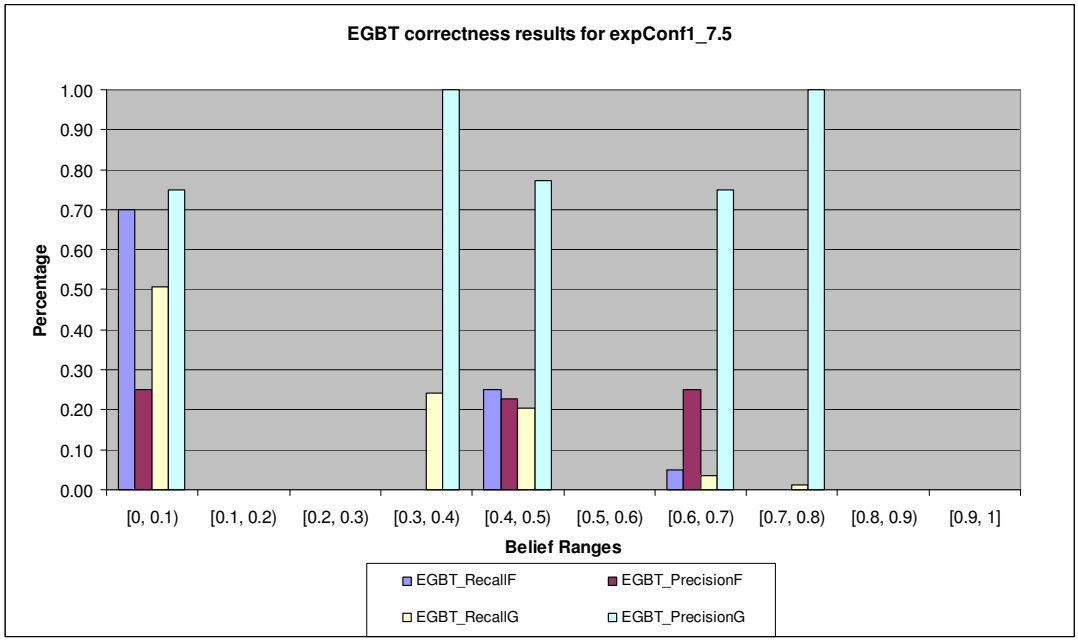

Figure 6-19 – EGBT correctness results for expConf1_7.5

6.5.1.5.2 VDT correctness results

The following table (Table 6-19) accumulates the results of *VDT correctness* metrics for experiment expConf1_7.5, whilst Figure 6-20 illustrates a representative chart of the given experimental results.

Table 6-19 - VDT correctness results for experiment expConf1_7.5

Confirmation criterion	VDT_Recall$_F$	VDT_Precision$_F$	VDT_Recall$_G$	VDT_Precision$_G$
Bel(Genuine(P)) ≥ Bel(¬Genuine(P))	0.95	0.19	0.05	0.80

Table 6-19 presents a quite satisfying VDT_Recall_F experimental result. In particular, *VDT* has classified as *unconfirmed* events the 95% of the total *fake violation observations*. On the other hand, the $VDT\_Precision_F$ result is rather undesired due to the fact that only the 19% of the total *unconfirmed violation observations* happened to be *fake* events.

VDT_Recall_G result is quite undesired due to fact that only the 5% of the total *genuine violation observations* have been flagged as *confirmed* events by *VDT*. On the contrary $VDT\_Precision_G$ result is quite satisfying, as the 80% of the total *confirmed violation observations* happened to be *genuine* events.

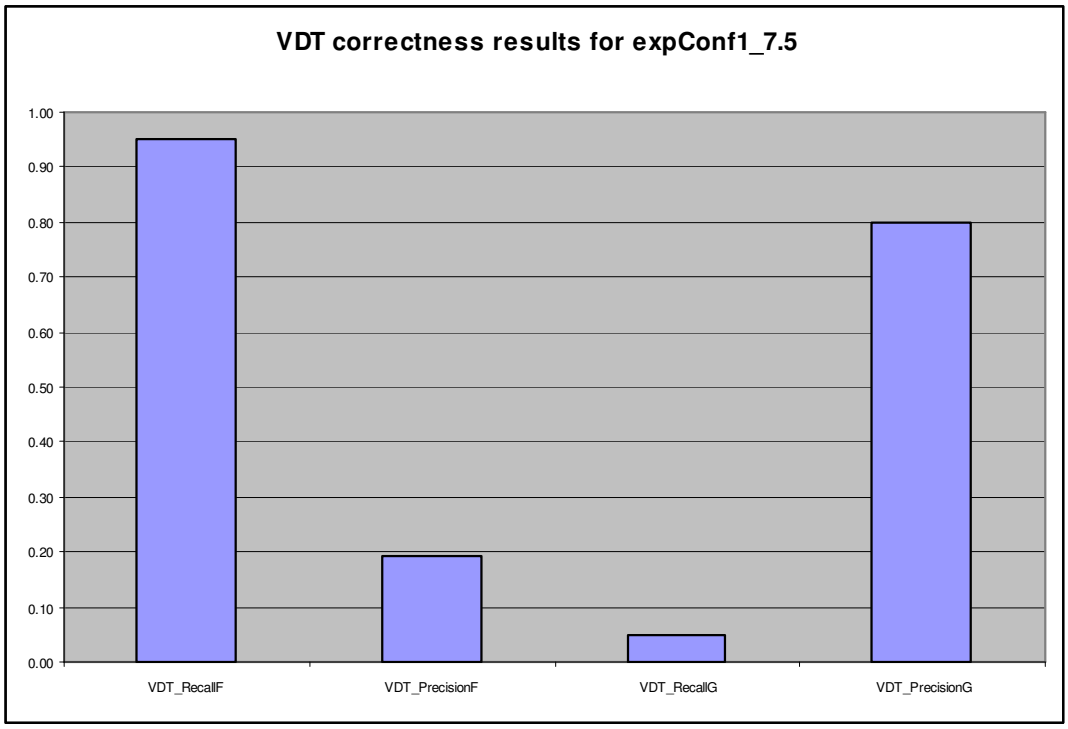

Figure 6-20 – VDT correctness results for expConf1_7.5

6.5.1.5.3 Responsiveness results

The following table (Table 6-20) presents the results of responsiveness metrics as specified in Section 6.3.2.

Table 6-20 - EGBT and VDT responsiveness results for experiment expConf1_7.5

Computational time types	Mean (sec)	Standard deviation (sec)	Max (sec)	Min (sec)
EGBT belief computational time	138.28	115.87	619.53	0.19
VDT diagnosis generation time	276.57	231.04	1237.75	0.00

According to the results presented in above table, the time *EGBT* needed to compute the belief and plausibility of an event was 138.28 sec in average, with a standard deviation equal to 115.87 sec. The max and min computational times occurred are 619.53 sec and 0.19 sec respectively.

The time *VDT* consumed to generate a final diagnosis for a violation was 276.57 sec in average, with a standard deviation equal to 231.04 sec. The max and min final diagnosis generation times occurred were 1237.75 sec and 0 sec respectively.

6.5.1.6 expConf1_10 results

The following results have been generated by setting the diagnosis window equal to 10 sec.

6.5.1.6.1 EGBT correctness results

Table 6-18 contains the results for the *EGBT correctness* metrics for experiment expConf1_10, whilst Figure 6-21 illustrates a representative chart of the given experimental results.

Table 6-21 - EGBT correctness results for experiment expConf1_10

Belief Range	$EGBT\_Recall_F$	$EGBT\_Precision_F$	$EGBT\_Recall_G$	$EGBT\_Precision_G$
[0, 0.1)	1.00	0.24	0.75	0.76
[0.1, 0.2)	0.00	N/A	0.00	N/A
[0.2, 0.3)	0.00	N/A	0.00	N/A
[0.3, 0.4)	0.00	0.00	0.24	1.00
[0.4, 0.5)	0.00	N/A	0.00	N/A
[0.5, 0.6)	0.00	N/A	0.00	N/A
[0.6, 0.7)	0.00	N/A	0.00	N/A
[0.7, 0.8)	0.00	N/A	0.00	N/A
[0.8, 0.9)	0.00	N/A	0.00	N/A
[0.9, 1]	0.00	0.00	0.01	1.00

The $EGBT\_Recall_F$ results in above table are quite satisfying. This is due to facts that all *fake* events found with belief values lying within [0, 0.1), and the percentages of *fake* events with respect to ranges [0.7, 1] are zero as ideally expected. On the other hand, $EGBT\_Precision_F$ results are rather complicated. Only the 24% of the events, whose belief value computed within [0, 0.1), happen to be *fake*. On the contrary, a rather satisfying result indicated that all events having belief values within [0.9, 1], were no *fake*.

Regarding $EGBT\_Recall_G$, the result with respect to range [0, 0.1) is undesired as the 75% of *genuine* events found with a belief value within [0, 0.1). On the contrary, the $EGBT\_Recall_G$ results in belief range [0.1, 0.3) are satisfying, as no *genuine* event having a belief value within the aforementioned range. The zero and very low (1%) percentages of *genuine* events with belief values within [0.7, 1] are totally undesired observations. Similarly, the $EGBT\_Precision_G$ result for the low range [0, 0.1) is undesired as the 76% of events with such low belief values happened to be *genuine*. On the contrary, there is a totally satisfying experimental observation with regards to events having belief values within [0.9, 1]. In particular, the 100% of such events happened to be *genuine*.

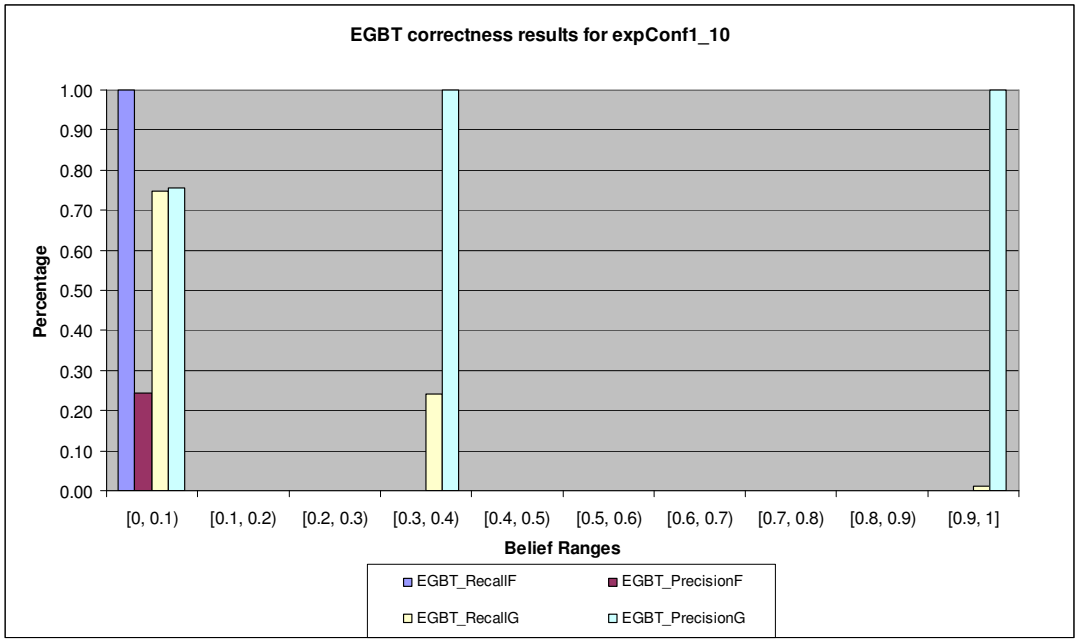

Figure 6-21 – EGBT correctness results for expConf1_10

6.5.1.6.2 VDT correctness results

The following table (Table 6-22) accumulates the results of *VDT correctness* metrics for experiment expConf1_10, whilst Figure 6-22 illustrates a representative chart of the given experimental results.

Table 6-22 - VDT correctness results for experiment expConf1_10

Confirmation criterion	VDT_Recall$_F$	VDT_Precision$_F$	VDT_Recall$_G$	VDT_Precision$_G$
Bel(Genuine(P)) ≥ Bel(¬Genuine(P))	1.00	0.20	0.01	1.00

Table 6-22 presents a totally satisfying *VDT_Recall$_F$* experimental result. In particular, *VDT* has classified as *unconfirmed* events all of the *fake violation observations*. On the other hand, the *VDT_Precision$_F$* result is rather undesired due to the fact that only the 20% of the total *unconfirmed violation observations* happened to be *fake* events.

VDT_Recall$_G$ result is quite undesired due to fact that only the 1% of the total *genuine violation observations* has been flagged as *confirmed* events by *VDT*. On the contrary *VDT_Precision$_G$* result is quite satisfying, as all of the *confirmed violation observations* happened to be *genuine* events.

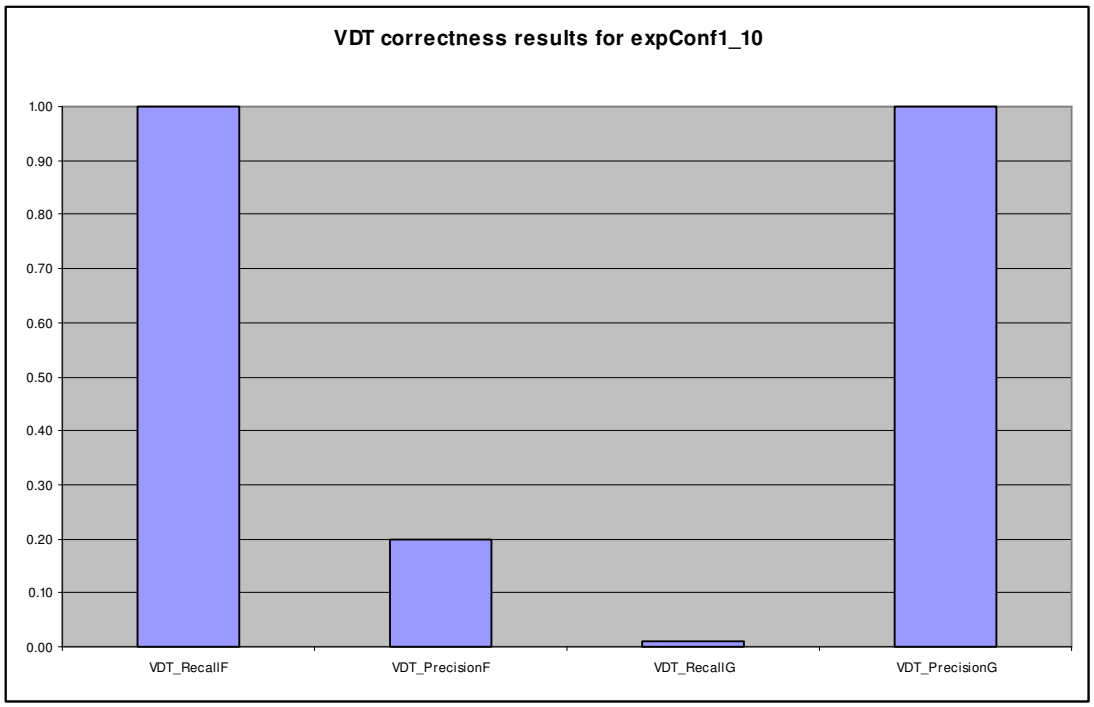

Figure 6-22 – VDT correctness results for expConf1_10

6.5.1.6.3 Responsiveness results

The following table (Table 6-23) presents the results of responsiveness metrics as specified in Section 6.3.2.

Table 6-23 - EGBT and VDT responsiveness results for experiment expConf1_10

Computational time types	Mean (sec)	Standard deviation (sec)	Max (sec)	Min (sec)
EGBT belief computational time	323.56	348.40	2106.11	0.19
VDT diagnosis generation time	647.13	665.56	3679.74	0.00

According to the results presented in above table, the time *EGBT* needed to compute the belief and plausibility of an event was 323.56 sec in average, with a standard deviation equal to 348.40 sec. The max and min computational times occurred are 2106.11 sec and 0.19 sec respectively.

The time *VDT* consumed to generate a final diagnosis for a violation was 647.13 sec in average, with a standard deviation equal to 665.56 sec. The max and min final diagnosis generation times occurred were 3679.74 sec and 0 sec respectively.

6.5.1.7 ExplanationConfiguration1 overall charts and discussion

In this section, we present charts that accumulate and compare the results of the individual *explanationConfiguration1* experiments. Based on the aforementioned charts, discussion on the experimental observations against the objective of the relevant experimental configuration follows.

6.5.1.7.1 EGBT correctness results charts and discussion

The following sections include charts and discussion on the overall *experimentalConfiguration1* results for each *EGBT correctness* metric i.e., $EGBT\_Recall_F$, $EGBT\_Precision_F$, $EGBT\_Recall_G$, and $EGBT\_Precision_G$. For each aforementioned metric, we provide charts with respect to low belief ranges, i.e., ranges within [0, 0.3), and high belief ranges, i.e., ranges within [0.7, 1]. It should be noted that results found within [0.3, 0.7) have been excluded from these charts and the following results analysis, as they might be considered as non useful and enough indicative information to be taken into account by a recovery decision making process.

6.5.1.7.1.1 $EGBT\_Recall_F$

Regarding the overall experimental $EGBT\_Recall_F$ results, Figure 6-23 and Figure 6-24 present the experimental $EGBT\_Recall_F$ behaviour against different diagnosis windows. More specifically, Figure 6-23 illustrates $EGBT\_Recall_F$ behaviour with respect to low belief ranges, i.e., ranges within [0, 0.3). Similary, Figure 6-24 shows $EGBT\_Recall_F$ behaviour with respect to belief ranges within [0.7, 1].

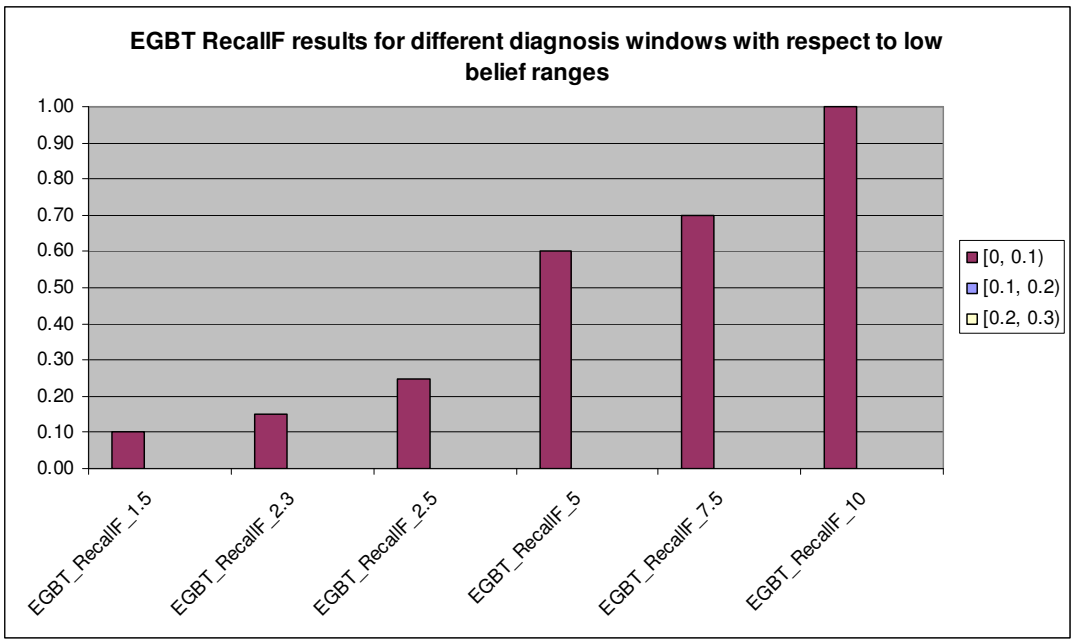

Figure 6-23 – EGBT_Recall$_F$ results for different diagnosis windows with respect to low belief ranges

By observing the chart of Figure 6-23, *EGBT_Recall$_F$* seems to increase as the diagnosis window increases. It should be noted that, *EGBT_Recall$_F$* should ideally have values much greater than 0.5 in low belief ranges. Therefore, although there are undesired low *EGBT_Recall$_F$* values for diagnosis windows less than 5 sec, *EGBT_Recall$_F$* increases quite satisfyingly as the diagnosis window is increased. Therefore, *EGBT* seems to compute belief values for *fake* events more correctly in case where long diagnosis windows are given and therefore more evidence is available, rather than when shorter diagnosis windows are given, and consequently less evidence is available.

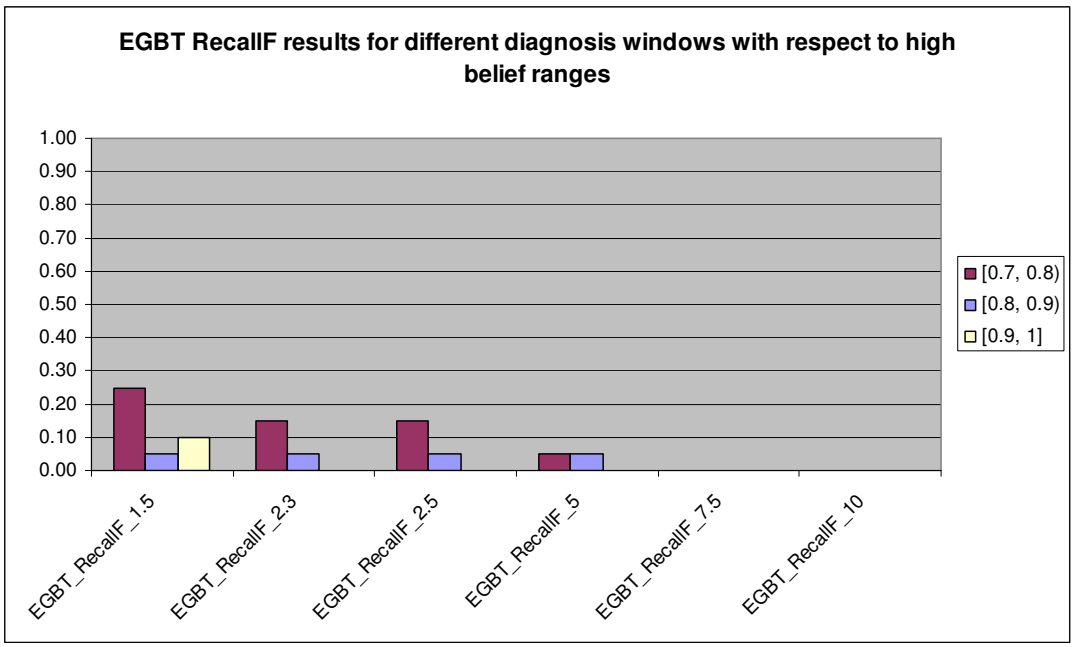

Figure 6-24 - EGBT_Recall$_F$ results for different diagnosis windows with respect to high belief ranges

Regarding belief ranges within [0.7, 1], it is expected that *EGBT_Recall$_F$* values should be quite low, i.e., much less than 0.5. For all *explanationConfiguration1* diagnosis windows, Figure 6-24 illustrates that indeed values are quite satisfying, as the max *EGBT_Recall$_F$* value is 0.25. Finally, one can observe that *EGBT_Recall$_F$* values are improving while the diagnosis window is increased.

As an overall observation about sensitivity of *EGBT* with respect to the belief values of *fake* events, we could say that the longer the given diagnosis window is, the greater probability is that *EGBT* would compute the *fake* events belief values within low ranges, Therefore, *EGBT* operates as expected with respect to *fake* events when long diagnosis windows are given.

6.5.1.7.1.2 EGBT_Precision$_F$

The charts regarding the overall experimental *EGBT_Precision$_F$* results with respect to low and high belief ranges are presented in Figure 6-25 and Figure 6-26 respectively.

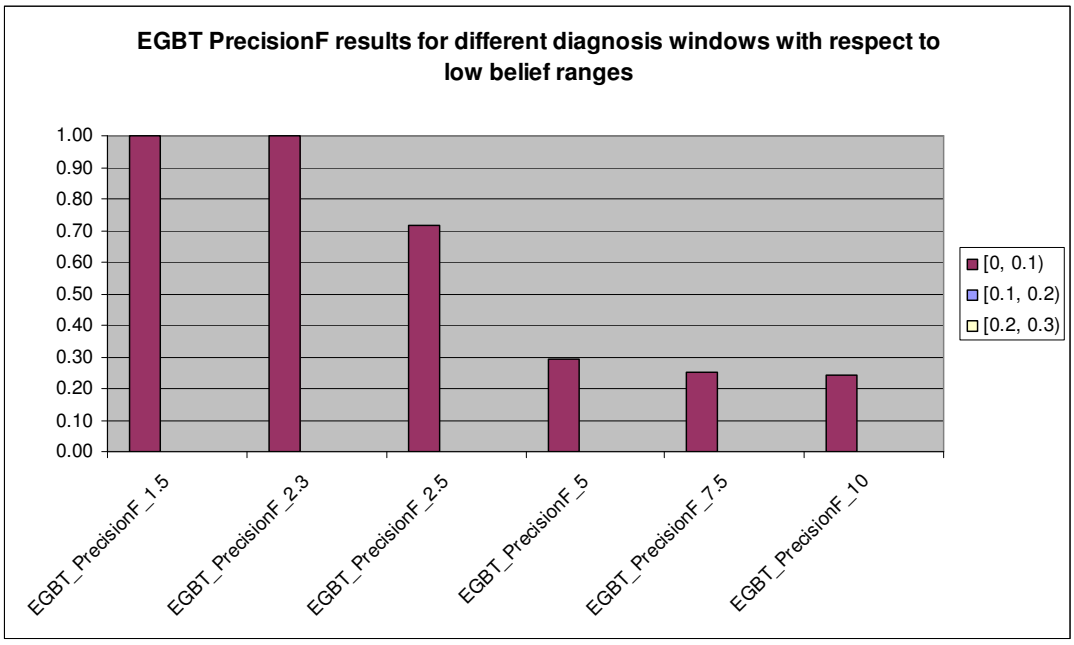

Figure 6-25 - EGBT_Precision$_F$ results for different diagnosis windows with respect to low belief ranges

Regarding belief ranges within [0, 0.3), *EGBT_Precision$_F$* should ideally be much greater than 0.5. For all *explanationConfiguration1* diagnosis windows, Figure 6-25 illustrates that *EGBT_Precision$_F$* values are quite satisfying only for the shortest diagnosis windows we have been experimenting with, as the min *EGBT_Precision$_F$* value for time windows 1.5, 2.3 and 2.5 sec. is 0.7. On the other hand, one can observe that *EGBT_Recall$_F$* values are rather undesired for diagnosis windows greater or equal to 5 sec. It seems that *EGBT* computes correctly low belief values for events that happen to be *fake* in cases that the given diagnosis windows are quite close to the *inter-event delay* and the time ranges medians average of the underlying monitoring theory. More specifically, for diagnosis windows around 2.3 sec, *EGBT_Precision$_F$* results with respect to low belief ranges are optimal. It should be recalled that both the *inter-event delay* and the time ranges medians average of the underlying monitoring theory we have used for the series of *experimentalConfiguration1* experiments are both equal to 2.3 (see also Section 6.4.2.1).

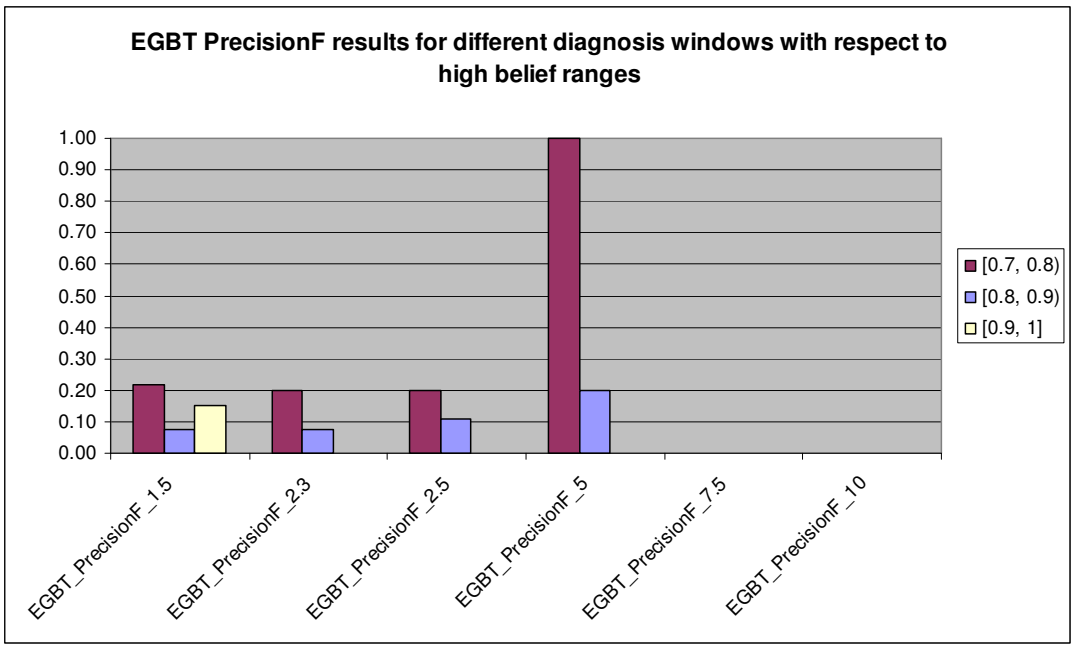

Figure 6-26 - EGBT_Precision$_F$ results for different diagnosis windows with respect to high belief ranges

Ideally, *EGBT_Precision$_F$* results with respect to high belief ranges should be much less than 0.5. By observing the chart illustrated in Figure 6-26, *EGBT_Precision$_F$* results, except the result with respect to range [0.7, 0.8) and diagnosis window equal to 5 sec, seem quite satisfying. Also, one can observe that *EGBT_Recall$_F$* values are minimized while the diagnosis window is increased and is set equal to or greater than 7.5 sec.

As an overall observation about *EGBT_Precision$_F$* presents sensitivity with respect to the *inter-event delay* and the time ranges medians average of the underlying monitoring theory. More specifically, the closest the time window is to the *inter-event delay* and the time ranges medians average of the underlying monitoring theory, the higher the probability is to have *EGBT* computing belief values less than 0.5 for *fake* events.

6.5.1.7.1.3 EGBT_Recall$_G$

The charts regarding the overall experimental *EGBT_Recall$_G$* results with respect to low and high belief ranges are presented in Figure 6-27 and Figure 6-28 respectively.

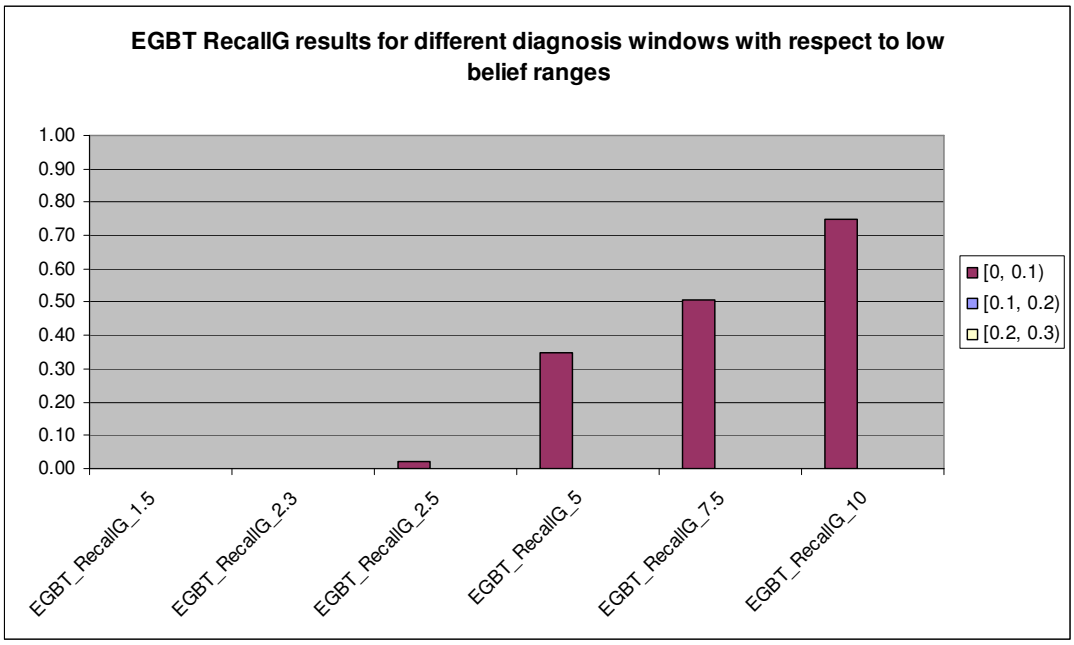

Figure 6-27 – EGBT_Recall$_G$ results for different diagnosis windows with respect to low belief ranges

The chart in above figure (Figure 6-27) illustrates that *EGBT* has not correctly computed low belief values for *genuine* events, as expected, in cases that diagnosis windows were set to values within [1.5, 5]. More specifically, the percentage of *genuine* events found to have belief values within [0, 0.3) is almost zero for time windows of 1.5, 2.3, and 2.5 sec, while this percentage increased to 35% when the time window was set to 5 sec. On the other hand, we got some low belief values for *genuine* events as the time windows increased. Half of the genuine events found to have belief values within [0, 0.1] for time window equal to 7.5 sec, while this percentage increased to 70% when the time window increased to 10 sec.

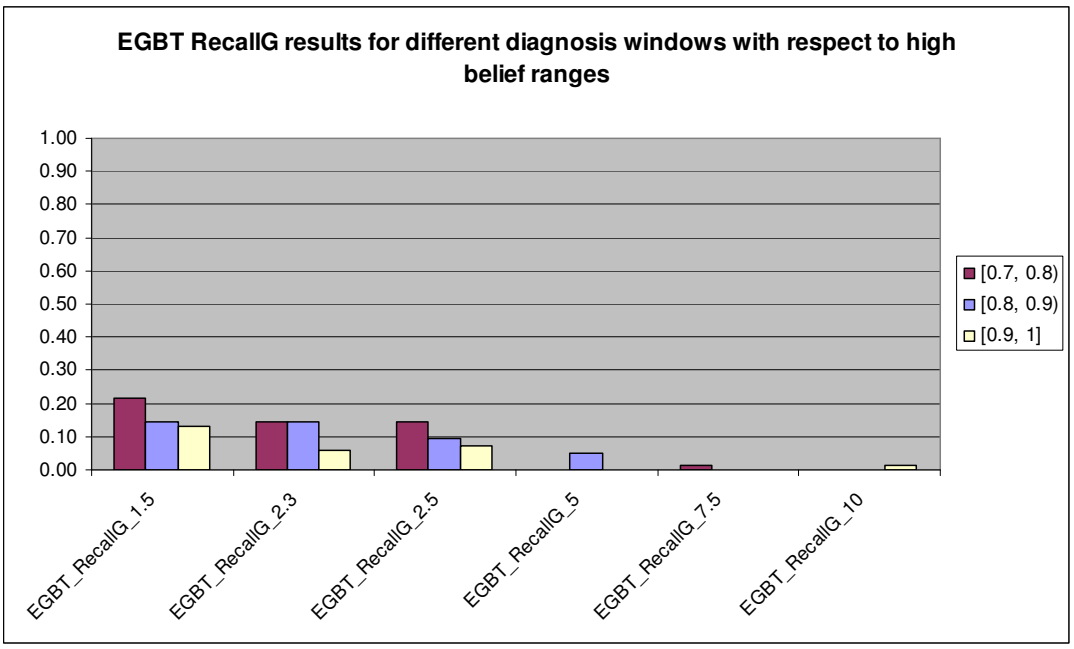

Figure 6-28 - EGBT_Recall$_G$ results for different diagnosis windows with respect to high belief ranges

Regarding belief ranges within [0.7, 1], *EGBT_Recall$_G$* should ideally be much greater than 0.5. For all *explanationConfiguration1* diagnosis windows, Figure 6-28 illustrates that *EGBT_Recall$_G$* values are quite undesired for all the time windows we have been experimenting with.

As an overall observation of the *EGBT* recall sensitivity with respect to *genuine* events, we could say that *EGBT* operated as expected only in cases that the time window was set to values around the *inter-event delay* and the time ranges medians average of the underlying monitoring theory, with respect only to lower belief ranges.

6.5.1.7.1.4 EGBT_Precision$_G$

The charts regarding the overall experimental *EGBT_Precision$_G$* results with respect to low and high belief ranges are presented in Figure 6-29 and Figure 6-30 respectively.

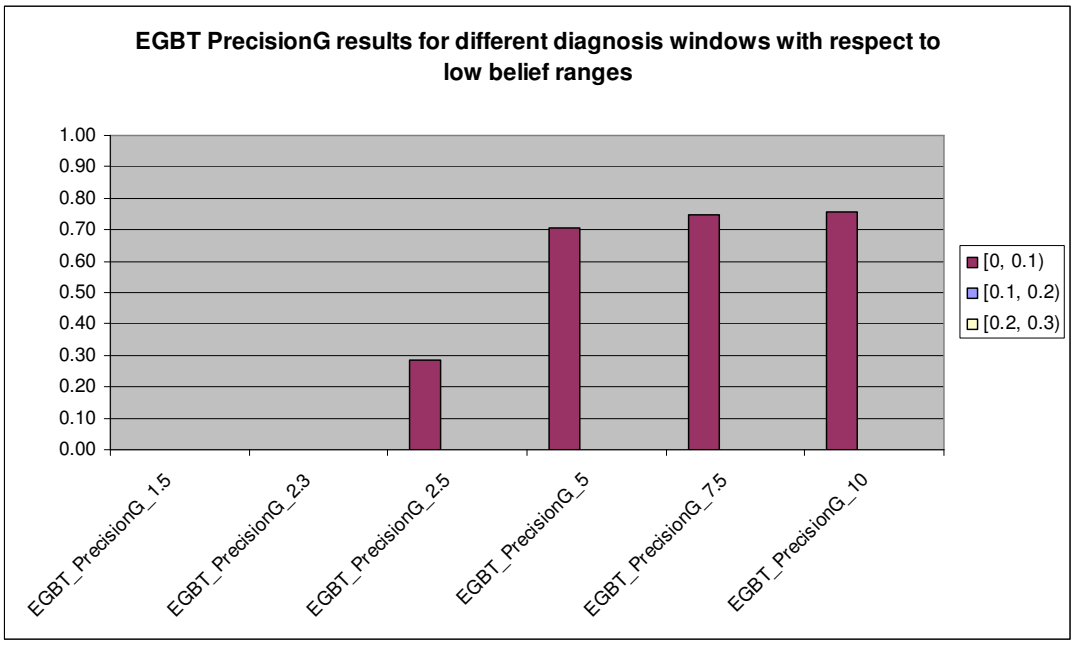

Figure 6-29 - EGBT_Precision$_G$ results for different diagnosis windows with respect to low belief ranges

By observing the chart of the above figure (Figure 6-29), EGBT precision with regards to genuine events was as low as expected only for the shortest time windows we have used in the current series of experiments. More specifically, the *EGBT_Precision$_G$* max value was 0.3 for the cases where diagnosis window was set to 1.5, 2.3, 2.5, and 5 sec. On the other hand, *EGBT_Precision$_G$* results are quite undesired for longer diagnosis windows, as the percentage of genuine events whose belief values were computed within [0, 01) is greater than 70% for both diagnosis windows of 7.5 and 10 sec.

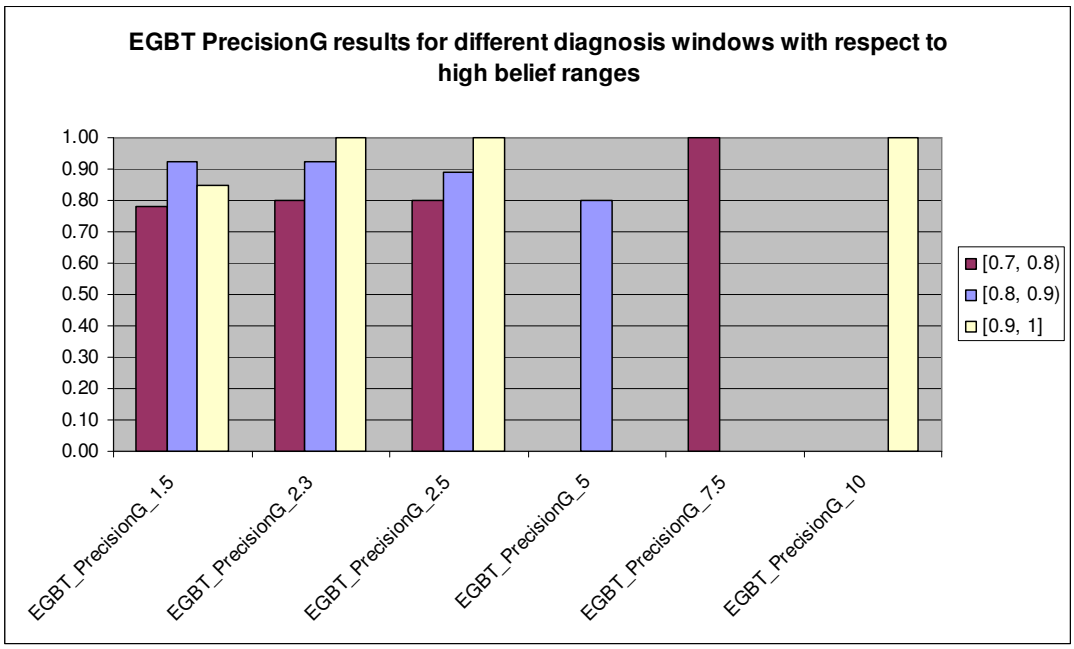

Figure 6-30 - EGBT_Precision$_G$ results for different diagnosis windows with respect to high belief ranges

Regarding belief ranges within [0.7, 1], it is expected that *EGBT_Precision$_G$* values should be quite high, i.e., greater than 0.5. For all *explanationConfiguration1* diagnosis windows, Figure 6-30 illustrates that indeed values are quite satisfying, as the min *EGBT_ Precision$_G$* value is 0.78.

As an overall observation about sensitivity of *EGBT* with respect to the belief values of *genuine* events, we could say that *EGBT_Precision$_G$* found to have a rather unsatisfying behaviour regarding long (i.e., greater than 5 sec) diagnosis windows and with respect to low belief ranges. For any other case, *EGBT_Precision$_G$* results were satisfying.

6.5.1.7.2 VDT correctness results charts and discussion

Similarly to above charts and discussion regarding *EGBT correctness*, the following sections include charts and discussion on the overall *experimentalConfiguration1* results for each *VDT correctness* metric i.e., *VDT_Recall$_F$, VDT_Precision$_F$, VDT_Recall$_G$*, and *VDT_Precision$_G$*.

6.5.1.7.2.1 VDT_Recall$_F$

The charts regarding the overall experimental *VDT_Recall$_F$* results are presented in Figure 6-31.

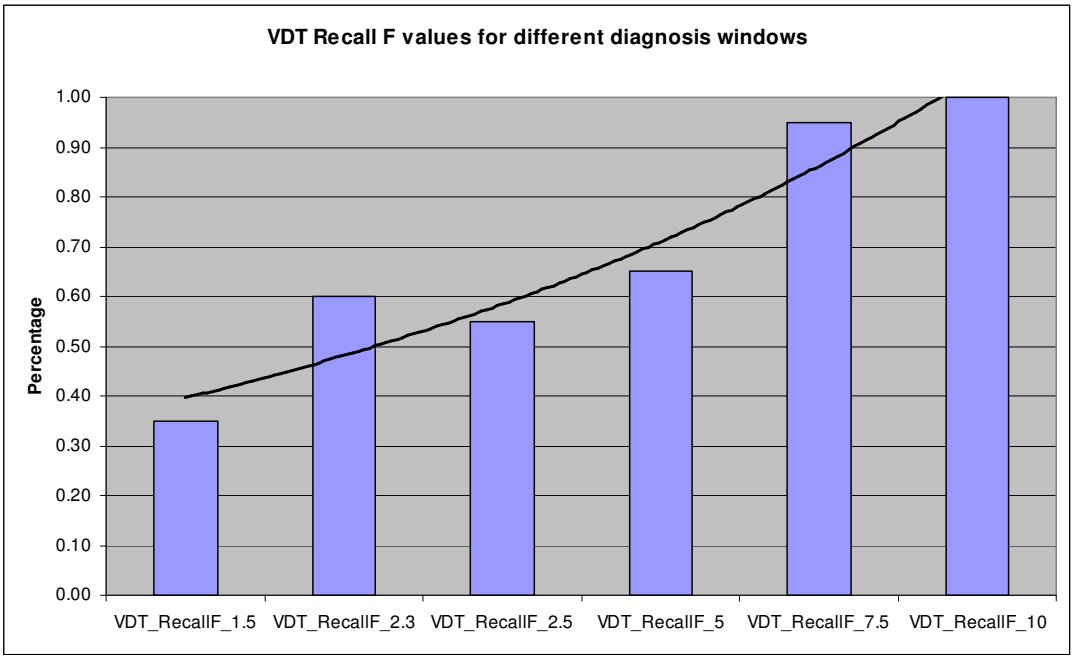

Figure 6-31 - VDT_Recall$_F$ results for different diagnosis windows

From the above chart (Figure 6-31), we could say that *VDT_Recall$_F$* results are quite satisfying. As it is expected, *VDT_Recall$_F$* were greater than 0.5, except for the case that the diagnosis window was set equal to 1.5 sec. Also one could observe that *VDT_Recall$_F$* was improving while the diagnosis window was being increased. Therefore, *VDT* seems to classify correctly *fake* events as *unconfirmed*, especially in cases with long diagnosis windows.

6.5.1.7.2.2 VDT_Precision$_F$

While the *VDT_Recall$_F$* experimental results were quite satisfying, the *VDT_Precision$_F$* results are rather undesired, as they are presented in Figure 6-32.

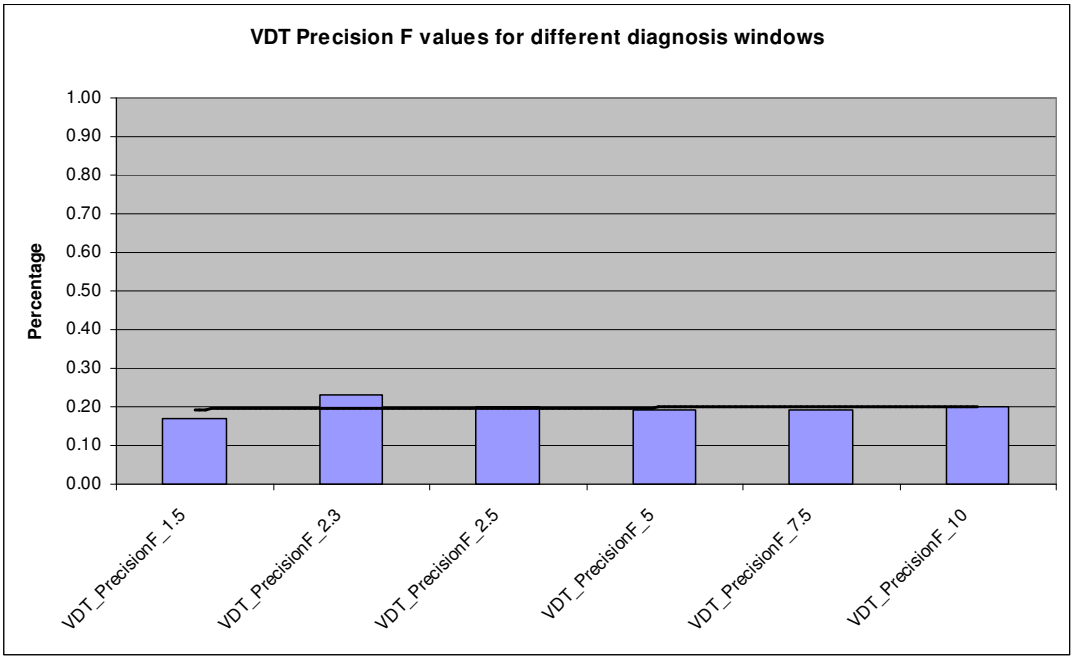

Figure 6-32 - VDT_Precision$_F$ results for different diagnosis windows

More specifically, while it is expected to have *VDT_Precision$_F$* high values (i.e., ideally much greater than 0.5), our *VDT_Precision$_F$* experimental results are quite low for all different diagnosis windows, with max value equal to 0.25. That means that *VDT* prototype classified unsuccessfully as *unconfirmed* events mostly *genuine* events rather than *fake* ones.

6.5.1.7.2.3 VDT_Recall$_G$

The charts regarding the overall experimental *VDT_Recall$_G$* results are presented in Figure 6-33.

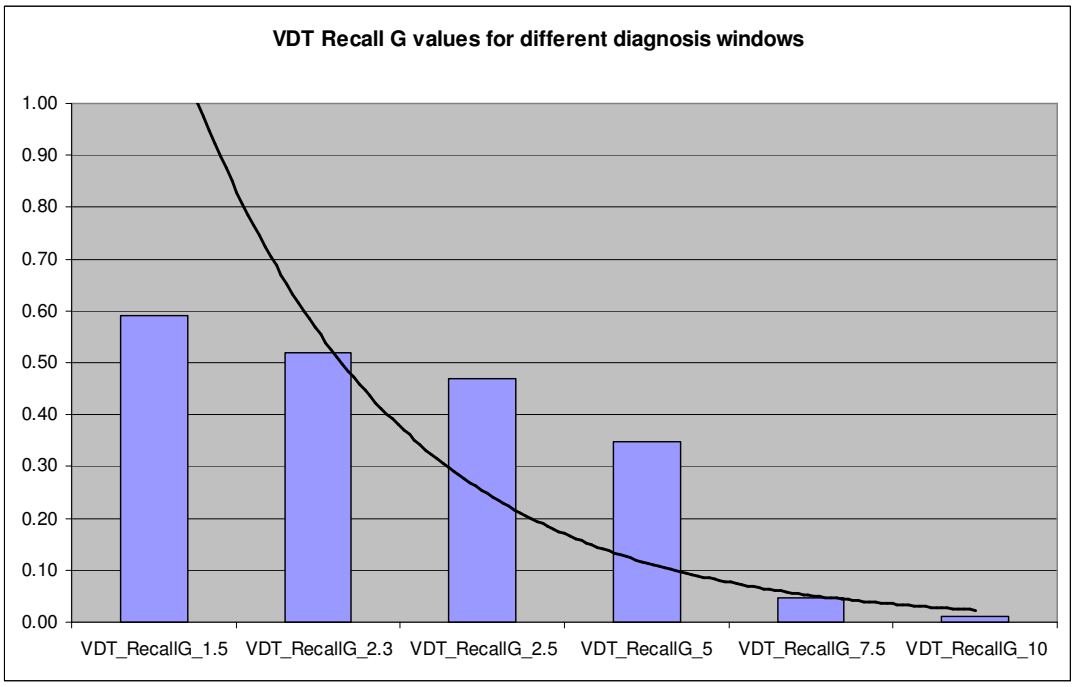

Figure 6-33 - VDT_Recall$_G$ results for different diagnosis windows

From the above chart (Figure 6-33), we could say that *VDT_Recall*$_G$ results are rather undesired; except for the cases that diagnosis window was set equal to 1.5 sec and 2.3. As it is expected, *VDT_Recall*$_G$ is greater than 0.5 only in the aforementioned cases. For the rest of the diagnosis windows we experimented with, *VDT_Recall*$_G$ was declining as the diagnosis window was being increased. Therefore, *VDT* seems to classify correctly *genuine* events as *confirmed* only in cases with diagnosis windows having values around the common *inter-event delay* and the time ranges medians average of the underlying monitoring theory, while it fails to do so as diagnosis window is being increased.

6.5.1.7.2.4 VDT_Precision$_G$

While the *VDT_Recall*$_G$ experimental results are rather undesired, the *VDT_Precision*$_G$ results are quite satisfying, as they are presented in Figure 6-34.

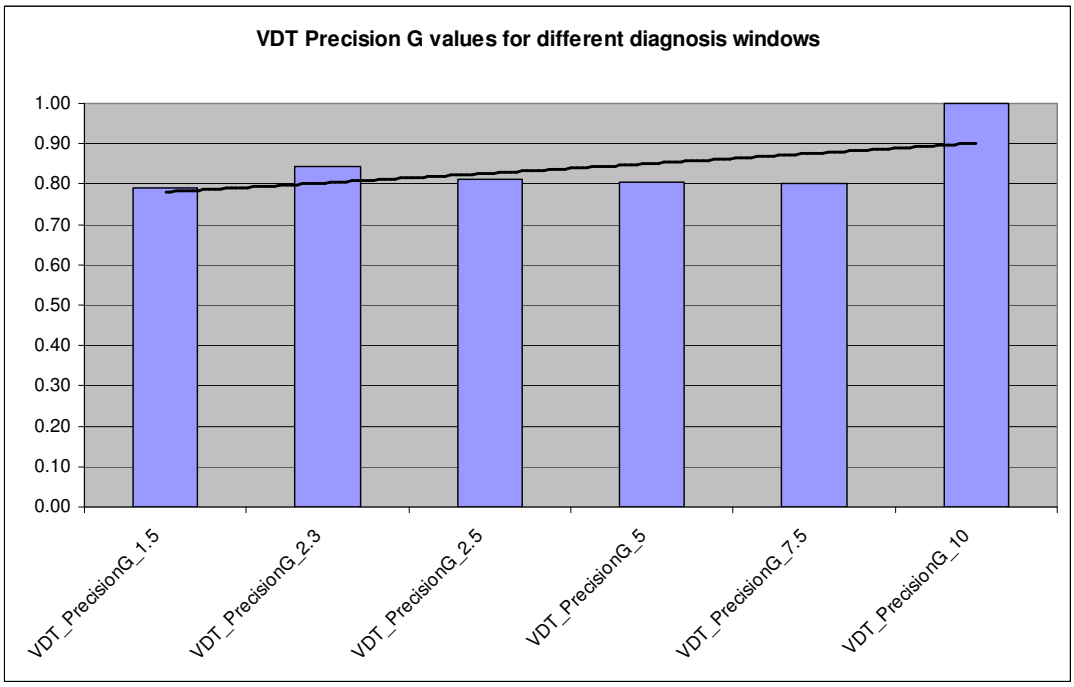

Figure 6-34 - VDT_Precision$_G$ results for different diagnosis windows

More specifically, our *VDT_Precision$_G$* experimental results values are quite high for all different diagnosis windows, as ideally expected. The above results show that *VDT* prototype classified successfully as *confirmed* events the *genuine* ones.

6.5.1.7.3 EGBT and VDT responsiveness results charts and discussion

In this section, we present charts regarding the *EGBT* and *VDT responsiveness* as occurred throughout the *explanationConfiguration1* experiments series. It should be noted that the charts in the flowing figures (Figure 6-35 and Figure 6-36) focus only on the mean *EGBT* and *VDT* computational times.

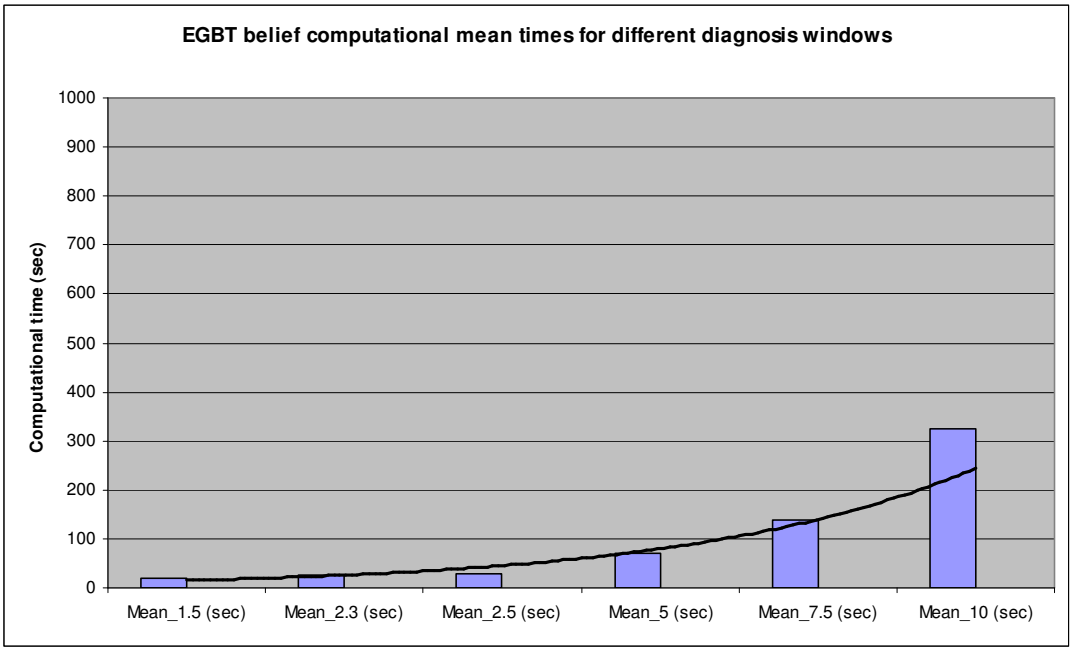

Figure 6-35 – EGBT belief computational mean times for different diagnosis windows

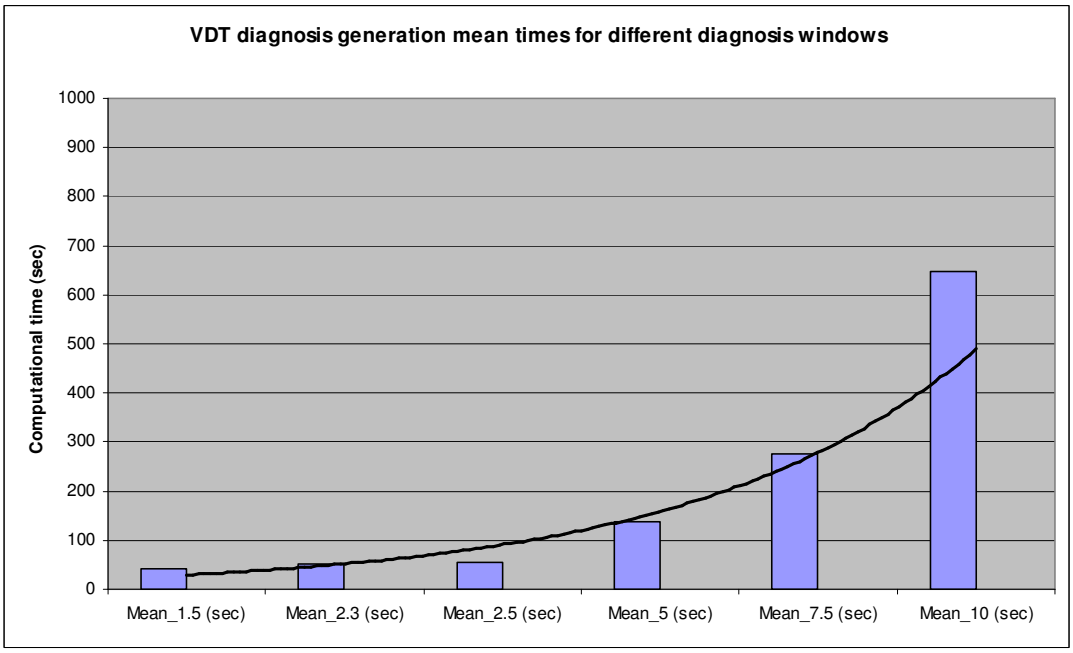

Figure 6-36 - VDT belief computational mean times for different diagnosis windows

From both charts, we can observe that the mean computational times for both *EGBT* and *VDT* are increasing exponentially while the diagnosis window is being increased. The behaviour of both *EGBT* and *VDT* computational times against the increment of diagnosis window is expected. This is because the increment of diagnosis window implies the increment of the event set *EGBT* should take into account as evidence pool.

Therefore, the larger event set *EGBT* takes into account, the longer time *EGBT* needs to compute the belief and plausibility of a given event.

Regarding *VDT*, it should be recalled that given a violation, *VDT* requests the belief values of *violation observations* to be computed by *EGBT*. Therefore, it is reasonable to observe the *VDT* final diagnosis generation time depending on the given diagnosis window. Also, it should be noted that the *VDT* computational times are quite high as the analysis of the events in a diagnosis window of 10 seconds takes more than 8 minutes, and even in the case of a window of 1.5 seconds the computational time needed is about one minute. These quite high values occur due to the number of events a violated monitoring rule that *VDT* analyzes includes, the number of assumptions *EGBT* uses to compute the genuineness belief of the violation observations and finally the volume of evidence that is taken into account and depends analogously to the diagnosis window. Due to the fact that none of the above factors can be restricted in a real life scenario, the *EGBT* and (therefore) *VDT* computational times could be improved by the introduction of an extra step in our diagnostic process. More specifically, as discussed below in Section 7.2, a premature thought of a static analysis of the monitoring theory assumptions in order to generate explanations and consequences trees at symbolic level for each event included in the theory before the analysis of occurred violations could improve the *EGBT* and *VDT* computational times.

6.5.2 ExplanationConfiguration2 Experiments Results

In this section, the results of experiments expConf2_10%, expConf2_20%, and expConf2_30% specified in Section 6.4.2.2 are presented. As a reminder, the results of the above experiments have been generated by running the monitoring and diagnosis prototype with the following common inputs:

- a diagnosis window set equal to 2.3 sec as discussed in Section 6.4.2.2
- the LBACS monitoring theory as described in Section 6.4.2, and
- the belief functions a1 and a2 set to 0.3 and 0.4 respectively (see also Section 6.4.2)

6.5.2.1 expConf2_10% results

The following results have been generated by using a set of 5000 events that have been generated by six *adversaries*. Each *adversary* was specified to intercept randomly and delay the 10% of its incoming events by introducing a delay capable to cause violations for the monitoring rules LBACS.R1 to LBACS.R4 (see Appendix A).

6.5.2.1.1 EGBT correctness results

Table 6-24 contains the results for the *EGBT correctness* metrics for experiment expConf2_10%, whilst Figure 6-37 illustrates a representative chart of the given experimental results.

Table 6-24 - EGBT correctness results for experiment expConf2_10%

Belief Range	$EGBT\_Recall_F$	$EGBT\_Precision_F$	$EGBT\_Recall_G$	$EGBT\_Precision_G$
[0, 0.1)	0.00	N/A	0.00	N/A
[0.1, 0.2)	0.00	N/A	0.00	N/A
[0.2, 0.3)	0.00	N/A	0.00	N/A
[0.3, 0.4)	0.00	0.00	0.26	1.00
[0.4, 0.5)	0.00	0.00	0.19	1.00
[0.5, 0.6)	0.00	N/A	0.00	N/A
[0.6, 0.7)	0.00	0.00	0.13	1.00
[0.7, 0.8)	0.00	0.00	0.11	1.00
[0.8, 0.9)	0.00	0.00	0.28	1.00
[0.9, 1]	1.00	0.33	0.04	0.67

Observing the $EGBT\_Recall_F$ results in above table, we could say that *EGBT* failed totally to compute low belief values for the *fake* events of expConf2_10%, as all the *fake* events found to have belief values within range [0.9, 1], while there is an undesired zero percentage of *fake* events whose belief values lie within [0, 0.3). On the other hand, $EGBT\_Precision_F$ results are more encouraging, as only the 33% of the events whose belief values ranged within [0.9, 1] happened to be *fake*.

Regarding $EGBT\_Recall_G$, the results in low belief ranges are quite satisfying as none *genuine* event with belief value within [0, 0.1) found. However, the $EGBT\_Recall_G$

results in higher belief ranges are not satisfying due to the fact that there are low percentages (11%, 28%, and 4%) of *genuine* events with belief values within [0.7, 1]. Finally, the $EGBT\_Precision_G$ results are quite satisfying because all events that found to have a belief value within [0.7, 0.9] were also *genuine* events. Also, the 67% of the events that found to have belief values within [0.9, 1] were *genuine* events as well.

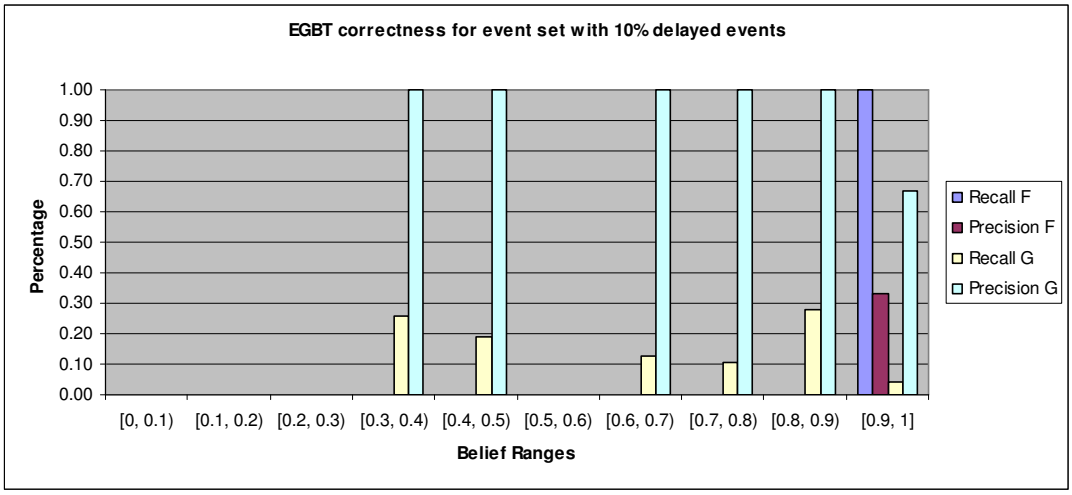

Figure 6-37 – EGBT correctness results for expConf2_10%

6.5.2.1.2 VDT correctness results

The following table (Table 6-25) accumulates the results of *VDT correctness* metrics for experiment expConf2_10%, whilst Figure 6-38 illustrates a representative chart of the given experimental results.

Table 6-25 - VDT correctness results for experiment expConf1_10%

Confirmation criterion	VDT_Recall_F	$VDT\_Precision_F$	VDT_Recall_G	$VDT\_Precision_G$
Bel(Genuine(P)) ≥ Bel(¬Genuine(P))	0.00	0.00	0.55	0.96

Table 6-25 presents a rather undesired VDT_Recall_F result. In particular, *VDT* has classified as *unconfirmed* events none of the *fake violation observations*. Similarly, the $VDT\_Precision_F$ result is again rather undesired due to the fact that none *unconfirmed violation observation* happened to be *fake* event.

VDT_Recall_G result is rather satisfying due to fact that the 55% of the total *genuine violation observations* have been flagged as *confirmed* events by *VDT*. Similarly and

evenly better, *VDT_Precision*$_G$ result is quite satisfying, as the 96% of the total *confirmed violation observations* happened to be *genuine* events.

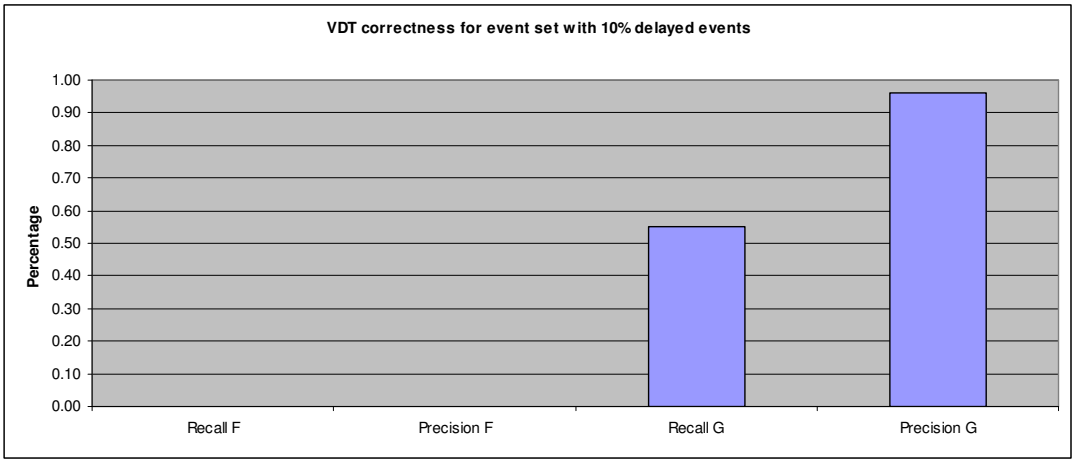

Figure 6-38 - VDT correctness results for expConf2_10%

6.5.2.1.3 Responsiveness results

The following table (Table 6-26) presents the results of responsiveness metrics as specified in Section 6.3.2.

Table 6-26 - EGBT and VDT responsiveness results for experiment expConf2_10%

Computational time types	Mean (sec)	Standard deviation (sec)	Max (sec)	Min (sec)
EGBT belief computational time	24.55	14.38	50.64	0.17
VDT diagnosis generation time	50.23	28.19	100.77	0.00

According to the results presented in above table, the time *EGBT* needed to compute the belief and plausibility of an event was 24.55 sec in average, with a standard deviation of 14.38 sec. The max and min computational times occurred are 50.64 sec and 0.17 sec respectively.

The time *VDT* consumed to generate a final diagnosis for a violation was is 50.23 sec in average, with a standard deviation of 28.19 sec. The max and min final diagnosis generation times occurred were 100.77 sec and 0 sec respectively.

6.5.2.2 expConf2_20% results

The following results have been generated by using a set of 5000 events that have been generated by six *adversaries*. Each *adversary* was specified to intercept randomly and delay the 20% of its incoming events by introducing a delay capable to cause violations for the monitoring rules LBACS.R1 to LBACS.R4 (see Appendix A).

6.5.2.2.1 EGBT correctness results

Table 6-27 contains the results for the *EGBT correctness* metrics for experiment expConf2_20%, whilst Figure 6-39 illustrates a representative chart of the given experimental results.

Table 6-27 - EGBT correctness results for experiment expConf2_20%

Belief Range	$EGBT\_Recall_F$	$EGBT\_Precision_F$	$EGBT\_Recall_G$	$EGBT\_Precision_G$
[0, 0.1)	0.15	1.00	0.00	0.00
[0.1, 0.2)	0.00	N/A	0.00	N/A
[0.2, 0.3)	0.00	N/A	0.00	N/A
[0.3, 0.4)	0.00	0.00	0.24	1.00
[0.4, 0.5)	0.45	0.31	0.24	0.69
[0.5, 0.6)	0.00	N/A	0.00	N/A
[0.6, 0.7)	0.20	0.22	0.17	0.78
[0.7, 0.8)	0.15	0.20	0.14	0.80
[0.8, 0.9)	0.05	0.08	0.14	0.92
[0.9, 1]	0.00	0.00	0.06	1.00

The $EGBT\_Recall_F$ experimental results in above table show that *EGBT* failed to compute low belief values for the *fake* events of expConf2_20%, as only the 15% of the *fake* events found to have belief values within range [0, 0.1). On the other hand, it is rather satisfying result to have only the 20% (i.e. 15% and 5%) of the *fake* events found to have belief values within range [0.7, 0.9). Similarly, $EGBT\_Precision_F$ results are quite encouraging. In particular, all events that found to have belief values within [0, 0.1) were *fake* as well. In addition to that, only the 9.3% in average of the events found with belief values ranged within [0.7, 1] happened to be *fake* too. It should be noted that the above

9.3% in average was computed by measuring the average of the $EGBT\_Precision_F$ results for the belief ranges [0.7, 0.8), [0.8, 0.9) and [0.9, 1] which were 20%, 8% and 0% respectively.

Regarding $EGBT\_Recall_G$, the results in low belief ranges are quite satisfying as none *genuine* event with belief value within [0, 0.3) found. However, the $EGBT\_Recall_G$ results in higher belief ranges are not satisfying due to the fact that there are low percentages (14%, 14%, and 6%) of *genuine* events with belief values within [0.7, 1]. Finally, the $EGBT\_Precision_G$ results are quite satisfying because the 91% in average of events that found to have a belief value within [0.7, 1] were also *genuine* events. It should be noted that the above 91% in average was computed by measuring the average of the $EGBT\_Precision_G$ results for the belief ranges [0.7, 0.8), [0.8, 0.9) and [0.9, 1] which were 80%, 92% and 100% respectively.

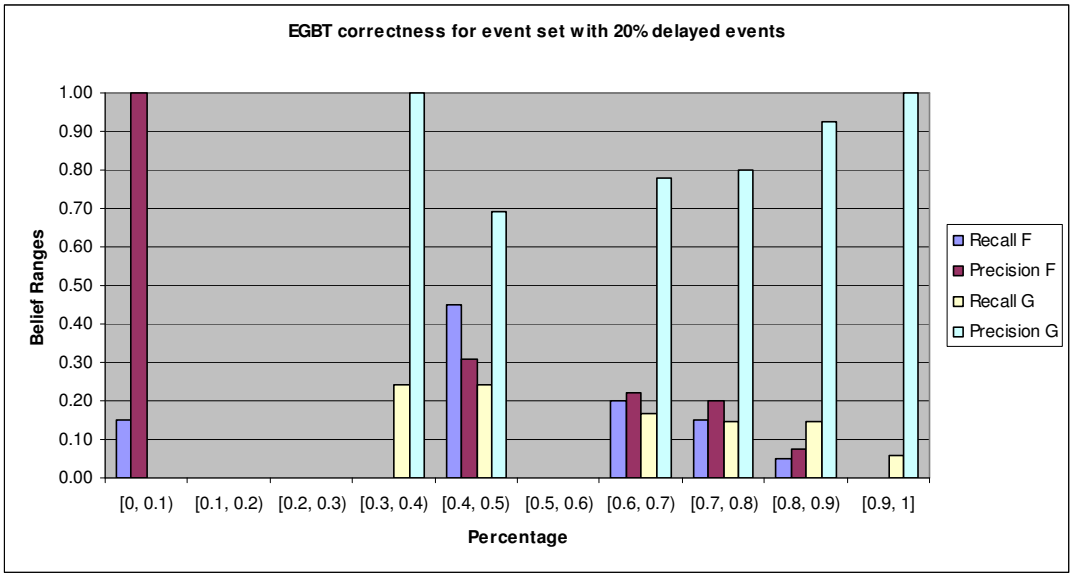

Figure 6-39 – EGBT correctness results for expConf2_20%

6.5.2.2.2 VDT correctness results

The following table (

Table 6-28) accumulates the results of *VDT correctness* metrics for experiment expConf2_20%, whilst Figure 6-40 illustrates a representative chart of the given experimental results.

Table 6-28 - VDT correctness results for experiment expConf1_20%

Confirmation criterion	VDT_Recall_F	$VDT\_Precision_F$	VDT_Recall_G	$VDT\_Precision_G$
Bel(Genuine(P)) ≥ Bel(¬Genuine(P))	0.60	0.23	0.52	0.84

Table 6-28 presents a rather satisfying VDT_Recall_F result. In particular, *VDT* has classified as *unconfirmed* events the 60% of the total *fake violation observations*. On the contrary, the $VDT\_Precision_F$ result is rather undesired due to the fact that only the 23% of the *unconfirmed violation observations* happened to be *fake* events.

VDT_Recall_G result is almost satisfying due to fact that the 52% of the total *genuine violation observations* have been flagged as *confirmed* events by *VDT*. Similarly and evenly better, $VDT\_Precision_G$ result is quite satisfying, as the 84% of the total *confirmed violation observations* happened to be *genuine* events.

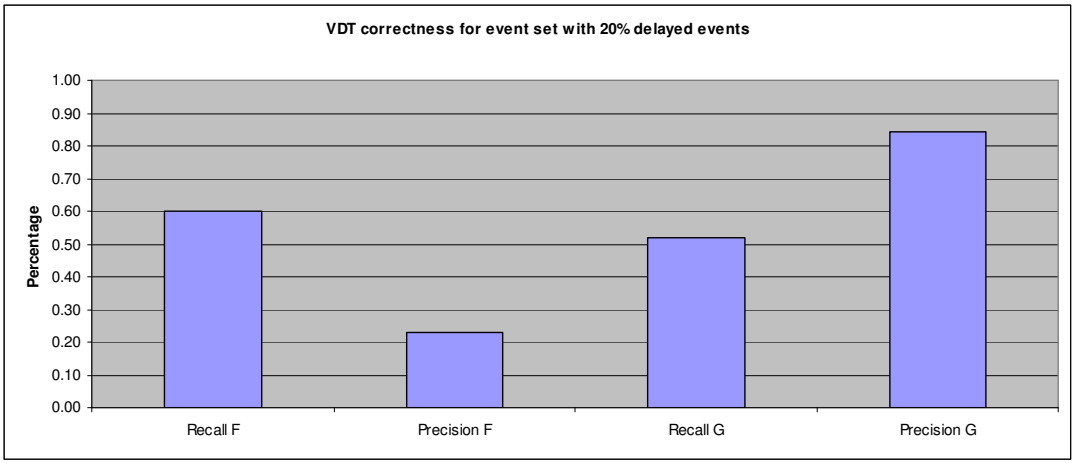

Figure 6-40 - VDT correctness results for expConf2_20%

6.5.2.2.3 Responsiveness results

The following table (Table 6-29) presents the results of responsiveness metrics as specified in Section 6.3.2.

Table 6-29 - EGBT and VDT responsiveness results for experiment expConf2_20%

Computational time types	Mean (sec)	Standard deviation (sec)	Max (sec)	Min (sec)
EGBT belief computational time	26.42	14.34	55.08	0.19
VDT diagnosis generation time	52.86	28.42	109.66	0.00

According to the results presented in above table, the time *EGBT* needed to compute the belief and plausibility values of an event was 26.42 sec in average, with a standard deviation of 14.34 sec. The max and min computational times occurred are 55.08 sec and 0.19 sec respectively.

The time *VDT* consumed to generate a final diagnosis for a violation was is 52.86 sec in average, with a standard deviation of 28.42 sec. The max and min final diagnosis generation times occurred were 109.66 sec and 0 sec respectively.

6.5.2.3 expConf2_30% results

The following results have been generated by using a set of 5000 events that have been generated by six *adversaries*. Each *adversary* was specified to intercept randomly and delay the 30% of its incoming events by introducing a delay capable to cause violations for the monitoring rules LBACS.R1 to LBACS.R4 (see Appendix A).

6.5.2.3.1 EGBT correctness results

Table 6-30 contains the results for the *EGBT correctness* metrics for experiment expConf2_30%, whilst Figure 6-41 illustrates a representative chart of the given experimental results.

Table 6-30 - EGBT correctness results for experiment expConf2_30%

Belief Range	$EGBT\_Recall_F$	$EGBT\_Precision_F$	$EGBT\_Recall_G$	$EGBT\_Precision_G$
[0, 0.1)	0.24	0.90	0.01	0.10
[0.1, 0.2)	0.00	N/A	0.00	N/A
[0.2, 0.3)	0.00	N/A	0.00	N/A
[0.3, 0.4)	0.00	0.00	0.16	1.00
[0.4, 0.5)	0.27	0.30	0.27	0.70
[0.5, 0.6)	0.00	N/A	0.00	N/A
[0.6, 0.7)	0.11	0.17	0.22	0.83
[0.7, 0.8)	0.11	0.19	0.20	0.81
[0.8, 0.9)	0.16	0.46	0.08	0.54
[0.9, 1]	0.11	0.50	0.05	0.50

The $EGBT\_Recall_F$ experimental results in above table show that *EGBT* failed to compute low belief values for the *fake* events of expConf2_20%, as only the 24% of the *fake* events found to have belief values within range [0, 0.1). On the other hand, it is rather satisfying result to have only the 38% (i.e. 11%, 16% and 11%) of the *fake* events found to have belief values within range [0.7, 1]. Similarly, $EGBT\_Precision_F$ results are quite encouraging. In particular, the 90% of events that found to have belief values within [0, 0.1) were *fake* as well. In addition to that, only the 19% of the events found with belief values ranged within [0.7, 0.8) happened to be *fake* too. Finally, the $EGBT\_Precision_F$ results regarding events found to have belief values ranged within [0.8, 0.9) and [0.9, 1] are not that optimal. More specifically, a rather high and almost undesired percentage (46%) of the events found to have belief values within [0.8, 0.9) happened to be *fake*. Also, we can observe similar behaviour for the events found to have belief values within [0.9, 1]. The percentage of such events that happened to be *fake* as well was quite high too (50%).

Regarding $EGBT\_Recall_G$, the results in low belief ranges are quite satisfying as only the 1% of the *genuine* events with belief value within [0, 0.3) found. However, the $EGBT\_Recall_G$ results in higher belief ranges are not satisfying due to the fact that there are quite low percentages (20%, 8%, and 5%) of *genuine* events with belief values within

[0.7, 1]. On the contrary, the *EGBT_Precision*$_G$ results with respect to low belief ranges are quite satisfying because only the 10% of events found to have belief values within [0. 0.1) happened to be *genuine* events. Also, the 81% of events that found to have a belief value within [0.7, 0.8) were also *genuine* events. Finally, the *EGBT_Precision*$_G$ results regarding events found to have belief values ranged within [0.8, 0.9) and [0.9, 1] could have been better. More specifically, a rather low percentage (54%) of the events found to have belief values within [0.8, 0.9) happened to be *genuine*. Also, we can observe similar behaviour for the events found to have belief values within [0.9, 1]. The percentage of such events that happened to be *genuine* events as well was quite low too (50%).

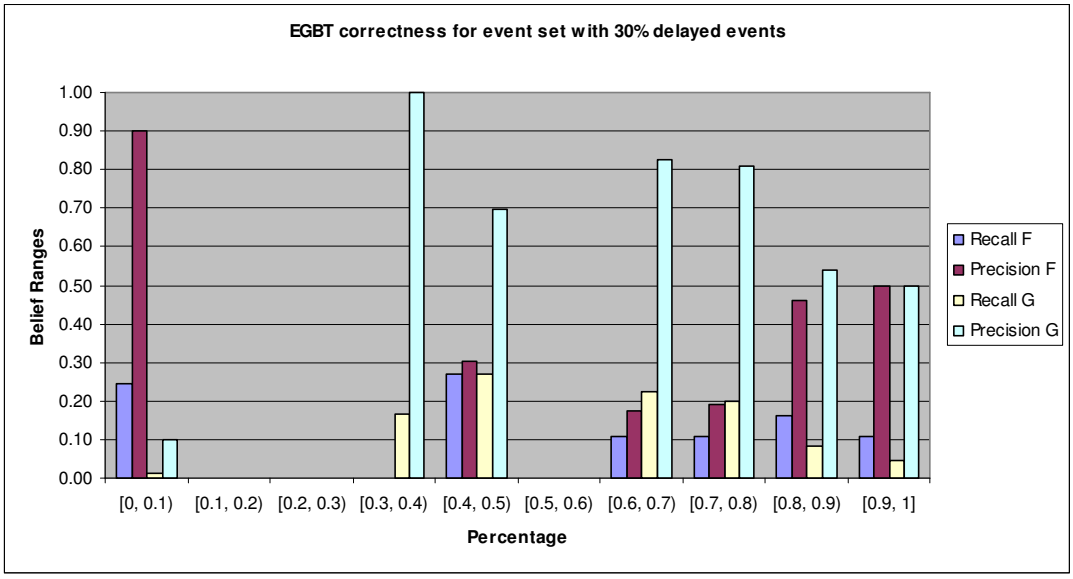

Figure 6-41 – EGBT correctness results for expConf2_30%

6.5.2.3.2 VDT correctness results

The following table (Table 6-31) accumulates the results of *VDT correctness* metrics for experiment expConf2_30%, whilst Figure 6-42 illustrates a representative chart of the given experimental results.

Table 6-31 - VDT correctness results for experiment expConf1_30%

Confirmation criterion	VDT_Recall$_F$	VDT_Precision$_F$	VDT_Recall$_G$	VDT_Precision$_G$
Bel(Genuine(P)) ≥ Bel(¬Genuine(P))	0.51	0.33	0.55	0.72

Table 6-31 presents a rather neutral VDT_Recall_F result. In particular, *VDT* has classified as *unconfirmed* events the 51% of the total *fake violation observations*. On the contrary, the $VDT\_Precision_F$ result is rather undesired due to the fact that only the 33% of the *unconfirmed violation observations* happened to be *fake* events.

VDT_Recall_G result is almost satisfying due to fact that the 55% of the total *genuine violation observations* have been flagged as *confirmed* events by *VDT*. Similarly and evenly better, $VDT\_Precision_G$ result is quite satisfying, as the 72% of the total *confirmed violation observations* happened to be *genuine* events.

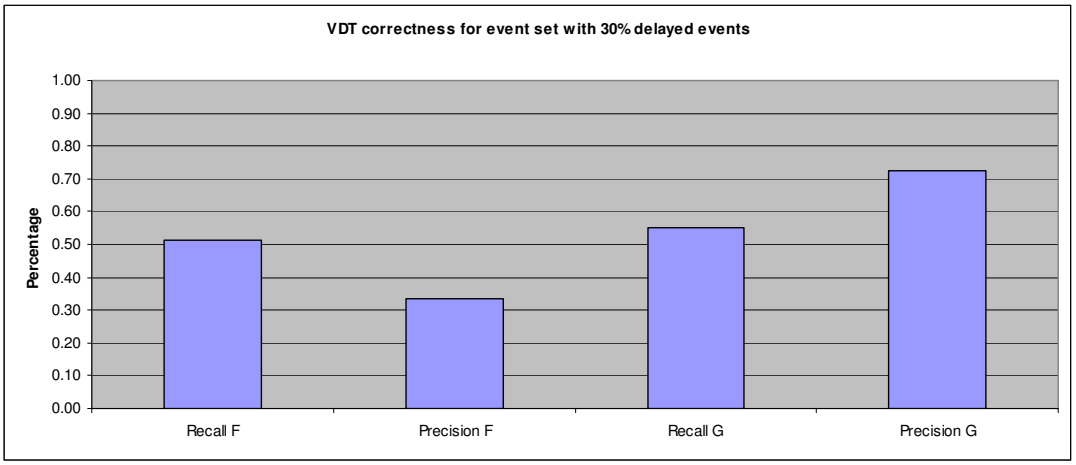

Figure 6-42 - VDT correctness results for expConf2_30%

6.5.2.3.3 Responsiveness results

The following table (Table 6-32) presents the results of responsiveness metrics as specified in Section 6.3.2.

Table 6-32 - EGBT and VDT responsiveness results for experiment expConf2_20%

Computational time types	Mean (sec)	Standard deviation (sec)	Max (sec)	Min (sec)
EGBT belief computational time	29.22	13.78	68.36	0.19
VDT diagnosis generation time	59.19	26.20	136.67	0.00

According to the results presented in above table, the time *EGBT* needed to compute the belief and plausibility values of an event was 29.22 sec in average, with a standard deviation of 13.78 sec. The max and min computational times occurred are 68.36 sec and 0.19 sec respectively.

The time *VDT* consumed to generate a final diagnosis for a violation was is 59.19 sec in average, with a standard deviation of 26.20 sec. The max and min final diagnosis generation times occurred were 136.67 sec and 0 sec respectively.

6.5.2.4 ExplanationConfiguration2 overall charts and discussion

In this section, we present charts that accumulate and compare the results of the individual *explanationConfiguration2* experiments. Based on the aforementioned charts, discussion on the experimental observations against the objective of the relevant experimental configuration follows.

6.5.2.4.1 EGBT correctness results charts and discussion

The following sections include charts and discussion on the overall *experimentalConfiguration2* results for each *EGBT correctness* metric i.e., $EGBT\_Recall_F$, $EGBT\_Precision_F$, $EGBT\_Recall_G$, and $EGBT\_Precision_G$. For each aforementioned metric, we provide charts with respect to low belief ranges, i.e., ranges within [0, 0.3), and high belief ranges, i.e., ranges within [0.7, 1]. It should be noted that results found within [0.3, 0.7) have been excluded from these charts and the following results analysis, as they might be considered as non useful and enough indicative information to be taken into account by a recovery decision making process.

6.5.2.4.1.1 $EGBT\_Recall_F$

Regarding the overall experimental $EGBT\_Recall_F$ results, Figure 6-43 and Figure 6-44 present the experimental $EGBT\_Recall_F$ behaviour against different diagnosis windows. More specifically, Figure 6-43 illustrates $EGBT\_Recall_F$ behaviour with respect to low belief ranges, i.e., ranges within [0, 0.3). Similarly, Figure 6-44 shows $EGBT\_Recall_F$ behaviour with respect to belief ranges within [0.7, 1].

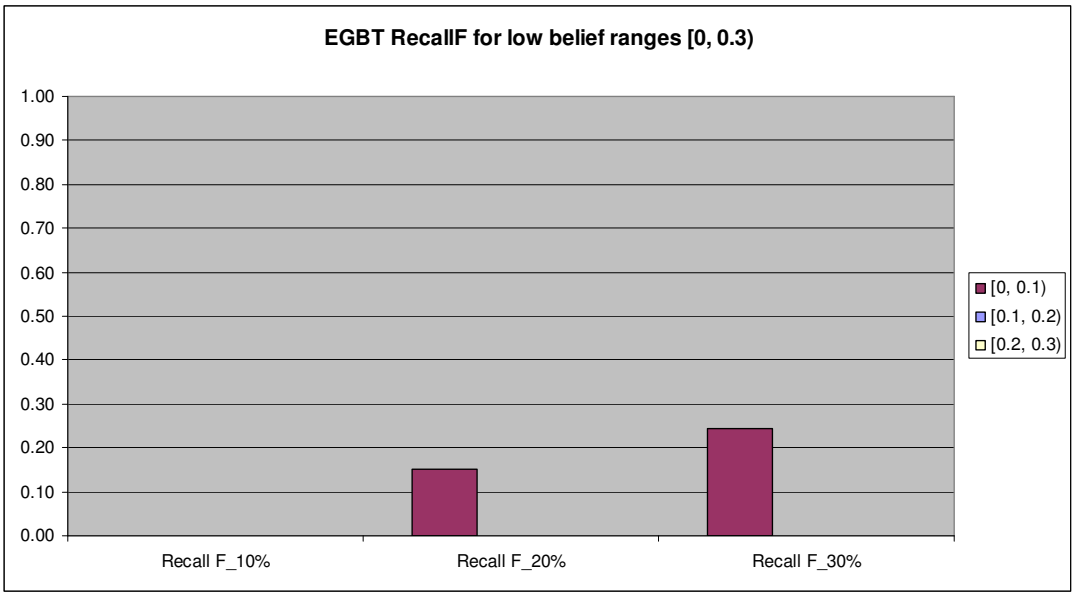

Figure 6-43 – EGBT_Recall$_F$ results for different delayed events percentages with respect to low belief ranges

The chart in Figure 6-43 illustrates that even though *EGBT_Recall$_F$* is quite low in general, it seems to increase as the percentage of randomly delayed (*fake*) events increases. It should be noted that, *EGBT_Recall$_F$* should ideally have values much greater than 0.5 in low belief ranges. Therefore, we could say that *EGBT* seems to converge to its intended behaviour, i.e., computing low belief values for *fake* events, as long as the number of *fake* events increases.

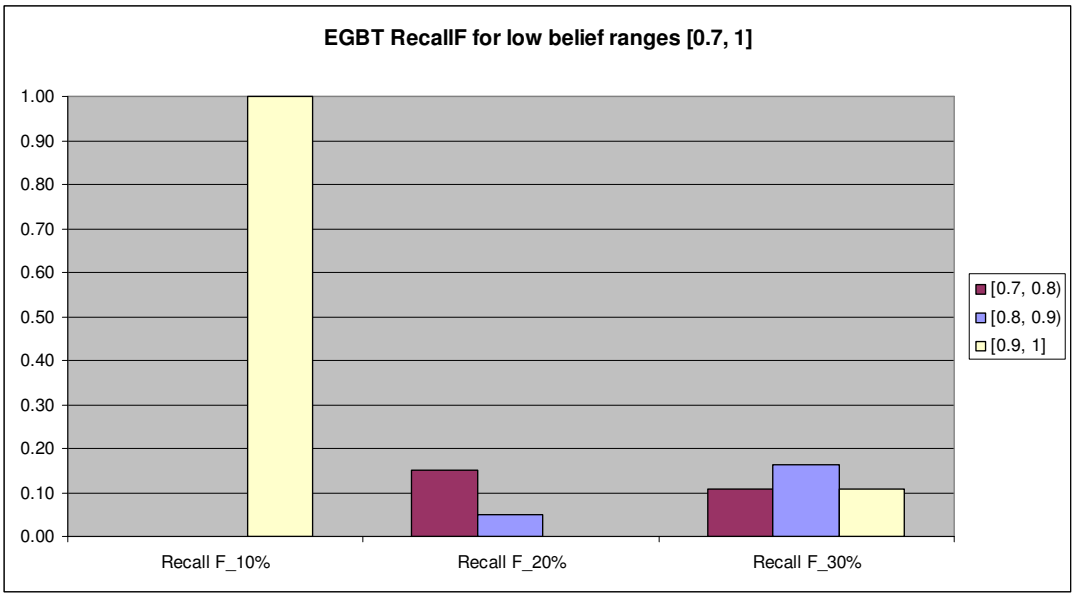

Figure 6-44 - EGBT_Recall$_F$ results for different delayed events percentages with respect to high belief ranges

Regarding belief ranges within [0.7, 1], it is expected that *EGBT_Recall$_F$* values should be quite low, i.e., much less than 0.5. Analyzing expConf2_10% experiment, where the percentage of randomly delayed (*fake*) events was 10%, Figure 6-44 illustrates that *EGBT* behaved rather undesirably, as all fake events found to have belief values within [0.9, 1). On the contrary, the other two experiments, expConf20% and expConf30%, where the percentage of delayed (*fake*) events was 20% and 30% respectively, *EGBT* behaved as expected, as the max *EGBT_Recall$_F$* value was less than 0.20.

As an overall observation about sensitivity of *EGBT* with respect to the belief values of *fake* events, we could say that the larger the *fake* event set we introduce, the greater probability is that *EGBT* would compute the *fake* events belief values within low ranges, Therefore, *EGBT* operates as expected with respect to *fake* events when more *fake* events occur and are taken into account.

6.5.2.4.1.2 EGBT_Precision$_F$

The charts regarding the overall experimental *EGBT_Precision$_F$* results with respect to low and high belief ranges are presented in Figure 6-45 and Figure 6-46 respectively.

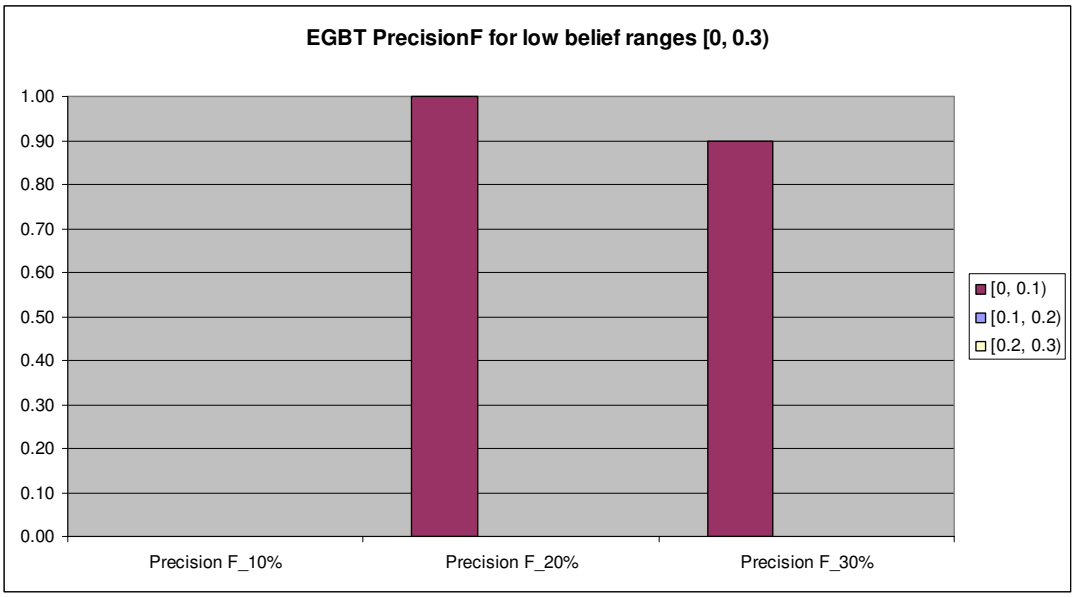

Figure 6-45 - EGBT_Precision$_F$ for different delayed events percentages with respect to low belief ranges

Regarding belief ranges within [0, 0.3), *EGBT_Precision$_F$* should ideally be much greater than 0.5. Figure 6-45 illustrates that *EGBT_Precision$_F$* result with respect to the case we had the 10% of events delayed is undesired, as was found equal to zero. On the other hand, the *EGBT_Precision$_F$* results with respect to higher percentages of delayed events, namely 20% and 30%, are quite satisfying, as were found equal to 0.9 and 1, respectively. A meaning to the above observations could be that, while the number of *fake* events was increasing, more events that found to have belief values within [0, 0.3) happened to be *fake*.

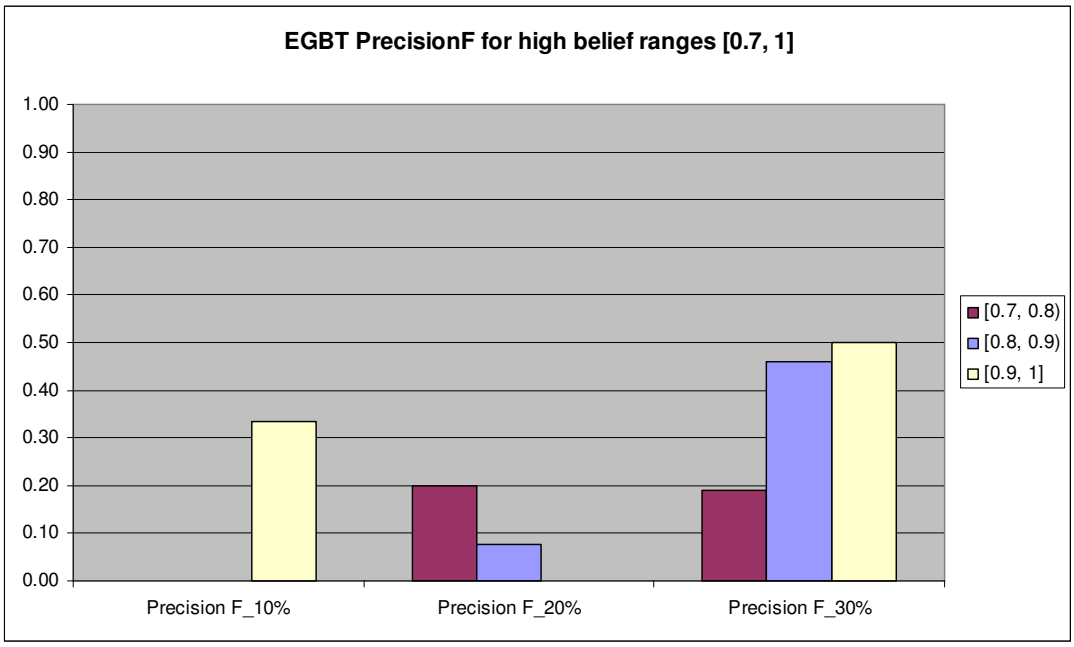

Figure 6-46 - EGBT_Precision$_F$ results for different delayed events percentages with respect to high belief ranges

Ideally, *EGBT_Precision$_F$* results with respect to high belief ranges should be much less than 0.5. By observing the chart illustrated in Figure 6-46, *EGBT_Precision$_F$* results seem almost satisfying. While *EGBT_Precision$_F$* results with respect to 10% and 20% delayed events configurations are quite low ranging within 0.08 and 0.32, there is a rather undesired high *EGBT_Precision$_F$* value (0.5) regarding the case we had 30% of the events delayed and the belief range [0.9, 1]. The *EGBT_Precision$_F$* result regarding the same case and the belief range [0.8, 0.9]) found quite high (0.46) as well.

As an overall observation we could say that *EGBT_Precision$_F$* presents sensitivity with respect to the delayed events percentages. More specifically, *EGBT_Precision$_F$* seem to increase as the percentages of the delayed events are increasing, independently to the belief ranges we might use for analysis. Therefore, provided that the number of *fake* events is increasing, we might get *EGBT_Precision$_F$* values with respect to low belief ranges, which would be, as ideally desired. However, the corresponding values with respect to high belief ranges would be undesirably high as well.

6.5.2.4.1.3 EGBT_Recall$_G$

The charts regarding the overall experimental *EGBT_Recall$_G$* results with respect to low and high belief ranges are presented in Figure 6-47 and Figure 6-48 respectively.

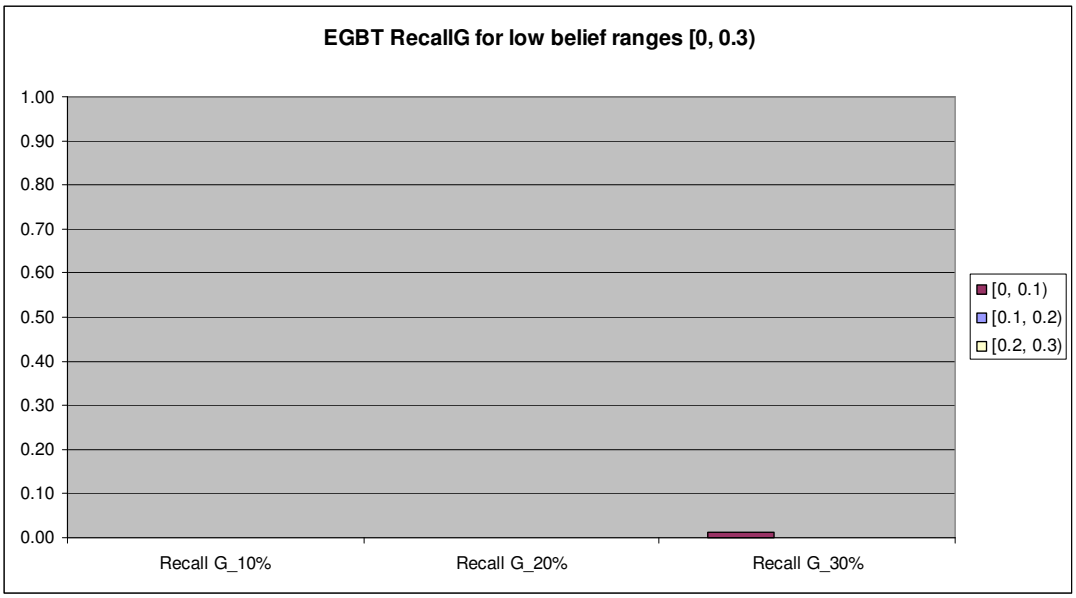

Figure 6-47 – EGBT_Recall$_G$ results for different delayed events percentages with respect to low belief ranges

The chart in above figure (Figure 6-47) illustrates that as ideally expected *EGBT* has not computed low belief values for *genuine* events in all three cases of different delayed events percentages. More specifically, the percentage of *genuine* events found to have belief values within [0, 0.3) is almost zero for all three cases.

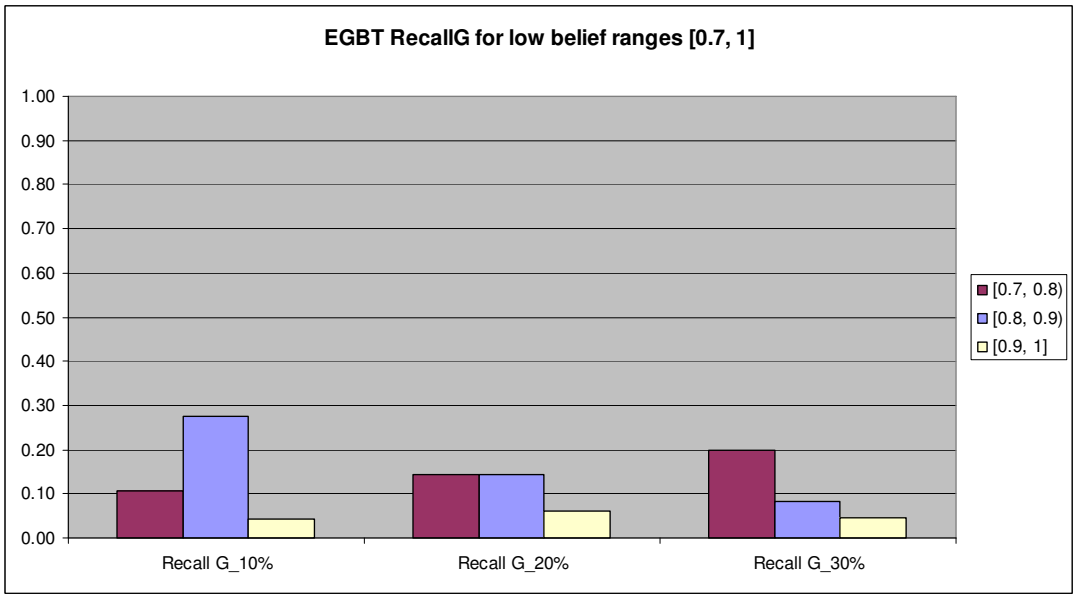

Figure 6-48 - EGBT_Recall$_G$ results for different delayed events percentages with respect to high belief ranges

On the contrary, Figure 6-48 illustrates that *EGBT_Recall$_G$* values with respect to high belief ranges are quite undesired for all three cases of delayed events we have been

experimenting with. While, $EGBT\_Recall_G$ values should ideally be much greater than 0.5, the experimental results we got were quite low, with a max $EGBT\_Recall_G$ value less than 0.3.

As an overall observation of the *EGBT recall* sensitivity with respect to *genuine* events, we could say that *EGBT* failed to compute belief values greater than 0.5 for *genuine* events.

6.5.2.4.1.4 EGBT_Precision$_G$

The charts regarding the overall experimental $EGBT\_Precision_G$ results with respect to low and high belief ranges are presented in Figure 6-49 and Figure 6-50 respectively.

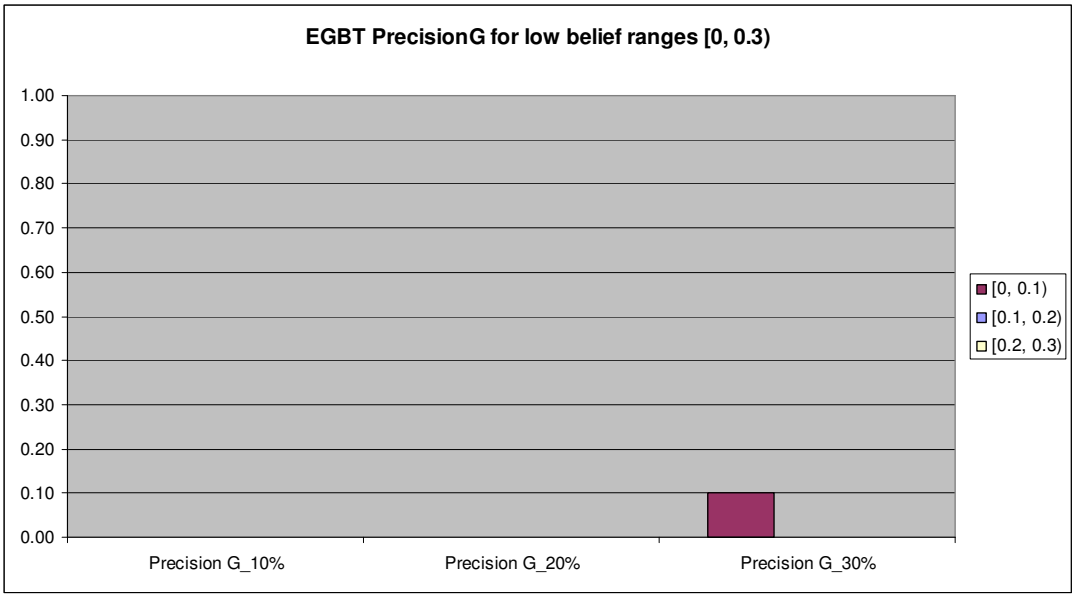

Figure 6-49 - EGBT_Precision$_G$ results for different diagnosis windows with respect to low belief ranges

By observing the chart of the above figure (Figure 6-49), *EGBT precision* with regards to *genuine* events was as low as expected for all three cases of delayed events. More specifically, the $EGBT\_Precision_G$ max value we got was equal to 0.1 for the case where the 30% of events were delayed.

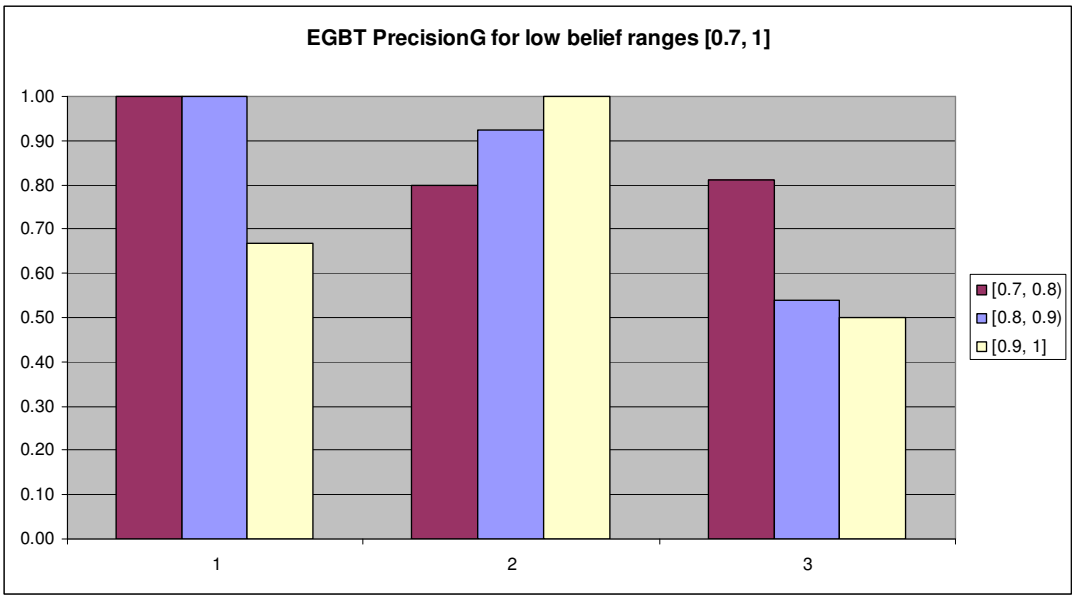

Figure 6-50 - EGBT_Precision$_G$ results for different diagnosis windows with respect to high belief ranges

Regarding belief ranges within [0.7, 1], it is expected that *EGBT_Precision$_G$* values should be quite high, i.e., greater than 0.5. For all *explanationConfiguration2* experiments, Figure 6-50 illustrates that indeed values are quite satisfying, as the min *EGBT_ Precision$_G$* value is 0.5.

As an overall observation about sensitivity of *EGBT_Precision$_G$* with respect to the percentage of delayed events, we could say that *EGBT_Precision$_G$* found to have no particular sensitivity.

6.5.2.4.2 VDT correctness results charts and discussion

Similarly to above charts and discussion regarding *EGBT correctness*, the following sections include charts and discussion on the overall *experimentalConfiguration2* results for each *VDT correctness* metric i.e., *VDT_Recall$_F$, VDT_Precision$_F$, VDT_Recall$_G$*, and *VDT_Precision$_G$*.

6.5.2.4.2.1 VDT_Recall$_F$

The charts regarding the overall experimental *VDT_Recall$_F$* results are presented in Figure 6-51.

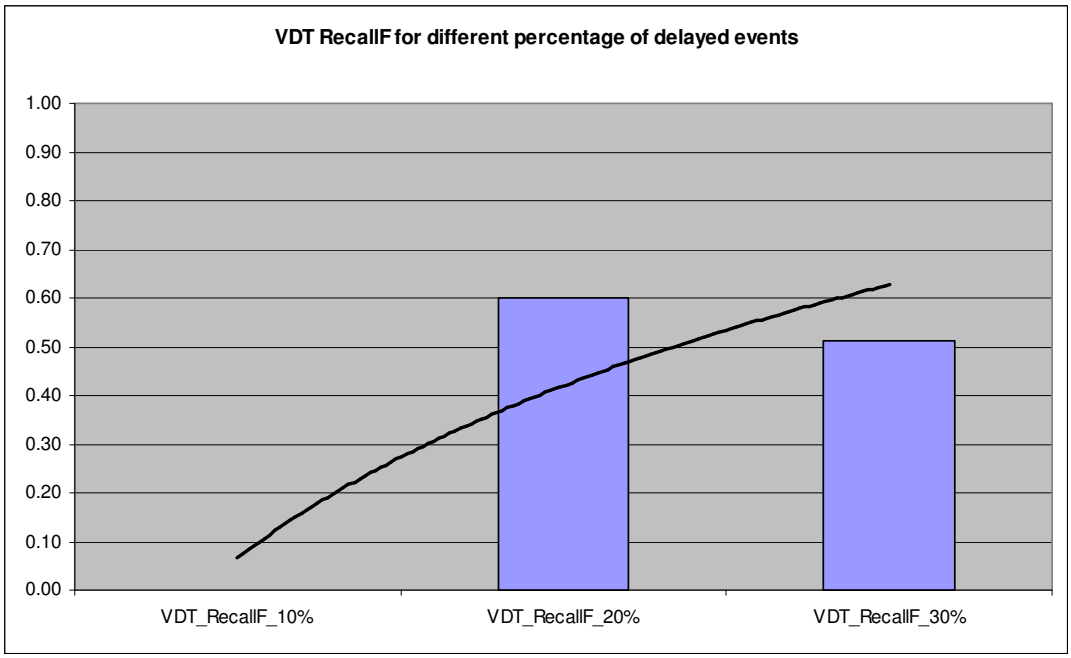

Figure 6-51 - VDT_Recall$_F$ results for different percentages of delayed events

From the above chart (Figure 6-51), we could say that *VDT_Recall$_F$* results are quite satisfying. As it is expected, *VDT_Recall$_F$* were greater than 0.5, except for the case that the 10% of the events were delayed. Therefore, VDT seems to classify correctly *fake* events as *unconfirmed*, especially while the number of *fake* events increases.

6.5.2.4.2.2 VDT_Precision$_F$

While the *VDT_Recall$_F$* experimental results were quite satisfying, the *VDT_Precision$_F$* results are rather undesired, as they are presented in Figure 6-52.

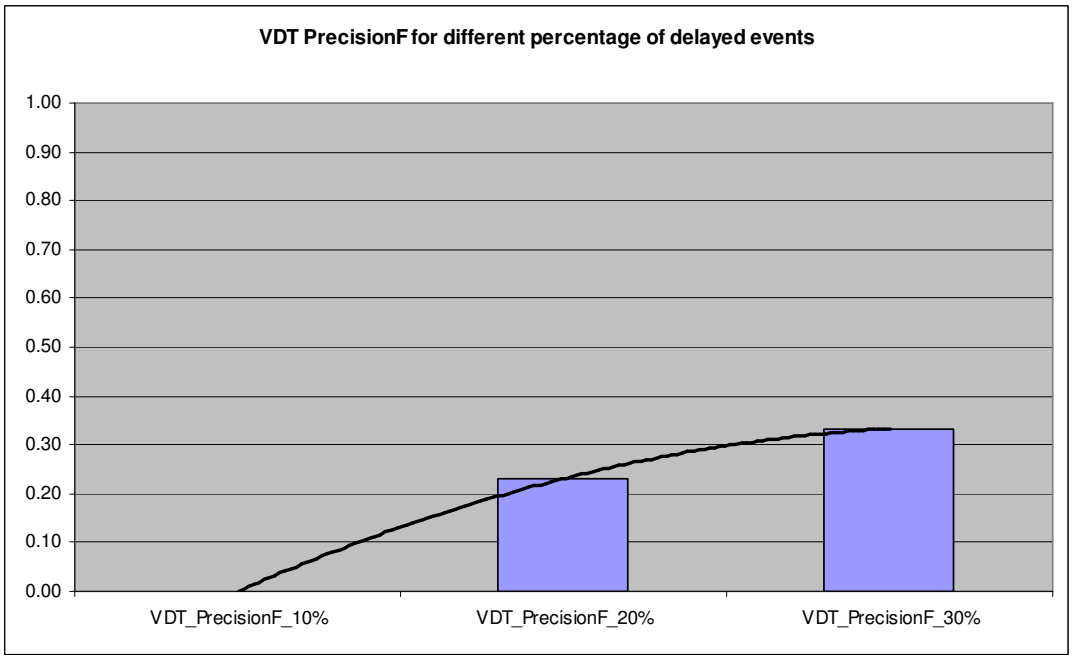

Figure 6-52 - VDT_Precision$_F$ results for different percentages of delayed events

More specifically, while it is expected to have *VDT_Precision$_F$* high values (i.e., ideally much greater than 0.5), our *VDT_Precision$_F$* experimental results are quite low for all different percentages of delayed events, with max value equal to 0.32. That means that *VDT* prototype classified unsuccessfully as *unconfirmed* events mostly *genuine* events rather than *fake* ones.

6.5.2.4.2.3 VDT_Recall$_G$

The charts regarding the overall experimental *VDT_Recall$_G$* results are presented in Figure 6-53.

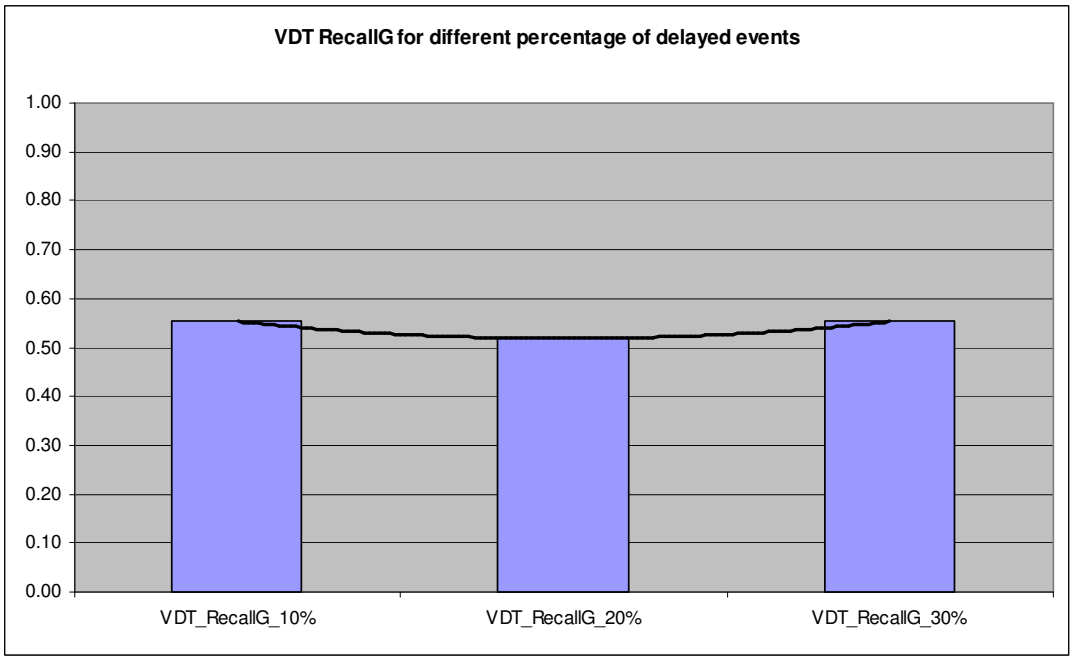

Figure 6-53 - VDT_Recall$_G$ results for different percentages of delayed events

From the above chart (Figure 6-53), we could say that *VDT_Recall$_G$* results are rather satisfying. As it is expected, *VDT_Recall$_G$* is greater than 0.5 for all three different cases we experimented with. Therefore, VDT seems to classify correctly *genuine* events as *confirmed* independently to the number of *fake* events the event set might contain.

6.5.2.4.2.4 VDT_Precision$_G$

VDT_Precision$_G$ results are quite satisfying, as they are presented in Figure 6-54.

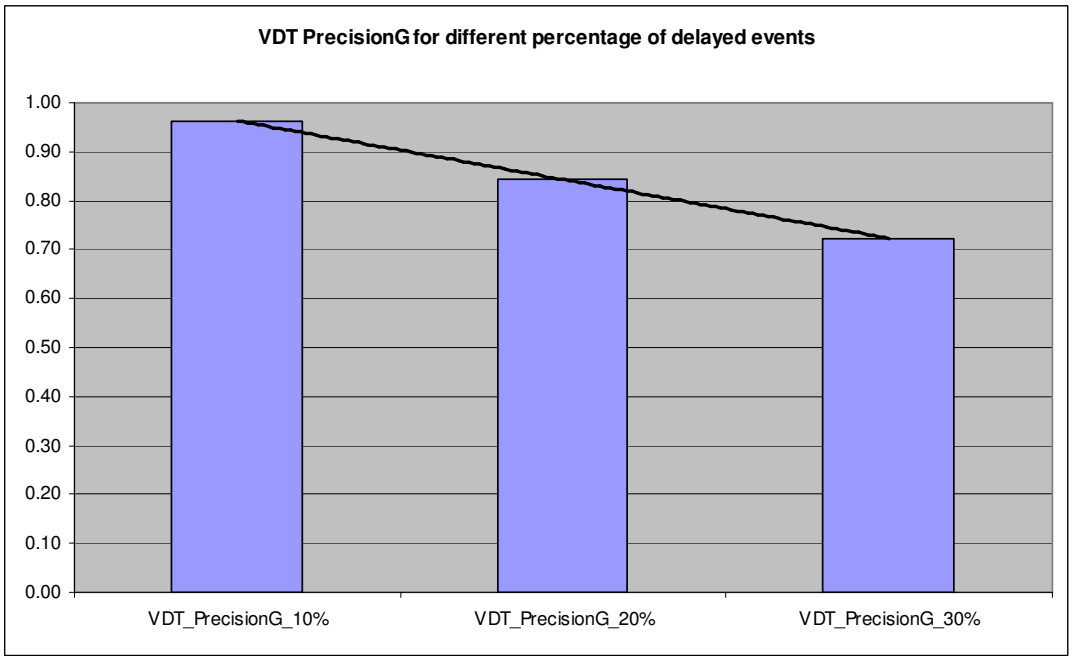

Figure 6-54 - VDT_Precision$_G$ results for different percentages of delayed events

More specifically, our *VDT_Precision$_G$* experimental results values are quite high for all different percentages of delayed events, as ideally expected. However, one can observe that while the percentage of delayed events is increasing the *precision* value decreases. Therefore, the above results show that *VDT* prototype classified successfully as *confirmed* events the *genuine* ones, but with a declining rate while the percentage of delayed events was increasing.

6.5.2.4.3 EGBT and VDT responsiveness results charts and discussion

In this section, we present charts regarding the *EGBT* and *VDT responsiveness* as occurred throughout the *explanationConfiguration2* experiments series. It should be noted that the charts in the flowing figures (Figure 6-55 and Figure 6-56) focus only on the mean *EGBT* and *VDT* computational times.

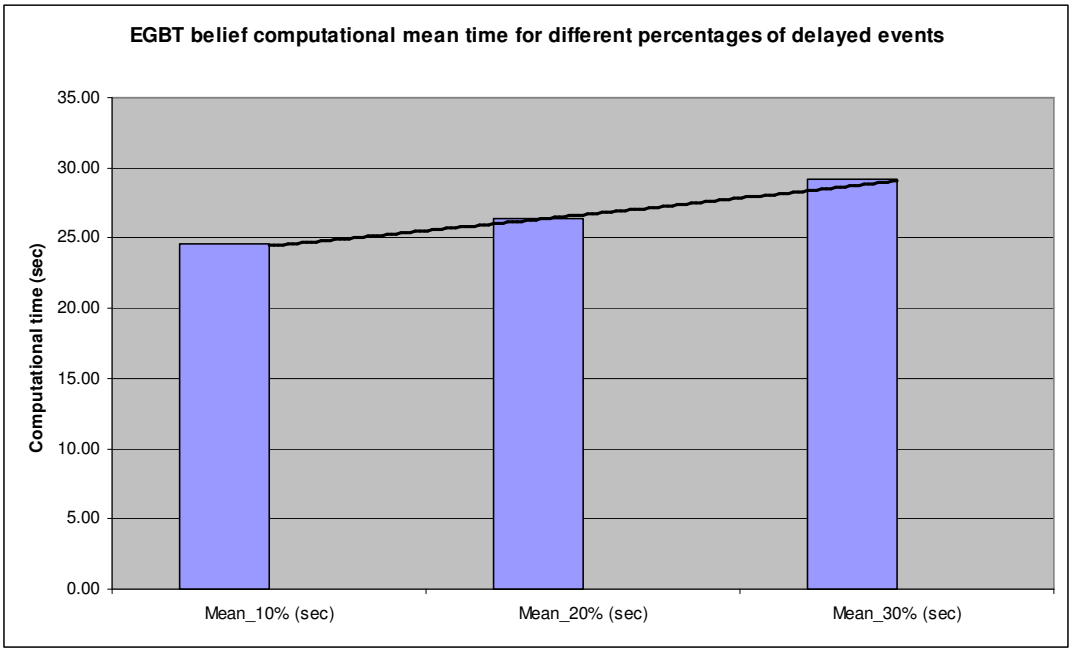

Figure 6-55 – EGBT belief computational mean times for different percentages of delayed events

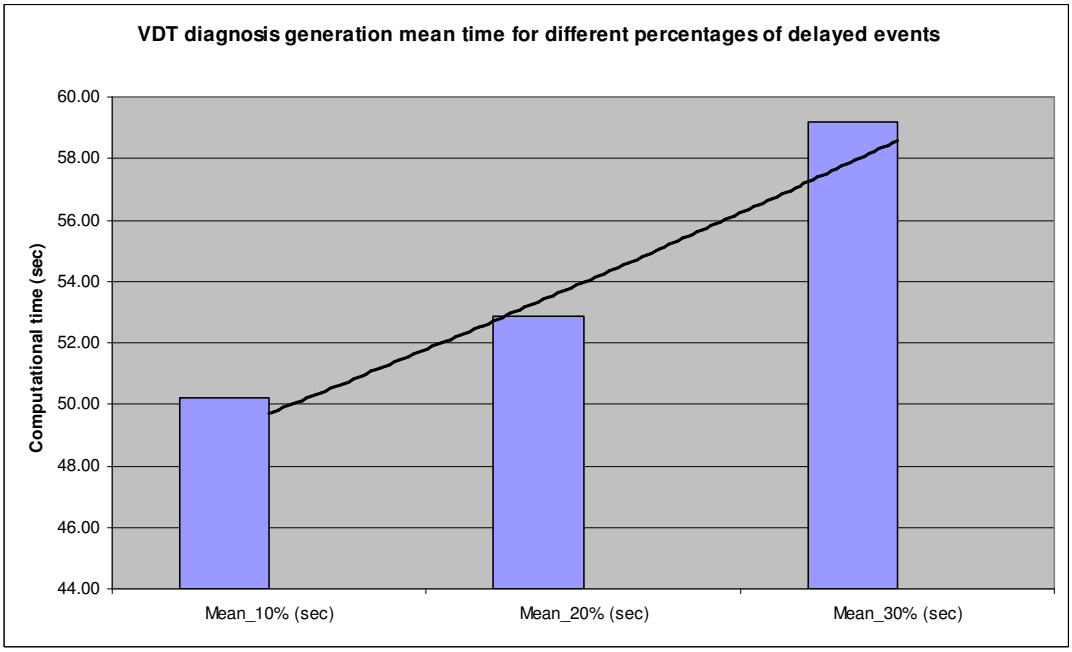

Figure 6-56 - VDT belief computational mean times for different percentages of delayed events

From both charts, we can observe that the mean computational times for both *EGBT* and *VDT* are increasing almost linearly while the percentage of events randomly delayed is being increased. The behaviour of both *EGBT* and *VDT* computational times against the increment of the percentage of delayed events is as expected. This is because, given a

common diagnosis window, the increment of events randomly delayed implies the increment of the time ranges *EGBT* should take into account when it computes the belief values of events. Consequently, for the belief computation of a given event, the longer the time ranges *EGBT* should take into account, the more events should be processed by *EGBT*. Therefore, the more delayed events there are, the longer time *EGBT* needs to compute the belief and plausibility of a given event.

Regarding *VDT*, it should be recalled that given a violation *VDT* requests the belief values of *violation observations* to be computed by *EGBT*. Therefore, it is reasonable to observe *VDT* final diagnosis generation time depending on the percentage of delayed events.

Chapter 7: Open Research Issues and Future Work

7.1 Overview

The aim of this chapter is to provide the reader with some of our insights regarding the open issues that emerged from our work in diagnosis of security and dependability violations, and we would like to put research effort on, hopefully, the near future.

The reason we would like to put extra research effort to improve the performance and extend the capabilities of the diagnostic prototype, as discussed in the rest of this chapter, is due to the significance of the role diagnostic mechanisms could play in the evolution of systems from a security perspective. EVEREST, and subsequently its diagnostic extension presented in this thesis, has already been used as the monitoring framework of the SERENITY project [146, 154]. Briefly, one of the key objectives of SERENITY has been the support of systems which operate in dynamic environments to configure, deploy and adapt mechanisms for realising S&D Properties dynamically. In particular, SERENITY project has produced a runtime framework, known as SERENITY Runtime Framework (SRF), enabling the dynamic selection, configuration and deployment of components that realise S&D Properties according to *S&D Patterns*. An S&D Pattern in SERENITY specifies a reusable S&D Solution for realising a set of S&D properties. It also specifies the contextual conditions under which this solution becomes applicable, and invariant conditions that need to be monitored at runtime in order to ensure that the solution described by the pattern behaves correctly. EVEREST has been used as a service to the SRF and when an S&D Pattern is activated it undertakes responsibility for analysing and checking conditions regarding the runtime operation of the components that implement the pattern. Runtime analysis was complemented by the diagnosis mechanism, described in this thesis, which deduces belief metrics for the plausible reasons for a mismatch between service/component modelling and actual behaviour. Further work on the limitations of our diagnostic approach could improve the performance of the process itself and the quality of the generated diagnostic information by aiming to provide sufficient information for preserving the evolvable systems users' privacy and security.

By analyzing the evaluation results, presented in Chapter 6:, one of the first lines of work is the optimization of our diagnostic approach. Some premature thoughts for optimization are discussed in Section 7.2.

Another interesting line of future work refers to further experimentation with our approach and is discussed in Section 7.3. The objective of this line of work is to provide us with an extended view of the potential and weaknesses of our diagnostic approach. To this direction, it would be interesting to explore the sensitivity of the diagnostic approach against factors and experimental configurations that we have not experimented with so far. In particular, future work plans, discussed in Section 7.3.1, aim in experimentation with an extended variation of adversaries capabilities and simulated attacks besides the delay attack used for the experimental evaluation of this thesis (see Section 6.2.3). Also, investigating the sensitivity of our approach against an extended set of values for our approach belief functions constants α_1 and α_2 (see Section 5.4.6.2) is another line of further experimentation work and is discussed in Section 7.3.2. The results of such experimentation could be also analyzed on theoretical basis to study the relation of the constants values and the uncertainty interpretations that could be generated. Finally, another line of further experimentation work, discussed in Section 7.3.3, is to investigate the sensitivity of our diagnostic approach against some characteristics of the underlying monitoring theory like the number of assumptions and the coverage of theory against the set of observed runtime events.

From a security perspective, the generation of notifications that would indicate faulty system components or components sensors could be valuable to the recovery action decision making process. In particular, Section 7.4 discusses some of our premature thoughts regarding a notification scheme that could generate notification reports, indicating the likelihood of system components or components sensors to be faulty, based on the diagnosis results of detected violations our diagnostic approach produces. It should be noted that the present version of out diagnostic approach generates diagnostic results, which flags events involved in detected violations as confirmed or confirmed by taking into account belief metrics in the event genuineness.

Finally, Section 7.5 introduces briefly other interesting lines of work and open research questions emerged during our work on diagnosis.

7.2 Optimization of the diagnostic prototype

The results of our diagnostic prototype experimental evaluation presented in Section 6.5 reveals some weaknesses that would concern us as a line of future work. More specifically, from an *EGBT* and *VDT correctness* (see Section 6.3.1) point of view, it would be interesting to investigate the reasons for having rather undesired results for our tools *precision* with respect to *fake* events (see definitions of $EGBT\_Precision_F$ and $VDT\_Precision_F$ in Sections 6.3.1.2 and 6.3.1.3 respectively) and *recall* regarding *genuine* events (see definitions of $EGBT\_Recall_G$ and VDT_Recall_G again in Sections 6.3.1.2 and 6.3.1.3 respectively). Of course, one reason to cause such results could be the event set that was used for the evaluation included in the present thesis. Therefore, rechecking how the event set was generated by our simulator, and the characteristics of the used event set would be one of the first tasks of this optimization line of future work.

From an *EGBT* and *VDT responsiveness* (see Section 6.3.2) perspective, it would be desired to have lower belief values and diagnosis generation *computational times* (see Sections 6.3.2). A premature thought to improve these *computational times* can be the introduction of an extra step to our approach after the diagnosis prototype is notified with the selected monitoring theory assumptions and diagnosis window. During this step the monitoring theory assumptions would be statically analyzed to compute and store all the possibly generated explanations and consequences trees at symbolic level. Particularly, for each predicate specified in a given theory assumptions, an explanation and consequences tree would be computed; however no specific values would be assigned to the involved predicates parameters, timestamps and time ranges. Therefore, when a diagnosis result would be required at runtime for some monitoring rule violation, our diagnostic approach could make use of the explanation and consequences trees generated during the static analysis. More specifically, for each violation observation the corresponding explanations and consequences tree could be retrieved and instantiated with the actual runtime values of the observation itself and other available runtime recorded events that could be unified with the predicates of the retrieved tree. The gain of the assumptions static analysis phase would be the reduction of time consumed at runtime by invocating multiple times the processes implementing the algorithms *Explain* and *Derive_AE_Consequences* (see Sections 5.2.1 and 5.3.1 respectively) to compute sets of possible explanations and expected consequences of events that are modeled by the same predicate at symbolic level.

7.3 Further Experimentation

7.3.1 Extended adversaries capabilities experiments

It would be interesting to investigate how various *adversaries* capabilities and therefore simulated attacks may affect the performance of our approach. As a reminder, it should be noted that the delay attack simulated for the present thesis experimental evaluation specified adversaries implemented to intercept randomly events of a set of predefined event types and introduce a delay to events dispatch time stamp (see Sections 6.2.3 and 6.4.1.3). Another attack, we would like to experiment with, would specify adversaries implemented to intercept randomly and block events of a set of predefined event types. Our intuition is that this block attack should affect the performance of our approach prototype by reducing the event set and therefore the potential evidence our approach uses to compute belief values and deciding whether observations involved in a violation are confirmed or unconfirmed, according to the criterion introduced in Section 5.5.1.

Two more complicated and interesting simulated attack would specify adversaries implemented to have memory. The first of these attacks specify adversaries having memory and being capable of replicating events intercepted in the past, and dispatch random numbers of replicated events with an updated timestamp. The effect of such replicating attack would be the increment of number of events with same payload (if events are considered as messages) but different time stamps. Our intuition is that especially for long diagnosis windows the diagnostic prototype would take into account the events generated by this attack as genuine events. Therefore, the prototype performance would not be that optimal in such circumstances.

The second attack that could use adversaries with memory could specify adversaries implemented to alter the content and context information of the events. It should be noted that, as specified in Section 3.3.1, the event signature arguments could be considered as the event content, whilst context information could be considered the event parameters *_sender*, *_receiver*, *_status*, and *_source*. It would be interesting to evaluate the performance of our diagnostic approach with such altered events, as at the moment there is no specific insight regarding the performance of our approach under such events conditions.

Whilst experimental evaluation taking into account the above adversary models individually is interesting to investigate, the experimentation with simulated attacks that

specify the orchestration of multiple instances of different adversary models is intriguing due to the fact that at the moment we cannot foresee the performance of our approach against such attacks. Such orchestration attacks could specify adversaries orchestration to cause undesired effects on different components of the simulated system, like a combined denial of service attack on two key components of the simulated system. In the same direction, the usage of an operation trace, which has been used for benchmarking intrusion detection systems like the DARPA data set [27], could be another alternative to evaluate our approach against attacks occurred and recorded at real time system operations.

7.3.2 Extended belief function constants experiments

Another alternative line of further experimentation refers to evaluation of our approach against an extended set of values for constants α_1 and α_2 that are used in the belief functions of our approach. As discussed in Section 6.4.2, we used only a couple of constants values for the experimental evaluation included in the present thesis. Future plans presume an extended set of values within range [0, 0.4] and a number of possible combinations of values for the two constants. It should be noted that the above range is specified with relatively low valued boundaries due to the fact that α_1 and α_2 constants are used as the actual masses assigned to events for which no consequences can be identified (by using α_2) and no explanations can be generated (by using α_1). Therefore, as discussed in Section 5.4.6.2, such cases with no identified consequences or generated explanations should result with low belief values.

Besides the examination of our approach sensitivity against the belief functions constants, analysis of the experiment results generated from a range of constants values combinations could also be worthy to study on theoretical basis the relation between the two belief function constants against the uncertainty interpretations that would be generated for a common set of events. It should be noted that, as discussed in Section 5.4.6.2, α_2 should be greater than α_1 to favor cases where no consequences can be identified within a given time window against cases where no explanations can be generated. However, it would be interesting to examine cases where constants are equal or α_1 is greater than α_2. Examining and comparing the results of such cases, we might get indications of the means that the relation between the constants affects the interpretation of the uncertainty an event set could carry. Of course, the uncertainty interpretations

would only be expressed in terms of belief metrics in the genuineness of the examined events.

7.3.3 Extended underlying monitoring theory experiments

Two are the monitoring theory characteristics whose impact on our diagnostic prortotype performance seems quite interesting for extended investigation. More specifically, the number of assumptions, which are used during the diagnostic process abductive and deductive phases, as well as, the theory coverage against the set of the observed runtime events may affect the diagnosis result, according to our intuition.

From an abductive process-wise point of view, the more assumptions we have for a specific type of event, the more explanations can be generated for it. From a deductive perspective, the more assumptions there are to identify the effects of a type of event, the more consequences are generated. Due to the fact that our event genuineness belief assessment scheme is based on additive functions, and therefore compute belief values analogously to the cardinality of sets taken into account, our intuition is that a theory A with relatively bigger number of assumptions than theory B will generate higher belief values. Of course, the frequency of recorded events that could be taken into account as matching recorded events by our process as well as the values of constants α_1 and α_2 that are used in belief functions could play significant role to the diagnosis outcome. Moreover, the number of assumptions of a theory is likely to affect the responsiveness of the diagnostic process by introducing analogously computational delays.

The theory coverage against the set of the observed runtime events is another characteristic that could play significant role in the performance of our diagnostic process. More specifically, the higher coverage a theory has against the set of the runtime and recordable events, the higher possibility there is that we entail with high belief values and belief computational times. As a counter example that makes our intuition stronger, assume the explanation generation process for an event *e*. In case that there are no assumptions formulas containing *e* in their head, no explanations for *e* can be generated and effectively according to belief function m_i^{EX} (see Definition 10 in Section 5.4.6.2), *e* belief value is set instantly equal to α_1.

7.4 Combining Diagnosis Results

From a security perspective, a significant research question that we have pointed out is whether and how reasoning on diagnosis results of multiple monitoring rules violations could generate indications (containing perhaps likelihoods) for faulty components or components sensors. It should be noted that the diagnosis results generated by the current version of our diagnostic approach contains only belief metrics in the genuineness of the events involved in monitoring rules violations. It is these belief metrics that perhaps an administrator of the monitored system should take into account in order to initiate recovery actions after detection of violations. What we are suggesting as future line of work in the present section is a notification scheme that includes indications for faulty system components or sensors.

The notification scheme should specify a reasoning module that should take into a predefined number of monitoring rules violations and their diagnosis results within a predefined time window. The notification reasoning module would then reason on the belief values included in the diagnosis results and generate notification reports about the likely faulty components and sensors involved in the examined violations. It should be noted that the notification scheme should specify a structured notification report schema that would allow the communication of notification reports among the relevant parties. More specifically, the notification reasoning module should generate notification reports according to the notification schema in order that a recovery action decision making process or a security administrator could be able to use the reports to decide for and initiate the appropriate recovery actions.

To illustrate our premature thoughts, please consider the following example. Assume that during a given time window and for a given threshold of detected violations, the framework that monitors and diagnoses violation occurred in LBACS, generates a number of violations that exceeds the predefined threshold for rules LBACS.R1 and LBACS.R2 as are specified below.

LBACS.R1. $\forall t1 \in Time, \exists t2 \in Time, \forall\_LServerId \in LocationServers,$
$\forall\_ACServerId \in AccessControlServers, \forall\_deviceId \in Devices, \forall\_source.$

Happens(e(_Id1,_ACServerId,_LServerId,REQ-A,locationRequest
(_deviceId),_source), t1, R(t1,t1)) ⇒
Happens(e(_Id2,_LServerId,_ACServerId,REQ-A,locationResponse
(_deviceId),_source), t2, R(t1+1,t1+3000))

LBACS.R2. ∀t1∈Time, ∃t2∈Time, ∀_LServerId ∈ LocationServers,
∀_ACServerId ∈ AccessControlServers, ∀_deviceId∈Devices,
∀_receiver1∈Sensors, ∀_source1, ∀_source2.

Happens(e(_Id1,_ACServerId,_LServerId,REQ-A,locationRequest
(_deviceId),_source1), t1, R(t1,t1)) ∧
¬**Happens**(e(_Id1,_ACServerId,_LServerId,REQ-A,locationRequest
(_deviceId),_source1), t2, R(0,t1-1)) ⇒
Happens(e(_Id2,_deviceId,_receiver1,REQ-A,signal(_deviceId),
_source2), t3, R(t1-2000,t1-1))

The monitoring rule LBACS.R1 is violated in all cases where, provided that the access control server of LBACS requests location information for a device from the location server of LBACS, the location server does not provide such information within the next 3 seconds after the corresponding request occurrence. Therefore, a violation of LBACS.R1 contains:

- an occurrence of a *locationRequest* event at R1.t1 – referred to as *locationRequest@R1.t1* henceforth – and

- a negated *locationRequest* event time-stamped at R1.t1+3000 – referred to as ¬*locationRequest@R1.t1+3000*, which is generated by negation as failure and indicates that no *locationRequest* event occurred within [R1.t1+1, R1.t1+3000] .

It should be noted that all events specified in LBACS.R1 are captured from the receiver side sensors, as all predicates status is set to *REQ-A*. The *REQ-A* predicate status value signifies that the event represented by the given predicate is captured after request from receiver's sensor (see Section 3.3.1).

Rule LBACS.R2 checks when the first signal from a device should have occurred. In particular, the first signal from a given device is expected within the last two seconds before the first request for the device location made by the LBACS access control server. A violation of LBACS.R2 contains:

- an occurrence of a *locationRequest* event at R2.t1 – referred to as *locationRequest@R2.t1* henceforth;

- a non occurrence of a *locationRequest* event at R2.t1-1 - referred to as ¬*locationRequest@R2.t1-1* henceforth – indicating that no *locationRequest* occurred within [0, R2.t1-1], and

- a negated *signal* event time-stamped at R2.t1-1 – referred to as ¬*signal@R2.t1-1*- that is generated by negation as failure and indicates that no *signal* event occurred within [0, R2.t1-1].

It should be noted again that all events specified in LBACS.R2 are captured from the receiver side sensors, as all predicates status is set to *REQ-A*.

Moreover, assume that in the most of the final diagnosis results the diagnosis module generates for rule LBACS.R1 violations, event ¬*locationRequest@R1.t1+3000* is flagged as confirmed, whilst event *locationRequest@R1.t1* is flagged as unconfirmed. Also, by taking into account the average of the belief values for the involved events for all examined LBACS.R1 violations, assume that the average of belief in genuineness of the event *locationRequest@R1.t1* is quite lower that the corresponding belief value of the event ¬*locationRequest@R1.t1+3000*. According to above diagnosis results and belief metrics, the occurrence of *locationRequest@R1.t1* seems less plausible from the occurrence of ¬*locationRequest@R1.t1*+3000. Therefore, a notification that the component that generates or the sensor that captures the event *locationRequest@R1.t1* is faulty can be made.

By reasoning on diagnosis results of rule Chapter 1: violations, the above notification might be enhanced. This could happen in case that initially there is a Chapter 1: violations number that exceeds the predefined violation threshold. Moreover, in the most of the Chapter 1: violations diagnosis results, event *locationRequest@R2.t1* should be flagged as unconfirmed, whilst the events ¬*locationRequest@R2.t1-1* and ¬*signal@R2.t1-1* should be flagged as confirmed. Finally, in the examined Chapter 1: violations, events *locationRequest@R2.t1* should have a lower average of belief values than the average belief values of events ¬*locationRequest@R2.t1-1* and ¬*signal@R2.t1-1*. By these means, events *locationRequest@R2.t1* seems again less plausible than the events ¬*locationRequest@R2.t1-1* and ¬*signal@R2.t1-1*, enhancing our hypothesis that the component that generates or the sensor that captures the event *locationRequest@R1.t1* is faulty.

Of course, the above notification might not be totally accurate; however, it can be taken into account as an indication during the recovery action decision making process. To try to ameliorate the notifications accuracy, research effort should be put on the business logic of the reasoning notification module, i.e., how the reasoning module

reasons on a set of diagnosis results. For instance, in the above example, we have mentioned a comparison on the average of the belief values of the events involved in a set of detected violations of the same rule. Also, the factors that might affect the accuracy of such notifications like the predefined violation threshold, and the time window, which examined violations lie within, might be of significance importance. On the other hand, regarding the notification report schema, it should be designed carefully to meet requirements regarding the support of recovery action decision making process.

7.5 Other open research issues

This section discusses briefly other open research questions emerged with our diagnosis oriented work. More specifically, other open issues we have pointed out are as follows:

- *Quality assessment and update of underlying monitoring theory assumptions*. Our diagnosis process could be extended to keep track of the number of times a monitoring theory assumption is used during the abductive and deductive phases. Premature thoughts presume that the generated assumption usage frequencies could be used to rank the monitoring theory assumptions and provide indications for the quality of the monitoring specifications. Having as reference research work on Bayesian networks, which are graphical models for encoding causal and probabilistic relationships among variables of interest of a given knowledge domain [84, 85, 123, 124, 125, 126], and especially Bayesian inference and learning techniques like approaches presented by Heckerman [76], monitoring theory assumptions with lower frequency could be reconsidered or restructured to generate new assumptions. The newly generated assumptions could then specify events correlations that have not been specified within the initial monitoring theory.

- *Extensive comparison between Dempster Shafer theory of evidence (DS theory)* [146] *and Bayesian reasoning* [84, 85, 123, 124, 125, 126] *for handling uncertainty.* Having been aware of approaches that handle uncertainty by using Bayesian networks as the approach by Pan et al. [121], an interesting theoretical line of research is a comparison of our present diagnostic process that is based on DS theory to a similar diagnostic process based on Bayesian reasoning. To this direction, another version of our diagnostic approach based on Bayesian reasoning would be necessary. To obtain such a diagnosis approach, the foreseen challenges

might be faced should refer to the theoretical and practical differences between DS theory and Bayesian reasoning. For instance, an issue might emerge regarding the specification of prior probabilities that are required by the Bayesian reasoning in order to function. Of course, we have faced something similar during the design of the present diagnosis approach when it was required to assign preliminary masses that reflect the initial knowledge of the examined system. The results of an extensive comparison between these two uncertainty handling frameworks against a common use case could provide the basis for a discussion on the relative merits and demerits, as the ones pointed out in [96, 151, 152].

Chapter 8: Conclusions

8.1 Overview

The final chapter of the present thesis provides an overview of the research work resulted in the diagnosis approach, which was presented through out the previous chapters. Besides the overview, the present chapter points out the diagnosis approach main novelties and the contributions that our research has made to the state of the art. Our claims are founded on a comparison with other diagnostic approaches. Finally, the diagnostic approach limitations are given.

8.2 Summary of the research work

Designed as an extension of *EVEREST* monitoring framework [109, 153, 155], the diagnostic framework this thesis presented aims to the identification of possible explanations for the violations of S&D properties specified as *EVEREST* monitoring rules. To design the diagnostic framework, we have specified some extensions in *EC-Assertion* language that *EVEREST* monitoring rules and assumptions are specified. These extensions were made to support the basic formulation of the diagnosis problem as discussed in Section 4.2.

As a mechanism of trying to find possible causes of the runtime events that have caused a violation of an S&D monitoring rule, abductive reasoning [122] is used. More specifically, to generate the possible explanations of S&D violations, we devised an abductive algorithm for generating explanations for events that are involved in the detected violations discussed in Section 5.2. This algorithm generates a list representing the alternative explanations for a particular event by taking into account the intended behaviour of the monitored system as it is specified in *EC-Assertion* assumptions. It should be noted that the aforementioned algorithm treats any occurring time constraint satisfaction problem as linear programming problem by using the Simplex method [63].

After the generation of the possible explanations for the events involved in the violation of a rule, the diagnosis process identifies the expected effects of these explanations and uses them to assess the plausibility of the explanations. The assessment of explanation plausibility is based on the hypothesis that if the expected effects of an explanation match with events that have occurred (and recorded) during the operation of

the system that is being monitored, then there is evidence about the validity of the explanation. To identify therefore any effects of the generated explanations, a deductive algorithm that generates all the possible derived observations from the abductive explanations by using the system assumptions has been devised. The consequences identification algorithm presented in Section 5.3 treats again any occurring time constraint satisfaction problem as linear programming problem by using the Simplex method [63].

Having identified the expected effects of the abductive explanations of the violation observations, the diagnosis mechanism assesses the genuineness of violation observations. Based on the hypothesis mentioned above, i.e. if the expected effects of an explanation match with observations that have occurred (and recorded) during the operation of the system that is being monitored, then there is evidence about the validity of the explanation, there is the possibility that we would not be able to confirm or disconfirm the validity of the explanation at the time that diagnostic process searches for evidence. To deal with this uncertainty, the diagnosis mechanism advocates an approximate reasoning approach which generates degrees of belief in the membership of observations in the log of the monitor and the existence of some valid explanation for it rather than strict logical truth values. These degrees of belief are computed by functions founded in the axiomatic framework of the Dempster-Shafer theory of evidence [146] (see also Section 5.4.6).

Finally, we have provided a scheme for final diagnosis reports of detected S&D violations. Based on the beliefs computed for the genuineness of the individual events involved in an S&D violation (i.e. *violation observations*), the scheme generates as a final diagnosis for the given violation a report of the *confirmed* and *unconfirmed violations observations*. As discussed in Section 5.5, a *violation observation* P will be classified as a *confirmed* event if the belief in the genuineness of P is greater than or equal to the corresponding disbelief, i.e., $Bel(P) \geq Bel(\neg P)$[7]. A negated *violation observation* ¬P, will be classified as a unconfirmed predicate if $Bel(P) \leq Bel(\neg P)$.

7 Bel(P) and Bel(¬P) represent the proposition Bel(Genuine(e,U_o,TR)) and Bel(¬Genuine(e,U_o,TR)) respectively.

8.3 Main novelties

To generate explanations for the violations of S&D properties specified as *EVEREST* monitoring rules, the diagnostic mechanism uses *abductive reasoning* [122] that takes into account the temporal aspects of the violation observations (i.e. time stamps) and the underlying monitoring rules and assumptions (i.e. time ranges have been specified to indicate the intended behaviour of the monitored system). In other temporal abduction approaches [28, 41, 42, 53, 140], temporal knowledge can be expressed as temporal constraints, which are associated to the rules of the underlying domain theory. Such temporal constraints must be satisfied by the temporal information associated to the generated explanations. On the other hand, our temporal abductive approach that is based on reasoning on events and formulas specified in *EC-Assertion* whose formal foundations are based in *Event Calculus*, temporal knowledge can be represented as information embedded in the underlying theory formulas. As mentioned in Section 8.2, our abductive mechanism treats any occurring time constraint satisfaction problem as linear programming problem by using the Simplex method [63]. Therefore, our approach draws upon work on temporal abductive reasoning [28, 41, 42, 53, 140] and its applications to diagnosis [52, 130], but is based on a newly developed algorithm for abductive search with *Event Calculus* that generates all the possible explanations of a formula (unlike [53, 140]).

Due to the fact that uncertainty is an inherent feature of abductive reasoning, the likelihood of abducible explanations truthness, can play significant role in the selection of the most preferable abductive explanation. Thus, probabilistic models and, in particular, Bayesian models have been used to identify the most plausible abductive explanation [50, 83, 90, 97, 123, 124, 125, 126, 131, 132, 140]. The use of Bayesian models imposes some limitations in the generality of logic-based abductive reasoning. In particular, the set of possible hypotheses must be determined in advance. Moreover, an a priori probability must be assigned to each of the possible hypothesis in advance, as well as, the conditional probabilities of consequences, given particular assumptions, must be predetermined. When these prerequisites are met, the Bayes' theorem can be applied in order to compute the conditional probabilities of the predefined possible hypotheses, given the observations to be explained. Based on the outputs of the Bayes's rule, the most possible combination of hypotheses, which jointly explain the observations, is selected. Our approach also uses a probabilistic explanation assessment approach. However, our

approach is not based on Bayesian abduction. The reason for this is to avoid the need to elicit the a-priori and conditional probability measures which are required by this approach. Furthermore, the choice of the Dempster-Shafer theory of evidence [146] as the framework for calculating the likelihoods of abduced explanations has been dictated by the need to represent the uncertainty regarding the confirmation of the consequences of these explanations as discussed Section 5.4 and reason in the presence of this uncertainty.

8.4 Limitations

The diagnosis approach for S&D properties violations we have presented in the thesis happens to have some limitations. These limitations are enlisted below:

- The property specification language of *EVEREST*, and therefore of our diagnostic framework, is expressive enough to support a wide spectrum of S&D properties. However, the use of the language for the specification of such properties may be difficult for users who are not familiar with formal languages.
- The diagnostic prototype as it is presented in Section 6.2.1 does not implement the function that computes the basic probability assignment in the genuineness of an event *e* for the 2.i case of Definition 9 (see Section 5.4.6.2). More specifically, the aforementioned case refers to circumstances that:
    - no recorded events matching with *e* were found in the event log, and
    - the last known value of the clock of *Captor(e)*, i.e., the timestamp of the last event in the log that has produced by *Captor(e)*, at the time of the search is greater than the upper boundary of the time range that is specified for *e*.

  At such cases, events occurred within the upper boundary of *e* and the last time stamp of *Captor(e)* are used to compute the basic probability assignment in the genuineness of *e*.
- As discussed in Section 7.2, the results of our diagnostic prototype experimental evaluation presented in Section 6.5 reveal some weaknesses that would concern us as a line of future work. More specifically, from an *EGBT* and *VDT correctness* (see Section 6.3.1) point of view, it would be interesting to investigate the reasons for having rather undesired results for our tools *precision* with respect to *fake*

events (see definitions of $EGBT\_Precision_F$ and $VDT\_Precision_F$ in Sections 6.3.1.2 and 6.3.1.3 respectively) and *recall* regarding *genuine* events (see definitions of $EGBT\_Recall_G$ and VDT_Recall_G again in Sections 6.3.1.2 and 6.3.1.3 respectively).

References

1. Abercrombie P and Karaorman M (2002) jContractor: Bytecode instrumentation techniques for implementing design by contract in java. In Electronic Notes in Theoretical Computer Science, volume 70. Elsevier Science Publishers
2. Álvarez G and Petrović S (2003) A new taxonomy of web attacks suitable for efficient encoding. Computers and Security, 22(5), pp. 435-449
3. Alpern B and Schneider FB (1987) Recognizing safety and liveness. Distrib. Comput. 2, 117–126
4. Alur R, Fix L and Henzinger TA (1994) A determinizable class of timed automata. In Proc. 6th Int. Conf. on Computer Aided Verification (CAV'94), vol. 818 of LNCS, pp. 1–13. Springer
5. Anderson T and Lee PA (1990) Fault Tolerance: Principles and Practice. Springer-Verlag, Wien - New York
6. Andrieux A et al. (2004) Web Services Agreement Specification. Global Grid Forum, available from: http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf
7. Angluin D, Frazier M, and Pitt L (1992) Learning Conjunctions of Horn Clauses. Machine Learning 9, 2-3 (Jul. 1992), 147-164. DOI= http://dx.doi.org/10.1007/BF00992675
8. Appelt DE, Pollack M (1992) Weighted abduction for plan ascription. Technical report, Artificial Intelligence Center and Center for the Study of Language and Information, SRI International, Menlo Park, California
9. Arnold A (1987) Transition systems and concurrent processes. In Mathematical Problems in Computation Theory, Banach Center, Warsaw, pp. 9–21
10. Artho C, Havelund K, and Biere A (2003) High-level data races. Journal on Software Testing, Verification & Reliability (STVR), 13(4).
11. Artho C, Biere A, and Havelund K (2004) Using block-local atomicity to detect stale value concurrency errors. In Farn Wang, editor, Proc. ATVA '04. Springer.

12. Artho C, Schuppan V, Biere A, Eugster P, Baur M. and Zweimüller B (2004) JNuke: Efficient Dynamic Analysis for Java. In Proc. 16th Intl. Conf. On Computer Aided Verification (CAV 2004), volume 3114 of LNCS, pp. 462–465, Boston, USA. Springer

13. Artho C, and Biere A (2005) Combined Static and Dynamic Analysis. In Proc. AIOOL '05, Paris, France.

14. Avizienis A, Larpie JC and Randell B (2000) Fundamental Concepts of Dependability. In Information Survivability Workshop

15. Bandara AK, Lupu EC, and Russo A (2003) Using event calculus to formalise policy specification and analysis. Policies for Distributed Systems and Networks Proceedings, POLICY 2003, pp. 26- 39.

16. Baresi L and Guinea S (2005) Dynamo: Dynamic Monitoring of WS-BPEL Processes. ICSOC 05, 3rd International Conference On Service Oriented Computing, Amsterdam, The Netherlands

17. Baresi L, and Guinea S (2005) Towards Dynamic Monitoring of WS-BPEL Processes. ICSOC 05, 3rd International Conference On Service Oriented Computing, Amsterdam, The Netherlands.

18. Baresi L, Guinea S, and Plembani P (2005) Using WS-Policy in Service Monitoring. TES 05, 6th VLDB Workshop on Technologies for E-Services, Trondheim, Norway.

19. Barnett M, and Schulte W (2001) Spying on Components: A Runtime Verification Technique. In Proceedings of OOPSLA 2001 Workshop on Specification and Verification of Component Based Systems, Tampa, FL, USA.

20. Barringer H, Goldberg A, Havelund K and Sen K (2004) Rule-Based Runtime Verification. 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04), LNCS 2937, Springer, pp. 44-57

21. Bartetzko D, Fischer C, Moller M, and Wehrheim H (2001) Jass – Java with assertions. In Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01. Published in Electronic Notes in Theoretical Computer Science, K. Havelund and G. Rosu (eds.), 55(2).

22. Bouyer P, Chevalier F and D'Souza D (2005) Fault Diagnosis using Timed Automata. In Proc. 8th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'05), LNCS 3441, pp.219-233, Springer

23. Brat G, Drusinsky D, Giannakopoulou D, Goldberg A, Havelund K, Lowry M, Pasareanu C, Visser W, and Washington R (2004) Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. Formal Methods in System Design, 25(2).

24. Brisset P (2000) A Case Study in Java Software Verification. Appeared in Workshop on Security, Middleware, and Languages, Stockholm.

25. Brörkens M, and Möller M (2002) Dynamic event generation for runtime checking using the JDI. In Havelund, K. and Rosu, G., editors, Proceedings of the Federated Logic Conference Satellite Workshops, Runtime Verification, Copenhagen, Denmark. Electronic Notes in Theoretical Computer Science 70(4).

26. Brörkens M, and Möller M (2002) Jassda trace assertions, runtime checking the dynamic of java programs. In Schieferdecker, I., König, H., and Wolisz, A., editors, Trends in Testing Communicating Systems, International Conference on Testing of Communicating Systems, Berlin, Germany, pp. 39-48.

27. Brugger, S. T. and J. Chow (2007). An assessment of the DARPA IDS Evaluation Dataset using Snort. Technical Report CSE-2007-1, University of California, Davis, Department of Computer Science, Davis, CA. http://www.cs.ucdavis.edu/research/tech-reports/2007/CSE-2007-1.pdf.

28. Brusoni V, Console L, Terenziani P, Dupré DT (1997) An Efficient Algorithm for Temporal Abduction. In Proceedings of the 5th Congress of the Italian Association For Artificial intelligence on Advances in Artificial intelligence. M. Lenzerini, Ed. Lecture Notes In Computer Science, vol. 1321. Springer-Verlag, London, pp. 195-206

29. Brusoni V, Console L, Terenziani P, Pernici B (1997) LATER: Managing Temporal Information Efficiently. IEEE Expert: Intelligent Systems and Their Applications 12, 4.

30. Capra L, Emmerich W and Mascolo C (2001) Reflective middleware solutions for context-aware applications. In Yonezawa A, Matsuoka S, eds.: Proceedings of

Reflection 2001, The Third International Conference on Meta-level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan. LNCS 2192, AITO, Springer-Verlag, pp.126–133

31. Capra L, Emmerich W, and Mascolo C (2003) CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. In IEEE Transactions on Software Engineering, 29(10), pp.929-945.
32. Chang E, Pnueli A, Manna Z (1994) Compositional Verification of Real-Time Systems. Proc. 9'th IEEE Symp. On Logic In Computer Science, 1994, pp. 458-465.
33. Charniak E, McDermott D (1985) Introduction to Artificial Intelligence. Addison Wesley, Reading, MA.
34. Chen F and Rosu G (2003) Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In Electronic Notes in Theoretical Computer Science 89 No. 2, Published by Elsevier Science B.V.
35. Clark K (1978) Negation as Failure, Logic and Databases. Editors: H. Gallaire, J. Minker, Plenum Press, pp. 293-322, New York
36. Clarke EM, Grumberg O, and Peled D (1999) Model Checking. MIT Press
37. Clavel M, Durán FJ, Eker S, Lincoln Martí-Oliet N, Meseguer J, and Quesada JF (1999) The Maude System. In Proceedings of the 10th International Conference on Rewriting Techniques
38. Cohen D, Feather M, Narayanswamy K, and Fickas S (1997) Automatic Monitoring of Software Requirements. In Proc. of the 19th Int. Conf. on Software Engineering.
39. Cohen G, Chase J, and Kaminsky D (1998) Automatic Program Transformation with JOIE. In Proceedings of the 1998 USENIX Annual Technical Symposium.
40. Cohen S (1999) Jtrek, Developed by Compaq. http://www.compaq.com/java/download/jtrek.
41. Console L, Dupre DT, Torasso P (1991) On the Relationship between Abduction and Deduction. Journal of Logic and Computation, 1(5).
42. Console et al. (2002) Local Reasoning and Knowledge Compilation for Efficient Temporal Abduction. IEEE Trans. on Knowledge & Data Engineering 14(6): 1230-1248.

43. Corbett J, Dwyer M, Hatcliff J, and Robby (2001) Expressing Checkable Properties of Dynamic Systems: The Bandera Specification Language. KSU CIS Technical Report 2001-04.

44. Corbett J, Dwyer M, Hatcliff J, Pasareanu C, Robby, Laubach S, and Zheng H (2000). Bandera: Extracting Finite-state Models from Java Source Code. In Proceedings of the 22nd International Conference on Software Engineering, June.

45. Cousot P, and Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. Symp. of Principles of Programming Languages. ACM Press.

46. Damianou N, Dulay N, Lupu EC, and Sloman MS (2001) The Ponder Policy Specification Language. Presented at Policy 2001: Workshop on Policies for Distributed Systems and Networks, Bristol, UK.

47. d'Amorim M and Havelund K (2005) Event-based runtime verification of java programs. In Proceedings of the Third international Workshop on Dynamic Analysis (St. Louis, Missouri, May 17 - 17, 2005). WODA '05. ACM Press, New York, NY, pp.1-7

48. Dardenne A, van Lamsweerde A, and Fickas S (1993) Goal-Directed Requirements Acquisition. Science of Computer Programming, 20, pp. 3-50.

49. David PC, Ledoux T, and Bouraqadi-Saadani NMN (2001) Two-step weaving with reflection using AspectJ. In OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems.

50. Dawid AP (1992) Applications of a general propagation algorithm for probabilistic expert systems. Stat. Comput., vol. 2, pp. 25-36

51. Dechter R, Meiri I, Pearl J (1991) Temporal constraint networks. Artif. Intell. 49, 1-3, pp. 61-95

52. De Kleer J., and Williams B.C. (1987) Diagnosing Multiple Faults. Artif. Itell.32(1): 97-130.

53. Denecker M. et al. (1992) Temporal reasoning with abductive event calculus. 10th ECAI.

54. Denning D (1987) An Intrusion-Detection Model. IEEE Transactions on Software Engineering, Vol. SE-13, No. 2, pp. 222-232

55. Desai N (2003) Intrusion Prevention Systems: the Next Step in the Evolution of IDS. SecurityFocus, http://www.securityfocus.com/infocus/1670

56. Dingwall-Smith A, and Finkelstein A (2002) From Requirements to Monitors by Way of Aspects. Proc. of 1st Int. Conf. on Aspect-Oriented Software Development.

57. Drusinsky D (2000) The Temporal Rover and the ATG Rover. In K. Havelund, J. Penix, and W. Visser, editors, SPIN Model Checking and Software Verification, volume 1885 of LNCS, pp. 323–330, Springer.

58. Emmerich W (2000) Software Engineering and Middleware: A Roadmap. In The Future of Software Engineering - 22nd Int. Conf. on Software Engineering (ICSE2000), pages 117–129, ACM Press.

59. Eshgi K, Kowalski R (1988) Abduction through deduction. Technical report, Imperial College of Science and Technology, Department of Computing

60. Feather M, and Fickas S (1995) Requirements Monitoring in Dynamic Environments. In Proc. of Int. Conf. on Requirements Engineering.

61. Feather MS, Fickas S, van Lamsweerde A, and Ponsard C (1998) Reconciling System Requirements and Runtime Behaviour. Proc. of 9th Int. Work. on Software Specification & Design.

62. Firesmith D (2003) Engineering Security Requirements. In Journal of Object Technology, vol. 2, no. 1, January-February 2003, pp. 53-68. http://www.jot.fm/issues/issue_2003_01/column6

63. Gale D. (2007) Linear programming and the simplex method. Notices of the AMS, 54(3):364–369

64. Giannakopoulou D and Havelund K (2001) Automata-Based Verification of Temporal Properties on Running Programs. In Proceedings of International Conference on Automated Software Engineering (ASE'01), pp. 412–416. ENTCS. Coronado Island, California.

65. Goldberg A, and Havelund K (2003) Instrumentation of Java Bytecode for Runtime Analysis. In Proc. Formal Techniques for Java-like Programs, volume 408 of Technical Reports from ETH Zurich, Switzerland. ETH Zurich.

66. Grastien A, Cordier M, Largouët C (2005) Incremental Diagnosis of Discrete-Event Systems.15th Int. Work. On Principles of Diagnosis (DX05)

67. Grimes R (2004) Authenticode. Microsoft Corporation TechNet, Microsoft Authenticode Reference Guide.

68. Gritzalis S, Katsikas S and Gritzalis D (2003) Computer Network Security. (In Greek) Papasotiriou Publishers

69. Gurevich Y (1993) Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Saloma, editors, Current Trends in Theoretical Computer Science, pp. 266-292. World Scientific.

70. Gurevich Y, Schulte W, Campbell C, and Grieskamp W (2001) The Abstract State Machine Language. The Abstract State Machine Language, Microsoft Corporation.

71. Hatcliff J, and Dwyer, M, (2001) Using the Bandera tool set to model-check properties of concurrent Java software. In CONCUR 2001, LNCS 2154, pages 39–58.

72. Havelund K, and Rosu G (2001) Monitoring Java Programs with Java PathExplorer. In Proceedings of the 1st International Workshop on Runtime Verification (RV'01) [1], pp. 97–114.

73. Havelund K, and Rosu G (2001) Monitoring Programs using Rewriting. In Proceedings of International Conference on Automated Software Engineering (ASE'01), pp. 135–143. Institute of Electrical and Electronics Engineers. Coronado Island, California.

74. Havelund K and Rosu G (2002) Synthesizing Monitors for Safety Properties. In Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), volume 2280 of LNCS, pp. 342–356. Springer.

75. Havelund K and Roşu G (2004) An Overview of the Runtime Verification Tool Java PathExplorer. Methods Syst. Des. 24, pp.189-215

76. Heckerman, D. (1998) A tutorial on learning with Bayesian networks, in M.I. Jordan, ed., Learning in Graphical Models, Kluwer, Dordrecht, Netherlands.

77. Hirschfeld R, and Kawamura K (2004) Dynamic service adaptation. In Proceedings of the Fourth IEEE International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (with ICDCS'04), Tokyo, Japan.

78. Hoare C (1985-2004) Communicating Sequential Processes. Electronic version of Communicating Sequential Processes, first published in 1985 by Prentice Hall International, http://www.usingcsp.com/cspbook.pdf.

79. Hobbs JR, Stickel M, Martin P, Edwards D (1988) Interpretation as abduction. In 26th Annual Meeting of the Association for Computational Linguistics, Buffalo, NY, pp.95-103

80. Holzmann GJ, and Smith MH (1997) The model checker SPIN. IEEE trans. SE, 23(5), pp. 279–295.

81. Jahanian J, Rajkumar R, and Raju S (1994) Runtime monitoring of timing constraints in distributed real-time systems. Technical Report CSE-TR 212-94, University of Michigan

82. Janicke H, Siewe K, Jones F, Cau A, and Zedan H (2005) Analysis and Run-time Verification of Dynamic Security Policies. AAMAS 05 workshop on Defence Applications of Multi-Agent Systems, Utrecht.

83. Jensen FV, Lauritzen SL, Olesen KG (1990) Bayesian updating in causal probabilistic networks by local computations. Computational Statistics Quarterly, 4:269-282

84. Jensen, F. (1996). An Introduction to Bayesian Networks. Springer.

85. Jensen, F. and Nielsen, D. (1996) Bayesian Networks and Decision Graphs – Second Edition, Information and Science Statistics, Springer.

86. Kakas A, Mancarella P (1990) Generalised stable models: a semantics for abduction. In Proceedings of the 9th European Conference on Artificial Intelligence, pp. 385-391

87. Kakas A, Kowalski R, Toni F (1992) Abductive logic programming. Department of Computer Science, University of Cyprus, Nicosia, and Imperial College of Science, Technology and Medicine, London

88. Kaler C, and Nadalin A (editors) (2005) Web Services Security Policy Language (WS-SecurityPolicy). http://www-128.ibm.com/developerworks/library/specification/ws-secpol/.

89. Karaorman M and Freeman J (2004). jMonitor: Java runtime event specification and monitoring library. Proceedings of 4th Workshop on Run-time Verification, 2004

90. Kiczales G and Lamping J (1997) Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, Proceedings European Conference on Object-Oriented Programming, volume 1241, pages 220-242. Springer-Verlag, Berlin, Heidelberg, and New York

91. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, and Griswold WG (2001). An Overview of AspectJ. In Proceedings of the 15th European Conference on Object-Oriented Programming, pages 327–353. Springer-Verlag.

92. Kim JH, Pearl J (1983) A computational model for causal and diagnostic reasoning in inference systems. In Proceedings of the 8th International Joint Conference on Artificial Intelligence, pp. 190 – 193

93. Kim M, Kannan S, Lee I, Sokolsky O, and Viswanathan M (2001) Java-mac: a run-time assurance tool for java programs. In Electronic Notes in Theoretical Computer Science, volume 55. Elsevier Science Publishers

94. Knight K (1989) Unification: a multidisciplinary survey. ACM Computing Surveys, 21(1):93-124. DOI= http://doi.acm.org/10.1145/62029.62030.

95. Ko C, Ruschitzka M and Levitt K (1997) Execution monitoring of security-critical programs in distributed systems: A specication-based approach. In Proceedings of the IEEE Symposium on Security and Privacy, pp. 175-187, Oakland, CA, USA

96. Koks D, and Chall S, (2005) An introduction to Bayesian and Dempster-Shafer Data Fusion. DSTO–TR–1436. DSTO Systems Sciences Laboratory,

97. Konolige K (1990) Closure + minimization implies abduction. In PRICAI-90, Nagoya, Japan

98. Lauritzen SL, Spiegelhalter DJ (1990) Local computations with probabilities on graphical structures and their application to expert systems. In Readings in Uncertain Reasoning, G. Shafer and J. Pearl, Eds. Morgan Kaufmann Publishers, San Francisco, CA, 415-448

99. Lazarevic A., Kumar V., Srivastava J. (2005) Intrusion detection: a survey, In Managing cyber-threats: issues approaches & challenges, Springer.

100. Leavens G, Baker A, and Ruby C (2003) Preliminary Design of JML: A Behavioural Interface Specification Language for Java. Technical Report 9806u, Iowa State University, Department of Computer Science, http://www.jmlspecs.org/.

101. Lee D and Yannakakis M (1996) Principles and Methods of Testing Finite State Machines – A Survey. Proceedings of the IEEE, vol. 84, n. 8, August, p. 1090-1123

102. Lee I, Kannan S, Kim M, Sokolsky O, and Viswanathan. M (1999) Runtime Assurance Based on Formal Specifications. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications.

103. Levesque H (1989) A knowledge-level account of abduction. In Proceedings of the 11th International Joint Conference on Artificial Intelligence, pp. 1061-1067

104. Ligatti J, Bauer L, and Walker D (2005) Edit Automata: Enforcement Mechanisms for Run-time Security Policies. International Journal of Information Security, 4(1–2).

105. Lindholm T, and Yellin F (1996) The Java Virtual Machine specification. Web document at URL http://www.javasoft.com/docs/books/vmspec/html/VMSpecTOC.doc.html, Sun Microsystems.

106. Lowe G (1995) An attack on the Needham-Schroeder public-key authentication protocol. Inf. Process. Lett. 56, 3, pp.131-133. DOI: http://dx.doi.org/10.1016/0020-0190(95)00144-2

107. Ludwig H, Keller A, Dan A, King RP, and Franck R (2003) Web Service Level Agreement (WSLA) Language Specification. Version 1.0, IBM Corporation, http://www.research.ibm.com/wsla

108. Lutz R (2000) Software Engineering for Safety: A Roadmap. Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 4-11, ACM

109. Mahbub K and Spanoudakis G (2004) A Framework for Requirements Monitoring of Service Based Systems. In Proceedings of the 2nd International Conference on Service Oriented Computing, NY, USA

110. Mascolo C, Capra L, Zachariadis S, and Emmerich W (2002) XMIDDLE: A Data-Sharing Middleware for Mobile Computing. In International Journal on Wireless Personal Communications, 21(1), pp.77-103. Kluwer Academic Publisher.

111. McGraw G, and Felten E (1999) Securing JAVA. Getting Down to Business with Mobile Code. Chapter 3, published by John Wiley & Sons, Inc., Securing Java: Getting Down to Business with Mobile Code.

112. Meyer B (2000) Object-Oriented Software Construction. 2nd edition. Prentice Hall, Upper Saddle River, New Jersey.

113. Mohnen M (2002) A graph-free approach to data-flow analysis. In Proc. 11th CC, pages 46–61, Grenoble, France. Springer.

114. Mok AK and Liu G (1997) Efficient run-time monitoring of timing constraints. In Real-Time Technology and Applications Symposium

115. Moller M, Bartetzko D, Fischer C, and Wehrheim H (2001) Jass - java with assertions. In Electronic Notes in Theoretical Computer Science, volume 55. Elsevier Science Publishers.

116. Moszkowski B (1996) The programming language Tempura. Journal of Symbolic Computation, 22(5/6):730—733.

117. Naldurg P, Sen K, and Thati P (2004) A Temporal Logic Based Framework to Intrusion Detection. In Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2004)

118. Necula G (1997) Proof-Carrying Code. In Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97), Paris, France.

119. Nelson S and Pecheur C (2002) V&V for advanced systems at NASA. TASK NO: 10 TA-5.3.3 (WBS 1.4.4.5.3), prepared for Northrop Grumman Corp

120. Ng HT, Mooney RJ (1990) On the role of coherence in abductive explanation. In Proceedings of the 8th National Conference on Artificial Intelligence, pp. 337-342

121. Pan R., Peng Y., and Ding Z. (2006) Belief Update in Bayesian Networks Using Uncertain Evidence. In Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '06). IEEE Computer Society, Washington, DC, USA, 441-444. DOI=10.1109/ICTAI.2006.39 http://dx.doi.org/10.1109/ICTAI.2006.39

122. Paul G (1993) Approaches to Abductive Reasoning: an overview. Artificial Intelligence, 7, pp. 109-152

123. Pearl J. (1988): "Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference". Morgan Kauffman, San Mateo, CA

124. Pearl J (1994) Bayesian networks. Tech. rep. R-216, Computer Science Department, University of California, Los Angeles

125. Pearl J (1995) Bayesian Networks. In Handbook of Brain Theory and Neural Networks, MIT Press

126. Pearl J and Russel S (2000) Bayesian networks. UCLA Cognitive Systems Laboratory, Technical Report (R-278)

127. Peirce CS (1931-1958) Collected Papers of Charles Sanders Peirce (eds. C. Hartshore et al.). Harvard University Press

128. Pencolé Y and Cordier M (2005) A formal framework for the decentralised diagnosis of large scale discrete event systems & its application to telecommunication networks. Artif. Intell. 164, pp.121-180.

129. Pnueli A (1977) The Temporal Logic of Programs. In Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, pp. 46–77.

130. Poole D (1989) Explanation and prediction: an architecture for default and abductive reasoning. Computational Intelligence, 5(2), pp.97-110

131. Poole D (1991) Representing Bayesian networks within probabilistic Horn abduction. In Proceedings of the Seventh Conference on Uncertainty in Artificial

intelligence (Los Angeles, California, United States). B. D. D'Ambrosio, P. Smets, and P. P. Bonissone, Eds. Morgan Kaufmann Publishers, San Francisco, CA, pp. 271-278

132. Poole D. (1993): "Probabilistic Horn abduction and Bayesian networks". Artif. Intell. 64(1), pp. 81-129

133. Porras PA and Neumann PG (1997) EMERALD: Event monitoring enabling responses to anomalous live disturbances. In Proc. 20th NISTNCSC National Information Systems Security Conference, pp. 353 – 365

134. Ragsdale D, Carver CA, Humphries J. and Pooch U (2000) Adaptation techniques for intrusion detection and intrusion response system. Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics at Nashville, Tennessee, pp. 2344 - 2349

135. Ray O and Kakas A (2006) ProLogICA: a practical system for Abductive Logic Programming. Proc. 11th Int. Workshop on Non-monotonic Reasoning, pp304-312

136. Reiter R. (1987) A theory of diagnosis from first principles, Artif. Intell. 32(1): 57-96.

137. Robinson W (2002) Monitoring Software Requirements using Instrumented Code. In proceedings of the Hawaii Int. Conf. on Systems Sciences

138. Russo A, Miller A, Nuseibeh B, and Kramer J (2002) An Abductive Approach for Analysing Event-Based Requirements Specifications. Presented at 18th Int. Conf. on Logic Programming (ICLP), Copenhagen, Denmark.

139. Sampath M, Sengupta R, Lafortune S, Sinnamohideen K and Teneketzis DC (1996) Failure diagnosis using discrete-event models. IEEE Trans. on Control Systems Technology, 4(2), pp.105-124

140. Santos E Jr (1996) Unifying time and uncertainty for diagnosis. Journal of Experimental and Theoretical Artificial Intelligence, 8 pp. 75-94

141. Savage S, Burrows M, Nelson G, Sobalvarro P, and Anderson T (1997) Eraser: A dynamic data race detector for multithreaded programs. In ACM Trans. on Computer Systems, 15(4).

142. Schlimmer J (editor) (2006) Web Services Policy Framework (WS-Policy Framework), http://www.ibm.com/developerworks/library/specification/ws-polfram/.

143. Schneider FB (1998) Enforceable Security Policies. Cornell University Technical Report TR98-1664.

144. Sekar R, Venkatakrishnan VN, Basu S, Bhatkar S, and DuVarney D (2003) Model - Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. ACM Symposium on Operating Systems Principles. (SOSP'03; Bolton Landing, New York).

145. Sen K, and Rosu G (2003) Generating Optimal Monitors for Extended Regular Expressions. In Proceedings of the 3rd International Workshop on Runtime Verification (RV'03) [1], pp. 162–181

146. SERENITY, System Engineering for Security & Dependability, http://www.serenity-project.org/ (last visited at 06/04/2011).

147. Shafer G (1975) A Mathematical Theory of Evidence. Princeton University Press.

148. Shanahan M (1989) Prediction is deduction but explanation is abduction. In Proceedings of the International Joint Conference on Artificial Intelligence, pp. 1055 - 1060, San Mateo, CA

149. Shanahan M (1999) The Event Calculus Explained. In Artificial Intelligence Today, LNAI 1600:409-430

150. Shanahan M, (2000) Abductive Event Calculus Planner. J. Logic Programming 44: 207-239.

151. Simon, C., and Weber, P. (2006) Bayesian Networks Implementation of the Dempster Shafer Theory to Model Reliability Uncertainty. ARES 2006: 788-793

152. Simon, C., Weber, P., and Levrat, E. (2007) Bayesian Networks and Evidence Theory to Model Complex Systems Reliability. JCP 2(1): 33-43.

153. Spanoudakis G, Mahbub K (2006) Non intrusive monitoring of service based systems. Int. J. of Cooperative Inform. Systems, 15(3):325–358

154. Spanoudakis G, Mana A, and Kokolakis S (Editors) (2009) Security and Dependability for Ambient Intelligence, Information Security Series, Springer.

155. Spanoudakis G, Kloukinas C and Mahbub K (2009) The Runtime Monitoring Framework of SERENITY. In Security and Dependability for Ambient Intelligence, Information Security Series, Springer, pp. 213-237

156. Stickel M (1988) A Prolog-like inference system for computing minimum-cost abductive explanations in natural-language interpretation. In International Computer Science Conference, Hong Kong, pp. 343-350

157. Sun Microsystems (2003) Securing Web Services - Concepts, Standards, and Requirements. White Paper

158. Tardo J, and Valente L (1996) Mobile Agent Security and Telescript. In Proceedings of IEEE COMPCON '96, Santa Clara, California, pp. 58-63, February 1996, IEEE Computer Society Press.

159. Tarr PL, Ossher H, Harrison WH, and Sutton SM Jr. (1999) "N degrees of separation: Multi-dimensional separation of concerns. In International Conference on Software Engineering, pages 107–119.

160. Thane H (2000) Design for deterministic monitoring of distributed real-time systems. Technical report, Malardalen Real-Time Research Centre

161. Tripakis S (2002) Fault diagnosis for timed automata. In Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02), vol. 2469 of LNCS, pp. 205–224, Springer

162. Tsigkritis T, Spanoudakis G (2008) Diagnosing Runtime Violations of Secrity and Dependability Properties. In Proc of 20th Int. Conference in Software Engineering and Knowledge Engineering, pp. 661-666.

163. Tsigkritis T, Spanoudakis G (2008) A temporal abductive diagnosis process fo runtime properties violations. ECAU 2008 Workshop on Explanation Aware Computing.

164. Tsigkritis T, Spanoudakis G, Kloukinas C, and Lorenzoli D (2009) Diagnosis and Threat detection capabilities of the SERENITY Runtime Framework, In *Security and Dependability for Ambient Intelligence*, Information Security Series, Springer, pp. 239-272

165. van Lamsweerde A. (1996) Divergent Views in Goal-Driven Requirements Engineering. In proc. Viewpoints '96 – ACM SIGSOFT Workshop of Viewpoints in Software Development, October

166. van Lamsweerde, A. (2004) Elaborating Security Requirements by Construction of Intentional Anti-Models. In Proceedings of ICSE'04, 26th International Conference on Software Engineering, Edinburgh, May. 2004, ACM-IEEE, pp. 148-157.

167. VeriSign (2005) VeriSign Code Signing for Netscape Object Signing, in Business Guide. Chapters 2 & 3, VeriSign, http://www.verisign.com/static/030997.pdf.

168. Visser W, Havelund K, Brat G, and Park SJ (2000) Model Checking Programs. In Proceedings of ASE-2000: The 15th IEEE Conference on Automated Software Engineering. IEEE CS Press. Grenoble, France.

169. Wagelaar D (2004) Towards a context-driven development framework for ambient intelligence. In Proceedings of the Fourth IEEE International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (withICDCS'04), Tokyo, Japan.

170. Yang Z, Cheng BH, Stirewalt RE, Sowell J, Sadjadi SM, and McKinley PK (2002) An aspect-oriented approach to dynamic adaptation. In Proceedings of the ACM SIGSOFT Workshop On Self-healing Software (WOSS'02).

171. Yellin F (1996) Low-level security in Java. Web document at URL http://www.javasoft.com/sfaq/verifier.html, Sun Microsystems.

Appendix A: Location Based Access Control System Monitoring Theory

LBACS.R1. ∀t1∈Time, ∃t2∈Time, ∀_LServerId ∈ LocationServers,
∀_ACServerId ∈ AccessControlServers, ∀_deviceId∈Devices,
∀_source.

Happens(e(_Id1,_ACServerId,_LServerId,REQ-B,locationRequest
(_deviceId),_source), t1, R(t1,t1)) ⇒
Happens(e(_Id2,_LServerId,_ACServerId,REQ-A,locationResponse
(_deviceId),_source), t2, R(t1+1,t1+3000))

LBACS.R2. ∀t1∈Time, ∃t2∈Time, ∀_LServerId ∈ LocationServers,
∀_ACServerId ∈ AccessControlServers, ∀_deviceId∈Devices,
∀_receiver1∈Sensors, ∀_source1, ∀_source2.

Happens(e(_Id1,_ACServerId,_LServerId,REQ-A,locationRequest
(_deviceId),_source1), t1, R(t1,t1)) ∧
¬**Happens**(e(_Id1,_ACServerId,_LServerId,REQ-A,locationRequest
(_deviceId),_source1), t2, R(0,t1-1)) ⇒
Happens(e(_Id2,_deviceId,_receiver1,REQ-A,signal(_deviceId),
_source2), t3, R(t1-2000,t1-1))

LBACS.R3. ∀t1∈Time, ∃t2∈Time, ∀_deviceId∈Devices,
∀_receiver1∈Sensors, ∀_source1.

Happens(e(_Id1,_deviceId,_receiver1,REQ-A, signal(_deviceId),
_source1), t1, R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId,_receiver1,REQ-A,signal(_deviceId),
_source1), t2, R(t1+1,t1+2000))

LBACS.R4. ∀t1 ∈ Time, ∃t2 ∈ Time, ∀t3 ∈ Time, ∀_deviceId ∈ Devices,
∀_userId ∈ Users, ∀_source1, ∀_source2.

Happens(e(_Id1,intranetRouter,_deviceId,REQ-B,
loginAcknowledgment(_userId),_source1),
t1,R(t1,t1)) ∧
Happens(e(_Id2,internetRouter,_deviceId,REQ-B,
loginAcknowledgment(_userId),_source2),
t2,R(t1+1,t2)) ⇒
Happens(e(_Id3,intranetRouter,_deviceId,REQ-B,
logoutAcknowledgment(_userId)

_source1),t3,R(t1+1,t2-1))

LBACS.A1. ∀t1∈Time, ∃t2∈Time, ∀_deviceId∈Devices, ∀_receiver, ∀_sender, ∀_source.

Happens(e(_Id1,_sender,_receiver,REQ-A,operableInPremises (_deviceId),_source),t1,R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId,_receiver,REQ-A,signal(_deviceId), _source),t2,R(t1-2000,t1))

LBACS.A2. ∀t1 ∈ Time, ∃t2 ∈ Time, ∀_deviceId ∈ Devices, ∀_resourceId ∈ Resources, ∀_ACServerId ∈ AccessControlServers, ∀_sender, ∀_receiver, ∀_source.

Happens(e(_Id1,_sender,_receiver,REQ-A,operableInPremises (_deviceId),_source),t1,R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId_ACServerId,REQ-A,accessTo (_deviceId, _resourceId), _source), t2, R(t1-2000,t1))

LBACS.A3. ∀t1 ∈ Time, ∃t2 ∈ Time, ∀_deviceId ∈ Devices, ∀_resourceId ∈ Resources, ∀_receiver1 ∈ AccessControlServers, ∀_receiver2 ∈ Sensors, ∀_source1, ∀_source2.

Happens(e(_Id1,_deviceId,_receiver1,REQ-A,accessTo(_deviceId, _resourceId),_source1), t1, R(t1,t1))⇒
Happens(e(_Id2,_deviceId,_receiver2,REQ-A,signal(_deviceId), _source2), t2, R(t1-2000,t1))

LBACS.A4. ∀t1 ∈ Time, ∃t2 ∈ Time, ∀_deviceId ∈ Devices, ∀_resourceId ∈ Resources, ∀_ACServerId ∈ AccessControlServers, ∀_receiver ∈ LocationServers, ∀_source1, ∀_source2.

Happens(e(_Id1,_ACServerId,_receiver,REQ-B,locationRequest (_deviceId), _source1), t1, R(t1,t1))⇒
Happens(e(_Id2,_deviceId,_ACServerId,REQ-A,accessTo (_deviceId,_resourceId),_source2),t2,R(t1-5000,t1))

LBACS.A5. ∀t1 ∈ Time, ∃t2 ∈ Time, ∀_deviceId ∈ Devices, ∀_resourceId ∈ Resources, ∀_ACServerId ∈ AccessControlServers, ∀_source.

Happens(e(_Id1,_ACServerId,_deviceId,REQ-B,accessToResponse (_deviceId,_resourceId),_source), t1, R(t1,t1))⇒
Happens(e(_Id2,_deviceId,_ACServerId,REQ-A,accessTo

(_deviceId,_resourceId),_source),t2,R(t1-5000,t1))

LBACS.A6. ∀t1 ∈ Time, ∃t2 ∈ Time, ∀_deviceId ∈ Devices, ∀_LServerId ∈ LocationServers, ∀_ACServerId ∈ AccessControlServers, ∀_source.

Happens(e(_Id1,_LServerId,_LServerId,REQ-B,locationResponse (_deviceId),_source), t1, R(t1,t1))⇒
Happens(e(_Id2,_ACServerId,_LServerId,REQ-A,locationRequest (_deviceId),_source), t2, R(t1-2000,t1))

LBACS.A7. ∀t1 ∈ Time, ∃t2 ∈ Time, ∀_deviceId ∈ Devices, ∀_ACServerId ∈ AccessControlServers, ∀_resourceId ∈ Resources, ∀_LServerId ∈ LocationServers, ∀_source1, ∀_source2.

Happens(e(_Id1,_ACServerId,_deviceId,REQ-B,accessToResponse (_deviceId,_resourceId),_source1), t1, R(t1,t1))⇒
Happens(e(_Id2,_LServerId,_ACServerId,REQ-B,locationResponse (_deviceId),_source2), t2, R(t1-1000,t1))

LBACS.A8. ∀t1 ∈ Time, ∃t2 ∈ Time, ∀_sender, ∀_receiver, ∀_deviceId ∈ Devices, ∀_ACServerId ∈ AccessControlServers, ∀_resourceId ∈ Resources, ∀_source1, ∀_source2.

Happens(e(_Id1, _sender, _receiver, REQ-B, accessControlServerIsRunning(_ACServerId), _source1), t1, R(t1,t1))⇒
Happens(e(_Id2,_ACServerId,_deviceId,REQ-B,accessToResponse (_deviceId,_resourceId),_source2),t2,R(t1-1000,t1))

LBACS.A9. ∀t1 ∈ Time, ∃t2 ∈ Time, ∀_sender, ∀_receiver, ∀_LServerId ∈ LocationServers, ∀_deviceId ∈ Devices, ∀_source1, ∀_source2.

Happens(e(_Id1, _sender, _receiver, REQ-B, locationServerIsRunning(_LServerId),_source1),t1, R(t1,t1))⇒
Happens(e(_Id2,_LServerId,_LServerId,REQ-B,locate (_deviceId),_source), t2, R(t1-1000,t1))

LBACS.A10. ∀t1 ∈ Time, ∃t2 ∈ Time, ∀_sender, ∀_receiver, ∀_LServerId ∈ LocationServers, ∀_ACServerId ∈ AccessControlServers, ∀_deviceId ∈ Devices, ∀_source1, ∀_source2.

Happens(e(_Id1, _sender, _receiver, REQ-B,

locationServerIsRunning(_LServerId),_source1),t1,
R(t1,t1))⇒
Happens(e(_Id2,_LServerId,_ACServerId,REQ-B,locationResponse
(_deviceId),_source2), t2, R(t1-1000,t1))

LBACSNT.A1. ∀t1∈Time,∃t2∈Time,∀_sender,∀_receiver,∀_deviceId∈Devices,
∀_routerId∈Routers, ∀_userId∈Users, ∀_source1, ∀_source2.

Happens(e(_Id1,_sender,_receiver,REQ-A,operableInPremises
(_deviceId),_source),t1,R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId,_routerId,REQ-A,login(_userId,
_deviceId,_routerId),_source),t2,R(t1-1000,t1))

LBACSNT.A2. ∀t1∈Time,∃t2∈Time,∀_deviceId∈Devices, ∀_routerId∈Routers,
∀_sensorId∈Sensors, ∀_userId∈Users, ∀_source1, ∀_source2.

Happens(e(_Id1,_deviceId,_routerId,REQ-A, login(_userId,
_deviceId,_routerId),_source1),t1,R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId,_sensorId,REQ-A, signal
(_deviceId),_source2),t2,R(t1-2000,t1))

LBACSNT.A3. ∀t1∈Time,∃t2∈Time,∀_routerId∈Routers, ∀_deviceId∈Devices,
∀_userId∈Users, ∀_source.

Happens(e(_Id1, _routerId, _deviceId, REQ-B,
loginAcknowledgment(_deviceId,_routerId),
_source),t1,R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId,_routerId,REQ-A,login(_userId,
_deviceId,_routerId),_source),t2,R(t1-5000,t1))

LBACSNT.A4. ∀t1∈Time, ∃t2∈Time, ∀_sender, ∀_receiver, ∀_routerId∈
Routers, ∀_deviceId ∈ Devices, ∀_source.

Happens(e(_Id1, _sender, _receiver, REQ-A,
routerIsRunning(_routerId),_source),t1,
R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId,_routerId,REQ-B,
loginAcknowledgment(_deviceId,_routerId),
_source),t2, R(t1-1000,t1))

LBACSNT.A5. ∀t1∈Time,∃t2∈Time,∀_sender,∀_receiver,∀_deviceId∈Devices,
∀_routerId∈Routers, ∀_userId∈Users, ∀_source1, ∀_source2.

Happens(e(_Id1,_sender,_receiver,REQ-A,operableInPremises
(_deviceId),_source),t1,R(t1,t1)) ⇒

Happens(e(_Id2,_deviceId,_routerId,REQ-A,logout(_userId,
_deviceId,_routerId),_source),t2,R(t1-1000,t1))

LBACSNT.A6. ∀t1∈Time,∃t2∈Time,∀_deviceId∈Devices, ∀_routerId∈Routers,
∀_sensorId∈Sensors, ∀_userId∈Users, ∀_source1, ∀_source2.

Happens(e(_Id1,_deviceId,_routerId,REQ-A,logout(_userId,
_deviceId,_routerId),_source1),t1,R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId,_sensorId,REQ-A, signal
(_deviceId),_source2),t2,R(t1-2000,t1))

LBACSNT.A7. ∀t1∈Time,∃t2∈Time,∀_routerId∈Routers, ∀_deviceId∈Devices,
∀_userId∈Users, ∀_source.

Happens(e(_Id1, _routerId, _deviceId, REQ-B,
logoutAcknowledgment(_deviceId,_routerId),
_source),t1,R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId,_routerId,REQ-A,login(_userId,
_deviceId,_routerId),_source),t2,R(t1-5000,t1))

LBACSNT.A8. ∀t1∈Time, ∃t2∈Time, ∀_sender, ∀_receiver, ∀_routerId∈
Routers, ∀_deviceId ∈ Devices, ∀_source.

Happens(e(_Id1, _sender, _receiver, REQ-A,
routerIsRunning(_routerId),_source),t1,
R(t1,t1)) ⇒
Happens(e(_Id2,_deviceId,_routerId,REQ-B,
logoutAcknowledgment(_deviceId,_routerId),
_source),t2, R(t1-1000,t1))