A4.D4.1 – Mechanisms for detecting potential S&D threats

N. Amalio, V. Di Giacomo, C. Kloukinas, G. Spanoudakis

Document Number	A4.D4.1
Document Title	Mechanisms for detecting potential S&D threats
Version	1.0
Status	final
Work Package	WP 4.4
Deliverable Type	Report
Contractual Date of Delivery	28/02/2008
Actual Date of Delivery	14 May 2008
Responsible Unit	CUL
Contributors	
Keyword List	
Dissemination level	PU

Change History

Version	Date	Status	Author (Unit)	Description
0.1	19/12/2007	Draft	Nuno Amalio (CUL)	First version of document structure. Initial draft of sections 1 and 3.
0.2	8/01/2008	Draft	Valentina Di Giacomo (ENG)	Inputs on modelling aspects
0.3	18/01/2008	Draft	Nuno Amalio (CUL)	Drafted Section 3 (threat detection mechanisms, an overview).
0.4	22/01/2008	Draft	George Spanoudakis (CUL)	Revised introduction of section 3, and terminology (section 3.1). Introduced section 3.2 (overview of detection mechanisms of SERENITY).
0.5	25/01/2008	Draft	Nuno Amalio (CUL)	Re-structured section 3. Introduced section 7 for case study and examples.
0.6	18/4/2008	Draft	Nuno Amalio (CUL)	Re-structured document. Added sections 2, 3, 4, and 7.
0.7	18/4/2008	Draft	Christos Kloukinas (CUL)	Initial draft of section 6
0.8	21/4/2008	Draft	George Spanoudakis (CUL)	Revised draft of Sections 4 and 6, Executive summary
0.9	28/4/2008	Draft	Nuno Amalio (CUL)	Revision of Section 4
0.10	29/4/08	Draft	Christos Kloukinas (CUL)	Revision of Sections 2 and 4
0.11	30/4/2008	Draft	Nuno Amalio (CUL	Extension of Section 7
0.12	7/5/2008	Draft submitted for quality review	George Spanoudakis (CUL)	Final editing
1.0	14/5/2008	Final	Nuno Amalio (CUL)	After quality review

Executive Summary

This document presents the mechanisms for detecting threats at run-time in the context of the SERENITY framework. These mechanisms are being implemented and in their final version will constitute an integrated part of the SERENITY framework serving two main functional objectives within it. The first objective is to support the automatic generation of monitoring policies and attack signatures that can be used for the detection of runtime violations of S&D properties security. The second objective is to estimate the likelihood of potential violations of S&D properties (aka S&D threats).

The generation of attack signatures starts from the specification of a *security objective* for a specific *system asset*. Starting from a pair of a security objective and an asset, the *attack signature generator* identifies the possible *system operations* whose execution can potentially violate the required security property that underpins the goal. Subsequently, it creates initial *monitoring specifications* using pre-specified monitoring rule templates that cover basic security properties. These initial monitoring specifications might not include directly monitorable events and/or make references to system states which cannot be directly monitored at runtime. Hence, it is necessary to be transformed into *attack signatures* that can be monitored merely on the basis of runtime events received by the monitor during the operation of a system. This transformation is based on *planning* that uses an abductive reasoning procedure and a set of *assumptions* (EC formulas) that indicate how concrete events that can be monitored during the operation of a system can set and modify system states and/or generate other non directly observable events.

If the generation of monitorable attack signatures is succesfull then the usets of the SERENITY monitoring framework can turn them into *monitoring policies* and use them to detect *security threats* for the system being monitored. It should be noted, however, although the mechanisms that we have presented in this report support the generation of attack signatures from security objectives and assets there is no guarantee that the specification of a security objective for a specific system asset can always lead to the generation of monitorable attack signatures.

The detection of security threats at runtime is based on the basic monitoring capability of the SERENITY monitoring framework. More specifically, as soon as some runtime event instantiates a security monitoring rule and can, therefore, possibly lead to a violation of this rule, the event constitutes a security threat. To enable, however, the users of the SERENITY monitoring framework concentrate on security threats which are more likely to lead to a violation of security property in some future state of the system, the framework should calculate the likelihood of a violation given the current state of a system. The computation of this likelihood is the second main objective of the mechanisms described in this report and the computation of this likelihood is based on the *Dempster Shafer* theory of evidence.

Table of Contents

1. Introduction

This document presents the mechanisms for detecting threats in the context of SERENITY framework. These mechanisms serve the following functional objectives within the framework:

— They support the automatic generation of attack signatures that can be stored in S&D patterns as security monitoring rules and used for the detection of security threats at runtime. The generation of attack signatures starts from the specification of a *security objective* for a specific *system asset* that is defined in the class model of an S&D pattern. Starting from a pair of a security objective and an asset, an *attack signature generator* identifies generic monitoring templates for the objective and possible *system operations* whose execution can potentially violate the objective. Subsequently, it creates initial *attack signatures* by instantiating the identified monitoring templates for the objective using the system operations. These initial attack signatures might not include monitorable events or make references to system states which cannot be directly monitored at runtime. Hence, it is necessary to be transformed into attack signatures that can be monitored. This transformation is carried out by an abductive reasoning procedure using *assumptions* about the system that indicate how concrete events that can be monitored during the operation of a system can set and modify system states and/or generate other non directly observable events. If the generation of monitorable attack signatures is succesfull then the users of the SERENITY monitoring framework can give them the status of *security monitoring rules* and store them S&D patterns to detect *security violations* and *threats* for the system using the mechanisms described below.

— They support the detection of security threats (i.e., potential violations of security monitoring rules) at runtime by extending the basic monitoring capability of the SERENITY monitoring framework. More specifically, as soon as some runtime event instantiates a security monitoring rule and can, therefore, possibly lead to a violation of this rule, the event constitutes a security threat. To enable, however, the users of the SERENITY monitoring framework concentrate on security threats which are more likely to lead to a violation of security monitoring rule, the framework calculates the likelihood of a violation given the current state of a system. The calculation of likelihood is based on the *Dempster Shafer* theory of evidence.

The rest of this report is structured as follows. In Section 2, we provide an overview of the overall approach for the detection of threats. In Sections 3 and 4, we discuss the generation of attack signatures. In Section 5, we discuss the mechanisms for the estimation of the likelihood of threats. In Section 6, we provide an overview of the relation of the mehanisms described in this report with the runtime and development framework of SERENITY. Finally, in Section 7 we provide an overview of related work and, in Section 8, we summarise our approach.

2. An overview of our threat detection approach

Run-time threat detection is aimed at detecting potential violations of S&D properties, before these violations actually occur, and estimating their likelihood. Threat detection is the responsibility of SERENITY's overall monitoring framework (shortly referred to as "monitor" in the rest of this report) and the mechanisms supporting it are being implemented as part of this framework [2].

This section provides an overview of our approach for threat detection that is introduced based on a motivating example.

2.1. Motivating Example

To appreciate the notion of threats as seen in SERENITY, consider an example drawn from the e-healthcare system that is described in [15][16]. This e-healthcare system supports the monitoring of patients with critical medical conditions and the provision of assistance and medication to them based on smart-item technology.

In this system, patients who have been discharged from hospitals with potentially threatening medical conditions can use an *e-health terminal* (EHT) – that is an e-health application installed on their PDAs – to contact an *emergency response centre* (ERC) for medical assistance and fast ordering of medication. To guarantee the availability of medical advice to patients, one of the workflow level S&D patterns for this system suggests the search for an alternative doctor in cases where following a health threatening incident for a given patient, the doctor who is normally associated with the particular patient is not available. Consider also that in this system, doctors have access the medical data of patients through the execution of specific system operations but subject to the satisfaction of the requirements shown in Table 1.

R1	A patient's medical file may be seen by the doctor of the patient only.
R2	All doctors may see partial pieces of information belonging to a patient's medical file in a way that preserves requirement R1.

Table 1. Examples of requirements for the e-health care system (based on [15]).

The first of these requirements (R1) states that the complete medical file of a patient may be seen only by the doctor of the patient. This requirement is a confidentiality requirement whose aim is to protect the privacy of patients.

In cases, however, where an alternative doctor is identified and given responsibility for handling a patient incident by ERC according to the alternative doctor search pattern there is a potential for violation of R1. More specifically, suppose that the patient in question is Mr Anderson, the doctor of this patient is Dr Jones, and in a specific incident requiring medical help for Mr Anderson, another doctor, namely Dr. Smith, who is not Anderson's doctor is identified and appointed for handling the case since Dr. Jones is not available (Figure 1). In this scenatio, although Dr Jones would have access to the full medical file of Mr. Anderson, Dr Smith should not enjoy the same access level as he is not Anderson's doctor. Note, however, that due to requirement *R2*, the system will have operations providing access to parts of a patient's medical file which in the particular scenario may conflict with *R1*.

A scenario illustrating how *R2* may conflict with *R1* is shown in Figure 1. In this scenario, Dr Smith can infer all the information in Anderson's medical file by obtaining gradually partial information about Anderson which enables him infer other information in Anderson’s file without having accessing it directly. This indirect way of accessing confidential data in this case is an instance of the so-called *inference attacks* [17].

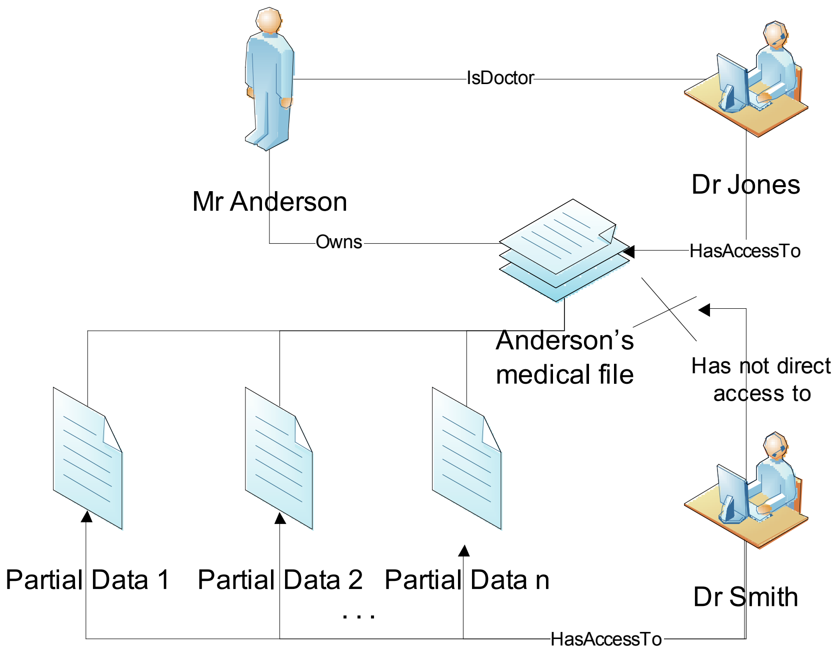

Figure 1. A partial data inference scenario in the e-health care system.

In this example, whilst it is relatively easy to specify a monitoring rule for detecting runtime behaviour that would violate directly requirement R1 (this rule could simply state that only the doctor of a patient has the authorisation to execute the operation which retrieves the whole medical file of the patient), it is relatively difficult to do the same for the latter requirement. This is because the specification of a rule enabling the monitoring of R2 requires the identification of: (a) the combinations of the partial medical data that would reveal confidential information to an unauthorised doctor, and (b) the different possible ways of getting access to these medical data within the system so as to monitor them.

Furthermore, when a rule for R2 (and R1 for the same matter) is specified, it is necessary for the monitor to have a means of assessing how “close” the system is to an inference attack at each state of its operation. This, however, would require looking at the set of the partial data that an unauthorised doctor has got up to a certain point and the likelihood of getting the remaining data which are required for the inference.

Our approach to threat detection supports these two activities (i.e., the specification of complex monitoring rules and the evaluation of the likelihood of potential violations of rules) as we describe in the following.

2.2. Stages of threat detection

As discussed earlier, the threat detection mechanisms that we have designed for SERENITY support: (a) the off-line, automated construction of security monitoring rules for an S&D Pattern, based on its constituent components/parameters and the S&D Properties it guarantees, and (b) the continuous evaluation of the current level of threat for an S&D Pattern based on the collection of run-time information on the likelihood of the occurrence (or non-occurrence) of the events which appear in these rules and ways to update this information. In the remaining of this section, we provide an overview of the approach underpinning the design of these mechanisms following an introduction of the basic concepts used in it. These concepts are:

— *System model:* A system model is a specification of the components of a software system that is being monitored and the operations provided by these components at an interface level. A system model may also include relationships between the components that it refers to as well as relationships between the object types of the parameters of the operations of these components. Typically a system model comprises the specification of the parts of an S&D pattern and the S&D class that the pattern conforms to and is extracted from an S&D pattern.

— *Asset:* An asset is a *type* of information that is used and/or held by a system or an operation of a system (e.g. the type of an operation parameter, part of the internal structure of a component etc).

— *S&D Property.* An S&D property is a property related to the security and dependability of an asset.

— *S&D objective.* An S&D objective is a specification that expresses the intention to achieve a specific S&D Property in connection with one or more assets (e.g. confidentiality of an operation parameter, availability of a specific operation of a component of an S&D pattern, integrity of an operation of the S&D class of an S&D pattern).

— *Event.* Any observable (or deduced) occurrence in a network or a system.

— *S&D monitoring rule.* A specification of a combination of events and circumstances (e.g. system states), representing a property, whose violation can be detected at runtime. In SERENITY, S&D monitoring rules are specified by Event Calculus formulas as part of S&D patterns.

— *Assumption.* An Event Calculus formula which specifies how events that are directly observed during the system operation at runtime affect the state of the system or can lead to the establishment of other events. During monitoring, assumptions are used by the SERENITY monitor to deduce information about the state of the monitoring system or derived events. Unlike S&D monitoring rules, assumptions do not need to be satisfied by runtime events and their satisfiability is not checked by the monitor.

— *Monitoring policy.* Part of the specification of an S&D pattern that specifies the conditions that need to be monitored at runtime in order to ensure the correctness of the pattern's operation and, therefore, the achievement of the S&D property that the pattern is meant to provide. A monitoring policy has two parts: a part that includes the S&D monitoring rules that should be checked at runtime and a part that includes the assumptions which can be used to derive information about the state of the system that is being monitored and event about it which are necessary for checking the S&D monitoring rules.

— *Rule violation.* A state in which a monitoring rule is known to have been violated.

— *Threat.* Any circumstance or event with the potential to violate a monitoring rule.

— *Threat likelihood.* A measure that indicates how likely is for a threat to occur.

— ***Attack signature.*** An attack signature is a combination of events whose occurrence in certain violates an S&D Property. An attack signature consists of only of events that can be detected during the system operation at runtime without referring to system states that should be established during monitoring. Thus, appart from the absence of any references to system states, attack signatures are otherwise equivalent to S&D monitoring rules and can acquire this status if an S&D pattern developer includes them in the monitoring policy of an S&D pattern.

— *Attack.* An attack is an explicit attempt to execute an attack signature that violates an S&D property.

As indicated in the above terminology, each S&D Pattern contains a monitoring policy with two parts. The first of these parts describes the behaviour of the components of the pattern at some high level of abstraction and how this behaviour can change the state of the monitored system (i.e. the system that deploys the pattern). This part is expressed by Event Calculus formulas known as *assumptions* [2]. *Assumptions* are used by the monitor at runtime to deduce important facts about the state of the system which are relevant to monitoring. The second part of a monitoring policy specifies the specific properties which should be monitored at run-time to ensure that the S&D Properties of the S&D Pattern are indeed guaranteed. These properties are the *monitoring rules* that the monitor is checking at runtime to detect if they are violated.

The specification of monitoring rules is generally more difficult than the specification of the assumptions of an S&D Pattern. This is because assumptions tend to be rather simple and straightforward descriptions of how the execution of a system operation changes the state of the system (e.g. the state of some internal variable, etc). Unlike assumptions, however, the specification of monitoring rules should describe of ALL the possible ways in which the combination of the known runtime events can render an S&D Property invalid.

To address this problem which arises in the specification of monitoring policies in S&D patterns, we have introduced monitoring templates which specify in an abstract parametric form monitoring rules for general S&D Properties including, for example, properties such as confidentiality, integrity and availability [19].

Through the use of these monitoring templates, S&D Pattern developers can produce the monitoring rules needed for the monitoring policy of the pattern more easily. This is possible by selecting the S&D property they are interested in, the assets in the pattern that need protection with respect to the selected property, and the elements interacting with these assets (operations, data types). Subsequently, they can instantiate the monitoring template which corresponds to the selected property using the operations and data types which relate to the aspect that needs protection. Our initial template-based approach to the generation of monitoring policies that was described in [19] assumed that this process would be carried out manually. To ease it further, however, we have developed a scheme for automating it. Our expectation of this scheme is that it will increase not only the usability of the whole monitoring framework of SERENITY but also the degree of confidence that users have in it, since the automated construction of the monitoring rules can ensure their correctness.

As in [19], the automatic generation of monotoring rules starts from the identification of the assets of an S&D pattern and the S&D Properties required for them in order to automatically construct the monitoring rules needed for the pattern. The information needed for the instantiation comes from the identified assets and the assumptions which describe exactly how these assets are used in the context of the specific S&D Pattern. Additional information is taken from the basic architectural description of the S&D Pattern components and parameters within the pattern, which effectively identifies the relationship of these elements with the abstract events and states used in the monitoring templates.

These automatically generated monitoring rules can then be used along with the monitoring assumptions to automatically derive all possible ways through which the rules can be violated at run-time, i.e., the attack signatures. Knowing these attack signatures is essential for computing the threat likelihood during run-time, since it is by matching these possible attack signatures that one is able to estimate the likelihood of an attack becoming successful.

The process of generating monitoring rules, attack signatures and detecting the likelihood of violations of them is shown in Figure 2. As shown in the figure, the security objective and the design information of an S&D Pattern are used by the *Template Instantiator* to produce the monitoring policy. Using this policy, the *Attack Signature Generator* produces the attack signatures for the S&D Pattern and, finally, the *Threat Evaluator* uses the signatures and runtime observations to evaluate the likelihood of threat at runtime.

The creation of the monitoring policy itself is shown in more detail in Figure 3. Using a pair of a security objective and an asset (i.e., a pair of the form ⟨Security Objective, Asset⟩), the monitoring template which is associated with the specific security objective is identified in the catalogue of monitoring templates initially. Also, the asset og interest is used to select appropriate elements from the design of the S&D Pattern, which are used subsequently to instantiate the chosen monitoring template. Then the instantiated monitoring template forms a monitoring policy that can be used to detect violations of S&D properties but not threats. For the latter, it is necessary to derive all possible attack plans, i.e., the different ways that the interaction of the S&D Pattern's elements can violate each monitoring rule within the policy. These attack plans are computed by the *Attack Signature Generator* component of Figure 2 and form the *attack signature* shown in Figure 4.

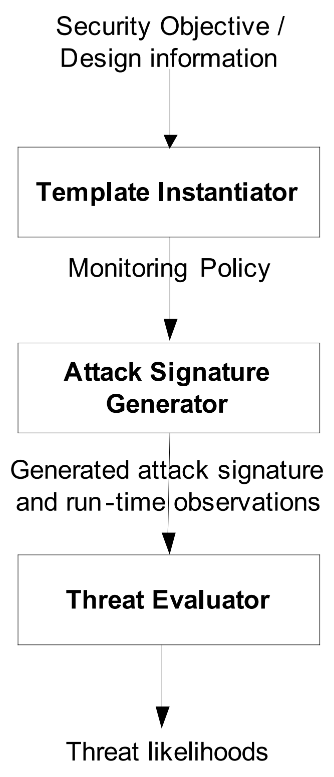
Security Objective / Design information
Template Instantiator
Monitoring Policy
Attack Signature Generator
Generated attack signature and run-time observations
Threat Evaluator
Threat likelihoods

Figure 2. Main components for threat detection.

It should be noted that, the existence of different attack plans does not mean that these will always take place at runtime. Therefore, the final component of Figure 2, namely the *Threat Evaluator*, uses the information concerning known events and the currently known degree of correlation among events to evaluate the likelihood of each of these attack plans materialising and therefore the overall threat level.

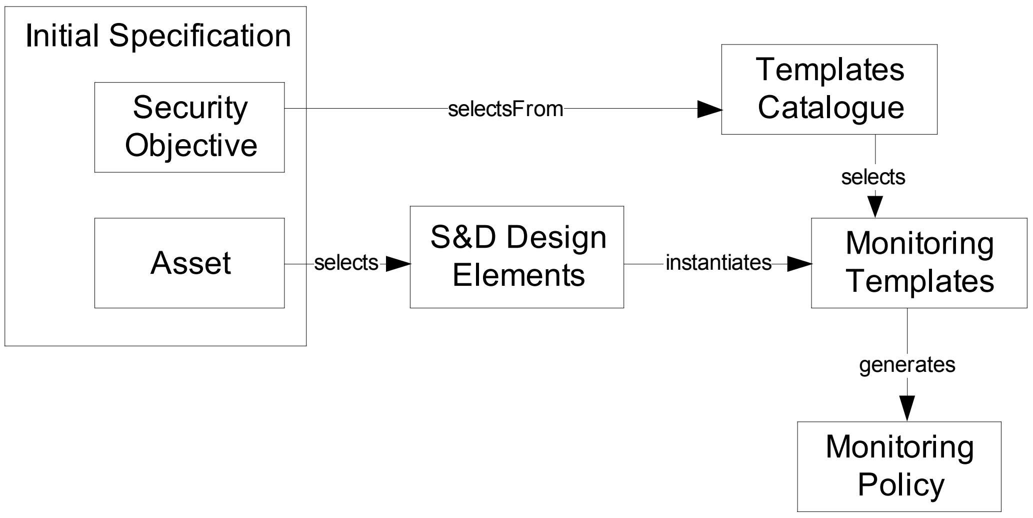

Figure 3. Generation of a monitoring policy from monitoring templates and design models.

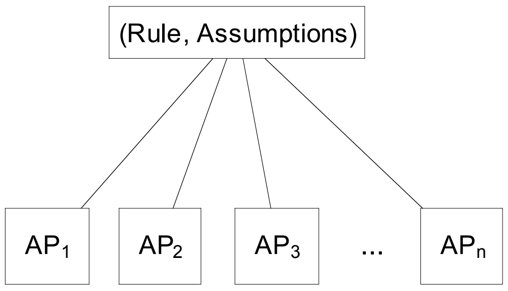

Figure 4. From *goals* to *attack* signatures.

Given the underlying uncertainty which exists with various aspects of the system, the evaluation of threat likelihood is based on the Dempster Shafer theory of evidence [3]. Initially, when some event that (partially) matches the attack signature is observed then this constitutes a possible threat and its likelihood is evaluated. As more observations are gathered, this likelihood of threat may be confirmed and become stronger or refuted and become weaker. The evaluation of threat likelihood is carried out by the component signified as *Threat Evaluator* in Figure 2.

3. Generation of monitoring policies

This section describes the scheme for the generation of monitoring policies by instantiating monitoring templates. This is what drives the software component *Template Instantiator* identified in Figure 2.

3.1. Overview of monitoring templates

The monitoring templates described herein were first introduced in [19] and then in [20], in order to facilitate the specification of the monitoring rules for the monitoring policies of S&D Patterns. They were derived by the observation that the basic security properties, i.e., confidentiality, integrity & availability in the CIA model, need specific types of monitoring rules, whose structure is independent from the specific system they are defined for. The system specific differences have in general to do with the identification of the asset/data to be protected, the operations that access/modify the asset and the principals which perform these operations. Thus, if one identifies these design elements – asset, operations, and principals – then it is possible to produce a specialised version of the monitoring template for the specific system.

Monitoring rules:	
∀ _o:Operation; _i:InformationTerm; t1 :Time _sender, _receiver, _owner, _agent1:Agent; Happens(E1, t1, ℜ(t1, t1)) ∧ HoldsAt(exposes(<_o>, <_owner>, <_i>), t1) ⇒ HoldsAt(authorised(<_owner>, <_receiver>, E1),t1) where: E1 = e(_eID1[1],<_sender>,<_receiver>, [REQ-*|RES-*], <_o>, <_sender>)	CSR1
Assumptions:	
Initially(exposes(<_o>, <_owner>, <_i>))	CSA1
∀ _authorO: Operation; _sender, _receiver,_agent1, _owner:Agent; _authorisedValue, _result_authorO: Term; t:Time; Happens(E2, t,ℜ(t,t)) ∧ HoldsAt(equalTo(<_result_authorO>, <_authorisedValue>),t) ⇒ Initiates(E2, authorised(<_owner>,<_receiver>,E1),t) where E2 = e(_eID2,<_agent1>,<_sender>, RES-A, <_authorO>, <_sender>)	CSA2

Figure 5. Pattern for confidentiality monitoring (adapted from [19])

[1] The variables indicating the identifiers of events in all the formulas in the paper are assumed to be of type *String* and universally quantified.

An example of such a template for confidentiality is the one shown in Figure 5. The template uses as template parameters the elements such as <_o>, <_owner>, etc. and uses these to specify when an asset <_i> owned by agent <_owner> is exposed to someone else.

However, the templates of [19] were described in an ad-hoc language, which is a combination of the Event Calculus language used for the monitoring rules and assumptions, and of an informal language which introduced the template parameters, such as the asset <_i>, as can be seen in Figure 5.

The need to automate the process of template instantiation made it necessary to describe all parts of the templates templates in a formal language and, for this puprose, we have recasted templates using a language that is a combination of Event Calculus and the Formal Template Language (FTL) [10][11]. FTL is used to express the monitoring templates, as well as, to formalise the generation of monitoring policies from the templates given an instantiation, i.e., a mapping of the template parameters to specific modelling elements.

In the following, we start by giving a description of FTL and then, we express the confidentiality monitoring template shown in Figure 5, and illustrate the monitoring policy generation process using this template and a design model from our motivating example.

3.2. The Formal Template Language

The Formal Template Language (FTL) [10][11] is a generic formal language for expressing templates of any target language. A key characteristic of FTL is that it is *generative*; that is, it describes sentences of some target language (here we use EC) and can generate sentences when provided with an instantiation. FTL has a denotational semantics based on its abstract syntax (see [10][11] for further details).

The following starts by giving a short introduction to FTL. Then, we present FTL's syntax and show how we can use FTL to define templates of the Event Calculus. Finally, we discuss FTL's semantics in more detail and discuss the implementation of its instantiation mechanisms.

3.2.1. A short introduction to FTL

The main constructs of FTL are *placeholders*, *lists, choice, template definitions* and *template references* (see [10][11] for further details). The following briefly describes each of these constructs.

Placeholders. Placeholders are represented by enclosing one variable within ⟨⟩. When not within lists, placeholders denote one variable occurrence and they are substituted by the value assigned to the variable when the template is instantiated. The template,

$\langle X\rangle=\langle Y\rangle \vee \langle X\rangle=\langle Z\rangle$,

includes four placeholders and three variables, *X*, *Y* and *Z*. This can be instantiated with the substitution set,

$\{X \mapsto \text{"A"}, Y \mapsto \text{"1"}, Z \mapsto \text{"2"}\}$,

to yield:

$A=1 \vee A=2$

List. A list comprises one list term, a list separator (the separator of the instantiated list terms) and a string representing the empty instantiation of the list, and it is represented by enclosing the list term within **[ ]**. The list term is a combination of text, placeholders and possibly other lists. Often, the abbreviated form of lists, without separator and empty instantiation, is used.

A placeholder within a list denotes an indexed set of variable occurrences. This means that *⟨X⟩ and **[**⟨X⟩ **]*** actually denote different variable occurrences; *⟨X⟩* denotes an occurrence of the variable *X*, but ***[** ⟨X⟩**]*** denotes the occurrence of the indexed set of variables, $\{x_1, \ldots, x_n\}$

The template,

[⟨X⟩=⟨Y⟩]$_{(\vee,\ \lambda)}$,

can be instantiated with the sequence of substitution sets

⟨{X↦ "A", Y↦ "1"}, { X↦ "B", Y ↦ "2"}⟩,

to yield :

A=1 ∨ B=2

Choice. The FTL choice construct expresses *choice* of template expressions. That is, only one of the available choices is present in the instantiation. There are two kinds of choice: optional and multiple. Optional choice is represented by enclosing a template expression within **( )**$^?$ and it means that the enclosed expression may be present in the instantiation or not. Multiple choice is represented by enclosing the choice expressions within **( )** and separated by [] (see below); it means that one of the choice expressions must be present in the instantiation. Choices are instantiated with a choice-selection, a natural number, indicating the selected choice; non-selection takes the value zero.

The template *(⟨X⟩ = ⟨Y⟩)?* can be instantiated with *(1, {X → "A", Y → "1"),* to yield: A = 1. To avoid the presence of the expression in the instantiation, the template can be instantiated with *(0 {}),* which simply yields the empty string. In the multiple-choice template,

(<X> = <Y> [] <X> = {<Z>})

the first choice is instantiated with (1, *{X → "A", Y → "1"*}), to yield: *A = 1*; the second with

(2, *{X ↦ "A", Z ↦ "2"*} to yield: A = {2}.

Template Definitions. FTL provides a construct to define templates, which associates some template name with a set of template expressions. For example, we may associate the name "SimpleDisjunctionOfEquals" with the template expression used above to illustrate *placeholders*:

⟨SimpleDisjunctionOfEquals⟩$_{\text{tdef}}$ == *⟨X⟩ =⟨Y⟩ ∨ ⟨X⟩=⟨Z⟩*

We may also associate a name with the template expression used above to illustrate the *list* constructor:

⟨CompositeDisjunctionOfEquals⟩$_{tdef}$ == [⟨X⟩=⟨Y⟩]$_{(\vee,\ \lambda)}$,

Template Reference. Names are associated with templates so that they can be referenced by other templates. We define references to templates by using the "template reference" construct. For example, from the template defined above we can define a new template:

⟨SimpleOrCompositeDisjunction ⟩$_{\text{tdef}}$==

⟨SimpleDisjunctionOfEquals⟩$_{\text{tref}}$ [] ⟨CompositeDisjunctionOfEquals⟩$_{\text{tdef}}$

which defines a new template as a choice of two template references.

The formal definition of FTL's syntax in the Backus-Naur form (BNF) is given in Figure 6.

T ::=	TD | TD T
TD ::=	$\langle I\rangle_{tdef}$ == E
E ::=	CA | CA E
A :: =	$\langle I\rangle$ | T | $\langle I\rangle_{tref}$
C ::=	$(E)^{?}$ | (CL)
CL ::=	E_1 [] E2 | E [] CL
L ::=	$[\,LT\,]_{(SEP,\,EI)}$ | [LT]
LT ::=	CA | CA LT
SEP ::=	Str
T ::=	Str

Figure 6. The syntax of FTL.

3.2.2. *Example of FTL template of EC predicate*

To further illustrate FTL in the context of the EC, suppose the following FTL template of an EC predicate, which specifies a number of event preconditions associated with the template event *E*:

$\langle FTLECDef\rangle_{tdef}$ == (∀ t : Time) Happens (⟨E⟩, t, R(t, t)) ⇒ **[** HoldsAt (⟨F⟩, t) **]**

This template includes two placeholders and one list term. It basically says that a number of pre-conditions (*HoldsAt* predicate inside the list with placeholder *F*) may be associated with some event (*E* placeholder in *Happens* predicate). The template can be instantiated to give the following EC formula:

(∀ t : Time) Happens (Eat, t, R(t, t)) ⇒ HoldsAt (IsHungry, t)∧ HoldsAt (DinnerServed, t)

Below, we give further examples of FTL templates of EC predicates and their instantiations.

3.2.3. *FTL semantics and instantiation mechanisms*

FTL has a semantics based on substitutions. Given an instantiation (a set of substitutions) a template is substituted with values for placeholders until target language text is generated. FTL has been given two alternative semantics: the *total-instantiation* semantics and the *partial-instantiation* semantics [10][11]. Our implementation of the instantiation mechanisms follows the *total-instantiation* semantics of FTL, which means that all substitutions are performed on some template until it has been totally instantiated.

The instantiation process is done in two steps:

1. First all template references constructs are substituted by the template they refer to. This is done by fetching the appropriate template from the template catalogue and performing the appropriate substitution. We call this process template reference resolution, and its underlying algorithm is described in Figure 7.

2. After all template references have been resolved, substitutions are performed according to the given instantiation. This effectively performs the substitutions of placeholders, lists and choices until the final text of the target language (in our case the Event Calculus) is generated. A pseudo-code description of the algorithm that performs this is given in Figure 8.

```
Resolve_Template_References ( ftl_template : String)
1.  FTL_Abs_Syn_Tree = Parse_FTL_Def (ftl_template);
2.  For each node in FTL_Abs_Syn_Tree do
3.    If node.type == tpl_ref then
4.     Find_And_Replace (node, node.value);
5.    end if
6.  end for
end Resolve_Template_References
```

Figure 7. Template reference resolution algorithm.

```
Instantiate_Template (ftl_template : String; Instantiation)
1.  FTL_Abs_Syn_Tree = Parse_FTL_Def (ftl_template);
2.  If Has_references (FTL_Abs_Syn_Tree) then
3.    raise exception "FTL Def not resolved";
4.  end if
5.  For each node in FTL_Abs_Syn_Tree do
6.    switch(node.type)
7.      case tpl_placeholder then  substitute_placeholder(node, instantiation);
8.      case tpl_choice then substitute_choice (node, instantiation);
9.      case tpl_list then substitute_list (node, instantiation);
10.   end case;
11. end for
end Instantiate_Template
```

Figure 8. Algorithm for instantiation of placeholders, lists and choices.

The software component *FTL* template *instantiator*, as shown in Figure 9, is responsible for carrying out the tasks described above, i.e., the implementation of the instantiation mechanisms as these are defined in the semantics of FTL [10][11].

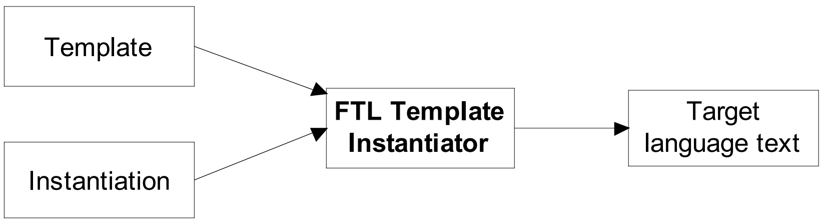

Figure 9. The FTL template instantiator

3.3. Expressing Monitoring Templates in FTL

In order to better illustrate our approach we will first re-specify the template for confidentiality that we introduced earlier (Section 3.1. in FTL and apply it to our motivating example (see Section 2) showing how these template can be specialised for the purpose of modelling partial access to some asset's data, and then be automatically instantiated from elements of the system model of an S&D pattern.

3.3.1. Confidentiality Monitoring Template in FTL

The confidentiality monitoring template that we introduced earlier in Section 3.1. is expressed in FTL as shown in Figure 10. The core part of this tempalte comprises one monitoring rule (formula 1) and two assumptions (formulas 2–3). Formula 1 says that if an agent is exposed to some asset he must be authorised to do so at the time of the exposition. Formula 2 says that an event "*⟨Authorise⟩*" initiates (sets to true) the fluent "*Authorised⟨Asset⟩*". Formula 3 says that an event "*⟨Disclose⟩*" initiates the fluent "*Exposed⟨Asset⟩*".

The template of Figure 10 expresses a basic and general security property, namely confidentiality in relation to some asset's data. By virtue of the monitoring rule that is expressed by formula 1, the template says that the asset's data may be disclosed to some agent only if the agent is authorised to access these. This is because if that's not the case the rule expressed by formula will be breached resulting in an S&D violation. Although quite general, however, this template does not support all possible needs of confidentiality monitoring and, in particular, it does not cover the case of partially accesing data which in the case of our motivating example in Section 2 is important.

$\langle\text{Confidentiality}\rangle_{\text{tdef}}$ ==

$(\forall\ ag : \langle\text{Agent}\rangle;\ a : \langle\text{Asset}\rangle;\ t : \text{Time})$ (1)

$\quad \text{HoldsAt}(\text{Exposed}\langle\text{Asset}\rangle(ag, a), t) \Rightarrow \text{HoldsAt}(\text{Authorised}\langle\text{Asset}\rangle(ag, a), t)$

$(\forall\ ag : \langle\text{Agent}\rangle;\ a : \langle\text{Asset}\rangle;\ t : \text{Time})$ (2)

$\quad \text{Happens}(\langle\text{Authorise}\rangle(ag, a), t, R(t, t))$

$\quad \Rightarrow \text{Initiates}(\langle\text{Authorise}\rangle(ag, a), \text{Authorised}\langle\text{Asset}\rangle(ag, a), t)$

$(\forall\ ag : \langle\text{Agent}\rangle;\ a : \langle\text{Asset}\rangle;\ t : \text{Time})$ (3)

$\quad \text{Happens}(\langle\text{Disclose}\rangle(ag, a), t, R(t, t))$

$\quad \Rightarrow \text{Initiates}(\langle\text{Disclose}\rangle(a, ag), \text{Exposed}\langle\text{Asset}\rangle(ag, a), t)$

Figure 10. Core confidentiality template.

To deal with this case, we need to extend the core template so as to cover partial accesses to the asset's data and the template for confidentiality with inference, given in Figure 11, extends the core confidentiality template to monitor exactly these indirect accesses to the asset. The FTL expression "$\langle\text{Confidentiality}\rangle_{\text{ref}}$" in the template of Figure 11 means that the core confidentiality template of Figure 10 is to be included at that point, and so instantiations of 'Confidentiality.Inference' also comprise an instantiation of the core 'Confidentiality' template. The 'Confidentiality.Inference' template is made entirely of assumptions (formulas (4)-(6)) with no additional monitoring rules. In particular, formula 4 in the extended template says that the event "*⟨DiscloseP⟩*" initiates the template fluent "*KnowsPD⟨Asset⟩*". Formula 5 says that event *"⟨DiscloseP⟩"* initiates the fluent "*Exposed⟨Asset⟩*", provided the agent already knows all partial pieces of the medical file apart from the one being accessed currently (by event *⟨DiscloseP⟩*). Finally, formula 6 specifies which partial pieces of data are available.

$\langle \text{Confidentiality.Inference} \rangle_{tdef}$ ==

$\langle \text{Confidentiality} \rangle_{tref}$

$$\begin{array}{l}
\forall\ ag : \langle Agent \rangle;\ a : \langle Asset \rangle;\ pdt : \langle PDT \rangle;\ t : Time) \\
\quad Happens(\langle DiscloseP \rangle\ (ag, a, pdt), t, R\ (t, t)) \\
\quad \wedge\ HoldsAt(AvailablePD\langle Asset \rangle\ (pdt, t) \\
\quad \Rightarrow\ Initiates\ (\langle DiscloseP \rangle\ (ag, a, pdt), KnowsPD\langle Asset \rangle\ (ag, a, pdt), t)
\end{array} \tag{4}$$

$$\begin{array}{l}
(\forall\ ag : \langle Agent \rangle;\ a : \langle Asset \rangle;\ pdt : \langle PDT \rangle;\ t : Time) \\
\quad Happens\ (\langle DiscloseP \rangle\ (ag, a, pdt), t, R\ (t, t)) \\
\quad \wedge\ HoldsAt(AvailablePD\langle Asset \rangle\ (pdt), t) \\
\quad \wedge\ ((\forall\ pdt2 : \langle PDT \rangle)\ pdt \neq pdt2 \\
\quad \wedge\ HoldsAt(AvailablePD\langle Asset \rangle\ (pdt2), t) \\
\quad \Rightarrow HoldsAt(KnowsPD\langle Asset \rangle\ (ag, a, pdt2), t)) \\
\quad \Rightarrow\ Initiates(\langle DiscloseP \rangle\ (ag, a, pdt), \\
\quad Exposed\langle Asset \rangle\ (a, ag), t)
\end{array} \tag{5}$$

$$[\ Initially(AvailablePD\langle Asset \rangle(\langle PD \rangle))\ ] \tag{6}$$

Figure 11. Extension to the core template for monitoring access to partial data.

Given the extended template that is expressed in FTL we are now in a position to show how concrete monitoring policies can be generated from it (and other templates).

3.4. Generating monitoring policies from security objective and asset

The initial stage of the process for generating monitoring policies is concerned with how the templates are selected and how they are instantiated into the final monitoring policy.

As briefly explained in Section 2. this process is driven by a pair of ⟨*security objective*, *asset*⟩ indicating that the *security objective* is required for *asset* within an S&D pattern. This is what initiates the request for monitoring some security property (specified by the security objective) over some system asset. The former is used to select a template and the latter to gather information to instantiate the selected template.

The process of monitoring policy generation starts by selecting a template given the above pair, and proceeds with gathering information from the S&D pattern design model (if possible) to enable template instantiation and monitoring policy generation. These steps are explained below illustrated through the use of our motivating example.

3.4.1. *Template selection from Security Objective*

The security objective identifies uniquely both the security property to monitor and the template to be used. This is because the name of each security monitoring template identifies a specific security property. For example, the security objective *Confidentiality.Inference* results in the selection of the template of Figure 11.

Computationally, this involves a simple search query within the catalogue of templates, where given a template/property name one tries to find a matching template, if there is one.

Once retrieved, a template is parsed in order to identify any references to other templates, which are themselves then retrieved and their references are substituted with their contents and so on, recursively.

The final template, which contains no further references to other templates, identifies (among others) a set of placeholders. Collectively, these placeholders identify the required instantiation information. This instantiation information is then retrieved from the contents of the S&D Pattern, by using the asset description.

3.4.2. *Gathering instantiation information from the security asset*

The process of template selection from a security objective identifies the template to be instantiated. Once the template is identified, we need to retrieve the instantiation parameters that enable the instantiation of the template. These parameters are used along with the specified system asset to search for instantiation information, which is essentially a search for values to assign to placeholders. The algorithm that retrieves the instantiation parameters is described in Figure 12.

```
Retrieve_Instantiation_Parameters (ftl_template : String) : Instantiation_Parameters
1.  ins_params : Instantiation_Parameters;
2.  ins_params = {};
3.  FTL_Abs_Syn_Tree = Parse_FTL_Def (ftl_template);
4.  if Has_references (FTL_Abs_Syn_Tree) then
5.    raise exception "FTL Def not resolved";
6.  end if;
7.  For each node in FTL_Abs_Syn_Tree do
8.    switch(node.type)
9.      case tpl_placeholder then  add_to_ins_params_placeholder (ins_params, node);
10.     case tpl_choice then add_to_ins_params_choice (ins_params, node);
11.     case tpl_list then add_to_ins_params_list (ins_params, node);
12.   end case;
13. end for;
14. return ins_params;
end Retrieve_Instantiation_Parameters
```

Figure 12. Algorithm for retrieving instantiation parameters from templates.

The search for instantiation information involves going through the system model of a pattern to find instantiation information pertinent to the asset. This search process relies on tags or stereotypes that are attached to system models (illustrated below for UML stereotypes), where the name of a tag or stereotype corresponds to some template placeholder. This effectively establishes the link between user models and template placeholders.

Our generation process assumes that the system models in the SERENITY S&D patterns follow the meta-model of Figure 13. More specifically, according to this meta model, each system model is composed of modelling elements (class *ModellingElement*) having type (e.g. class, operation, attribute) and name attributes (the name of the class, attribute, etc). Furthermore, each modelling element may have relations with other modelling elements (e.g. a class has attributes, operations and associations) and be associated with zero or more tags or stereotypes (see association between classes *Tag* and *ModellingElement* in Figure 13)[2].

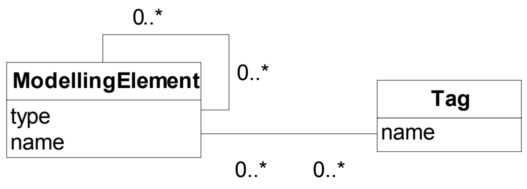

Figure 13. Meta model for system models.

Currently, the list of placeholders/tags that we use in the specification of monitoring templates and for stereotyping design models in S&D patterns includes the tags shown in Table 2. The same table also indicates the meaning of each placeholder/tag and the types of the modelling elements that the tag can be applied to.

[2] This is because the same modelling element may assume different roles across security policies. For instance, in the context of a confidentiality property the some operation (a modelling element) may assume the role of a “Disclose” operation and is, therefore, tagged as such, but in the context of integrity the same operation may assume the role of a “Tranforms” operation.

Tag / Placeholder	Description	Modelling Element type
Asset	Used to identify system assets.	Class, Entity
Agent	Used to identify a generic agent in some security policy.	Class, Entity
Disclose	Identifies an operation that discloses some data related to an asset.	Operation
DiscloseP	Identifies an operation that discloses some partial data that is related to some asset.	Operation
Authorise	Identifies an operation that authorises access to some asset's data.	Operation
PDT	Used in confidentiality monitoring policies to identify a type of partial data.	Class, Entity
Transform	Identifies a transformation of data, used in integrity security policies.	Operation

Table 2. Names of placeholders to be used in security templates and as tags/stereotypes

The list of tags shown in Table 2 has been identified from an analysis of the concepts related to the specification of the basic security properties of confidentiality, integrity and availability. It should, however, be noted that our approach is generic and can accommodate further tags/placeholders that may be defined by communities of S&D pattern developers and users of the SERENITY framework.

Given the structure assumed by the meta-model of Figure 13, the search algorithm that finds instantiation information works as follows:

1. It looks in the system models of an S&D pattern for modelling elements with the name of the specified asset and with the tag "Asset" to produce a set of *core* modelling elements.
2. It searches through the core modelling elements for their constituent modelling elements, using the template parameters, and gathering all required instantiation information in the process.

Once this is done, it should obtain an instantiation for the template that should, along with the template, result in a specific monitoring policy. The algorithm that produces an instantiation is described in Figure 14. An illustration of how this algorithm works is given in the next section using our earlier example.

```
TemplateInstantiation (asset_name : String, ins_params : Instantiation_parameters) : Instantiation
1.    asset_model_elements : Set(ModelligElement);
2.    param_m_elems : Set(ModellingElement);
3.    result : Instantiation;
```

```
4.    result = {};
5.    asset_model_elements =  Get_List_Modelling_Elements ("Asset", asset_name);
6.    if asset_model_elements == {} then
7.       raise exception "Asset Not Found";
8.       return Empty_Instantiation;
9.    end if;
10.   result = result ∪ {("Asset", asset_name)};
11.   For each param in ins_params do
12.      param_m_elems = Find_Modelling_Elements (asset_model_elements, param);
13.      For each m_elem in param_m_elems do
14.         result = result ∪ {(param.name, asset.name)};
15.      end for;
16.   end for;
17.   return result;
end TemplateInstantiation
```

Figure 14. Process for template instantiation.

3.4.3. Example

We now show how the templates of Figure 10 and Figure 11 are instantiated to produce a monitoring policy.

To start the generation process, we start from the object/asset pair:

⟨Confidentiality.Inference, Patient⟩

The above pair says that we want to monitor the "Confidentiality.Inference" security property over the "Patient" asset. The first element of the pair (security property) results in the selection of the template of Figure 11. The second element of the pair (*Patient* asset) would result in the selection of a set of system model elements in the relevant S&D pattern that specify properties of the asset. A simple design for the confidentiality security requirement expressed as a UML class diagram that refers to the *Patient* asset is shown in Figure 15. We can see that this model follows the structure of the meta-model of Figure 13 as certain elements in it have been tagged[3].

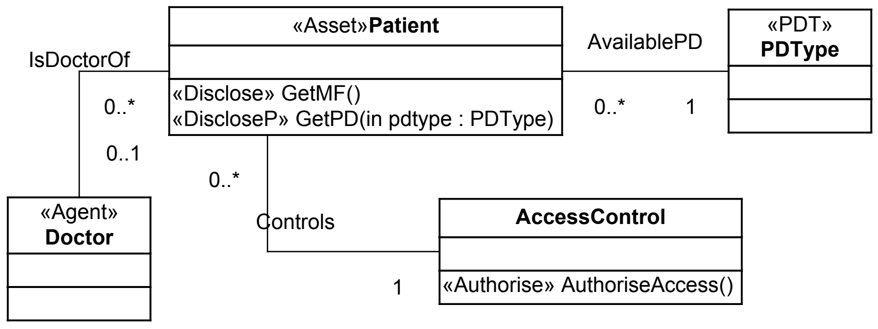

Figure 15. A simple system model for Patients.

[3] An extension of the S&D pattern specification language will be required for the attachement of such tags to elements of S&D patterns.

The class model of Figure 15 introduces the classes (object types) *Doctor*, *Patient*, *PDType* and *AccessControl*. The link with the confidentiality template is established through stereotypes, where each stereotype corresponds to the name of a placeholder. From this class model we obtain the following instantiation (or substitution set) of the monitoring template:

$$\{Agent \mapsto \text{``}Doctor\text{''},\ Asset \mapsto \text{``}Patient\text{''},\ Disclose \mapsto \text{``}GetMF\text{''},$$
$$Authorise \mapsto \text{``}AuthoriseAccess\text{''},$$
$$PDT \mapsto \text{``}PDType\text{''},\ DiscloseP \mapsto \text{``}GetPD\text{''}\}$$

Based on the above substitution set the instantiation process results in the generation of the EC monitoring policy of Figure 16.

(∀ ag : Doctor; a: Patient; t : Time) (7)
HoldsAt (ExposedPatient (ag, a), t) ⇒ HoldsAt (AuthorisedPatient(ag, a), t)

(∀ ag : Doctor; a: Patient; t : Time) (8)
Happens(AuthoriseAccess(ag, a), t, R (t, t))
⇒ Initiates (AuthoriseAccess(ag, a), AuthorisedPatient (ag, a), t)

(∀ ag : Doctor; a: Patient; t : Time) (9)
Happens(GetMF (ag, a), t, R (t, t))
⇒ Initiates (GetMF (a, ag), ExposedPatient (ag, a), t)

∀ ag : Doctor; a: Patient; pdt : PDType; t : Time) (10)
Happens (GetPD (ag, a, pdt), t, R (t, t))
∧ HoldsAt(AvailablePDPatient (pdt), t)
⇒ Initiates (GetPD (ag, a, pdt), KnowsPDPatient (ag, a, pdt), t)

(∀ ag : Doctor; a: Patient; pdt : PDType; t : Time) (11)
Happens(GetPD(ag, a, pdt), t, R (t, t))
∧ HoldsAt(AvailablePDPatient(pdt), t)
∧ ((∀ pdt2 : PDType) pdt ≠ pdt2
∧ HoldsAt(AvailablePDPatient (pdt2), t)
⇒ HoldsAt(KnowsPDPatient (ag, a, pdt2), t))
⇒ Initiates(GetPD(ag, a, pdt), ExposedPatient (a, ag), t)

Initially(AvailablePDPatient(PD1)) (12)

Initially(AvailablePDPatient(PD2)) (13)

Figure 16. Generated Event Calculus monitoring policy.

In this monitoring policy all the placeholders of the original template have been replaced by the model elements that were identified through the instantiation search process. The placeholder <*authorise*> in formula 3, for instance, has been replaced by the operation *AuthoriseAccess* in the model giving rise to formula 8 in the instantiated monitoring policy. This substitution was identified by the fact that the latter operation had the same stereotype as the placeholder in formula (3) (i.e., the stereotype <authorise>).

The next section shows how the generated monitoring theory is used to detect threats to the confidentiality security requirement at run-time. In particular, it shows how an attack signature can be derived for this monitoring policy.

3.5. Summary

The *template instantiator* software component implements all algorithms that have been described in this section. In particular:

- The resolution of templates references to produce a template without references to other templates (Figure 7).
- The generation process given a reference-resolved template and an instantiation of the given template to produce a monitoring policy (Figure 8).
- The derivation of instantiation parameters from a reference-resolved template (Figure 12).
- The algorithm that produces instantiation information by searching through user models (Figure 14).

The whole process of producing a monitoring policy uses these individual algorithms as follows:

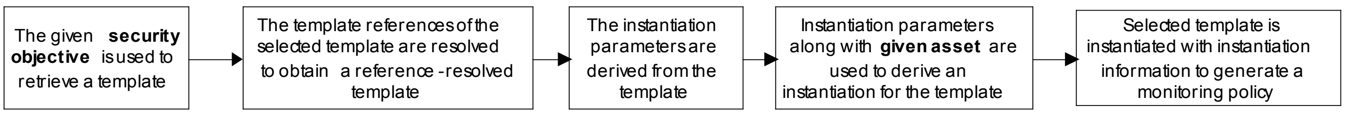
The given **security objective** is used to retrieve a template
The template references of the selected template are resolved to obtain a reference-resolved template
The instantiation parameters are derived from the template
Instantiation parameters along with **given asset** are used to derive an instantiation for the template
Selected template is instantiated with instantiation information to generate a monitoring policy

Figure 17. Process for producing a monitoring policy.

4. Attack signature generation

Attack signatures are generated from a monitoring policy by using AI planning techniques [12][13]. Our task does, however, differ from the work on *classical* planning, where the aim is to derive a concrete sequence of actions to achieve a *goal*. That is, given some specific state on some specific time-point, some formal theory and a goal, classical planning derives a concrete sequence of actions to achieve a goal; this specific sequence of actions is described in terms of specific instances. Given that attack signature generation is performed off-line, the process concentrates on deriving an *abstract* or *universal* plan representing all possible sequences of actions over time to achieve some goal. Such plans compactly represent every classical plan [21] involving both *conditional* and *partial-order* plans. Conditional plans may be generated as during the plan derivation process some information may be missing and different branches may have to be included [13]. Partial-order plans may also arise as it might not be possible to fully order the actions involved in them based on the available time constraints [12].

The next section presents a brief overview of the approach to derive an abstract signature from a monitoring policy based on planning.

4.1. Overview

In general, there are two approaches to derive plans to achieve some goal:

- The first approach proceeds *forwards* to derive either a set of paths that go from the initial state to the goal state (classical planning) or an abstract description of all these possible paths. This approach is known as *deductive planning* [22] as the forward derivation of plans is based on the logical mechanism of *deduction*.
- The second approach proceeds *backwards* from the goal state to different initial states that can lead to it, trying to derive either a set of paths (classical planning) or an abstract description of all possible paths (universal planning). This approach is known as *abductive planning* [13] because it is based on the logical mechanism of *abduction*.

Our attack signature generation mechanisms undertakes the second approach and is based on an adaptation of the algorithm that is used in the SERENITY monitoring framework to generate the diagnosis for violations of S&D monitoring rules which has been presented in [4].

Abduction is a process of backward reasoning to can generate a set of explanations for a given set of observations based on a specific theory [24] . It has been defined as a kind of inversion of Modus Ponens (or logical implication) [25]. The key idea behind abduction can be represented by the following inference rule [25]:

$$\frac{\varphi \Rightarrow \omega, \omega}{\varphi}$$

That is, from an observation ω and a rule "φ⇒ω" in a given theort, infer the occurrence of φ as a plausible hypothesis of what is the reason for the occurrence of ω or, equivalently, an explanation of ω. The idea of using abduction for planning in the Event-Calculus was introduced by Eshgi [12], whose aim was to compute all possible explanation of some goal by following an abductive strategy, going backwards from the goal.

In our approach abduction is used in order to find all the possible "explanations" or causes of the predicates that constitute a goal. More specifically, the plan generation process starts from a triplet *⟨R, MP, ECAxs⟩* which includes: (a) a monitoring rule *R* for which plans need to be generated, (b) a monitoring policy *MP* that includes R, other rules and assumptions about the system to be monitored, and (c) the set of axioms of event calculus *ECAxs*. Based on these inputs, the general steps of the plan generation process are the following:

1. The goal G for which plans need to be generated is created by negating the monitoring rule, i.e. $G = \neg R$
2. *G* is transformed into a conjunctive normal form (i.e., a conjunction of atomic conditions).
3. An abductive plan generation algorithm is deployed to derive a plan for each atomic condition in G. As there may be alternative ways to reach a particular atomic condition, this step may generate a set of alternative subplans (AS) for each condition. This algorithm identifies the set of the alternative primitive possible causes for each atomic condition, i.e., causes which cannot themselves be the result of another cause.
4. The valid combinations of all the different plans for each subgoal (atomic condition) are identified and alternative global plans for G are formulated as a conjunction of the valid combinations of the plans for the subgoals.
5. The global attack signature for G, which identifies all the possible ways of compromising R (aka attacks against R) is obtained by taking the disjunction of the alternative global plans.

A simple example of undertaking the above steps and combining the alternative plans which are generated for the different atomic conditions of a goal of the form $G = S_1 \wedge S_2$ in given in Figure 18.

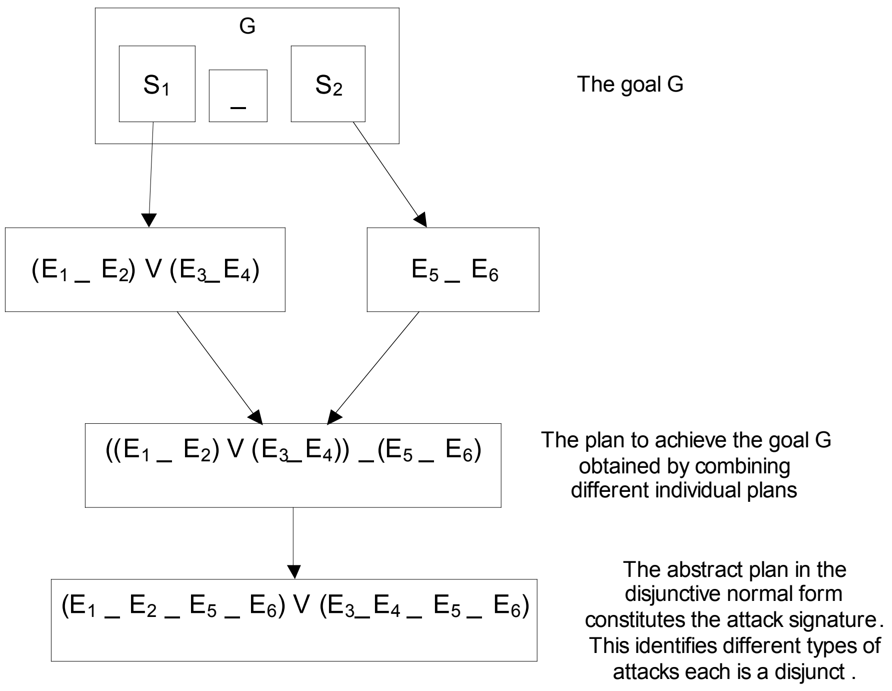
G
S1
_
S2
The goal G
(E1 _ E2) V (E3_E4)
E5 _ E6
((E1 _ E2) V (E3_E4)) _(E5 _ E6)
The plan to achieve the goal G obtained by combining different individual plans
(E1 _ E2 _ E5 _ E6) V (E3_E4 _ E5 _ E6)
The abstract plan in the disjunctive normal form constitutes the attack signature. This identifies different types of attacks each is a disjunct .

Figure 18. The derivation of a plan for the conjunctive goal $S_1 \wedge S_2$.

4.2. The abductive plan generation algorithm

The abductive plan generation algorithm is shown in Figure 19.

```
FindPlansForFormula(FR, Constraints): $\Phi_e$
/* FR: formula */
/* Constraints: temporal constraints related to FR */
/* $\Phi_e$: a list keeping the disjunction of possible attack signatures for e */
  1.  $\Phi_e$ = [ id(FR):e $]_{OR}$  /* a list keeping the possible attack signatures for e */
  2.  For all atomic formulas e in FR do
  3.    For all $f \in$ AS do /* try to derive plans from all given assumptions AS*/
  4.      u = mgu(head(f), e) /* mgu returns the most general unifier of e and a predicate p if this unifier exists*/
  5.      If u $\neq \varnothing$ Then /* e matches head(f) */
  6.        Copy atomic conditions in the body(f) into $CND_f$
  7.        If $CND_f = \varnothing$ Then /* assumption f is an atomic formula of the form: $\Rightarrow$head(f) */
  8.          FormulaFailed = False
  9.          $C_u$ = ApplyUnification(u, head(f))
  10.         $\Phi_f = [(f{:}C_u, t_{min}(C_u) < t_{var}(C_u), t_{var}(C_u) \leq t_{max}(C_u)]_{AND}$
  11.       Else /* assumption f is an non atomic formula of the form: body(f)$\Rightarrow$head(f) */
  12.         TemporalConstraints = Constraints $\bigcup_{c \in CND_f} \{t_{min}(c) < t_{var}(c), t_{var}(c) \leq t_{max}(c)\}$
  13.         If TemporalConstraints are consistent Then
  14.           FormulaFailed = False
  15.           $\Phi_f = []_{AND}$  /* return an empty $\Phi_f$ */
  16.           While FormulaFailed = False and $CND_f \neq \varnothing$ Do /* find plans for all the conditions of f */
  17.             Remove some condition C from $CND_f$
  18.             $C_u$ = ApplyUnification(u, C)
  19.             $\Phi_C$ = Explain(C, f, TemporalConstraints)
  20.             If $\Phi_C$ is empty Then
  21.               FormulaFailed = True
  22.             Else
  23.               $\Phi_f$ = append($\Phi_f$ , $\Phi_C$)
  24.             End If
  25.           End While
  26.         Else /* Constraints $\cup \{t_{min}(C) < t_{var}(C), t_{var}(C) \leq t_{max}(C)\}$ are not consistent */
  27.           FormulaFailed = True
  28.           $\Phi_f = []_{AND}$
  29.         End If /* temporal constraints test */
  30.       End if /* atomic vs non atomic assumption test */
  31.       If FormulaFailed = False Then /*add the plans generated from f if none of its conditions failed */
  32.         $\Phi_e$ = append ($\Phi_e$,$\Phi_f$)
  33.       End if
  34.     End if /* unification test */
  35.   End For
  36.   End If
  37. End For
  38. return($\Phi_e$)
  END FindPlansForFormula
```

Figure 19. Algorithm for abductive generation of attack signatures

This algorithm is an adaptation of the algorithm used for the generation of diagnostic explanations by abduction within the SERENITY monitoring framework that has been discussed in [4].

The algorithm of Figure 19 tries recursively to produce an attack signature for a formula. For each atomic condition *e* of it searches exhaustively through the assumptions *AS* of the given monitoring policy to establish if any of them has a predicate in its head that matches *e*. If such an assumption *f* is identified the algorithm checks if it is an atomic formula and if that's the case it adds f to the possible attack signatures for *e*. If *f* is not an atomic formula, the algorithm gets the constraints of the time variables of *f* and checks if they are consistent with the constraints of the time variables of *e*. In case that they are, it proceeds recursively by trying to generate alternative attack signatures for each of the conditions of *f*. In cases where the generation of alternative attack signatures fails for any of the conditions of *f*, the entire set of partial signatures that might have been generated from *f* up to that point is disregarded. Note that the algorithm might return an empty list for attack signatures for a formula if there are no such signatures.

The main difference between the above algorithm and the algorithm for the abductive generation of diagnostic explanations presented in [4] is that the algorithm for the generation of attack signatures, does not take into account only grounded predicates with specific time ranges as the algorithm for the generation of diagnostic information. Instead it deals with ungrounded predicates and time variables which are constained by symbolic constraints expressed as linear inequalities over other time variables instead of having specific numeric boundaries. Thus, it needs to perform symbolic computations to establish if sets of such constraints are consistent (see [23] for a more detailed discussion of this issue).

4.3. Example

As an example of the derivation of an attack signature for a monitoring rule abductive reasoning consider the monitoring policy of Figure 16. The monitoring rule in this policy is expressed by the EC formula 7:

```
R =
(∀ _ag : Doctor; _a: Patient;  t : Time)
HoldsAt (ExposedPatient (_ag, _a), t) ⇒
HoldsAt (AuthorisedPatient(_ag, _a), t)
```

According to this formula when the medical file of a patient *_a* is exposed to a doctor *_ag*, *_ag* must be authorised to have access to the file.

An attack signature can be generated for this monitoring rule based on the algorithm of Figure 19 assuming that the inputs to it are:

- The monitoring rule R above
- The monitoring policy consisting of the formulas (8)–(13) in Figure 16; and
- The following two EC's axioms:

```
EC1:  HoldsAt(f,t)     ⇐     Initially(f) ∧ ¬∃e,t1 Happens(e,t1, R(0,t))
      ∧ Terminates(e,f,t1)

EC2:  HoldsAt(f,t2)    ⇐     Happens(e,t1,R(t1,t1)) ∧ Initiates(e,f,t1) ∧
      (t1 < t2)  ∧ ¬∃e,t1 Happens(e,t1, R(0,t)) ∧ Terminates(e,f,t1)
```

Based on the above inputs the algorithm will generate the attack signature for R through the following steps.

Step 1: Obtain goal G by negating the monitoring rule R:

```
¬R   →    → (∃ _ag : Doctor; _a: Patient;  t : Time)
          ¬( ¬HoldsAt (ExposedPatient (_ag, _a), t) ∨
          HoldsAt (AuthorisedPatient(_ag,_a), t))

     →    (∃ _ag : Doctor; _a: Patient;  t : Time)
          HoldsAt(ExposedPatient(_ag, _a), t) ∧
          ¬HoldsAt (AuthorisedPatient(_ag, _a), t)                   (R1)
```

Step 2: Generate plan for the first conjunct *HoldsAt(ExposedPatient(_ag, _a), t)* using assumption (11) in the monitoring theory in Figure 16; that is the formula:

Assumption (11):

```
    (∀ _ag : Doctor; _a: Patient; _pdt : PDType; t : Time)
    Happens(GetPD(_ag, _a, _pdt), t, R (t, t)) ∧
    HoldsAt(AvailablePDPatient(_pdt), t) ∧
    ((∀ _pdt2:PDType) _pdt ≠ _pdt2 ∧ HoldsAt(AvailablePDPatient (_pdt2), t) ⇒
    HoldsAt(KnowsPDPatient (_ag, _a, _pdt2), t))
    ⇒ Initiates(GetPD(_ag, _a, _pdt), ExposedPatient (a, ag), t)
```

From axiom EC2 and this assumption, R1 is transformed into:

```
R1    →     (∃ _ag : Doctor; _a: Patient; _pdt:PDType; t: Time)
            (Happens(GetPD(_ag, _a, _pdt), t, R (t, t))
                  ∧ HoldsAt(AvailablePDPatient(_pdt), t)
                  ∧ (∀ _pdt2:PDType) (_pdt ≠ _pdt2 ∧
                  HoldsAt(AvailablePDPatient(_pdt2), t)
                  ⇒ HoldsAt(KnowsPDPatient(_ag, _a, _pdt2), t))
            ∧ ¬HoldsAt(AuthorisedPatient(_ag, _a), t)

      →     (∃ _ag : Doctor; _a: Patient; _pdt:PDType;  t : Time)
            (Happens(GetPD(_ag, _a, _pdt), t, R (t, t))
            ∧ HoldsAt(AvailablePDPatient(_pdt), t)
            ∧ ((∀ _pdt2:PDType)
                  ¬(_pdt ≠ _pdt2 ∧ HoldsAt(AvailablePDPatient(_pdt2), t))
                  ∨ HoldsAt(KnowsPDPatient(_ag, _a, _pdt2), t))
            ∧ ¬HoldsAt(AuthorisedPatient(_ag, _a), t)                (R2)
```

Step 3: Generate plan for the conjunct *HoldsAt(KnowsPDPatient(_ag, _a, _pdt2), t1)* in R2 using assumption (10) in the monitoring theory in Figure 16; that is the formula:

Assumption (10):

```
    ∀ _ag : Doctor; _a: Patient; _pdt : PDType; t : Time)
     Happens(GetPD(_ag, _a, _pdt), t, R (t, t))
     ∧ HoldsAt(AvailablePDPatient(_pdt), t)
     ⇒ Initiates(GetPD(_ag, _a, _pdt), KnowsPDPatient(_ag, _a, _pdt), t)
```

From axiom EC2 and this assumption, R2 is transformed into the following formula:

```
R2   →     (∃ _ag : Doctor; _a: Patient;  _pdt : PDType; t, t1 : Time)
           (Happens(GetPD(_ag, _a, _pdt), t, R (t, t))
           ∧ HoldsAt(AvailablePDPatient(_pdt), t)
           ∧ (∀ _pdt2:PDType)
           (¬(_pdt ≠ _pdt2 ∧ HoldsAt(AvailablePDPatient(_pdt2), t1))
                 ∨ (Happens (GetPD(_ag, _a, _pdt2), t1, R(t1,t1)) ∧ (t1 < t)
                       ∧ HoldsAt(AvailablePDPatient(_pdt2),t))))
           ∧ ¬HoldsAt(AuthorisedPatient(_ag, _a), t)                 (R3)
```

Note that in the formula R3 above the constraint *t1* < *t* arises due to axiom EC2 and that since there are no terminating predicates in the theory of Figure 16 we do not need to consider the inexistance of an event which could terminate this fluent.

Step 4: Generate plan for predicate *HoldsAt(AvailablePDPatient(_pdt), t))* in R3 using the axiom EC1 (again, ignoring the terminating part of the axiom):

```
R3   →     (∃ _ag : Doctor; _a: Patient; _pdt : PDType;  t, t1 : Time)
           (Happens(GetPD(_ag, _a, _pdt), t, R (t, t))
           ∧ Initially(AvailablePDPatient(_pdt))
           ∧ (∀ _pdt2:PDType)
           (¬(_pdt ≠ _pdt2 ∧ HoldsAt(AvailablePDPatient(_pdt2), t1))
                 ∨ (Happens(GetPD(_ag, _a, _pdt2), t1, R(t1,t1)) ∧ (t1 < t)
                       ∧ HoldsAt(AvailablePDPatient(_pdt2),t))))
           ∧ ¬HoldsAt(AuthorisedPatient(_ag, _a), t)                 (R4)
```

Similarly, the predicate *HoldsAt(AvailablePDPatient(_pdt2),t)* can be replaced (twice) in R4 based on EC1:

```
R4   →     (∃ _ag : Doctor; _a: Patient; _pdt : PDType;  t, t1 : Time)
           (Happens(GetPD(_ag, _a, _pdt), t, R (t, t))
           ∧ Initially(AvailablePDPatient(_pdt))
           ∧ (∀ _pdt2:PDType)
           (¬(_pdt ≠ _pdt2 ∧ Initially(AvailablePDPatient(_pdt2))
                 ∨ (Happens (GetPD(_ag, _a, _pdt2), t1, R(t1,t1)) ∧ (t1 < t)

                       ∧ Initially(AvailablePDPatient(_pdt2))))
           ∧ ¬HoldsAt(AuthorisedPatient(_ag, _a), t)                  (R5)
```

Step 5: Generate different instantiations of R5 using the following assumptions in the monitoring theory in Figure 16 and the axiom EC1:

Assumption (12):

```
Initially(AvailablePDPatient(PD1))
```

Assumption (13):

```
Initially(AvailablePDPatient(PD2))
```

The first alternative instantiation is generated by substituting *Initially(AvailablePDPatient(PD1))* for *Initially(AvailablePDPatient(_pdt)* and *Initially(AvailablePDPatient(PD2))* for *Initially(AvailablePDPatient(_pdt2)*, respectively, thus considering the instantiation of *{(_pdt, PD1), (_pdt2, PD2)}*:

```
R5    →     (∃ _ag : Doctor; _a: Patient;  t, t1: Time)
            (Happens(GetPD(_ag, _a, PD1), t, R (t, t))
            ∧ Initially(AvailablePDPatient(PD1))
            ∧ (∀ _pdt2:PDType)
            ¬(PD1 ≠ PD2 ∧ Initially(AvailablePDPatient(PD2)))
                  ∨ (Happens(GetPD(_ag, _a, PD2), t1, R(t1,t1)) ∧

                   Initially(AvailablePDPatient(PD2)) ∧ (t1 < t))
            ∧ ¬HoldsAt(AuthorisedPatient(_ag, _a), t)                  (R6)

R6    →     (∃ _ag : Doctor; _a: Patient;  t, t1: Time)
            (Happens(GetPD(_ag, _a, PD1), t, R (t, t))
            ∧ (Happens(GetPD(_ag, _a, PD2), t1, R(t1,t1)) ∧  (t1 < t))
            ∧ ¬HoldsAt(AuthorisedPatient(_ag, _a), t)                  (R7)
```

The transformation of R6 to R6 is because *Initially(AvailablePDPatient(PD1)* is a given fact in the monitoring policy and therefore always True as well as PD1 ≠ PD2.

(PD1 ≠ PD2 ∧ Initially(AvailablePDPatient(PD2)))

is True (Initially(AvailablePDPatient(PD2)) is part of the monitoring theory) and therefore

¬(PD1 ≠ PD2 ∧ Initially(AvailablePDPatient(PD2))) is always False.

Also, *Initially(AvailablePDPatient(PD1))* and *Initially(AvailablePDPatient(PD2))* are always True and therefore can be removed.

The second alternative instantiation is generated by substituting *Initially(AvailablePDPatient(PD2))* for *Initially(AvailablePDPatient(_pdt)* and *Initially(AvailablePDPatient(PD1))* for *Initially(AvailablePDPatient(_pdt2)*, respectively, thus considering the instantiation of *{(_pdt, PD2), (_pdt2, PD1)}*. This results into the formula:

```
R5    →      (∃ _ag : Doctor; _a: Patient;  t, t1: Time)
             (Happens(GetPD(_ag, _a, PD2), t, R (t, t))
             ∧ Initially(AvailablePDPatient(PD2))
             ∧ (∀ _pdt2:PDType)
             ¬(PD2 ≠ PD1 ∧ Initially(AvailablePDPatient(PD1)))
                   ∨ (Happens(GetPD(_ag, _a, PD1), t1, R(t1,t1)) ∧
                     Initially(AvailablePDPatient(PD1)) ∧ (t1 < t))
             ∧ ¬HoldsAt(AuthorisedPatient(_ag, _a), t)

      →      (∃ _ag : Doctor; _a: Patient;  t, t1: Time)
             (Happens(GetPD(_ag, _a, PD2), t, R (t, t))
             ∧ (Happens (GetPD(_ag, _a, PD1), t1, R(t1,t1)) ∧ (t1 < t))
             ∧ ¬HoldsAt(AuthorisedPatient(_ag, _a), t)                (R8)
```

<u>Step 6:</u> Generate different plans for *¬HoldsAt(AuthorisedPatient(_ag, _a), t)* in R7 using assumption (8) in the monitoring theory in Figure 16, that is the formula:

Assumption (8)

```
     (∀ _ag : Doctor; _a: Patient;  t : Time)
     Happens(AuthoriseAccess(_ag, _a), t, R (t, t))
     ⇒ Initiates (AuthoriseAccess(_ag, _a), AuthorisedPatient(_ag, _a), t)
```

```
R7    →      (∃ _ag : Doctor; _a: Patient;  t, t1, t2: Time)
             (Happens(GetPD(_ag, _a, PD1), t, R (t, t))
             ∧ (Happens(GetPD(_ag, _a, PD2), t1, R(t1,t1)) ∧  (t1 < t)
             ∧ ¬Happens(AuthorisedAccess(_ag, _a), t2) ∧ (t2 < t)       (R9)
```

Similarly from R8 we can generate

```
R7    →      (∃ _ag : Doctor; _a: Patient;  t, t1, t2: Time)
             (Happens(GetPD(_ag, _a, PD2), t, R (t, t))
             ∧ (Happens(GetPD(_ag, _a, PD1), t1, R(t1,t1)) ∧  (t1 < t)
             ∧ ¬Happens(AuthorisedAccess(_ag, _a), t2) ∧ (t2 < t)       (R10)
```

Formulas R9 and R10 constitute two alternative ways of failing R or, equivalently, two alternative attack signatures. Their disjunction constitutes part of overall attack signature for the rule (an alternative attack signature can be generated if at Step 2 above we use assumption (9) for the *HoldsAt(ExposedPatient(_ag, _a), t)* conjunct which would generate a formula representing a direct access attack). In the next section we show how we use the attack signature and run-time observations to compute the likelihood of threats.

5. Evaluation of threat likelihood

5.1. Overview

This section describes the estimation of the level of threat that is faced by a system given the rules and attack signatures which are being monitored. As discussed earlier, an S&D threat in SERENITY is defined as the potential for the violation of a monitoring rule or an attack signature.

The level of threat for a system monitored by the SERENITY framework is a measure of the likelihood of a potential occurrence of a violation of an S&D monitoring rule or attack signature defined for this system. This likelihood becomes 1 when a violation has indeed occurred given the runtime events that have been recorded in the log of the SERENITY framework. Thus, an S&D threat with a likelihood that is equal to 1 is equivalent to an S&D violation as defined in [1] .

Our objective, however, in threat detection is to estimate the threat likelihood for a particular rule or attack signature before a violation occurs. To measure this likelihood we use beliefs founded in reasoning framework of the Dempster Shafer theory of evidence [3]. This is due to the need to cope with uncertainty regarding the events that have been seen at the different monitoring states of the system, which makes the use of classic probabilistic reasoning inappropriate for our needs.

In the following, we introduce these beliefs and the reasoning underpinning their computation following an overview of the uncertainties regarding the events which are made available to the monitor.

5.2. Event uncertainties

Monitoring in SERENITY is based on runtime events which are produced by the components of the system that is being monitored. These events are captured by captors (sensors) associated with the components of the system that is being monitored. Captors transmit the events that they intercept to the framework through different communication channels. In this event capturing and communication setting, there are two forms of uncertainty associated with events:

1. There might be events which have occurred but not received yet by the monitor
2. There might be events which have been received by the monitor without having really occurred at the component where they appear to be coming from

The former type of uncertainty arises due to the use of distributed event captors which communicate with the SERENITY monitoring framework via different channels. Individual channels are assumed to provide a *first-in-first-out* (FIFO) transmission (i.e., the events arive at the destination of the channel in exactly the same order in which they are dispatched at the channel's origin). Therefore, the reception of an event from a channel informs the monitor that all the events that preceed this event, i.e., all the events with a timestamp earlier than the timestamp of the current event, have also been received[4]. In this case, the monitor may still in the future receive other events from the channel with the same timestamp as the last event seen from it but no events with a later timestamp. Generally, however, the monitor may be getting events from more than one sources using different

[4] As we describe in [2], even if the events which are transmitted through the same channel are captured at different sources, they have timestamps which are expressed using the monitor's clock (this is achieved because sources synchronise their clocks with the clck of the monitor by using the NTP protocol).

communication channels to send them to the monitor. When this is the case, the order of the arrival of events at the monitor cannot be guaranteed to be the same as the order of their dispatch from the sources where the events were captured. This is because, as we have discussed in [2], in the general case of multiple communication channels, there can be different transmission delays on each one of the channels. Thus, the reception of an event E_i from channel c_i timestamped at time t (translated to the monitor's clock) does not inform the monitor that all events E_j coming from a different channel c_j and timestamped with $t' < t$ have already been received. In such cases, upon the reception of E_i the monitor will be unable to make a decision about the occurrence or not of the event E_j if the latest event seen from the c_j channel has a timestamp that is less than t.

In the presence of such uncertainty, the monitor would have to wait until it can guarantee that an E_j event could not have been observed. This would be possible when the monitor receives an event from c_j that has a timestamp t'' that is greater than t'. However, as our objective in threat detection is to generate early forecasts of the potential of violations of rules, delaying monitoring decisions is not desirable for this purpose[5]. It should also be noted that this type of event uncertainty increases when channels fail temporarily or become the subject of attacks that prevent them from sending events. In such cases, delaying the estimation of the likelihood of threats involving E_j events might not be desirable as an early estimate might be necessary for taking preemptive action to avoid problems arising from the violation of the relevant rules. Thus, the mechanisms used by the monitor to estimate threat likelihood should be able to reason about an event without certainty about its presence or absence.

The second form of event uncertainty arises due to the possibility of having events in the log of the monitor which are the result of an attack. An event captor may, for example, be attacked and start producing events which are not coming from the system component which it is asscociated with. In such cases, the genuineness of the events involved in a potential violation of a rule needs to be assessed before estimating the likelihood of a threat. The diagnosis mechanisms of the SERENITY monitoring framework [4] can be used for this assessment. The basic principle realised by these mechanisms is that an event is considered to be genuine only if an explanation can be found for it and this explanation is confirmed by other runtime events. However, the confirmation or not of an event explanation depends on whether its expected consequences match with runtime events and, therefore, due to possible delays in the transmission of events there might be explanation consequences which are neither confirmed nor disconfirmed. Thus, there is further uncertainty associated with the validity of the explanations of events. To deal with the latter form of uncertainty whilst producing diagnostic information, we adopted the Dempster-Shafer theory of evidence. And for the same reasons, this theory is adopted as the belief reasoning framework that underpins threat detection.

5.3. Threat likelihood

A threat is defined in reference to a specific S&D monitoring rule or attack signature. In SERENITY, both rules and attack signatures are expressed as *Event Calculus* (EC) formulas, formed by predicates which refer to occurences of system events and conditions regarding the state of the system that is being monitored (aka *fluents*). Since at a representation level both monitoring rules and attack signatures are exactly the same, in the remaining of this document we will refer to both of them as "rules".

[5] The monitor waits only for the detection of definite S&D violations but not for threats.

The predicates involving events – *Happens(e(sender,recipient,type,src,operation-name,op-params,timestamp), t, R(t1, t2))* – are effectively event parameters, with associated constraints on their sender, recipient, type, source, operation name and parameters, as well as the temporal constraints on the time *t* when an event is observed (this is the translation of the event's timestamp into the monitor's clock). When events arrive at the monitor and they can be unified with a predicate of a rule, the monitor performs the unification and generates a new rule template to represent the instantiation of the rule by the particular event. If more than one unification of an event with a rule is possible (this is the case when the event is unifiable with different predicates of the same rule), then more than one templates are created; one for each of the possible unifications. And, if at some point one of these templates has all its predicates unified and evaluates to *False*, the rule that is instantiated by the template is known to be violated. Thus, the threat likelihood of a rule R at each stage of the monitoring process needs to be a measure of the likelihood that any of the current templates (partial instantiations) of R will evaluate to *False* in a future state rather than a measure over just one template. To reflect this, we define the threat likelihood of a rule as:

Definition 1: The threat likelihood of a rule R is the belief in the potential of the occurrence of a violation of any of the templates of R that have been currently generated.

To clarify this definition, consider the case of the following monitoring rule:

Rule-1:

$\texttt{Happens}(E_1,\ t_1,\ \Re(t_1,t_1)) \wedge \texttt{Happens}(E_2,\ t_2,\ \Re(t_1,t_1+a)) \Rightarrow \texttt{Happens}(E_3,t_3,\Re(t_2,t_2+b))$

The above rule states that when an event E_1 occurs at some time point t_1 and another event E_2 occurs at some time point t_2 within the time range $(t_1, t_1+a]$ then a third event of type E_3 should take place within the time range $(t_2, t_2+b]$. In the case of this rule, the likelihood of a threat needs to be estimated in the following states of the monitoring process:

3. when one (or more) events that match E_1 but no events that match E_2 have been received by the monitor
4. when one (or more) events that match E_2 but no events that match E_1 have been received by the monitor
5. when one (or more) events that match E_1 and one (or more) events that match E_2 have been received by the monitor
6. when one (or more) events that match E_1 and one (or more) events that match E_2 have been received by the monitor and an event E has been received from the event captor that should send E_3 indicating that E_3 will not arrive. The absence of the E_3 event (or, equivalently, $\neg E_3$) can be derived from E in this case if the monitor receives E from the same event captor that should sent E_3 with a timestamp t' that is greater than t_2+b. This derivation is based on the application of the principle of negation as failure (NAF) − since t' > t_2+b the monitor knows that it cannot receive any event with a timestamp earlier than t' and therefore ealier than t_2+b. Thus, it knows that it cannot also receive an E_3 event and, hence, deduces its absence.

In case (1) above, the threat likelihood for *Rule-1* will be a measure of the belief that the event E_1 which has already matched the rule is genuine and an event of type E_2 matching the rule will occur within the time range $(t_1, t_1+a]$ and no event matching E_3 will occur in the range $(t_1, t_1+b]$. In case (2), the threat likelihood for *Rule-1* will be a measure of the belief that the event E_2 which has already matched the rule is genuine and an event of type E_1 matching the rule has already occurred within the time range *[max(t_2 − a, latestTime(captor(E_1))), t2)* but not received by the monitor and an event matching E_3 will occur in the range $(t_2, t_2+b]$. In case (3), the threat likelihood of *Rule-1* threat will be a measure of the belief that the E_1 and E_2 events that match the rule are genuine and an event of type E_3 matching the rule will occur in the time range $(t_2, t_2+b]$. Finally, in case (4), the threat likelihood of *Rule-1* will be a measure of the belief that the E_1 and E_2 events that match the rule are genuine and the event of type E which provided the basis for deriving $\neg E_3$ is genuine and, therefore, the application of the NAF principle in deriving $\neg E_3$ is valid. The functions used to measure these beliefs are discussed in the following section.

5.4. Basic probabilities for event occurrences

As indicated for the example of *Rule-1* in Section 5.3. the calculation of the threat likelihood of a rule requires the measurement and combination of beliefs of three basic types: (i) beliefs in the genuiness of events that have been recorded in the log of the monitor, (ii) beliefs in the possibility of occurrence of an event of a specific type within a time range that is determined by another event, (iii) beliefs in the validity of the derivation of the negation of an event when another event's occurrence indicates that the time range whithin which the former event should occurred has elapsed.

The beliefs of the former type can be computed on the basis of the diagnostic information that the SERENITY monitoring framework generates for individual events when it needs to diagnose the reasons underpinning a rule violation. More specifically, as it has been discussed in [4], the SERENITY monitoring framework calculates a belief in the genuineness of an event as a belief that there is at least one explanation of this event and this explanation has expected consequences which are confirmed by matching other runtime events recorded in the log of the framework. The framework for the calculation of beliefs in the genuineness of individual events in SERENITY is that of the Dempster Shafer theory of evidence [3], referred to as "D-S theory" in the following. The adoption of the D-S theory for this purpose has been dictated by the need to reason on the basis of uncertainty about the occurrence or not of events which have not been recorded yet in the log of the monitoring framework. The same need arises in the case of threat beliefs and therefore the D-S theory has been selected as the belief computation framework for threats.

The definition and combination of beliefs in the D-S theory requires the existence of a *frame of discernment* representing a set of mutually exclusive propositions that beliefs will need to be computed for. Given such a frame of discernment θ, a *basic probability assignment (BPA)* or *mass function* in the D-S theory is a function from the powerset of θ, $\wp(\theta)$, to the range [0…1] or, equivalently, a function *m* of the following form:

(a1) $m: \wp(\theta) \rightarrow [0\ldots1]$

A function *m* of this form provides a measure of basic 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 (splitted) 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:

(a2) $m(\emptyset) = 0$

(a3) $\Sigma_{P\subseteq\theta}\, m(P) = 1$

The first of these axioms prevents basic probability assignments from assigning a non zero basic beliefs to an empty proposition set. The second axiom requires that the sum of the basic beliefs 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 D-S theory induces a unique "belief" function *Bel* which is defined as:

(a4) $Bel: \wp(\theta) \rightarrow [0\ldots1]$

(a5) $Bel(A) = \Sigma_{B \subseteq A}\, m(B)$

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:

(a6) $Bel(\varnothing) = 0$

(a7) $Bel(\theta) = 1$

(a8) $\Sigma_{I \subseteq \{1,\ldots,n\},\ and\ I \neq \theta}\, (-1)^{|I|+1} Bel(\cap_{i \in I} P_i) \leq Bel(\cup_{i=1,\ \ldots,\ n} P_i)$ where $n = |\wp(\theta)|$ and $P \subseteq \theta$, $(i=1,\ldots,n)$

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

(a9) $m_1 \oplus m_2\ (P) = (\Sigma_{X \cap Y = P}\, m_1(X) \times m_2(Y)) / (1 - k_0)$

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

k_0 in *(a9)* 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.

In the case of rule threats, a *frame of discernment* needs to describe all the different combinations of occurrences and non occurrences of the events involved in a rule. For a rule with n different events, these combinations can be described by using a vector of n Boolean variables $<e_1, e_2, \ldots, e_n>$ where the variable e_i denotes whether or not the event E_i in the rule has occurred or not by taking the values 1 and 0, respectively. A vector of such variables will (by convention) denote the conjunction of the elementary propositions denoted by the values of its variables. For a rule with n different events there will be 2^n different such vectors, which will constitute the frame of discernment θ of the rule, and the powerset of this frame will have 2^{2^n} elements.

Thus, in the case of *Rule-1* above, for example, the frame of discernment will contain the following vectors:

$$\theta_R = \{<e_1,e_2,e_3>, <e_1,e_2,\underline{e_3}>, <e_1,\underline{e_2},e_3>, <e_1,\underline{e_2},\underline{e_3}>, <\underline{e_1},e_2,e_3>, <\underline{e_1},e_2,\underline{e_3}>, <\underline{e_1},\underline{e_2},e_3>, <\underline{e_1},\underline{e_2},\underline{e_3}>\}^{6}$$

and the powerset of θ_R will contain all the possible subsets of it − 256 (2^{2^3}) elements in total. Certain elements of this powerset will represent complex propositions which will be relevant to the estimation of threat likelihood whilst others not. For example,

— the proposition that the event E_1 has occurred will be represented by the set

$\{<e_1,e_2,e_3>, <e_1,e_2,\underline{e_3}>, <e_1,\underline{e_2},e_3>, <e_1,\underline{e_2},\underline{e_3}>\}$, and

[6] Underlined variables are variables having a 0 value and non underlined variables are variables having 1 as a value.

— the proposition that the event E_1 has occurred along with an E_2 event in the time range that it determines will be represented by the set

$\{<e_1,e_2,e_3>, <e_1,e_2,\underline{e_3}>\}$.

Given the frame of discernment θ_R for a rule R, we define the basic probability assignments for the events of the rule as follows:

Definition 2: The basic probability assignment m_i of an event E_i is defined as:

$$m_i(X) = \begin{cases} k_i = \Sigma_{J\subseteq\{1,\ldots,L\} \text{ and } J\neq\varnothing}(-1)^{|J|+1}\{\Pi_{j\in J}\, m_{ij}(\mathrm{Valid}(\Phi_{ij}))\} & \text{if } X = \{<e_1, e_2, \ldots, e_n> \mid e_i = 1\} \\ k_i' = \Pi_{j=1,\ldots,L}\, m_{ij}(\neg \mathrm{Valid}(\Phi_{ij})) & \text{if } X = \{<e_1, e_2, \ldots, e_n> \mid e_i = 0\} \\ 1 - k_i - k_i' & \text{if } X = \theta \\ 0 \quad \text{Otherwise} & \end{cases}$$

where

- Φ_{ij} (j=1,…,L) are the alternative explanations that can be constructed for the event e_i based on the approach that we have described in [4];
- $\mathrm{Valid}(\Phi_{ij})$ denotes whether the explanation Φ_{ij} is valid; and
- $m_{ij}(\mathrm{Valid}(\Phi_{ij}))$ is the basic probability assignment in the validity of an explanation Φ_{ij}, which, as we discuss in [4] (see definition 2, pg. 31), is defined as

  $m_{ij}(\mathrm{Valid}(\Phi_{ij}))$ = #observed-consequences-of-Φ_{ij} / #expected-consequences-of-Φ_{ij}

According to the above definition, the basic probability assignment in an event E_i which has been recorded in the log of the monitor is defined as the basic belief in the existence of at least one valid explanation for E_i. Thus, our approach is to question the validity of each of the events which have appeared in the event log and accept as valid only those events for which at least one explanation can be found, and this explanation is confirmed by the presence of other events in the log that match its expected consequences. It should be noted that the validity of explanations is also assessed probabilistically using the basic probability assignment m_{ij} in *Definition 2*. This basic probability assignment measures the likelihood of the validity of an explanation as the proportion of the expected consequences of the explanation which match events recorded in the log of the monitoring framework.

The second type of basic probability assignments that we use for the assessment of the likelihood of threats are assignments that provide belief measures in the occurrence or not of an event e_i when another event e_j has occurred. These assignments are similar to conditional probabilities in the classic probability theory but in our framework are treated as basic belief functions which are defined as follows:

Definition 3: The basic probability assignment $m_{i|j}$ in the occurrence of an event E_i within the time range determined by another valid event E_j is defined as:

$$m_{i|j}(X) = \left\{ \begin{array}{ll} k_{ij} = \Sigma_{e\in Log(E_j)} m(e)\{\Sigma_{I\subseteq Log(E_i \mid e) \text{ and } I\neq\varnothing}(-1)^{|I|+1}\{\Pi_{Ei\in I} m_i(E_i))\}\}/\Sigma_{e\in Log(E_j)} m_j(e) & \\ & \text{if } X = \{<e_1,e_2,\ldots,e_n> \mid e_j = 1 \text{ and } e_i=1\} \\ k_{ij}' = \Sigma_{e\in Log(E_j)} m(e)\{\Pi_{Ei\in Log(E_i \mid e)} m_i(\neg E_i))\}\}/\Sigma_{e\in Log(E_j)} m(e) & \\ & \text{if } X = \{<e_1,e_2,\ldots,e_n> \mid e_j = 1 \text{ and } e_i=0\} \end{array} \right.$$

$$\begin{array}{ll} \vert\ 1 - k_{ij} - k_{ij}' & \text{if } X = \theta \\ \lfloor\ 0 & \text{Otherwise} \end{array}$$

where

- *$Log(E_j)$* is the set of the events of type E_j in the event log up to the time point when $m_{j|i}$ is calculated
- *$Log(E_i|e)$* is the set of the events of type E_i in the event log that have occurred within the time period determined by *e* and up to the time point when $m_{i|j}$ is calculated
- m(e) is the basic probability assignment $m_j(e)$ defined by Definition 2 in the case of non negated events E_j or the basic probability assignment $m^{NAF}_{j|u}$ defined in Definition 4 for negative events $\neg E_j$

$m_{i|j}(X)$ measures the belief in the occurrence of a valid event of type E_i within the time range determined by events of type E_j, as the average belief of seeing a valid event of type E_i within the time range determined by a valid event of type E_j. More specifically, for each occurrence of an E_j event, *$m_{i|j}(X)$* calculates the basic probability of seeing at least one valid event of type E_i within the period determined by the E_j event. Assuming that the set of such E_i events is *$Log(E_i|e)$*, this basic probability is calculated by the formula $\Sigma_{I\subseteq Log(E_i|e) \text{ and } I\neq\emptyset} (-1)^{|I|+1}\{\Pi_{Ei\in I} m_i(E_i))\}$. This formula measures the basic probability of at least one of the events in *$Log(E_i|e)$* being a valid event, i.e., being an event that has at least one explanation confirmed by other events in the log of the system, and uses the basic probability assignments of individual events *$m_i(E_i)$* defined in Definition 2 above for positive events or the basic probability assignment $m^{NAF}_{j|u}$ for negative events $\neg E_j$ which are themselves established by a third event E_u (see Definition 4 below). Thus, *$m_{i|j}(X)$* discounts occurrences of events of type *E_i* which are not assessed as valid. Also, according to the definition of *$m_{i|j}(X)$*, the more the occurrences of valid events of type *E_i* within the period determined by an *E_j* event, the higher the belief in the occurrence of at least one such valid event in the relevant period will be. Finally, it should be noted that *$m_{i|j}(X)$* takes also into account the basic probability of the validity of each occurrence of an event of type *E_j* (i.e., *$m_j(e)$*) and uses it to discount the evidence collected from cases of *E_j* events which are not genuine (i.e., they do not have at least one confirmed explanation themselves).

The basic probability assignments that we have introduced above provide beliefs in the occurrence or not of an event that has occurred or that depends on another event which has happened. They do not cover, however, the case where the absence of an event is deduced by the NAF principle. As we discussed earlier, the SERENITY monitor uses this principle to deduce the absence of an event *E* (i.e. ¬ *E*) that is expected to occur within a specific time range $[t_L, \ldots, t_U]$ when it receives another event *E'* from the same event captor that should sent *E* with a timestamp *t'* that is greater than t_U ($t' > t_U$) and has not received *E* up to that point. Considering, however, that the event *E'* which triggers the application of the NAF principle might not be a genuine event itself, in such cases it is necessary to estimate the belief in ¬*E*. This belief is provided by the following basic probability assignment:

Definition 4: The basic probability assignment $m_{i|j}$ in the absence of an event E_i or, equivalently, $\neg E_i$ due to the application of the NAF principle when another event E_j occurs is defined as:

$$m^{NAF}_{i|j}(X) = \begin{cases} m_j(E_j) & \text{if } X = \{<e_1,e_2,\ldots,e_n> \mid e_j = 1 \text{ and } e_i=0\} \\ 1 - m_j(E_j) & \text{if } X = \theta \\ 0 & \text{Otherwise} \end{cases}$$

where

- $m_j(E_j)$ is the basic probability assignment in the genuineness of the event E_j defined by Definition 2.

According to this definition, the basic probability of the genuineness of the event E_j which has triggered the application of the NAF principle becomes also the basic probability of the absence of E_i. Note, however, that according to Definition 4, the probability that *$m_j(E_j)$* generates for the non genuineness of the event E_j is not evidence in favour or against E_i. This is because believing that E_j is not a genuine event is equivalent to believing that E_j has not occurred and in this case, although $\neg E_i$ cannot be derived by the NAF principle there is uncertainty regarding the occurrence of E_i. Hence, $m^{NAF}{}_{i|j}(X)$ assigns the remaining belief $1 - m_j(E_j)$ to the entire frame of discernment θ which represents the proposition $E_i \vee \neg E_i$.

5.5. Combination of basic probability assignments through DS belief networks

In Section 5.4. we have defined the general functions which are used for the calculation of the basic probability assignments for event occurrences. The exact functions, however, that will be required in the case of each rule need to be determined by analysing the time dependencies between the events involved in the rule. This involves both the functions that will be required for the calculation of basic probabilities in the genuineness of individual events and the functions required for the calculation of basic probability assignments to expected occurrences of events which are constrained by other events in a rule. To clarify this point, consider again Rule-1. As we discussed in Section 5.3. the assessment of the threat likelihood of this rule will need to be carried out in the following states of the monitoring process:

- Case 1: when one (or more) events that match E_1 but no events that match E_2 have been received by the monitor
- Case 2: when one (or more) events that match E_2 but no events that match E_1 have been received by the monitor
- Case 3: when one (or more) events that match E_1 and one (or more) events that match E_2 have been received by the monitor
- Case 4: when one (or more) events that match E_1 and one (or more) events that match E_2 have been received by the monitor and $\neg E_3$ is derived by the NAF principle due to another event E_u. In this case an estimation of the threat likelihood will still be necessary due to the uncertainty about the genuineness of the event E_u.that has led to the derivation of $\neg E_3$.

Thus, it will be necessary to use and combine the following basic belief functions for this rule: m_1, m_2, m_3, $m_{1|2}$, $m_{2|3}$, $m_{2|1}$ and $m^{NAF}{}_{3|U}$ These functions will need to be combined in the above cases as follows:

- In Case 1: $(m_1 \oplus m_{2|1}) \oplus m_{3|2}$
- In Case 2: $(m_2 \oplus m_{1|2}) \oplus m_{3|2}$
- In Case 3: $(m_1 \oplus m_2) \oplus m_{3|2}$
- In Case 4: $(m_1 \oplus m_2) \oplus m^{NAF}{}_{3|u}$

The combinations of the above basic probability functions can be obtained through the application of the rule of the orthogonal sum of the D-S theory (see formula (a9)). The different cases of combining event evidence along with the basic probability functions that will be used in each of them can be determined by analysing the time dependencies between the events of a rule. To represent these cases for a monitoring rule, we construct a graph with the different events as vertices and *directed, labelled* edges among them indicating dependencies between their time variables. These edges can be derived from the time variables which constrain the occurrence of an event, and indicate how evidence can be propagated at runtime by combining the different basic probability assignments that are associated with the observed events. This graph is called "rule belief graph" and the algorithm for constructing it is listed in Figure 20.

According to this algorithm, the events of the given input rule R (i.e., the negated form of a rule R or an attack signature which has been derived from the negated form of a monitoring rule as we discussed in Section 4. are identified initially and a node is constructed to represent the starting point of the accumulation of evidence at runtime (see line 2). Then for each event, the algorithm constructs a node to represent its occurrence at runtime (line 4) and identifies the dependencies of the event to other events (line 5). At this step an event E_j is taken to depend on all other events E_i whose time variables appear in the expressions that define the lower and upper bound of the time variable of E_j.

After identifying these dependencies, the algorithm creates a directed edge from all the events E_i that an event E_j depends on to E_j (see line 9). These edges indicate the paths for obtaining a basic probability for E_j when any of the events E_i is observed. Also an opposite edge from E_j to each of the events E_i is created if E_j is not a negated event (see lines 10-12). These edges will be used when E_j is observed before E_i in order to indicate how the basic probability of E_i can be computed given E_j (see Case 2 above). Note that no backward edges are constructed from an event E_j to the events that it depends on, if E_j is a negated event (see condition in line 10). This is because, in the monitoring framework of SERENITY, negated events can only be derived through the application of the NAF principle when their ranges have fully determined boundaries (an event expected in a fully determined time range [a,b] is known to not have happened when the monitor receives the first event from its captor with a time stamp $t > b$ without having received the event itself up to that point). Fully detemined boundaries, however, will not be possible to have for E_j unless E_i has already occurred. Hence, it will not be possible to derive the truth value of E_j before that of E_i and therefore compute a basic probability assignment for the latter event based on the basic probability of the former. The label attached by the algorithm on an edge from an event E_i to an event E_j will be $m_{i|j}$, i.e., it will represent the basic probability assignment of observing (or not) E_j given that E_i has already been observed (note that this is different to $m_{j|i}$).

Following the generation of edges between events, the algorithm constructs edges to link the *Start* node of the graph with the nodes representing the different non negated events of the rule (see lines 14−20). These edges are labelled by the basic probability assignment corresponding to the event E_i that they point to (i.e., the basic probability assignment m_i). Negated events, on the other hand, are linked with the *Start* node only if they have a time range defined by constant values at the time of application of the algorithm (i.e., prior to runtime) and, therefore, it will be possible to establish their absence or not prior the seeing any other event at runtime (see conditions in lines 14 and 17) [7].

[7] Such events may typically appear in rules of the form *¬Happens(e1,t1,R(a,b))* ⇒ *Happens(e2,t2,R(t1,t1+c))*. The event *¬Happens(e1,t1,R(a,b))* in this rule has a time range with fully

The edge linking the *Start* node with a negated event E_i is labelled by the basic probability function $m^{NAF}_{i|<x>}$. This function is partially determined as it includes the placeholder <x>. At runtime this placeholder will be bound to the identifier of the event E_j that triggers the application of the NAF principle to derive the absence of E_i creating a fully determined basic probability function $m^{NAF}_{i|j}$ which will be used to estimate the basic probability of $\neg E_i$.

```
Construct_DS_Belief_Graph(R, $DSG_R$)
1.  Find all $n$ events $e_i$ in R /* R is assumed to be the negated form of a monitoring rule or an attack signature */
2.  Construct a node representing the starting point in the assessment of the threat likelihood of R, called "Start" node.
3.  For each event $e_i$ ($i \le n$) do:
4.      Construct a node for $e_i$ and store the mapping of the time variable of this event as $M(t_i)= e_i$ (i.e. store the fact that time variable $t_i$ has been used to declare the time of occurrence of event $e_i$.)
5.      Build a list $TVARS_i$ of all time variables $t_k$ appearing in the lower and upper bound of the time variable $t_i$ of $e_i$.
6.  end for
7.  For each event $e_i$ ($i \le n$) do:
8.      For each time variable $t \in TVARS_i$ such that $t \ne t_i$ do:
9.          Construct an edge to $e_i$ from $e_p=M(t)$, labelled by $m_{i|p}$, i.e., the basic probability of observing (or not) $e_i$ given $e_p$.
10.         If $e_i$ is not a negated event Then
11.             Construct an edge from $e_i$ to $e_p=M(t)$, labelled by $m_{p|i}$, i.e., the basic probability of observing (or not) $e_p$ given $e_i$.
12.         end if
13.     end for
14.     If $e_i$ is not a negated event then
15.             construct an edge from the "Start" node to $e_i$, labelled by the basic probability $m_i$ of $e_i$.
16.     else /* negated events */
17.             if $e_i$ has a time range defined by constant values then
18.                 construct an edge from the "Start" node to $e_i$, labelled by the basic probability $m^{NAF}_{i|<x>}$ of $e_i$
19.             end if
20.     end if
21. end for
end Construct_DS_Belief_Graph
```

Figure 20. Algorithm for constructing DS belief graphs

---

determined boundaries (a and b) prior to runtime and will remain as a negated event in the negated form of the rule, i.e., $\neg Happens(e1,t1,R(a,b)) \wedge \neg Happens(e2,t2,R(t1,t1+c))$

An example of a DS belief graph is shown in Figure 21. This graph has been built for *Rule-1* above. The graph reflects that the occurrence of E_2 in the rule depends on the occurrence of E_1 since the range of the time variable of E_2 (i.e., $\Re(t_1,t_1+a)$) refers to the time variable of E_1 but not vice versa (the range $\Re(t_1,t_1)$ of t_1 indicates that E_1 is an event with a not constrained time variable). Thus, an edge from E_1 to E_2 labelled by $m_{2|1}$ has been inserted in the graph as well as another edge from E_2 to E_1 labelled by $m_{1|2}$. Similarly, as the time range of the event $\neg E_3$ (i.e., $\Re(t_2,t_2+b)$) refers to the time variable t_2 of the event E_2, the graph contains an edge from E_2 to E_3. It does not contain, however, an edge from $\neg E_3$ to E_2 as the former event cannot be derived by NAF unless E_2 is seen first. Finally, the graph includes edges from the starting node to E_1 and E_2 which complete the graph. These edges are labelled by m_1 and m_2 representing the basic probability functions that are to be used when the occurrence or absence of the events E_1 or E_2 is established from the starting node.

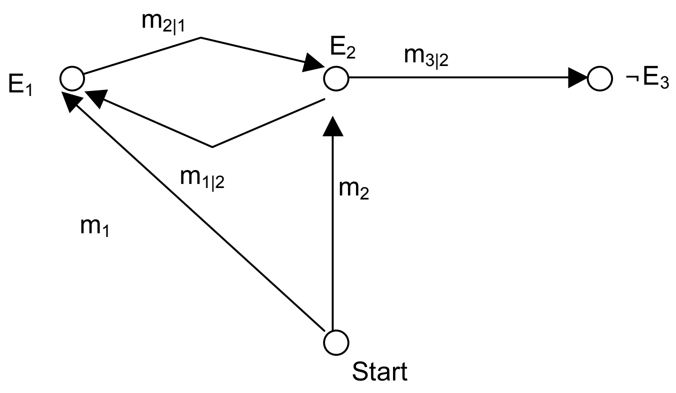

Figure 21. D-S belief graph for Rule-1

The effect of observing the presence or deducing the absence of new events during monitoring is propagated through the DS belief graph of a rule in order to compute the current threat likelihood of the rule. This computation is carried out using the algorithm of Figure 22. More specifically, given a new event E that can be unified with a rule R or can trigger the derivation of a negated event in R by NAF and the DS belief graph for the rule (DSG_R), this algorithm initially identifies the known (KE) and unknown (UE) events in the belief graph DSG_R (see line 1). Then it combines the basic probability assignments of the known events (see lines 4-7), finds the paths from E to the unknown events UE in DSG_R which do not include any events in KE (see line 9), and combines the basic probability assignments identified by the labels of the edges of these paths that have not been combined so far (see lines 12-15). Finally, the algorithm returns the belief assigned to the events in the negation of the rule and the events in the rule (see line 19).

Using the *Compute_Threat_Likelihood* algorithm with the DS belief graph of Rule-1, when an event E_1 becomes known and there are no other known events in graph, the possible paths to unknown events in the network of Figure 21 will be $E_1 \rightarrow E_2$ and $E_1 \rightarrow E_2 \rightarrow \neg E_3$. Given these paths, the threat estimation will derived from the combination of the following basic probability assignments:

Threat = $(m_1 \oplus m_{2|1}) \oplus m_{3|2}\,(E1 \wedge E2 \wedge \neg E3)$

```
Compute_Threat_Likelihood(E_i, DSG_R, R)
1.  Find the sets of the known events KE and the set of the unkown events UE in DSG_R;
2.  m = basic_probability_assignment(<start, E_i>)
3.  CombinedBPA = {}
4.  For each E_k in KE Do /* combine the BPAs of events in KE */
5.      m = m ⊕ basic_probability_assignment(<start, E_k>)
6.      CombinedBPA = CombinedBPA ∪ basic_probability_assignment(<start, E_k>)
7.  end for
8.  For each e_j ∈ UE do
9.      find all the paths from e_i to e_j, which do not include any event in KE and insert them in P_ij
10.     For each p ∈ P_ij do
11.         For each edge L in p Do /* combine the BPAs of paths to unkwown events */
12.             If basic_probability_assignment(L) ∉ CombinedBPA then
13.                 m = m ⊕ basic_probability_assignment(L)
14.                 CombinedBPA = CombinedBPA ∪ basic_probability_assignment(L)
15.             end if
16.         end for
17.     end for
18. end for
19. mark E_i as a known event in DSG_R
20. return (m(events(¬R), m(events(R)))
End Compute_Threat_Likelihood
```

Figure 22. Algorithm for computing threat likelihood

In the case where the event E_2 becomes known first, the set of the unknown events in the network will include the events E_1 and E_3 and, therefore, there will be two possible paths, $E_2 \rightarrow E_1$ and $E_2 \rightarrow E_3$. As the basic probabilities that label the edges in these paths are $m_{1|2}$ and $m_{3|2}$ respectively, the threat estimation will be derived from the combination of the following functions:

Threat = $m_2 \oplus m_{1|2} \oplus m_{3|2}$

There are other cases where more than one event is adding evidence on the possible existence of another event. If in the network of Figure 21, for example, when the event E_2 occurs E_1 is also known then the unknown event will be $\neg E3$ and the only path to it will be $E_2 \rightarrow E_3$. Thus, the threat likelihood will be computed by the following combination of basic probability assignments:

Threat = $(m_1 \oplus m_2) \oplus m_{3|2}\,(E1 \wedge E2 \wedge \neg E3)$

A more detailed example of the computation of threat likelihood is also given in the next section.

5.6. Example of threat likelihood evaluation

As an example of estimating threat likelihoods at runtime consider the attack signatures that were derived using our abduction based generation algorithm in Section 4.3. represented by the EC formulas R9 and R10. Whilst the attack signature that was derived for the monitoring rule MR in that section was formulated as a disjunction of these two EC formulas (i.e., R9 ∨ R10), for simplicity, we will only concentrate on the formula R9 in our example below assuming that this formula constitutes the entire attack signature for the MR rule. This formula can be re-written as indicated below to express the constraints of the different time variables using the range based EC notation of the SERENITY monitoring framework:

```
R9’: (∃ _ag : Doctor; _a: Patient;  t, t1, t2: Time)
     Happens(GetPD(_ag, _a, PD1), t, R (t, t)) ∧
     Happens(GetPD(_ag, _a, PD2), t1, R(0,t-1)) ∧
     ∧ ¬Happens(AuthorisedAccess(_ag, _a), t2, R(0,t-1))
```

Given this formula, the algorithm of Figure 20 will produce the D-S belief graph shown in Figure 23. This graph assumes that the predicates in R9’ map onto the following events:

— E_1: *Happens(GetPD(_ag, _a, PD1), t, R (t, t))*

— E_2: *Happens(GetPD(_ag, _a, PD2), t1, R (0, t −1))*

— ¬E_3: *¬Happens(AuthorisedAccess(_ag, _a), t2, R(0, t −1))*

The edges in the graph represent the time dependencies between the different events of the attack signature, i.e., that the events E_2 and E_3 should occur before E_1 but they are not time-ordered themselves (this is indicated by the absence of an edge between them). Also the presence of edges from the *Start* node of the network to E_1 and E_2 signifies the possible ways of triggering the estimation of the likelihood with respect to R9’, i.e., when events that match E_1 and/or E_2 have been received by the monitor.

.

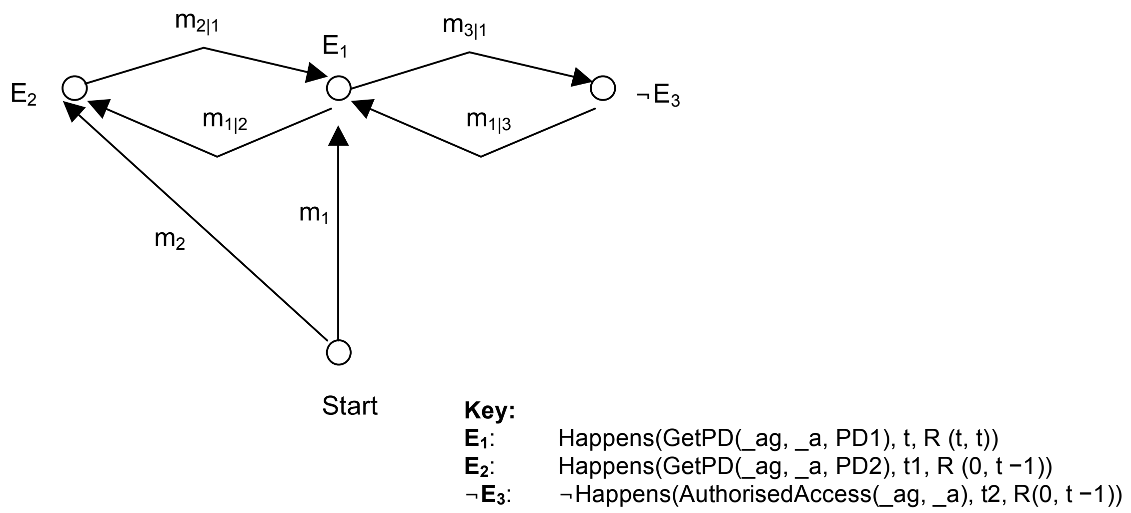
m2|1
E1
m3|1
E2
¬E3
m1|2
m1|3
m1
m2
Start
Key:
E1: Happens(GetPD(_ag, _a, PD1), t, R (t, t))
E2: Happens(GetPD(_ag, _a, PD2), t1, R (0, t −1))
¬E3: ¬Happens(AuthorisedAccess(_ag, _a), t2, R(0, t −1))

Figure 23. A simple D-S network for the attack signature R9’

Suppose that the following events arrive at the monitor in the order listed below:

— *EV2: Happens(GetPD(p1, p2, PD2), 10, R (10, 10))*

— *EV1: Happens(GetPD(p1,p2, PD1), 15, R (15, 15))*

— *EVx: Happens(AuthorisedAccess(p1, p2), 18, R(18, 18))*

and that EV1 and EV2 are produced from the same event captor whilst EVx is produced by a different captor that is the same as the captor of the events of type E_3 in the rule.

Given the graph of Figure 23, when EV2 arrives the computation of the threat likelihood for the attack signature will be based on the combination of the basic probability functions: $(m_1 \oplus m_{2|1}) \oplus m_{3|2}$.

Based on the generic definitions of these functions, in Section 5.4. (see Definition 2 and 3), it can be shown that the application of the rule of the orthogonal sum will result in the following functional form for $(m_1 \oplus m_{2|1}) \oplus m_{3|2}$:

$$(m_1 \oplus m_{2|1}) \oplus m_{3|2} (E1 \wedge E2 \wedge \neg E3) = (k_{32}'\, k_{21}\, k_1 + k_{32}'\, k_1\, (1 - k_{21} - k_{21}') + k_{32}'\, k_{21}\, (1 - k_1 - k_1')) \,/\, (1 - (k_{32}\, k_{21}'(1 - k_1') + k_{32}' k_{21}'(1 - k_1'))$$

Then assuming the following basic probability assignments:

$$m_1(X) = \begin{cases} k_1 = 0.8 & \text{if } X = \{<e_1, e_2, e_3> \mid e_1 = 1\} \\ k_1' = 0.1 & \text{if } X = \{<e_1, e_2, e_3> \mid e_i = 0\} \\ 1 - k_1 - k_1' = 0.1 & \text{if } X = \theta \\ 0 & \text{Otherwise} \end{cases}$$

$$m_{2|1}(X) = \begin{cases} k_{21} = 0.6 & \text{if } X = \{<e_1, e_2, e_3> \mid e_1 = 1 \text{ and } e_2 = 1\} \\ k_{21}' = 0.4 & \text{if } X = \{<e_1, e_2, e_3> \mid e_1 = 1 \text{ and } e_2 = 0\} \\ 1 - k_{21} - k_{21}' = 0 & \text{if } X = \theta \\ 0 & \text{Otherwise} \end{cases}$$

and

$$m_{3|2}(X) = \begin{cases} k_{32} = 0.2 & \text{if } X = \{<e_1, e_2, e_3> \mid e_2 = 1 \text{ and } e_3 = 1\} \\ k_{32}' = 0.6 & \text{if } X = \{<e_1, e_2, e_3> \mid e_2 = 1 \text{ and } e_3 = 0\} \\ 1 - k_{32} - k_{32}' = 0.2 & \text{if } X = \theta \\ 0 & \text{Otherwise} \end{cases}$$

the threat likelihood will be estimated at:

$$(m_1 \oplus m_{2|1}) \oplus m_{3|2} (E1 \wedge E2 \wedge \neg E3) = (0.6*0.6*0.8 + 0.6*0.8*0 + 0.6*0.6*0.1) \,/\, (1 - (0.2*0.4*0.9 + 0.6*0.4*0.9) = .45$$

Subsequently, when E2 arrives the threat likelihood will be estimated by the combination

$(m_1 \oplus m_2) \oplus m_{3|2}\ (E1 \wedge E2 \wedge \neg E3) =$ $(k_{32}'\, k_2\, k_1 + k_{32}'\, k_1\, (1 - k_2 - k_2') + k_{32}'\, k_2\, (1 - k_1 - k_1'))\ /$ $(1 - (k_{32}\, k_2'(1 - k_1') + k_{32}' k_2'(1 - k_1'))$

Thus, if m_2 is

$$m_2(X) = \begin{cases} k_2 = 0.8 & \text{if } X = \{<e_1, e_2, e_3> \mid e_2 = 1\} \\ k_2' = 0.2 & \text{if } X = \{<e_1, e_2, e_3> \mid e_2 = 0\} \\ 1 - k_2 - k_2' = 0 & \text{if } X = \theta \\ 0 & \text{Otherwise} \end{cases}$$

the threat likelihood will be estimated at:

$(m_1 \oplus m_2) \oplus m_{3|2}\ (E1 \wedge E2 \wedge \neg E3) = 0.504$

The increase in the threat likelihood in this case is due to the fact that the basic probability of E2 given by $m_2(X)$ is higher than the basic probability of E2 that is computed by the combination $m_1 \oplus m_{2|1}$ (0.8 vs. 0.53).

Finally, when *EVx* arrives ¬E3 will be established by the monitor through the NAF principle and the threat likelihood will be estimated by the combination

$(m_1 \oplus m_2) \oplus m^{NAF}_{3|EVx}\ (E1 \wedge E2 \wedge \neg E3) = k_1\, k_2\, k_{3|EVx}'$

Thus, assuming that $k_{3|EVx}' = 0.9$, we will have that $(m_1 \oplus m_2) \oplus m^{NAF}_{3|EVx}\ (E1 \wedge E2 \wedge \neg E3) = 0.576$

6. Integration with SERENITY Framework

In the previous sections, we have presented in detail the design and functionality of the three components that constitute the threat detection mechanisms in SERENITY namely, the *Template instantiator*, *Attack Signature Generator* and *Threat Evaluator*. This section discusses how these three components are related with the overall SERENITY framework.

The *SERENITY* framework is subdivided into the *SERENITY Development Framework* (SDF) and *Run-time Framework* (SRF). Figure 24 shows how the three components that realise the threat detection mechanisms are related to either SDF or SRF and other components within these two parts of the SERENITY framework.

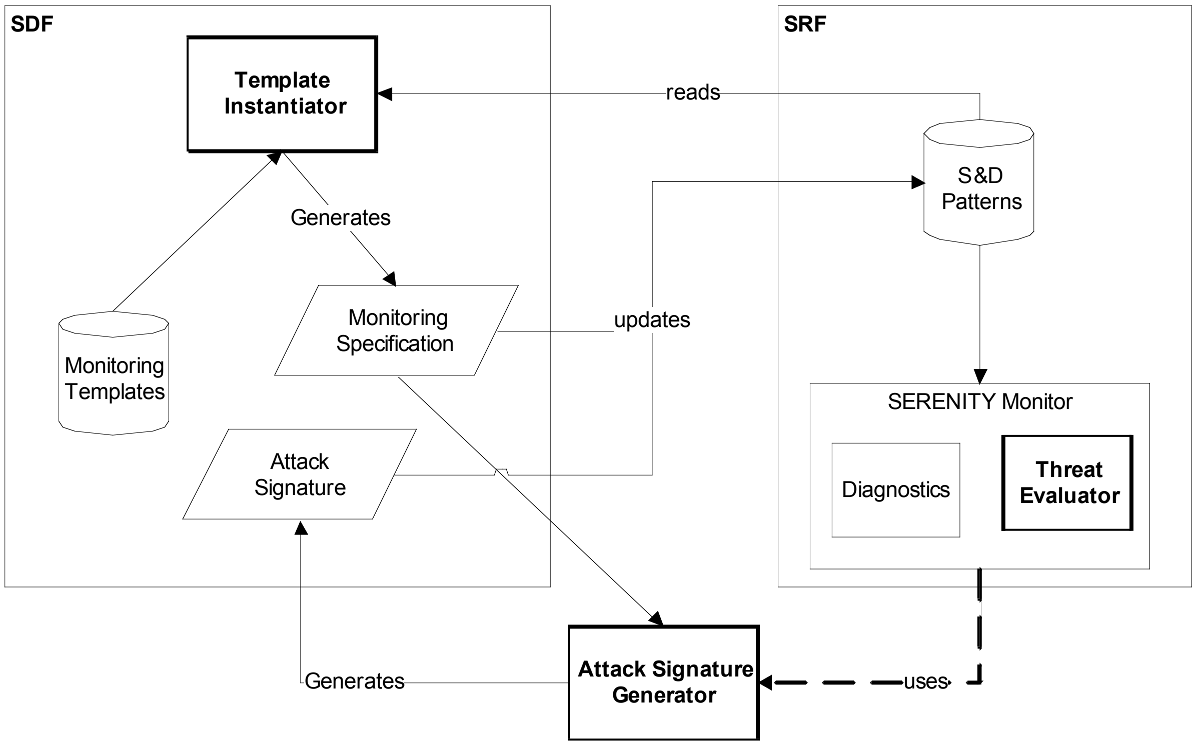

Figure 24. Threat detection mechanisms and the overall SERENITY framework.

More specifically, as shown in Figure 24,

1. The *Template Instantiator* is part of the SDF. As the primary functional purpose of this component is the generation of monitoring policies within SERENITY S&D patterns by instantiating abstract monitoring templates for particular security objectives, this component takes as input S&D patterns which do not have monitoring policies yet and supports the developers of these patterns to generate these polcies (the template instantiator generates a so-called monitoring specification which acquires the status of a monitoring policy and when S&D developers decide to include them in an S&D pattern). Note that, as shown in the figure, the Template Instantiator takes also as input a predefined catalogue of monitoring templates. This catalgue can be expanded by developers of S&D solutions if the latter wish to do so.

2. The *Attack signature generator* can be used by both the SDF and the SRF. Recall that the purpose of this component is to generate an attack signature expressing all the different ways in which a given monitoring rule may be violated given a set of assumptions about the system that is being monitored. The difference between a monitoring rule and an attack signature generated from it is that the latter expresses the former only in terms of the different combinations of runtime events that may violate it and without any reference to system states. Thus, at runtime it is possible to monitor the rule by merely observing the events sent to the monitor and without the need to deduce any information about the state of the system on the basis of these events. At development time, the use of the *Attack Signature Generator can* help developers produce, where possible, monitoring specifications for S&D patterns that do not incorporate any assumptions and use them as monitoring policies within S&D patterns. This removes the need to perform deduction at runtime and enables the estimation of threat likelihood according to the scheme introduced in Section 5. However, the *Attack Signature Generator* is also accessible by the SERENITY monitor. This is in order to transform monitoring rules, whose checking requires assumption-based deductions at runtime, into attack signatures which can subsequently be used both for monitoring without deduction and for the estimation of the threat likelihood of rules.

3. The *Threat Evaluator* is part of the SERENITY monitoring framework and the SRF. At runtime, this component runs in parallel with the SERENITY Monitor aggregating historical data that are required for the evaluation of threat likelihood (e.g., the computation of the basic probability assignmenets $m_{i|j}(X)$ in Section 5. and estimating the threat likelihood wrt different monitoring rules.

7. Related work

Our approach to threat detection is related to *intrusion detection (ID)* [8][31][32]. The goal of intrusion detection is to characterise attack manifestations to positively identify all true attacks without falsely identifying non-attacks [35]. Intrusion and threat detection systems continuously monitor some dynamic behavioural characteristic of a computer system to determine if an intrusion has occurred or is about to occur [36].

Most intrusion detection systems, however, only detect malicious actions that have already happened (intrusions), which is what the SERENITY monitor does (by detecting violations of S&D rules in patterns). Our approach to threat detection expands this basic capability by trying to predict malicious actions before they actually occur.

There are many ways to classify intrusion detection systems (IDSs) and their underlying approaches. One such classification concerns the source of events analysed by an IDS [39][40]. IDs that analyse the traffic exchanged on a network at the level of packets is called *network-based* [45]. An IDs can also target data produced locally at a host and it is therefore called *host-based*. The data analysed by a host-based system can be on different levels of abstraction, from below the OS level to the level of application running on top of an operating system. IDs that analyse data coming from higher-level applications are called *application-based* [40]; they not designated host-based because the applications being monitored may have a distributed nature. The data being sensed by the SERENITY monitor is application-based. Therefore, the following gives more focus to application-based approaches.

The most common discriminator in classifying ID concerns the strategy adopted towards detection. There are two main categories of detection approaches in ID: *anomaly-based* and *misuse-based* [31]. Anomaly-based approaches [8][34][36][43] assume that attacks will involve, somehow, abnormal behaviour of the system, and threats and intrusions are detected as deviations from normality. Misuse-based approaches [37][41][38], on the other hand, are based on models of known attacks, intrusions are detected by matching observations against representations of known attacks. The threat detection approach presented in this report is essentially *anomaly-based*, but it also has characteristics of misuse-based approaches. The next two sections review work related with these approaches in detail.

7.1. Anomaly-based approaches

There are, however, various strategies to follow in an anomaly-based approach. We consider two categories of strategies: profile-based and specification-based.

Profile-based strategies were introduced in the seminal work of Denning [8]. This strand of work considers that attacks come from subjects (users, groups, remote hosts, etc) and so it maintains profiles of normal subject behaviour. Intrusions are detected as departures from normality based on the profiles that characterise normality. In statistical approaches [8][41], a profile is described in terms of a set of intrusion-detection measures, and deviation-detection is based on statistics. Rather then building profiles of external subjects, [43] builds profiles of the software being monitored, and uses neural networks to train the IDs to know what is normality and then use the learned neural network to detect abnormality. Other approaches are more low-level, using profiles of sequences of system calls; deviations are detected by using immunology inspired algorithms [36].

A main source of criticism of profile-based approaches is that they tend to flag as intrusive previous unseen, but entirely legitimate behaviour. Model or specification-based approaches try to overcome this [34][42]. In this approach, there is a model or specification of the expected behaviour of the system or resource being protected, and intrusions are detected as deviations from the normal behaviour. This is the approach followed in the SERENITY Monitor proposed here. We have monitoring policies that specify the expected system behaviour and we detect threats and intrusion at the application level based on violations to the monitoring policy.

In particular our approach shares with [42] the emphasis on protecting system assets or resources and in building security policies with the goal of protecting them.

7.2. Misuse-based approaches

The threat detection approach proposed in this report shares many similarities with *misuse-based* approaches in that it uses attack models to detect threats. The main difference is that our approach derives the attack model from a model of the expected behaviour of the system: we try to predict what can go wrong in a system and build an attack model to represent it.

In STAT approach [37][41] describes attack models in terms of attack scenarios, where each attack scenario is represented as a sequence of transitions leading from an initial secure state to a set of compromised ending states. These attack scenarios are then used to detect intrusions. Our attack model differs from the STAT in that it is based on an abstract attack signature describing all possible attack scenarios.

In [38], Kumar and Spafford propose an approach to represent attack models as Coloured Petri Automaton (CPA). Intrusions are detected by using the matching mechanisms of CPA. CPA are generic descriptions like our attack signatures. Our approach also represents attacks models generically, but we use an EC formula to represent an attack signature rather than CPA.

In [46], Jajodia, Noel and O'Berry propose an approach to model attack graphs based on vulnerabilities. As in our approach, their approach derives a signature that concisely represents all possible attack paths or scenarios to achieve some goal. In their approach, however, the goal is some attack vulnerabilty whilst in ours the goal is the negation of a monitoring rule.

7.3. Other approaches

Our approach shares many similarities the approach to security-by-design of [33]. In particular, the use of security specification patterns selected from some security goal and which are instantiated with information coming from object models, and the derivation of attack representations (called attack trees in [33]) from security specifications. The most striking difference, however, is that [33] is a design approach; feedback coming from attack analysis is fed back into the design. Our approach detects threats to S&D requirements at *run-time*. Other differences include the formalisms being used, and the fact that we use belief based reasoning to estimate threat likelihood following the derivation of attack signatures.

8. Conclusions

This document presents the mechanisms for detecting threats at run-time in the context of the SERENITY framework. These mechanisms are being implemented and in their final version will constitute an integrated part of the SERENITY framework serving two main functional objectives within it. The first objective is to support the automatic generation of monitoring policies and attack signatures that can be used for the detection of runtime violations of S&D properties security. The second objective is to estimate the likelihood of potential violations of S&D properties (aka S&D threats).

The generation of attack signatures starts from the specification of a *security objective* for a specific *system asset*. Starting from a pair of a security objective and an asset, the *attack signature generator* identifies the possible *system operations* whose execution can potentially violate the required security property that underpins the goal. Subsequently, it creates initial *monitoring specifications* using pre-specified monitoring rule templates that cover basic security properties. These initial monitoring specifications might not include directly monitorable events and/or make references to system states which cannot be directly monitored at runtime. Hence, it is necessary to be transformed into *attack signatures* that can be monitored merely on the basis of runtime events received by the monitor during the operation of a system. This transformation is based on *planning* that uses an abductive reasoning procedure and a set of *assumptions* (EC formulas) that indicate how concrete events that can be monitored during the operation of a system can set and modify system states and/or generate other non directly observable events.

If the generation of monitorable attack signatures is succesfull then the usets of the SERENITY monitoring framework can turn them into *monitoring policies* and use them to detect *security threats* for the system being monitored. It should be noted, however, although the mechanisms that we have presented in this report support the generation of attack signatures from security objectives and assets there is no guarantee that the specification of a security objective for a specific system asset can always lead to the generation of monitorable attack signatures.

The detection of security threats at runtime is based on the basic monitoring capability of the SERENITY monitoring framework. More specifically, as soon as some runtime event instantiates a security monitoring rule and can, therefore, possibly lead to a violation of this rule, the event constitutes a security threat. To enable, however, the users of the SERENITY monitoring framework concentrate on security threats which are more likely to lead to a violation of security property in some future state of the system, the framework should calculate the likelihood of a violation given the current state of a system. The computation of this likelihood is the second main objective of the mechanisms described in this report and the computation of this likelihood is based on the *Dempster Shafer* theory of evidence.

References

[1] Androutsopoulos K., Ballas K., Kloukinas C., Mahbub K., and Spanoudakis G, "V1 of Dynamic Validation Prototype", Deliverable A4.D3.1, SERENITY Project. Available from http://www.serenity-forum.org/IMG/pdf/A4.D3.1_dynamic_validation_prototype_v1.2_final.pdf, 2006

[2] Mahbub K., Spanoudakis G., Kloukinas C., "V2 of dynamic validation prototype". Deliverable A4.D3.3, SERENITY Project. Available from: http://www.serenity-forum.org/IMG/pdf/A4.D3.3_-_V2_of_Dynamic_validation_Prototype.pdf, 2007

[3] Shafer G., "A Mathematical Theory of Evidence", Princeton University Press, 1975

[4] Tsigritis T., Spanoudakis G., "1st Version of Diagnosis Prototype". Deliverable A4.D5.1, SERENITY Project. Available from http://www.serenity-forum.org/Work-package-4-5.html, 2008

[5] Weiser, M., The Computer for the 21st Century. Scientific American, 265(3). 1991.

[6] Sánchez-Cid, F., A. Munoz, D. Serrano, and M. Gago. *Software Engineering Techniques Applied to AmI: Security Patterns*. In *Developing Ambient Intelligence*. 2006: Springer.

[7] Anderson, J.P. Computer Security Threat Monitoring and Surveillance. James P. Anderson Co., Technical Report,

[8] Denning, D., *An Intrusion Detection Model.* IEEE Transations on Software Engineering, 13(2): 222-232. 1987.

[9] F. Swiderski and W. Snyder, *Threat Modeling*. 2004: Microsoft Press.

[10] Amálio, N., S. Stepney, and F. Polack. *A formal template language enabling meta-proof*. In *FM 2006*. 2006: LNCS, Springer.

[11] Amálio, N., *Generative frameworks for rigorous model-driven development*. PhD Thesis. Dept Computer Science, Univ of York. 2007.

[12] Eshghi, K. *Abductive planning with Event Calculus*. In *5th International Conference on Logic Programming*. 1988: MIT Press.

[13] Levesque, H.J. *What is planning in the presence of sensing*. In *National Conference on Artificial intelligence (AAAI'96)*. 1996.

[14] Pearl, J., *Probabilistic reasoning in intelligent systems : networks of plausible inference*. 1988: Morgan Kaufmann.

[15] Campadello, S., L. Compagna, D. Gidoin, P. Giorgini, and S. Holtmanns. S&D Requirements specification. SERENITY Deliverable, A7.D2.1

[16] Campadello, S., L. Compagna, D. Gidoin, P. Giorgini, and S. Holtmanns. Scenario selection and definition. SERENITY Deliverable A7.D1.1

[17] Denning, D.E. and P.J. Denning, *Data Security.* ACM Comput. Surv., 11(3): 227--249. 1979.

[18] Tennent, R.D., *The denotational semantics of programming languages.* Communications of the ACM, 19(8): 437-453. 1976.

[19] Spanoudakis, G., C. Kloukinas, and K. Androutsopoulos. *Towards security monitoring patterns*. In *SAC '07: ACM symposium on Applied computing*. 2007: ACM.

[20] Kloukinas, C. and G. Spanoudakis. *A pattern-driven framework for Monitoring Security and Dependability*. In *TrustBus'07*. 2007: Springer.

[21] Schoppers, M.J. *Universal plans for reactive robots in unpredictable environments*. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI- 87)*. 1987.

[22] Manna, Z. and R. Waldinger, *How to clear a block: a theory of plans.* Journal of Automated Reasoning, 3: 343-377. 1987.

[23] Console, P., L.a. Terenziani, and D.T. Dupre, *Local reasoning and knowledge compilation for efficient temporal abduction.* IEEE Transactions on Knowledge and Data Engineering, 14(6): 1230 -1248. 2002.

[24] Eiter, T. and G. Gottlob, *The complexity of Logic-Based abduction.* Journal of the Association for Computing Machinery, 42(1): 3-42. 1995.

[25] Paul, G., *Approaches to abductive reasoning: an overview.* Artificial Intelligence Review, 7: 109-152. 1993.

[26] Stepney, S., F. Polack, and I. Toyn. A Z patterns catalogue, I. Department of Computer Science, University of York, YCS-2003-349

[27] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. 1995: Addison-Wesley.

[28] Yoder, J. and J. Barcalow. *Architectural patterns for enabling application security*. In *PLoP'97*. 1997.

[29] Cheng, B.H.C., S. Konrad, L.A. Campbell, and R. Wassermann. *Using Security Patterns to Model and analyze security requirements*. In *Requirements for high-assurance systems workshop (RHAS'03)*. 2003.

[30] Fernandez, E.B. and R. Pan. *A pattern language for security models*. In *PLoP 2001*. 2001.

[31] Lazarevic, A., V. Kumar, and J. Srivastava, *Intrusion detection: a survey*. In *Managing cyber-threats: issues approaches and challenges.*, V. Kumar, J. Srivastava, and A. Lazarevic, Editors. 2005, Springer.

[32] Lunt, T.F., *A survey of intrusion detection techniques.* Computers & Security, 12(4): 405-418. 1993.

[33] Lamsweerde, A.v., S. Brohez, R.D. Landtsheer, and D. Janssens. *From system goals to intruder Anti-goals: attack generation and resolution for security requirements engineering*. In *Requirements for high-assurance systems workshop (RHAS'03)*. 2003.

[34] C. Ko, M. Ruschitzka and K. Levitt. "Execution Monitoring of Security-Critical Programs in Distributed Systems: a Specification-Based approach." Proc. 1997 IEEE Symp. Security and Privacy. IEEE CS Press. pp. 175 – 187, 1997,.

[35] John McHugh, Alan Christie and Julia Allen. *The role of intrusion detection systems.* IEEE Software, 17 (5): 42–51. 2000.

[36] Hofmeyr, S. A., Forrest, S., and Somayaji, A. Intrusion detection using sequences of systems calls. Journal of Computer Security, 6(3): 151 – 180. 1998.

[37] K. Ilgun, R. A. Kemmerer, and P.A. Porras, *State Transition Analysis: A rule-based intrusion detection system.* IEEE Trans. Software Eng., 21(3): 181– 199. 1995.

[38] Kumar, S. and Spafford, E. H., *A pattern matching model for misuse intrusion detection.* In Proceedings of the 17th National Computer Security Conference, pages 11– 21.

[39] Christopher Kruegel, Frederik Valeur and Giovanni Vigna., *Intrusion Detection and Correlation: Challenges and Solutions. Springer. 2005.*

[40] Magnus Almgren and Ulf Lindqvist. *Application-Integrated Data Collection for Security Monitoring.* In RAID 2001, pages 22–36. Vol. 2212 of LNCS. Springer. 2001.

[41] Javitz, H. and Valdes, A. The NIDES statistical component description and justification. Tech. Rep., Computer Science Laboratory, SRI International, Menlo Park, Cal, USA. 1994.

[42] Suresh N. Chari and Pau-Chen Cheng. Bluebox: A policy-driven, host-based intrusion detection system. ACM Trans. Inf. Syst. Secur., 6(2):173–200. 2003.

[43] A. K. Gosh, J. Wanken, and F. Charron. Detecting Anomalous and unknown intrusions against programs. Proc. Annual Computer Security Application Conference (ACSAC'98), IEEE CS Press. 259–267. 1998.

[44] Richard A. Kemmerer and Giovanni Vigna. *Sensor families for intrusion detection infrastructures*. In [31]. pages 181–219. 2005.

[45] Philip K. Chan, Mattheq V. Mahoney and Muhammad H. Arshad. Learning rules and clusters for anomaly detection in network traffic. In [31]. pages 81–99. 2005.

[46] Sushil Jajodia, Steven Noel, Brian O'Berry. Topological Analysis of network attack vulnerability. In [31]. pages 81–99. 2005