



## A4.D2.1 – Basic traceability model for run-time S&D monitoring

K. Androutsopoulos, C. Kloukinas, G. Spanoudakis

<b>Document Number</b>	A4.D2.1
<b>Document Title</b>	Basic traceability model for run-time S&D monitoring
<b>Version</b>	1.0
<b>Status</b>	Final
<b>Work Package</b>	WP 4.2
<b>Deliverable Type</b>	Report
<b>Contractual Date of Delivery</b>	30 November 2006
<b>Actual Date of Delivery</b>	20 December 2006
<b>Responsible Unit</b>	CUL
<b>Contributors</b>	K. Androutsopoulos (CUL), C. Kloukinas (CUL), G. Spanoudakis (CUL)
<b>Keyword List</b>	Traceability, S&D monitoring
<b>Dissemination level</b>	PU

## Change History

<b>Version</b>	<b>Date</b>	<b>Status</b>	<b>Author (Unit)</b>	<b>Description</b>
0.1		Draft	Christos Kloukinas	Table of contents, indicative section contents
0.2	20/11/2006	Draft	Kelly Androutsopoulos	Contribution to all chapters.
0.3	24/11/2006	Draft	Christos Kloukinas	Feedback and made corrections on all chapters.
0.4	28/11/2006	Draft	George Spanoudakis	Feedback on all chapters.
0.5	8/12/2006	Draft	Kelly Androutsopoulos	Made changes to all chapters
0.6-0.9	16/12/2006	Drafts	Christos Kloukinas, George Spanoudakis	Editing, formalisation of example properties
1.0	21/12/2006	Final submission to consortium	Christos Kloukinas	Deliverable submitted for quality review

## Executive Summary

In this document we investigate and define traceability relations between the S&D modelling artefacts (described in A5.D2.1) that support the representation of S&D Solutions, i.e. S&D Properties, S&D Patterns, S&D Implementations, and S&D Configuration. We do not consider S&D classes as they are abstractions of a set of S&D Patterns and/or Integration Schemes and are mainly used by the SERENITY development tools. Our viewpoint is strongly oriented towards traceability relations which can be of use for runtime monitoring. We illustrate the usefulness of these traceability relations with respect to monitoring by applying them to an example (a solution described as an S&D Pattern for part of the smart items scenario). The document assumes that readers are familiar with the initial draft of the S&D Patterns specification schema that has been described in the A5.D2.1 deliverable of the project, including the EC-Assertion language, i.e. the formal language for specifying monitoring rules and assumptions within patterns.

## Table of Contents

Change History .....	2
Executive Summary .....	3
Table of Contents .....	4
1. Introduction.....	6
1.1. Glossary.....	9
2. Traceability between Monitoring Rules and S&D Properties .....	12
2.1. Traceability between S&D Properties and S&D Patterns .....	13
2.2. Traceability between Monitoring Rules and S&D Properties.....	16
3. Traceability between Monitoring Rules and other elements in S&D Patterns.....	18
3.1. Traceability between Monitoring Rules and Events .....	18
3.2. Traceability between Events and Components/Parameters.....	19
3.3. Traceability between Context and Monitoring Rules.....	22
3.4. Traceability between Monitoring Rules .....	23
4. Traceability between Monitoring Rules and S&D Implementations .....	27
4.1. Mapping of low-level events-to-high-level events.....	27
4.2. Traceability between event collectors and S&D Implementations .....	31
4.3. Traceability relation between events and event collectors.....	32
5. Traceability between Monitoring Rules and S&D Configuration.....	34
6. Use of Traceability Relations at Runtime.....	37
6.1. Activation of monitoring rules .....	37
6.2. Attaching event collectors to system components .....	38
6.3. Checking of monitoring rules.....	39
6.4. Deactivation of monitoring rules.....	40
6.5. Detaching event collectors from system components .....	40
7. Example .....	41
7.1. Smart Items scenario – A Solution.....	41
7.2. S&D Properties.....	45
7.3. S&D Pattern .....	48
7.4. System S&D Configuration.....	52
7.5. Traceability between S&D Properties and S&D Patterns .....	53
7.6. Traceability between S&D Properties and Monitoring Rules.....	54

7.7.	Traceability between monitoring rules and events.....	54
7.8.	Traceability between events and Components/Parameters .....	55
7.9.	Traceability between Context and Monitoring Rules.....	55
7.10.	Traceability between Monitoring Rules .....	55
7.11.	Traceability between Monitoring Rules and S&D Implementations .....	56
7.12.	Traceability between Monitoring Rules and S&D Configuration.....	57
8.	Conclusion and Future Work.....	58
8.1.	Summary of Required Extensions.....	59
	References.....	61

# 1. Introduction

---

In this document, we define and discuss traceability relations between the S&D modelling artefacts of SERENITY that support the representation of S&D Solutions, i.e. S&D Properties, S&D Patterns, S&D Implementations and S&D Configuration (described in A5.D2.1). These traceability relations complement the security and dependability modelling of systems in ways which are necessary to support the runtime S&D monitoring of such systems. Thus, our viewpoint is strongly oriented towards traceability relations which can be of use for run-time monitoring and is not concerned with other traceability relations which, although might be of help to other stages of the specification of systems and S&D solutions for them (e.g., at design time) are not relevant to monitoring. The usefulness of the traceability relations that we introduce in this report with respect to monitoring is illustrated through examples.

It should be noted that the traceability relations that we discuss in this report are inevitably preliminary, since some of these relations should exist between artefacts which are either not yet fully defined in the project (e.g., S&D Properties) or which may be revised and/or refined in subsequent phases of the project (e.g., S&D Patterns). In some cases, we refer to particular aspects which we believe should be supported by the specifications of different artefacts for traceability. To this end, we propose some extensions of the languages used to describe S&D Solutions, notably the S&D Pattern language and the S&D Property language. Some of these aspects refer to parts of these languages which have not yet been defined yet. In such cases our discussion points out what we believe should be included in the languages in order to support runtime S&D monitoring.

It should also be noted that in this report we do not consider S&D classes. This is because S&D classes are abstractions of a set of S&D Patterns and/or Integration Schemes and are mainly used by the SERENITY development tools, not the SERENITY monitors. Also, we do not consider traceability relations for S&D Integration Schemes. These relations will be considered in the next deliverable on traceability (i.e. A4.D2.3).

Our discussion of traceability assumes the definition of software traceability provided by Spanoudakis & Zisman in [7]:

*“Software traceability – that is the ability to relate artefacts created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other, the stakeholders that have contributed to the creation of the artefacts, and the rationale that explains the form of the artefacts”.*

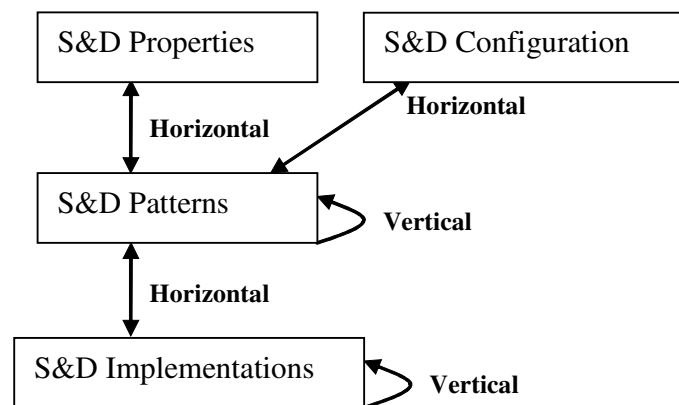
Figure 1 illustrates the type of traceability relations between the basic modelling artefacts that support the representation of S&D Solutions. The figure distinguishes these relations according to the two types of traceability relations which have been defined in [2], namely:

- **Vertical relations:** traceability relations defined between elements in the same model.
- **Horizontal relations:** traceability relations defined between elements belonging to different models.

Figure 2 illustrates the relations with respect to monitoring rules between the basic modelling artefacts in the SERENITY framework that we need to investigate for traceability. We discuss the traceability relations with respect to monitoring rules as these, together with events, are the key elements for monitoring and run-time support for security and dependability.

The traceability relations between the modelling artefacts which appear in Figures 1 and 2 are discussed in more detail in Chapters 2-5. In each of these chapters, we identify which artefacts or parts in them are being traced, the cardinality (or granularity) of the traceability relation, and how tracing is performed during all aspects of the lifecycle of S&D Solutions which, as defined in [4], include:

- The creation of new solutions and their description as S&D Patterns, each one fulfilling a number of S&D Properties.
- The implementation of the S&D Patterns and their descriptions as S&D Implementations.
- The SERENITY framework supports the development process of applications (or systems), by helping developers to select and adopt the most appropriate S&D Patterns for their requirements.
- The SERENITY framework supports the dynamic selection of S&D Implementations according to the requirements and context conditions, and provides a mechanism for monitoring the correctness of the execution of these implementations.



**Figure 1 – Relations between the basic modelling artefacts**





Except for the contribution relation, all of the other traceability relations describe associations between different software artefacts, such as specifications, software analysis, design, test models and code.

Finally, the definition of each traceability relation that we give in the following is structured according to the following aspects that need to be defined for each relation:

- **Type:** The type of traceability relation is given, for example, dependency.
- **Definition:** A definition of the traceability relation is given.
- **Attributes:** Any attributes will be described, such as, for example, how the artefacts will be obtained.
- **Cardinality:** The cardinality between the artefacts that are correlated in the traceability relation, for example, one A to many Bs, if A and B are the artefacts being correlated.
- **Constraints:** Definition of constraints on traceability relations that can be checked if necessary. For example, if there is a dependency traceability relation between A and B, then there cannot be a traceability relation of type satisfaction between A and B.
- **Directionality:** The direction of the traceability relation is given.
- **Place of definition:** The place where these traceability relations will be defined, for example, they could be defined in an extension of the S&D Pattern language.

Before discussing the traceability relations which are necessary for S&D monitoring in more detailed, however, we give a summary of the definitions of the various artefacts that we will be referring to and their features in the form of a glossary. This glossary is necessary for making this document self-contained and allowing the reader to follow it without a need to refer to other project deliverables that discuss the relevant artefacts in more detailed.

## 1.1. Glossary

The S&D modelling artefacts which are related to traceability are:

- **System:** A system is a software system that may have been either constructed using the SERENITY framework or that dynamically chooses patterns which are described in the SERENITY framework at runtime.
- **S&D Properties:** “*An S&D Property is a quality of a system that enhances its security or dependability in some way*” [4]. The S&D Requirements of a system describe a need for which properties should hold on a system or part of it. Therefore, an S&D Property must contain a (sub) system description i.e. a description of its architecture and the abstract interfaces of its components. Even though the language for describing S&D Properties has not yet been defined, we believe that it will resemble the one currently developed for S&D Patterns, to allow those who specify S&D Properties to describe the systems which have these properties. We consider the idea of S&D Properties (and indeed S&D Patterns) to be similar to the idea of “*architectural fragments*” in [9], where components which have not been fully specified are used as architectural “*placeholders*”.
- **S&D Patterns:** These describe self-contained S&D Solutions in an abstract way. S&D Solutions define mechanisms for realising the S&D Requirements and provide one or more

S&D Properties. S&D Patterns describe monitoring rules, preconditions (that must be true in order to apply the pattern), a solution description, any parameters, a reference to S&D Properties and an interface definition. S&D Integration Schemes are special types of S&D Patterns that are used to represent ways of combining other S&D Patterns. We do not discuss these in this report.

- **S&D Implementations:** These implement the solutions described by the S&D Patterns. For example, the S&D Pattern for a fair exchange protocol can be implemented in Java, or in C++. Each of these implementations is a separate S&D Implementation of the relevant S&D Pattern. All S&D Implementations of an S&D Pattern must conform directly to the monitoring capabilities, interfaces and all other characteristics described in the S&D Pattern.

S&D Implementations are also responsible for observing and capturing the (high level) events that appear in the definition of the monitoring rules described in the S&D Pattern. They use event collection mechanisms to catch the different events as they occur at run-time and these are broadcast to the monitor. Therefore, S&D Implementations consist not only of executable code implementing an S&D Pattern, but also of a specific set of event collectors for observing and emitting the events of interest.

- **Executable Implementation:** The executable implementation consists of the code for a particular S&D Implementation, and also the code for the event collectors required for monitoring this particular S&D Implementation.
- **S&D Configuration:** The S&D Configuration should instantiate the system components (environment/parameters of an S&D Pattern) and their interfaces. Monitoring rules use this information to check the interactions.
- **Event Collectors:** These are mechanisms for capturing events from the system or particular components in a system at runtime.
- **Events:** An event is either an operation call or response, or a communication of data (e.g. signals) which occurs during runtime. Events are distinguished into *high-level events* that are used in the monitoring rules of S&D Patterns and *low-level events* that are obtained from the event collectors of S&D Implementations. The main difference between high- and low-level events is that many low-level events can be mapped to one high-level event. For example, a high-level event of reading a file (*read\_file(agent, filename)*) may be mapped (and constructed) onto a sequence of three low-level POSIX events: *fp = open(pid, filename)*, *read(fp)*, and *close(fp)*. Both high level and low level events are described using the same XML schema; indeed, low-level information present in the low-level events (e.g., pid, IP addresses of machines, etc.) is fused into the description of the high-level events.

Elements used to describe an S&D Pattern:

- **ComponentDescription:** The components that are referred to in an S&D Pattern are the components that are used by the solution described in the pattern and express the specification of the S&D Implementation components which are needed to provide the particular solution at run-time [4]. These components are therefore a subset of all the components that make up the architecture of the system that deploys the solution and are introduced specifically for fulfilling S&D Properties. For monitoring, components that constitute the environment (i.e. all the components in the system architecture which are making use of some S&D Pattern and are abstractly described in these as parameters – see

below) should also be identified and described somewhere which is accessible from the S&D Pattern.

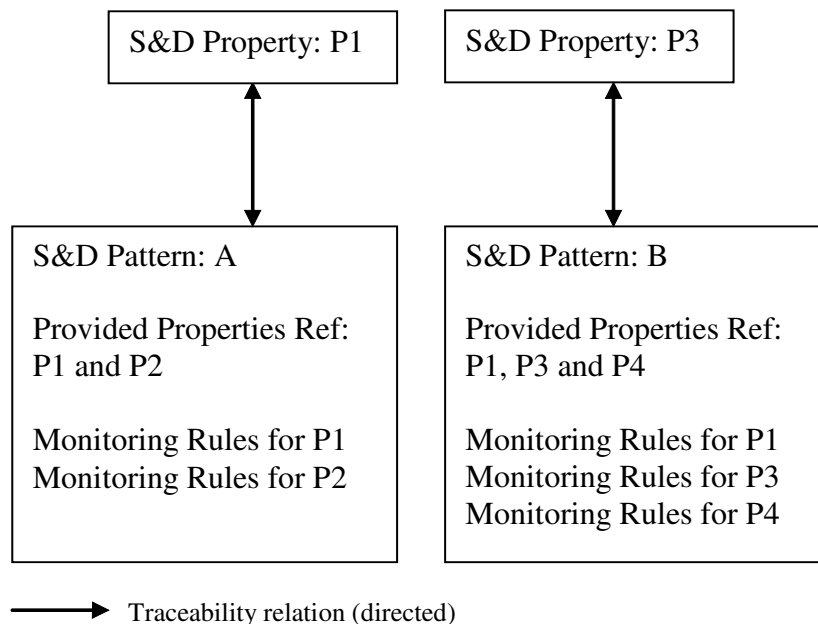
- **Parameters:** Parameters are given in order to provide some additional information about the system, i.e. description of the environment components, or constraints on existing components. They are used to make a solution more generic. The S&D Pattern language should provide detailed information about system components that are described as parameters, including their interface definition, which are required for monitoring.
- **Interface Definition:** Interface definitions describe the interface of an S&D Pattern, i.e. the interface of the components described in the pattern. However, for monitoring it might also be necessary to extend the notion of interface definitions in order to include definitions of the interfaces of pattern parameters as well. This is because calls and responses of the operations of these parameters may need to be monitored at run-time.
- **Monitoring Rules:** Monitoring rules are event calculus (EC) formulas that need to be checked at runtime. EC formulas are composed of (high-level) events and fluents. Monitoring rules can be either compulsory or recommended. Compulsory rules cannot be deactivated when an S&D Pattern is active, while recommended rules can be deactivated.
- **Context:** Context is described in S&D Patterns and consists of: a set of preconditions and a set of invariants. Preconditions are conditions regarding the applicability of an S&D pattern that must be true before a S&D Pattern can be selected. There are two types of preconditions: (i) preconditions that apply to the parameters of a pattern and (ii) preconditions that apply to the solution of a pattern. Invariants are conditions which must always be true, i.e. before the selection of the S&D Pattern and during the execution of the S&D Implementation that implements the S&D Pattern. Invariants have not been described as part of the language in A5.D2.1 [4].
- **Solution description:** This is a specification of the behaviour provided by the S&D Pattern including the behaviour of the components of the pattern. Although it is not currently, clear on whether a solution description is currently part of the first version of the S&D Pattern language, from a monitoring perspective this description should be given since deviations from the behaviour described in it which will need to be monitored at run-time.

## 2. Traceability between Monitoring Rules and S&D Properties

In this section we investigate the horizontal traceability relations between Monitoring Rules, S&D Properties and S&D Patterns. We require traceability in order to be able to determine which S&D Property is no longer true, when particular monitoring rule(s) are violated. Also, traceability can provide an explanation as to why a particular S&D Pattern was chosen, i.e. to satisfy particular S&D Properties.

The traceability relations are described at two levels of granularity. The traceability relation with a higher level of granularity is defined between S&D Properties and S&D Patterns. An S&D Pattern provides one or more S&D Properties. Moreover, each S&D Property can have many S&D Solutions defined by different S&D Patterns. It would be useful to be able to trace the S&D Properties that an S&D Pattern provides a solution for in order to have an explanation as to why a particular pattern was chosen.

A traceability relation with a finer level of granularity is defined between S&D Properties and monitoring rules. Each S&D Property has a set of monitoring rules that need to be checked. Some of these rules are compulsory and other are recommended. It would be beneficial if the SERENITY framework could automatically determine which S&D property failed as a result of a monitoring rule(s) being violated, for diagnosis and recovery actions. As such, a relation among the rules and the properties will need to be established.



**Figure 3 – Example of use of traceability between S&D Properties and S&D Patterns**

Let us consider an example. Figure 3 illustrates two S&D Patterns, A and B, that are chosen by the designer of an application App1 because they provide the S&D Properties P1 and P3, respectively. These S&D Patterns also provide some additional properties which the designer is not interested in. More specifically, the S&D Pattern A provides a solution for property P2 as well, and the S&D Pattern B provides a solution for properties P1 and P4 as well. Note that property P1 for S&D Pattern A and B is the same; however, the designer has chosen the solution provided for this property by the S&D Pattern A for some reason (e.g., its lower cost). By defining the traceability relation with the higher level of granularity between the S&D Properties and S&D Patterns, the SERENITY framework would be able to give an explanation as to why S&D Pattern A and B were used by App1, i.e. because S&D Pattern A provides a solution for S&D Property P1 and S&D Pattern B provides a solution for S&D Pattern P3. Note that an explanation cannot be provided as to why S&D Pattern A was chosen over S&D Pattern B for providing a solution for P1, unless it has been defined explicitly (e.g. as a string in natural language) by the developer. By defining the traceability relation with the finer level of granularity between the S&D Properties and monitoring rules, if a monitoring rule was violated, e.g. any of the monitoring rules for P3, the SERENITY framework could determine which S&D Property no longer holds, e.g. S&D Property P3. Also, if an S&D Pattern is chosen because of a particular S&D Property (we know because of the traceability relation with higher level of granularity), then all the recommended monitoring rules of other S&D properties can be deactivated by the framework. For example, in the S&D Pattern B, any recommended monitoring rules for P1 and P4 can be deactivated, which consequently improves the efficiency of the monitoring service because fewer rules will need to be checked.

The traceability relations between:

- (i) S&D Properties and S&D Patterns, and
- (ii) Monitoring Rules and S&D Properties

are investigated in more detail in the following and for each of these we discuss how the tracing is performed during all aspects of the lifecycle of S&D Solutions.

## 2.1. Traceability between S&D Properties and S&D Patterns

S&D patterns in SERENITY are assumed to be specified in order to represent solutions that deliver/realise certain S&D properties. To capture this relation we introduce a traceability relation, called “provides” between patterns and properties which is defined as follows:

<b>Definition of <i>Provides</i> Relation</b>	
<b>Name:</b>	Provides
<b>Type:</b>	Dependency
<b>Definition<sup>(1)</sup>:</b>	Patterns $\rightarrow$ $2^{\text{Properties}}$

<sup>1</sup> “ $2^{\text{Properties}}$ ” signifies the powerset of the set of S&D properties which are know in an instantiation of the SERENITY framework.

<b>Attributes:</b>	N/A
<b>Cardinality:</b>	One S&D Pattern to many S&D Properties
<b>Constraints:</b>	N/A
<b>Directionality:</b>	From Patterns to Properties
<b>Place of definition:</b>	Traceability is defined by a reference in the S&D Pattern (under an XML element called <i>Provided Property</i> ) to one or more Properties.

The *Provides* traceability relation means that a specific pattern realises a specific S&D property. Relations of this type should be defined during the creation of new solutions as S&D Patterns. In other words, the S&D Pattern is defined as offering one or more S&D Properties and a reference to these properties is explicitly defined in the S&D Pattern description under the XML clause *Provided Properties*. Therefore, the SERENITY framework is able to determine which S&D Patterns have been defined for which S&D Properties, which is useful during the development process of an application, as it helps the developers to select the most appropriate patterns for their requirements.

The *Provides* traceability relation between an S&D Pattern and S&D Properties is that of dependency, because an S&D Pattern relies on the existence of a specific S&D Property and if the S&D Property changes (e.g. because an error has been discovered in its description or because of changing legal requirements), these changes have to be reflected in the S&D Pattern. Because of this dependency, any changes in an S&D Property, must also be reflected in the monitoring rules as well (fine-grain traceability). For example, if an S&D Property specifies that a resource should be available in a any time range of 10 seconds, then the monitoring rule will check that the resource is available (e.g. can respond) in consecutive time intervals of 10 seconds. If the time value within which the resource should be available is changed in the S&D Property (e.g. 20 seconds), then the monitoring rule must be modified as well in order to check the new time value. As noted earlier, these changes can occur when legal requirements or standards change.

We have mentioned that each S&D Pattern has an explicit reference to the S&D Properties that it provides a solution for. However, by following this reference we are not always able to identify why the developers chose a particular S&D Pattern, as illustrated in the example of Figure 3. Therefore, for traceability between S&D Patterns and S&D Properties, the explicit reference to properties that is encoded through the *Provides* relation is not sufficient.

To represent the reason why an S&D Pattern has been selected in a specific case, we need to establish a second type of traceability relation between patterns and properties. This relation is called “SelectedFor” and is defined as shown below.

<b>Definition of <i>SelectedFor</i> Relation</b>	
<b>Name:</b>	SelectedFor
<b>Type:</b>	Rationalisation
<b>Definition:</b>	Patterns $\rightarrow$ 2 Properties
<b>Attributes:</b>	Explanation of intent (e.g. in text)
<b>Cardinality:</b>	One S&D Pattern to many Properties
<b>Constraints:</b>	The properties must be a subset of the properties provided by the pattern.
<b>Directionality:</b>	From S&D Patterns to Properties
<b>Place of definition:</b>	In the S&D Configuration - When a Pattern is selected, a traceability relation is defined that links it to some of its Properties, thus indicating the intent of the selection.

*SelectedFor* traceability relations are established, when an S&D Pattern is selected, and provide the link between the S&D Pattern and the S&D Property that the pattern realises in a specific application. Thus, *SelectedFor* relations are a kind of rationalisation relations. In the case of the example described in Figure 3, the *SelectedFor* relations for *App1* are those illustrated in Table 1.

<b>S&amp;D Pattern</b>	<b>S&amp;D Property</b>
A	P1
B	P3

**Table 1 – *SelectedFor* relations for *App1* of the example of Figure 3**

*SelectedFor* relations can be located in S&D Configurations by providing an association between an S&D pattern and the S&D Property that the pattern was chosen for. Such relations will be directional from the pattern to the S&D Property and will be updated dynamically when changes in the selected patterns of specific applications occur. Alternatively, a table, similar to Table 1, could be constructed for each application and kept up-to-date during the development process or during dynamic selection of S&D Patterns at run-time by the SERENITY framework. Thus, when *SelectedFor* relations are required by the Serenity Framework, such tables can be looked-up to establish the reason as to why a particular S&D Pattern was chosen.



## 2.2. Traceability between Monitoring Rules and S&D Properties

Fine grain traceability relations should also be established between the monitoring rules of a specific pattern and the S&D properties realised by the pattern in order to indicate which of these rules is necessary to be checked for the property to be satisfied. We call relations of this type as “Satisfies” relations and we define them as shown below.

<b>Definition of <i>Satisfies</i> Relation</b>	
<b>Name:</b>	Satisfies
<b>Type:</b>	Satisfaction
<b>Definition:</b>	$Satisfies \subseteq Rules \times \{ Compulsory, Recommended \} \times Properties$
<b>Attributes:</b>	{ Compulsory, Recommended }
<b>Cardinality:</b>	Many Rules to Many Properties
<b>Constraints:</b>	(i) $\forall r \in Rules, \exists p \in Properties, \exists a \in Attributes . (r, a, p) \in Satisfies$ (ii) $\forall r \in Rules, \forall p \in Properties, \forall a, a' \in Attributes . (r, a, p) \in Satisfies \wedge (r, a', p) \in Satisfies' \Rightarrow a = a'$
<b>Directionality:</b>	Bidirectional
<b>Place of definition:</b>	In the S&D Patterns - This traceability relation is defined by the additional XML elements required in the S&D Pattern language for monitoring rules. Monitoring rules must be distinguished into compulsory and recommended and also must refer to the S&D Property that it ensures.

The main reason for defining *Satisfies* relations is to enable the SERENITY framework to automatically determine which monitoring rules should be checked and which rules can be deactivated at runtime.

In the case of the example described in Figure 3, for instance, if the S&D Pattern A is chosen because the S&D Property P1 is required, then only the monitoring rules (both compulsory and recommended rules, unless the user has explicitly deactivated the recommended rules) for P1 should be checked. The monitoring rules for P2, on the other hand, can be deactivated by the framework, hence reducing the number of rules to be monitored and improving the efficiency of execution of rule checks. Note, however, that if there are other rules which depend on the active rules for P2, then these rules will have to be checked by the monitor (i.e. activated) as well, even though they might not provide the S&D Properties that the pattern was chosen for.

*Satisfies* traceability relations must be defined during the creation of S&D Patterns. More specifically, for each S&D Property that is provided by an S&D pattern, a set of monitoring rules may be defined and each of the rules in this set must be classified as either *recommended* or



*compulsory*. *Satisfies* traceability relations can be defined in the S&D Pattern with the addition of XML elements under the “Monitoring Rules” clause, which also has a reference to the S&D Property that it ensures. Alternatively, they can be described in a table that is accessible by the SERENITY framework via the S&D Pattern. The definition of *Satisfies* relations above includes also two constraints. The first of these constraints states that each rule in an S&D pattern must be associated to some property either as compulsory or recommended rule. The second constraint states that if a monitoring rule is associated with a property as a recommended or compulsory rule by a *Satisfies* relation then there cannot be another *Satisfies* relation between the same rule and property that is of a different type. Note that there can still be properties which are not associated with any rules via a *Satisfies* relation (e.g., because they are too expensive, impossible or not necessary to monitor).

Table 2 describes in a tabular form an example of possible monitoring rules assigned to S&D properties and their division into compulsory or recommended. For example, as shown in the table, *Rule 1* is compulsory for property *Pm* and *Rule 2* is recommended for properties *P1* and *P2*. Note that in cases where the S&D Pattern with the rules described in Table 2 is chosen for providing property *Pm*, then *Rule 1* and *Rule N* will be activated by default. Recommended rules, however, may be deactivated upon a request from an authorised user of the framework. Note also that if there are rules that depend on *Rule 1* or *Rule N* which are activated by default, then these rules will also be activated, as they are also required as a building block for offering the property *Pm*.

<b>Properties:</b> <b>Rules:</b>	<b>P1</b>	<b>P2</b>	<b>...</b>	<b>Pm</b>
<b>Rule 1</b>				Compulsory
<b>Rule 2</b>	Recommended	Recommended		
<b>...</b>				
<b>Rule N</b>		Recommended		Recommended

**Table 2 – *Satisfies* relation: an example of compulsory and recommended rules.**

It should be noted that *Satisfies* traceability relations are also beneficial when a monitoring rule gets violated. In this case, relations of this type can be used for determining the S&D Properties that are no longer true for that S&D Pattern. For example, if Rule 1 in Table 2 has been violated, then *Pm* is no longer true for the S&D Pattern. If a recommended monitoring rule, such as Rule 2 in Table 2, however has been violated, then it is not necessarily true that the S&D Properties which the rule refers to (via a *Satisfies* relation), i.e. P1 or P2, are not true. This is because by virtue of our definition only a failure of a compulsory rule will lead to the failure of the S&D Property it refers to.

## 3. Traceability between Monitoring Rules and other elements in S&D Patterns

---

In this section we investigate the vertical traceability relations between monitoring rules and other elements within S&D Patterns. The benefits of these traceability relations are that they help with runtime monitoring of the S&D Patterns and enhance the understanding of the solution described.

The S&D Pattern language is still under development, but the key elements, described in [4], are: components, parameters, interface, pre-conditions for parameters and solution (provides the context), monitoring rules. Monitoring rules are expressed in event calculus (EC) [5] (a first-order temporal formal language) in terms of events and fluents. Events signify the emission or reception of messages by different components of a system, and they are obtained or observed at runtime. High-level events are those that are described in the monitoring rules and low-level events are those that are captured from the system at runtime. Fluents represent changes in the (modelled) state of a system which are triggered by events. Fluents are not explicitly obtained or observed at runtime; they are derived by the monitoring engine using assumptions. Since events are obtained at runtime and also determine the values of fluents, we define traceability as the correlation of events between the monitoring rules and the key elements in S&D Patterns. Therefore, we investigate traceability relations between:

1. Monitoring rules and (high-level) events;
2. Monitoring Rules (events) and Components/Parameters (we include parameters here as well as they simply some under-specified components);
3. Context and Monitoring Rules, and,
4. Monitoring Rules.

### 3.1. Traceability between Monitoring Rules and Events

Monitoring rules and events are related by a traceability relation, called “Contains”. This traceability relation is required when monitoring rules refer to events which are not at a higher level than the events which can be obtained by the event captors associated with the implementation APIs contained in the pattern. In such cases, it is necessary to specify how the high level events which are referred to by the rules can be obtained from events that signify occurrences of API calls and responses to such calls. In such cases, the transformation from the events at the API level to the events at the rule level must be specified. The specification of this transformation is specified by virtue of an event calculus “theory” (i.e. a set of event calculus transformation rules). The presence of this theory is necessary so that: (i) the implementation developers know which “high” level events the “low” events of an implementation will need to be transformed to and how this transformation should be performed, and (ii) the monitor can derive the events through the API-level events.

The traceability relation *Contains* is defined as follows:

<b>Definition of <i>Contains</i> Relation</b>	
<b>Name:</b>	Contains
<b>Type:</b>	Dependency
<b>Definition:</b>	$\text{Contains} \subseteq \text{Rules} \times \text{Events}$
<b>Attributes:</b>	Transformation theory (i.e. a set of event calculus rules that specify transformations of implementation events to rule events).
<b>Cardinality:</b>	One monitoring rule to many events
<b>Constraints:</b>	N/A
<b>Directionality:</b>	Bidirectional
<b>Place of definition:</b>	In the definition of monitoring rules in the S&D Patterns.

In addition to the reasons identified above regarding the need for *Contains* relations, relations of this type are also beneficial for determining:

- (i) Which event collector to use in order to obtain the events required for a specific rule (this also requires the use of traceability relations between events and components or parameters which are described in Section 3.2).
- (ii) Whether the events that are related by *Contains* relations with a monitoring rule will still need to be captured if the rule is deactivated (i.e. it is no longer checked by the monitoring service). The SERENITY framework will need to determine this by checking not only the *Contains* relations of the specific rule that is deactivated but also whether the same events are contained in other monitoring rules which are still active. Determining which events are required leads subsequently to determining (by using some additional traceability relations) whether specific event collectors are still needed at runtime.

### 3.2. Traceability between Events and Components/Parameters

Events are captured at runtime by event capturing mechanisms that are placed at the system components of interest. The components of interest are determined by which events need to be captured for checking the monitoring rules. Therefore, a traceability relation between events expressed in monitoring rules and components is required in order to determine which component an event expressed in a monitoring rule belongs to and therefore should have an event captor that would be able to detect and report the event at runtime. This information consequently helps with diagnosis and recovery because when a monitoring rule is violated, we can determine which components were involved.

To support the identification of the components in the implementation of the system which need to provide the events which are required for monitoring, we introduce a special type of traceability relations, called “SourceOf”. *SourceOf* relations can be defined between events and S&D pattern component and parameters. Components and parameters constitute parts of the architecture that

realises the solution specified in an S&D Pattern and, according to A5.D2.1 [4], the main difference between them is that components are considered as complex types of parameters that are part of the solution described by an S&D patterns whose behaviour and other special characteristics are described within the pattern. Parameters on the other hand do not have detailed descriptions within an S&D pattern.

For the purposes of this document, we will use the term “component” to refer to both S&D pattern parameters and S&D pattern components. Given this assumption, the traceability relation *SourceOf* is defined as follows.

<b>Definition of <i>SourceOf</i> Relation</b>	
<b>Name:</b>	SourceOf
<b>Type:</b>	Dependency
<b>Definition:</b>	$\text{SourceOf} \subseteq \text{Components} \times \text{Events}$
<b>Attributes:</b>	N/A
<b>Cardinality:</b>	One component to many events
<b>Constraints:</b>	(i) $\forall e \in \text{Events} \exists c \in \text{Components} . (c, e) \in \text{SourceOf}$ $\wedge \neg \exists c' \in \text{Components} . c' \neq c \wedge (c', e) \in \text{SourceOf}$
<b>Directionality:</b>	Bidirectional
<b>Place of definition:</b>	In the definition of an event that has a reference to its source, i.e. to the component or parameter where it's produced. Events are described within the section of monitoring rules, within S&D Patterns.

It should be noted that although the *SourceOf* relations between components and events as well as between parameters and events can be described in a similar way and therefore it is possible to provide a common definition, the difference between components and parameters could be important when considering where the event capturing mechanisms that will provide the events will be placed. More specifically, the main difference here is that, in our view, event capturing mechanisms can always be placed on components if the mechanism to do so is available, because the components are introduced as part of the solution for the pattern. On the other hand, some of the parameters may be bound to existing components of the application architecture (i.e. during configuration the parameters will be unified with the real application components) and therefore it is not entirely clear if event capturing mechanisms could always be attached to these components. And in cases where it is not possible to attach event capturing mechanisms to parameters then the monitoring service will not be able check rules that contain events from such parameters and, as a consequence, the particular S&D Pattern should not be chosen. This has important repercussions for the dynamic selection of patterns at run-time by the SERENITY framework, since it must constrain the selection to only these patterns which demand events from the application components which

are indeed obtainable. As such, the definition of S&D Classes will need to take this aspect into consideration as well.

The traceability relation between events and components/parameters is defined during the construction of an S&D Pattern. The events expressed in the monitoring rules have the following generic form:

$$e(_ID, _sender, _receiver, _status, _o, _source)$$

where:

- *\_ID* is a unique identifier for the event
- *\_sender* is the name of the entity that sends the message *\_o*.
- *\_receiver* is the name of the entity that receives the message.
- *\_status* represents the processing status of an event. The status of the event can be: (i) REQ-B, that is a request for the invocation of an operation that has been received but whose processing has not started yet; (ii) REQ-A, that is a request for the invocation of an operation that has been received and whose processing has already started; (iii) RES-B, that is a response generated upon the completion of an operation that has not been dispatched yet; or (iv) RES-A, that is a response generated upon the completion of an operation that has been dispatched.
- *\_o* is a list of arguments and their types that the operation/event takes.
- *\_source* is the name of the component where this event has been captured at.

Given this event structure, it is clear that an event holds information about its *\_source* which should correspond to either a component or a parameter in an S&D Pattern. Note that parameters could represent other items besides components, such as key length, which a *source* of an event would never map to.

The cardinality of the *SourceOf* relation is one-to-many, since one component may provide many events. Table 3 illustrates examples of *sourceOf* relations between components and events.

Component A	Component B	...	Parameter C
ev(1, ..., A)	ev(3, ..., B)	...	ev(5, ..., C)
ev(2, ..., A)	ev(4, ..., B)	...	ev(6, ..., C)
...	...	...	...

**Table 3 – Examples of *SourceOf* relations between components and events**

Furthermore, it should be appreciated that the traceability relation *SourceOf* can also be used to check whether an event collector has failed. Thus, if, for example, the monitoring service has not received any events from an event collector at component X for a specific time period, such as 10 hours, then it can deduce that the event collector at X has failed.

### 3.3. Traceability between Context and Monitoring Rules

The context of an S&D Pattern consists of preconditions and invariants. Preconditions are differentiated between those that apply to parameters and those that apply to the solution, and they are currently described as a condition in natural language in A5.D2.1 [4]. Preconditions are properties that must be true *before* an S&D Pattern is selected. Thus, the validity of preconditions should be checked by the SERENITY framework before an S&D Pattern is adopted and, therefore, there is no need for monitoring them after a pattern is selected. Consequently, a traceability relation between preconditions and monitoring rules is of no interest for monitoring.

Contextual invariants are properties that should be true throughout the execution of the implementation of the solution provided by an S&D Pattern. For example, an invariant of an S&D Pattern could be that the pattern should not be used with WI-FI networks. Therefore, the monitoring service could check if at any time during the execution of an implementation of an S&D Pattern the solution uses WI-FI and if it does, the monitoring service should raise a signal to the SERENITY framework to request the deactivation of the pattern and possibly its substitution by a different S&D Pattern. Thus, it is necessary to establish traceability relations between invariants and monitoring rules.

For invariants, however, which are fully defined within a pattern a traceability relation to the monitoring rules that will be used to check it at runtime will need to be specified. These relations will be called “CheckedBy” relations and are defined as follows:

<b>Definition of <i>CheckedBy</i> Relation</b>	
<b>Name:</b>	CheckedBy
<b>Type:</b>	Generalisation
<b>Definition:</b>	Invariants $\rightarrow 2^{\text{Rules}}$
<b>Attributes:</b>	N/A
<b>Cardinality:</b>	Many invariants to many monitoring rules
<b>Constraints:</b>	N/A
<b>Directionality:</b>	Unidirectional, from invariants to rules.
<b>Place of definition:</b>	In the definition of monitoring rules in S&D Patterns.

Furthermore, for an invariant  $I$ , a monitoring rule can be described in the form of:  $f_I \Rightarrow C$ , where  $C$  is the action/condition required and  $f_I$  is a fluent that is true when the invariant is true. The monitoring service can receive events emitted from the SERENITY framework that is responsible for checking the invariants, which inform the monitor about the value of the invariant’s fluent.

Examples of events that it will receive are: *Initiates(f\_I)* and *Terminates(f\_I)*. In this way, the monitor can the framework respond to invariants which are violated.

We should also note that, as invariants are not as yet described in the S&D Pattern language in A5.D2.1 [4], the definition of *CheckedBy* relations might need to be amended when the specification language for invariants becomes fully defined.

### 3.4. Traceability between Monitoring Rules

Traceability relations need also to be expressed between the different monitoring rules which are specified in an S&D Pattern.

Dependency can be used by the monitoring service for detecting inconsistencies caused not only by the recorded but also by the expected behaviour of a system [6]. When checking whether a monitoring rule  $f:C \Rightarrow A$  is inconsistent with the expected behaviour of a system, the monitor takes into account not only the events that have been observed at runtime but also events that can be generated by other formulas and can affect the satisfiability of  $f$ . The definition of when a formula  $f$  is inconsistent with the expected behaviour of a system relies on a dependency relation between rules. This relation is defined in [11], as follows. Suppose that  $dep(f)$  is the set of formulas  $g:B \Rightarrow H$  that  $f$  depends on. A formula  $g:B \Rightarrow H$  belongs to  $dep(f)$ , if its head  $H$  has a literal  $L$  that unifies with: (i) some literal  $K$  in the body  $C$  or the head  $A$  of  $f$ , or (ii) some literal  $K$  in the body  $B'$  or head  $H'$  of another formula  $g'$  that belongs to  $dep(f)$  [11].

An example of dependency relations may be given in reference to a monitoring rule for checking the integrity of the e-healthcare system that we have given in [10]. In the e-healthcare scenario that is described in [8], 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 assistance and fast ordering of medication. In one scenario of this case study, a patient who had suffered from a cardiac arrest, feels unwell and sends through his EHT a request for assistance to ERC. To establish the cause of the problem, ERC retrieves the patient's medical record through the EHT. From this record, ERC establishes that the patient's doctor is on vacation and broadcasts a message to a group of doctors known to be able to substitute the patient's doctor. A doctor D receives this message on his own EHT and replies immediately. ERC verifies D's ability to substitute for the patient's doctor for the specific assistance request. Following this, D's EHT interrogates ERC to receive the patient's medical data. D analyses all these data, identifies the most appropriate treatment, and writes the electronic prescription on his/her EHT which subsequently sends the prescription to ERC which forwards it to the patient's EHT after registering it.

In the above scenario, the following integrity requirement has been identified:

*“Electronic prescriptions shall be issued only by doctors by means of an e-health terminal.”*  
(i.e., *Req. 2.2.1.15* in [8])



As we have indicated in [10], this requirement can be monitored by a rule stating that if an ERC receives an electronic prescription by a doctor then this doctor must be authorised to issue the prescription. The rule can be created by assuming that:

- (i) ERC provides the operation *createPrescription(docID:String, request:String, presc: Prescription)* to create new electronic prescriptions (*presc*) for a medical assistance request (*request*), and
- (ii) Doctors are authorised through the execution of an operation of ERC with the following signature: *verifyDoctor(docID:String, request:String, verified:Boolean)*. This operation verifies if a doctor (*docID*) can deal with a given request (*request*).

Assuming the above operations, the following rule can be specified to monitor the integrity requirement Req. 2.2.1.15 (this rule is derived from a monitoring pattern as described in [8]):

**Rule IR1:**

```

 $\forall$  _eID1,_ercID,_docEhtID:String; t:Time
Happens( e(_eID1,_docEhtID,_ercID,REQ-B,
createPrescription(_docID,_request,_presc), _ercID), t,  $\mathfrak{R}(t,t)$ )  $\wedge$ 
HoldsAt(transforms( createPrescription(_docID,_ request,_presc), _ercID), t)
 $\Rightarrow$ 
HoldsAt(authorised(_ercID, _docID, e(_eID1,_docEhtID, _ercID,REQ-B,
createPrescription(_docID,_ request,_presc), _ercID)), t)

```

The rule IR1 checks whether the doctor (*\_docID*) who invokes the operation *createPrescription* in ERC (*\_ercID*) is authorised to do so. IR1 effectively describes a delegation of the doctor's right to create prescriptions to his/her EHT, since it is the EHT which is the sender in this interaction (*\_docEhtID*), while it is the doctor who is being authorised (*\_docID*) for the action in reality. Following the pattern for monitoring integrity that we introduced in [10], the following assumption needs also to be specified in order to generate at runtime the information that is needed for checking the rule IR1:

**Assumption IA2:**

```

 $\forall$  _eID2,_ercID,_docEhtID:String; t:Time; _request: String, _verified:
Boolean
Happens(e(_eID2,_ercID,_ercID, RES-A, verifyDoctor(_docID,
_request,_verified), _ercID), t,  $\mathfrak{R}(t,t)$ )  $\wedge$ 
HoldsAt(valueOf(_verified, True),t)  $\Rightarrow$ 
Initiates( e(_eID2,_ercID,_ercID, RES-A,
verifyDoctor(_docID,_request,_verified), _ercID), authorised(_ercID,
_docEhtID, e(_eID1,_docEhtID, _ercID,REQ-B, createPrescription(_docID,_
request,_presc), _ercID)), t)

```



This assumption states that a doctor is authorised to call the operation *createPrescription* in ERC only if this is verified by the operation *verifyDoctor*. In this case, an appropriate authorisation fluent will be generated by *IA2* and by virtue of the EC axiom

$$\mathbf{HoldsAt}(f, t_B) \Leftarrow (\exists e, t) \mathbf{Happens}(e, t, \mathfrak{R}(t_A, t_B)) \wedge \mathbf{Initiates}(e, f, t) \wedge \neg \mathbf{Clipped}(t, f, t_B)$$

we can derive that the *HoldsAt* predicate in the head of the rule *IR1* is satisfied.

In this example, the rule *IR1* depends on the assumption *IA2* or, equivalently, *IA2* belongs to the set *dep(IR1)*. This is because the predicate

$$\mathbf{HoldsAt}(\text{authorised}(\_ercID, \_docID, e(\_eID1, \_docEhtID, \_ercID, REQ-B, \text{createPrescription}(\_docID, \_request, \_presc), \_ercID)), t)$$

of *IR1* can be unified with the predicate *HoldsAt(f,t)* in the above EC axiom. Thus the axiom belongs to *dep(IR1)*. Following this unification, however, the axiom takes the following form:

$$\begin{aligned} &\mathbf{HoldsAt}(\text{authorised}(\_ercID, \_docID, e(\_eID1, \_docEhtID, \_ercID, REQ-B, \\ &\text{createPrescription}(\_docID, \_request, \_presc), \_ercID)), t) \Leftarrow \\ &(\exists e, t) \mathbf{Happens}(e, t, \mathfrak{R}(t_A, t_B)) \wedge \\ &\mathbf{Initiates}(e, \text{authorised}(\_ercID, \_docID, e(\_eID1, \_docEhtID, \_ercID, REQ-B, \\ &\text{createPrescription}(\_docID, \_request, \_presc), \_ercID)), t) \wedge \\ &\neg \mathbf{Clipped}(t, \text{authorised}(\_ercID, \_docID, e(\_eID1, \_docEhtID, \_ercID, REQ-B, \\ &\text{createPrescription}(\_docID, \_request, \_presc), \_ercID)), t, t_B) \end{aligned}$$

The predicate

$$\mathbf{Initiates}(e, \text{authorised}(\_ercID, \_docID, e(\_eID1, \_docEhtID, \_ercID, REQ-B, \text{createPrescription}(\_docID, \_request, \_presc), \_ercID)), t)$$

in this new form of the axiom, however, can be unified with the predicate

$$\mathbf{Initiates}(e(\_eID2, \_ercID, \_ercID, RES-A, \text{verifyDoctor}(\_docID, \_request, \_verified), \_ercID), \text{authorised}(\_ercID, \_docEhtID, e(\_eID1, \_docEhtID, \_ercID, REQ-B, \text{createPrescription}(\_docID, \_request, \_presc), \_ercID)), t)$$

of *IA2*. Thus, according to the definition of dependent formulas above, *IA2* also belongs to *dep(IR1)*.

The dependency relations between monitoring rules will be called “DependsOn” relations and are defined as follows:

<b>Definition of <i>DependsOn</i> Relation</b>	
<b>Name:</b>	DependsOn
<b>Type:</b>	Dependency
<b>Definition:</b>	Rules $\rightarrow 2^{\text{Rules} \cup \text{Assumptions}}$
<b>Attributes:</b>	N/A
<b>Cardinality:</b>	One-to-many
<b>Constraints:</b>	If a monitoring rule MR1 is activated because it provides an S&D Property P1 of interest, then all monitoring rules on which MR1 depends on must also be activated.
<b>Directionality:</b>	Unidirectional
<b>Place of definition:</b>	In the definition of monitoring rules in S&D Patterns by a reference to other dependent rules. For example, if MR1 and MR2 are rules that depend on MR3, then they should have a reference to MR3 (a “depends on” XML reference is defined).

The same definition of dependency can be used for determining dependencies not only between monitoring rules and assumptions defined in S&D Patterns but also between monitoring rules themselves. Such dependency relations between the monitoring rules in an S&D Pattern are important to identify because they can be used for triggering the activation of rules and for recovery. If a monitoring rule MR1 is activated as it provides an S&D Property P1 of interest, for instance, then all the other monitoring rules that MR1 depends on will also be activated and checked. For recovery, the dependencies between the monitoring rules help to determine which rules have failed (i.e. a rule MR1 and all the rules which depend on it). Finally, it should be noted that *DependsOn* relations can be automatically generated using the definition of the set  $dep(f)$  that was given above. By identifying these dependencies statically and expressing them as *DependsOn* relations, the monitoring service can use them at runtime to identify the derived events that may affect the satisfiability of monitoring rules as described in [11].

## 4. Traceability between Monitoring Rules and S&D Implementations

---

In this section we describe the horizontal traceability relations between monitoring rules and S&D Implementations. Traceability in this case is fine-grain and focuses on the correlation of events between rules and S&D Implementations.

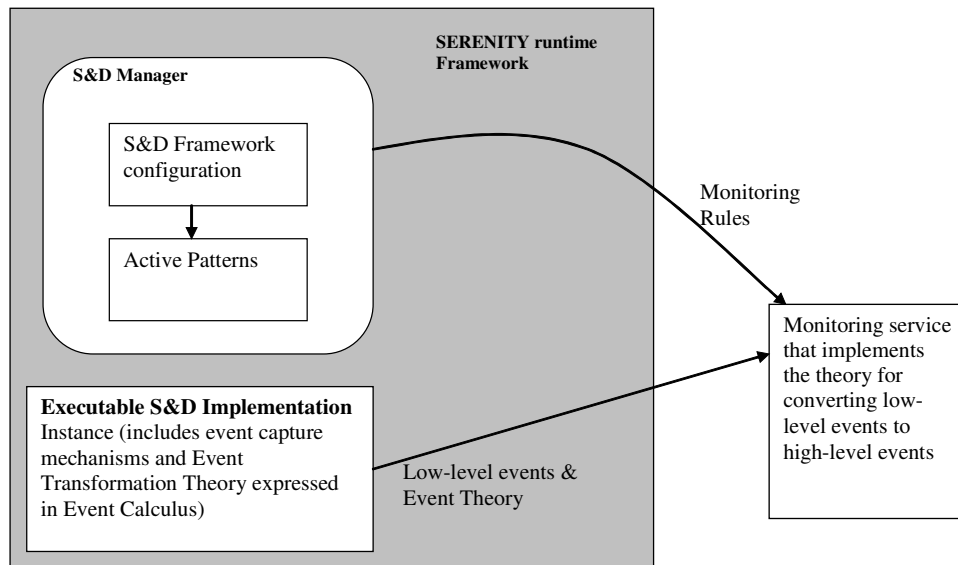
An S&D Implementation consists of an implementation of the solution described in an S&D Pattern and a set of event-capturing mechanisms (event collectors) for capturing and emitting low-level events from the implementation of the solution at runtime. In this section, we consider traceability relations between: (a) low-level events captured by the event collectors in S&D Implementations with high-level events which are described in the monitoring rules and (b) implementations with event collectors.

### 4.1. Mapping of low-level events-to-high-level events

The S&D Patterns describe the monitoring rules using high-level events. These monitoring rules will be given as input to the monitoring service in order to perform the checking. However, the events observed by the event collectors at runtime may be low-level events that need to be mapped onto the high-level events required by the monitoring service. Low-level events can be mapped into high-level events by using a theory of transformation formulas. For example, if a sequence of three low-level events ( $E1$ ,  $E2$  and  $E3$ ) should be mapped to a single high-level event  $E4$ , the transformation could be expressed by a formula  $f: E1 \wedge E2 \wedge E3 \Rightarrow E4$ . Following such transformation the high level events are sent to the monitoring service.

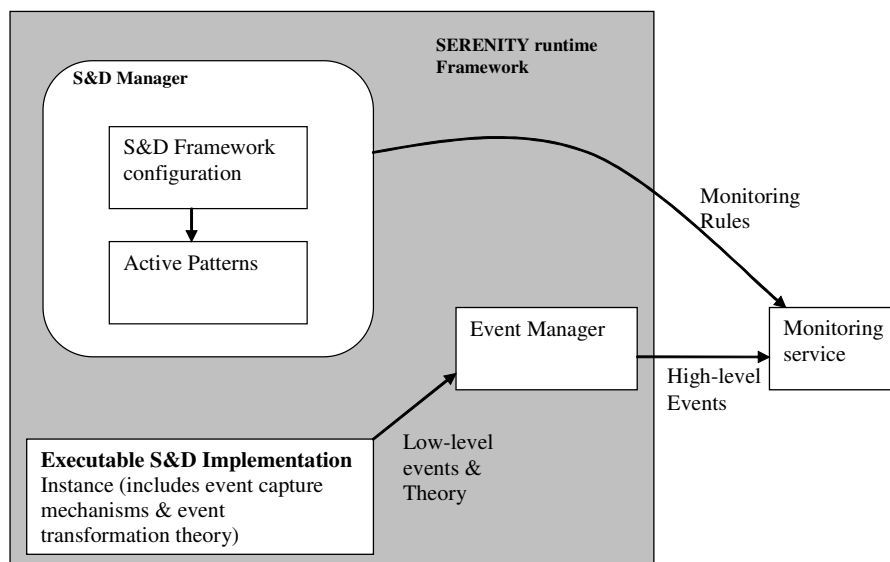
It should be noted that there are three possible options for performing the mapping of low-level events to high-level events:

1. The mapping occurs in the monitor and requires an Event Calculus theory. This theory can be expressed in the S&D Implementation and is sent to the monitoring service. Subsequently, the mapping is performed within the monitoring service according to the theory. The theory must be written by the developer who knows how the mapping should be done. Figure 4 shows the flow of events and their transformation in the various components of the SERENITY framework in the case of this option.



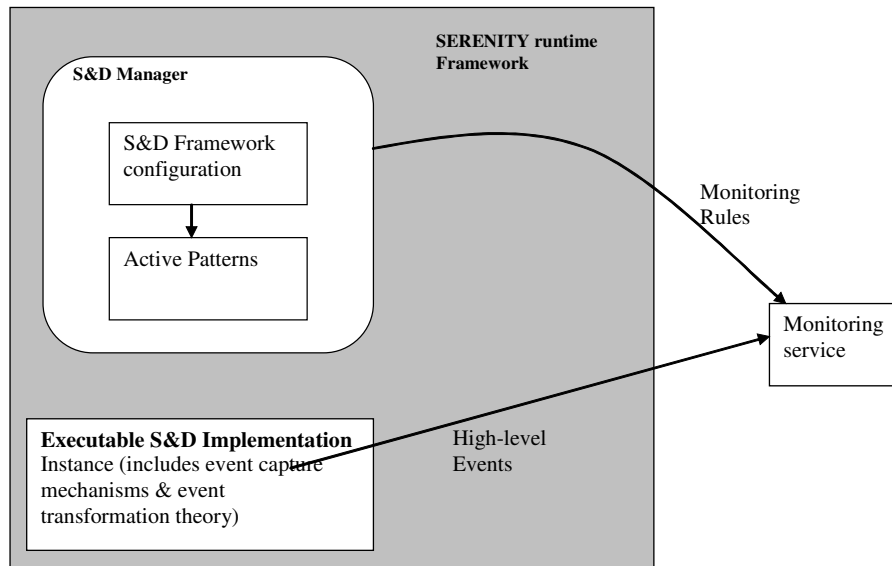
**Figure 4 – Part of SERENITY runtime framework that interacts with the monitoring service**

2. The mapping occurs in the SERENITY framework (for example, in the event manager) and it requires a theory that is described in the S&D Implementation. The language in which this theory will be described must be defined by A6. Figure 5 illustrates how the active patterns and executable implementation instances interact with the monitoring service. More specifically, according to this figure the executable implementation instances send the low level events and the transformation theory to the SERENITY framework which subsequently transforms them into the high level events expected by the monitor and forwards the latter to the monitoring service.



**Figure 5 – Part of SERENITY runtime framework that interacts with the monitoring service**

- The mapping occurs in the S&D Implementation, which also contains a description of the theory required for it (see Figure 6). The implementation directly maps the low-level events to high-level events and sends the latter to the monitoring service. In this option, the S&D implementation can express the event transformation theory in the language of its choice.



**Figure 6 – Part of SERENITY runtime framework that interacts with the monitoring service**

By defining the mapping between the low-level and high-level events, we are also able to trace a particular high-level event defined in a monitoring rule to its corresponding low-level event(s) and vice versa. This is of particular interest when a monitoring rule is found by the monitor to have been violated so that fine-grain diagnosis can be performed.

The traceability relation between high level and low level events is called “Mapped” and is defined as follows:

<b>Definition of <i>Mapped</i> Relation</b>	
<b>Name:</b>	Mapped
<b>Type:</b>	Refinement
<b>Definition:</b>	$Mapped \subseteq \text{High-level events} \times \text{Low-level events}$ This traceability relation is defined by a correlation of low-level events to high-level events.
<b>Attributes:</b>	<i>Transformation:</i> Theory of how of high-level events are produced from low-level events.

	low-level events.
<b>Cardinality:</b>	Many high-level events to many low-level events
<b>Constraints:</b>	Formulas that connect high-level and low-level events can be used to check if the transformations are correct.
<b>Directionality:</b>	Bidirectional
<b>Place of definition:</b>	In the monitor or the S&D Implementation.

The advantages and disadvantages of each of the three mapping approaches are summarised in Table 4.

<b>Location of Mapping</b>	<b>Advantages</b>	<b>Disadvantages</b>
1) In the monitor	Good for fine-grain diagnosis.	Burdens the monitor.
2) In the SERENITY framework	Clear separation of functionality, i.e. monitor responsible for checking rules only, S&D Implementation for implementing solution and framework for any translations, mappings etc.	Burdens the Framework.
3) In the S&D Implementation	Not burdening the monitor or framework. The implementation is not burdened either. In fact, it might improve performance because there will be less communication with the SERENITY framework.	Diagnosis is coarse-grain, since the traceability relation is effectively missing.

**Table 4 – Advantages and disadvantages of different architectures**

Regardless of which of the above three options is chosen, it should be noted that when it comes to the diagnosis of detected rule violations is required, the *Mapped* relation from high-level to low-level events must be defined, and for each high-level event this relation must provide a theory in event calculus specifying how the transformation occurs. Such a theory can be specified as the value of the attribute Transformation of each of the elements of the *Mapped* relation. Along with this relation, the set of low-level events which have occurred so far will be needed. This set should either be held at the runtime framework, through which the monitor can retrieve it, or the S&D Implementations should support requests from the monitor for retrieving these events.

## 4.2. Traceability between event collectors and S&D Implementations

The second type of traceability relations that we consider is a relation between event collectors and S&D implementations.

An S&D Pattern may have many executable implementations of the solution it describes. Each of these implementations is expected to have a set of event collectors that will be attached to components & parameters of the S&D Pattern. An implementation can use the same event-collector as another implementation. Assuming that during runtime certain S&D Implementations are adopted, sometimes in place of other implementations; it would be beneficial if there was some mechanism in the SERENITY framework responsible for managing event collectors<sup>2</sup>. For example, if an event collector EC1 is used by two Java implementations, when none of these implementations is any longer applied in the application, the event collector EC1 should also stop executing. In this way, the performance of the system's execution will be improved as the use of event collectors delays the system's performance. Apparently, we cannot stop this event collector when only one of the implementations which are using it is replaced.

To be able to identify the event collectors which are required during the operation of the SERENITY framework, we can use a traceability relation between S&D Implementations and event collectors, which we call "Uses". This relation is defined as follows:

<b>Definition of <i>Uses</i> Relation</b>	
<b>Name:</b>	Uses
<b>Type:</b>	Dependency
<b>Definition:</b>	$Uses \subseteq Implementations \times Event\ Collectors$
<b>Attributes:</b>	N/A
<b>Cardinality:</b>	Many implementations to many event collectors
<b>Constraints:</b>	N/A
<b>Directionality:</b>	Bidirectional
<b>Place of definition:</b>	In the S&D Implementation we provide a list of event collectors used

The traceability relations *Uses* between event collectors and S&D Implementations are defined during the construction of an S&D Implementation by enumerating the event collectors used by the implementation. Using this information, the SERENITY framework can determine which event

<sup>2</sup> The event collector manager can be compared to garbage collection, i.e. its aim is to stop unused event collectors from executing.

collectors to deactivate (or clean up) based on whether they are currently being used by active S&D implementations. Table 5 illustrates an example of the *Uses* traceability relation between event collectors and active implementations. The relationship between implementations and event collectors is many-to-many (one implementation can use many event-collectors and one event-collector can be used by many implementations). In this example, if the S&D implementation *ImpC*, for instance, becomes inactive, the event collector *EC2* can also be deactivated. Unlike it, the event collector *EC1* cannot be deactivated if the S&D implementation *ImpA* becomes inactive since it would still be needed by *ImpB*.

Event Collector	Implementations
EC1	ImpA
EC1	ImpB
EC2	ImpC
ECn	ImpF
ECn	ImpD
ECn	ImpA

**Table 5 – Relation between event collectors and active implementations**

### 4.3. Traceability relation between events and event collectors

Each event collector should describe the set of events that it can capture. Capture relations of this form are represented by a traceability relation called “CollectedBy” that is defined as follows:

<b>Definition of <i>CollectedBy</i> Relation</b>	
<b>Name:</b>	CollectedBy
<b>Type:</b>	Dependency
<b>Definition:</b>	Event $\rightarrow$ Event Collector $\cup$ {nil}
<b>Attributes:</b>	N/A
<b>Cardinality:</b>	One event to one event collector
<b>Constraints:</b>	N/A
<b>Directionality:</b>	Unidirectional – from event to event collector



<b>Place of definition:</b>	In the S&D Implementation, for each event collector we provide a list of events that can be captured.
-----------------------------	---

Given the presence of such relations, when the time comes to attach an event collector to a component in order to capture the events of interest (i.e. the events that are described in the rules) it will be possible to determine which event collector should be used for an event. According to its definition above the cardinality of *CollectedBy* is one-to-one. This reflects our assumption that an event can be captured only by one event collector in a single S&D Implementation.

Note that high-level events do not have any event collectors associated with them, since they are formed by a theory over the low-level events that the collectors collect.

## 5. Traceability between Monitoring Rules and S&D Configuration

Each system has a single S&D Configuration (referred to as the “System S&D Configuration”) which unifies the parameters of the different S&D Patterns to system and/or other S&D Pattern components. At the same time, each S&D Pattern needs its own S&D Configuration, which unifies the parameters of the S&D Properties provided by the pattern to the S&D Pattern components/parameters. The S&D Configuration also must describe the concrete interfaces of the parameters, that is, say which of the real components’ operations are mapped to the abstract operations of the parameters. These are not described anywhere else but they are required by the monitoring service, since the monitoring rules use this information to check the interactions between components and parameters in order to ensure the S&D Properties.

For the purposes of monitoring, the operations of pattern parameters which are described in the events which are referenced by the monitoring rules must be traced to the actual operations of the components that have substituted for pattern parameters. To enable this tracing defined To enable this tracing, we introduce two types of traceability relations in the S&D Configurations. The first of these relations is called “UnifiesWithComponent”, associates pattern parameters with the actual system components that are bound to them at runtime and is defined as specified below:

<b>Definition of <i>UnifiesWithComponent</i> Relation</b>	
<b>Name:</b>	UnifiesWithComponent
<b>Type:</b>	Refinement
<b>Definition:</b>	$(\text{Patterns} \times \text{Parameters}) \rightarrow \text{Components}$
<b>Attributes:</b>	N/A
<b>Cardinality:</b>	One pattern, parameter pair to one component
<b>Constraints:</b>	N/A
<b>Directionality:</b>	Unidirectional
<b>Place of definition:</b>	S&D Configuration

The second relation is called “UnifiesWithOperation”, associates abstract operations of monitoring rules with the concrete component operations which are bound to them at runtime and is defined as specified below:

<b>Definition of <i>UnifiesWithOperation</i> Relation</b>	
<b>Name:</b>	UnifiesWithOperation
<b>Type:</b>	Refinement
<b>Definition:</b>	$UnifiesWithOperation \subseteq (Patterns \times Abstract\ Operations) \times Concrete\ Operations$
<b>Attributes:</b>	N/A
<b>Cardinality:</b>	Many abstract operations to many concrete operations.
<b>Constraints:</b>	N/A
<b>Directionality:</b>	Bidirectional
<b>Place of definition:</b>	S&D Configuration

Let us consider a simple example. Suppose that the following monitoring rule MR1 is expressed in an S&D Pattern  $P$ :

**Rule MR1**

$$(\forall t_1, t_2: Time) \quad \mathbf{Happens}(e(A, B, op1(p1), A), t_1) \Rightarrow \mathbf{Happens}(e(B, C, op2(p2, p3), B), t_2) \ \& \ t_2 \geq t_1$$

This rule states that if A invokes an operation  $op1(p1)$  which is received by B at time  $t_1$ , then B must invoke the operation  $op2(p2, p3)$  which is received by C at some time  $t_2$  after  $t_1$ . In this rule, A, B and C signify dispatchers and/or receivers of the invocations of the operations  $op1(p1)$  and  $op2(p2, p3)$ . More specifically, A dispatches the invocation of  $op1(p1)$ , B received the invocation of  $op1(p1)$  and dispatches the invocation of  $op2(p2, p3)$  and C receives the invocation of  $op2(p2, p3)$ . Suppose also that A and B have been defined as parameters in the S&D Pattern that incorporates MR1, and C has been defined as a component in the same pattern. The operations  $op1(p1)$  and  $op2(p2, p3)$  belong to B and C, respectively.

When this S&D Pattern is selected (activated) by the SERENITY framework for a specific system, the S&D Configuration of this system will unify the elements in the S&D Pattern with the “actual” components of the involved system and/or the implementation of the S&D solution described by the selected pattern. Thus, for example, the parameters A and B may be unified with the components  $c1$  and  $c2$  using the *UnifiesWithComponent* relation as follows:  $UnifiesWithComponent((P, A), c1)$ ,  $UnifiesWithComponent((P, B), c2)$ . The interface of the components should also be described in the configuration part of the pattern, i.e. all the operations that  $c1$  can invoke as  $op1$  and  $c2$  can invoke as  $op2$ . Then  $op1$  and  $op2$  must be unified with the actual operations using the *UnifiesWithOperation* relation, e.g.  $op1(p1)$ , could be  $send(data)$  &  $receive(data)$ , and  $op2(p2, p3)$  could be  $add(d1, d2)$ .

It should be noted that a concrete operation can be unified with many abstract operations and an abstract operation can be unified with many concrete operations. An abstract operation signifying

the disclosure of an information parameter  $i$  in an S&D pattern, called  $leaks(i)$  for instance, can be unified with concrete operations that result in information disclosure such as  $write(f)$  and  $print(r)$ . Similarly, one concrete operation can also be unified with many abstract operations. The concrete operation  $write(f)$ , for example, can be unified with the abstract operation  $leaks(d)$  above and another abstract operation signifying the transmission of information called  $transmit(m)$ .

## 6. Use of Traceability Relations at Runtime

---

Having defined the different types of traceability relations in the previous sections, we can now describe how these relations can be used by the SERENITY framework at runtime. Generally, these relations can be used at runtime to enable:

- the selection (activation) of monitoring rules when an S&D Pattern is selected
- the attachment of event collectors to system components
- the checking of the monitoring rules by the monitor
- the deactivation of monitoring rules
- the detachment of event collectors to system components

Note that the assumptions of monitoring rules are treated in the same way as monitoring rules. However, an assumption is always associated to a monitoring rule.

### 6.1. Activation of monitoring rules

Monitoring rules are activated when an S&D Pattern is selected from the library to provide a solution that ensures particular S&D Properties. The user who chooses the S&D Patterns/Classes during the development of the system must also provide the information required for the *SelectedFor* traceability relation, i.e. which S&D Properties this S&D Pattern has been chosen for. Note also that the SERENITY framework can dynamically change the *SelectedFor* relation for a particular software system.

The steps which must be performed in order to activate the monitoring rules are as follows:

1. The user/framework selects an S&D Pattern *pat* from the library. This selection can take place either during the development of a system by the user (not at runtime) or at runtime by the framework when an S&D Pattern is no longer capable of providing a particular solution and another S&D Pattern has to be chosen instead.
2. The *SelectedFor* traceability relation between S&D Patterns and S&D Properties must be updated by the framework with the required information that is found in the S&D Configuration to indicate which S&D Properties the S&D Pattern was chosen for. At this point a check to ensure that the set of the properties that the pattern was selected for is a subset of the properties that the pattern provides (as expressed by the *Provides* relation) must be performed. Subsequently, the initial set of compulsory and recommended monitoring rules for the selected S&D Property can also be determined. The identification of these monitoring rules will be based on the *SelectedFor* relations and the *Satisfies* relations between rules and S&D Properties. More specifically, for each S&D Property that is referenced by the *SelectedFor* relation, its compulsory and recommended monitoring rules are identified and activated using the *Satisfies* relations which refer to the property. When activated a rule is sent to the monitoring service to be checked. It should be noted that the recommended rules of a *SelectedFor* S&D Property can be deactivated. This, however, must be done explicitly, as the default is to activate both the compulsory and recommended rules of a desired S&D Property. Formally, the initial set of activated rules is defined as follows:

$$InitRules_{pat} = \{r \mid \exists q \in SelectedFor(pat) \wedge \exists a \in Attributes . (r,a,q) \in Satisfies_{pat} \}$$

where *Attributes* is the set  $\{Compulsory, Recommended\}$ .

3. Then, by using the *DependsOn* traceability relation between monitoring rules, all the monitoring rules that are dependent on to initial set of rules that have been activated for the S&D Pattern can also be determined and activated. Formally, the set of rules that will be activated at this step is defined as follows:

$$FinalRules_{pat} = InitRules_{pat} \cup \bigcup_{r \in InitRules_{pat}} DependsOn_{pat}(r)$$

4. Subsequently, all the parameters in the event references which are made within the monitoring rules must be instantiated with the actual values that are described in the S&D Configuration of the system. This is done by using the traceability relations *UnifiesWithComponent* and *UnifiesWithOperation* which as we discussed in Section 5 are defined in the system S&D Configuration. In this way, we can obtain instances of monitoring rules which refer to actual system components and operations, instead of abstract pattern parameters/components and operations. The resulting set of fully instantiated monitoring rules is called *InstantiatedFinalRules<sub>pat</sub>*.
5. Finally, the instantiated rules in *InstantiatedFinalRules<sub>pat</sub>* are activated, i.e. they are sent to the monitor to be checked.

## 6.2. Attaching event collectors to system components

S&D Implementations have sets of event collectors that can be used to provide the events required for the monitoring of the solutions that they realise. When an S&D Pattern is chosen, along with a corresponding S&D Implementation, the appropriate event collectors will need to be attached to or activated in the system components which runtime events need to be captured from. The process of determining the event collectors that will be required and activating/attaching them to system components is driven by the instantiated rule set for the specific system and takes place through the following steps::

1. For each rule the set *InstantiatedFinalRules<sub>pat</sub>* of the activated pattern *pat*, the events of interest that need to be captured by the event collectors are determined through the *Contains* traceability relation. This set of events is formally defined as follows:

$$EventsOfInterest_{pat} = \{e : \exists r \in InstantiatedFinalRules_{pat} . (r, e) \in Contains_{pat}\}$$

where *Contains<sub>pat</sub>* is the *Contains* traceability relation between rules and events in S&D Pattern *pat*.

2. Then, by using the *SourceOf* traceability relation for each event of interest *e*, we determine the component that *e* needs to be obtained from. In this way we can determine the set of all the components which are the sources of the events of interest and attach the corresponding event collectors to them. This set of components is defined as follows:

$$\text{ComponentOfInterest} = \{c : (c, e) \in \text{SourceOf}_{pat}\}^3$$

3. Also by using the *CollectedBy* traceability relation we can determine the event collector for each event of interest  $e$ .

$$\text{EventCollector} = \text{CollectedBy}_{imp}(e)$$

4. Finally, the SERENITY framework will attach the identified event collector to the identified component, if the former is not nil, since not all events have a corresponding event-collector.

### 6.3. Checking of monitoring rules

In order to check monitoring rules, the monitoring service requires the set of active monitoring rules and assumptions and the set of events which occur at runtime, i.e. those captured by the event collectors. Note that monitoring rules contain high-level events, while the events which are captured by the event collectors are low-level. Therefore, the steps that must be performed in order to check the monitoring rules are:

1. The active monitoring rules in the S&D Pattern are sent to the monitoring service.
2. The low-level events are mapped to high-level events and sent to the monitoring service. The *Mapped* traceability relation will allow for low-level events to be traced to high-level events and vice versa. The mapping can occur at different locations and this depends on the type of architecture that the SERENITY framework will adopt.
3. The *DependsOn* traceability relation will determine the dependencies between the monitoring rules (or assumptions) which will enable the detection of different types of inconsistencies as defined in [6].

By checking each monitoring rule, the monitor determines whether the rule holds or is violated when the system is executed (during runtime). The following steps outline what happens when a rule is violated:

1. If a compulsory rule is violated, then the S&D Property that the monitoring rule is related to will no longer hold for the selected S&D Pattern. The identification of the properties which are affected by violations of monitoring rules is based on the *Satisfies* traceability relation. Then, the *SelectedFor* traceability relation is used to determine which properties of interest have been violated, so as to report this to the S&D Framework.
2. If a recommended rule is violated then the monitoring service raises a violation signal to the SERENITY framework but continues to monitor the other active rules without affecting the activation of the S&D pattern necessarily.

---

<sup>3</sup> This set has a single element.

## 6.4. Deactivation of monitoring rules

As we discussed earlier, recommended monitoring rules can be deactivated but compulsory rules cannot. The recommended rules for an S&D Property can be determined by using the *Satisfies* traceability relation between S&D Properties and monitoring rules. A rule can only be deactivated explicitly by a user, the SERENITY framework or an action of another rule. The deactivation of a rule means that it will not be checked by the monitoring service.

## 6.5. Detaching event collectors from system components

When particular S&D Implementations are deactivated, i.e. they are no longer selected, their event collectors may be detached from the components of these implementations. This should, however, happen only if these collectors are not also being used by other S&D Implementations. Event collectors can be detached in cases where they are found to be unreliable, e.g. in the case where they do not capture the required events in a bounded length of time.

In the first case, the deactivation of an event collector from a system component takes place as follows. First by using the *Uses* traceability relation between event collectors and S&D Implementations, the set of active S&D Implementations that are using a particular event collector is determined. Then, if an S&D Implementation is deactivated, the SERENITY framework can check whether there are other implementations that are still using the specific event collector and, if not, the event collector can be detached from its component.

In the second case, the deactivation of an event collector from a system component takes place through the following steps:

1. The SERENITY framework checks whether events are captured by an event collector within a bounded amount of time (the time can be determined by performing some analysis on the system). Other checks can be performed as well for determining the reliability/availability of the event collector.
2. By using the *SourceOf* traceability relation, for each component which has an event collector, a list of expected events can be determined.
3. Then if none of these events in the list have been observed in the given length of time, it can be deduced that the event collector is unavailable. If an event is observed by the event collector, then it is mapped to a high-level event by using the *Mapping* traceability relation and then the *SourceOf* relation is used to determine whether the event is in the set of the expected events<sup>4</sup>.

---

<sup>4</sup> More generally, there might be other factors that could affect the trustworthiness of events (or how confident we are that their absence is genuine and not the effect of some attack). In the initial version of the runtime monitoring components of SERENITY, we assume that all the captured events can be trusted and that the absence of events should also be trusted. In subsequent stages of the project, we will investigate possible trust models for event collectors and may need to provide more elaborate mechanisms for assessing event trust.



## 7. Example

---

In this section we illustrate the use of traceability relations in reference to a part of the smart items scenario that has been described in A7.D2.1 (see Section 2 in [8]). To enable this illustration, in the following we describe an S&D Pattern that describes a partial solution for a selected part of this scenario.

### 7.1. Smart Items scenario – A Solution

The solution that we describe in the following refers to the following requirements in A7.D2.1 [8]:

*“Each communication between the patient’s e-health terminal and the ERC shall guarantee message delivery, integrity and confidentiality of the data exchanged, and mutual authentication.”* (Req. 2.2.1.2)

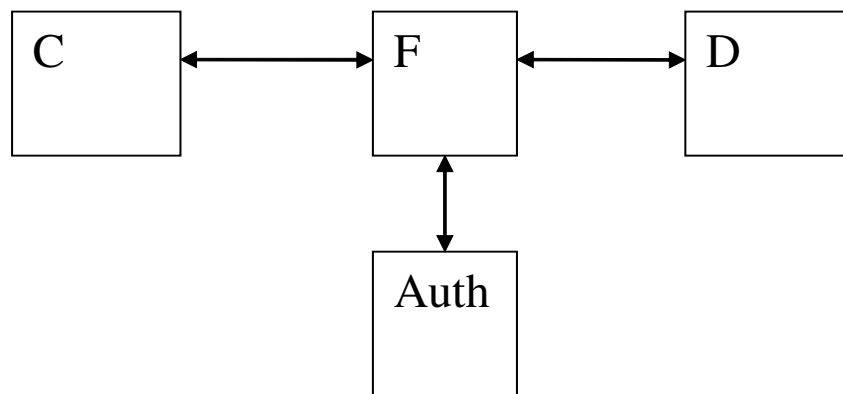
*“Each communication between the ERC and the e-health terminal of the selected doctor shall guarantee integrity and confidentiality of the data exchanged, and mutual authentication.”* (Req. 2.2.1.14)

The initial draft of our solution guarantees authorisation and confidentiality of the data exchanged between the ERC and the e-health terminals of patients and doctors but does not guarantee message delivery or authentication. This limitation is due to the fact that our objective here is to demonstrate the use of traceability relations rather than providing a complete solution to the above requirements.

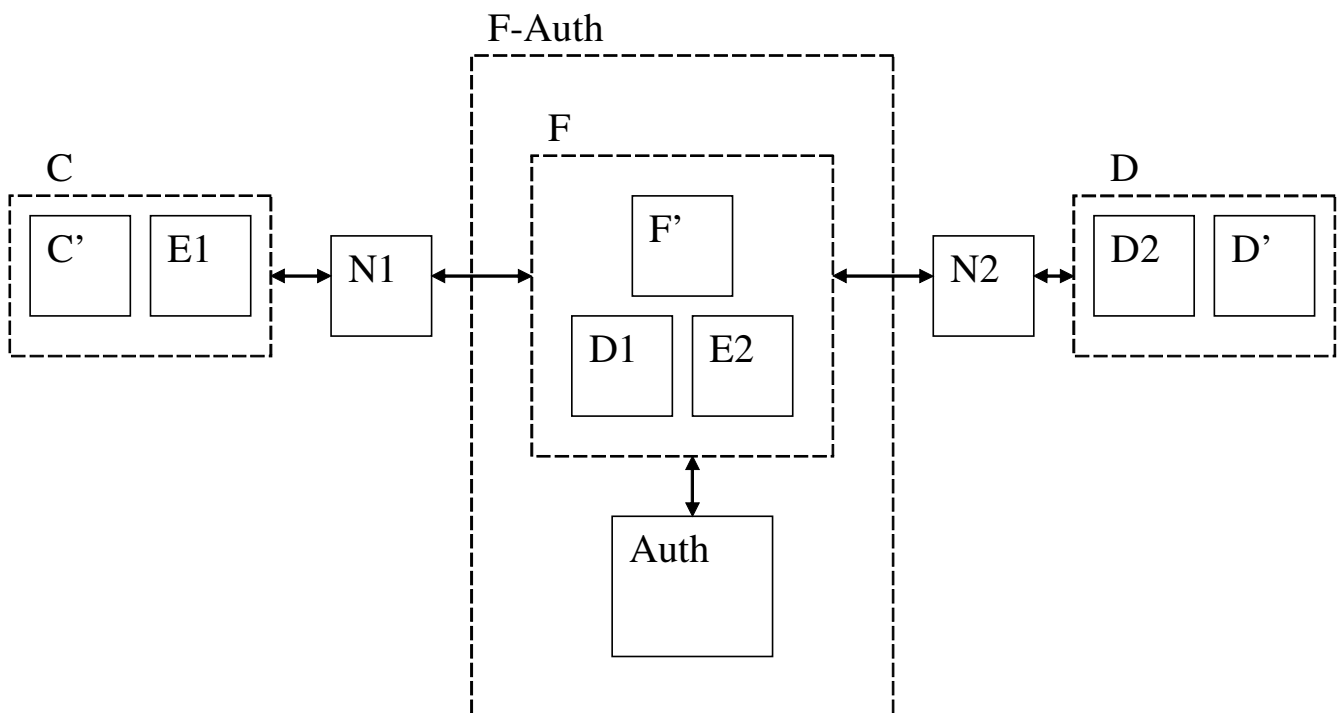
Also it should be noted that the description of our solution is based on the following assumptions:

- The patient’s and the doctor’s e-health terminals cannot communicate with each other directly during emergencies. All communications must go through the ERC (Emergency Response Centre). Therefore, in the case when a patient contacts the ERC with an emergency requiring a prescription, it is the ERC’s responsibility to ensure that only an authorised doctor can assign a prescription to this patient.
- Both the patient and doctor trust the ERC.
- Authorisation is provided by the ERC.
- There is a public key known to both the doctor and the ERC, and another public key known to both the ERC and the patient.

Furthermore, it should be noted that at this stage we do not focus on defining an executable implementation for this solution. Our focus is on defining a general S&D Pattern that could be used for the development of executable implementations. Thus, we consider only the general *architecture* of the solution and the *interactions* between the components in this architecture.



**Figure 4 – Architecture of the solution**



**Figure 5 – Decomposed architecture of the solution**

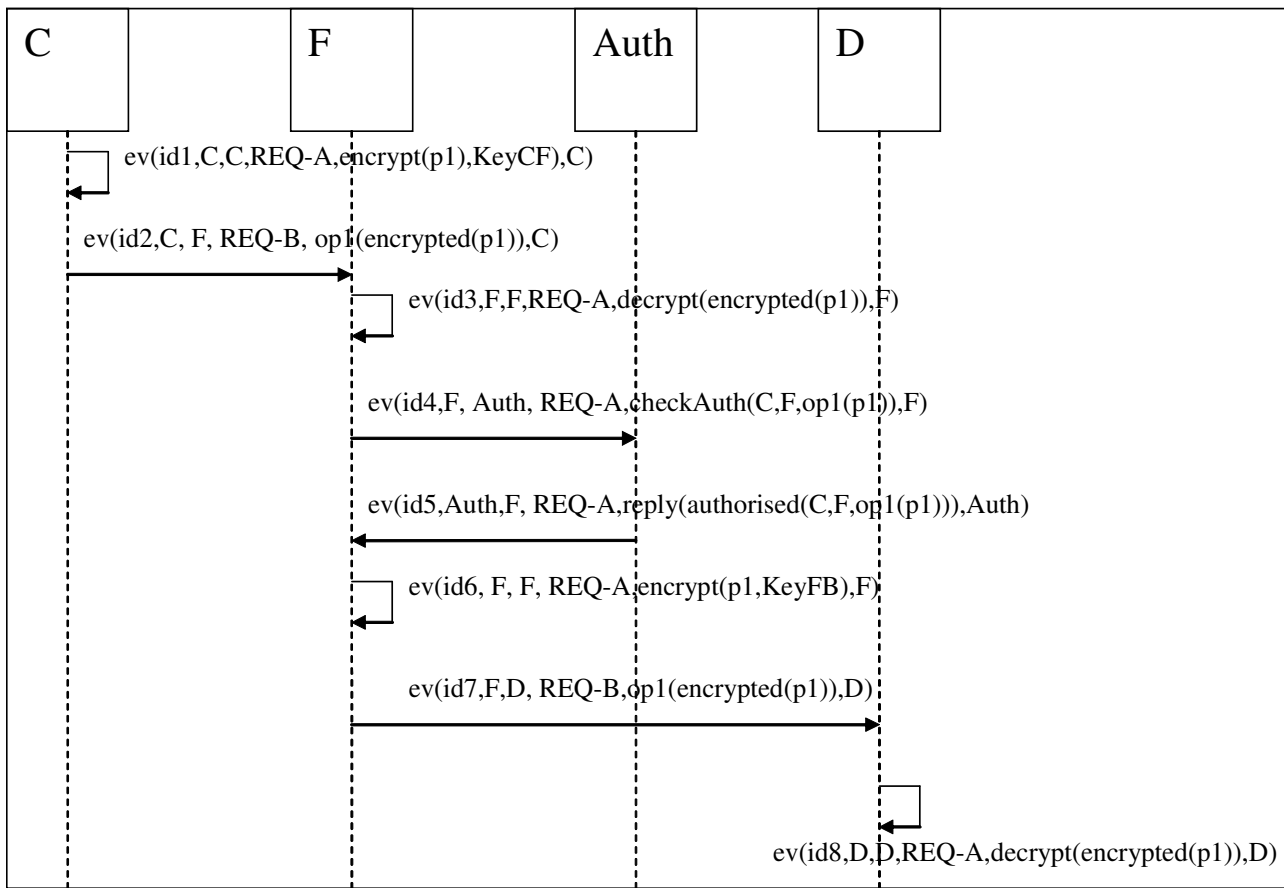
Figure 4 illustrates the architecture of the solution described using abstract components that could be unified with other components in different types of applications, i.e. not only in the smart items application. This architecture describes an indirect communication between two abstract components C and D that must take place through another intermediary abstract component F which also has responsibility for the authorisation of C and D in the communications. This authorisation is carried out by another abstract component called *Auth* that F communicates with. The architecture also describes the security realms in which the different parameters and components belong to, i.e., the fact that C' and E1 are in the same realm, just like the F', D1, E2 & Auth are in their own. For the smart items scenario, C would be unified with the doctor's e-health

terminal, *F* would be unified with the ERC, *D* would be unified with the patient's e-health terminal, and *Auth* would be unified with an authorisation function that is executed on the same machine as ERC.

The architecture of the solution can be further decomposed as illustrated in Figure 5, where the exact components that are responsible for performing certain functions are identified. For example, to ensure confidentiality, data encryption and decryption functions need to be introduced between the components *C* and *F*, and between *F* and *D*. Thus, the component *C* is refined into a decomposition that incorporates: (i) a component *C'* which provides the operations that are concerned with the normal system behaviour for *C*, and (ii) a component *E1* which provides the operations required for encryption/decryption. Similarly, the component *D* in Figure 4 is decomposed into a component *D'* that provides the normal operations of *D* and a component *D2* that provides the encryption/decryption operations for *D* and the component *F* is decomposed into the components *F1* that provides the normal operations of *F* and the components *D1* and *E2* that provide decryption and encryption functions for *F*. Note that encryption and decryption operations are not required for the communication between *F* and *Auth* since these two components are assumed to be running on the same machine and they may be event internal components of the same application. Thus, *Auth* does not have any encryption/decryption components.

The objective of this solution is to ensure that when a component *C* invokes an operation of *D* with some data, the data must be encrypted for ensuring confidentiality by using *E1* and together with the operation it is first sent to *F*, which decrypts it using *D1*, and checks that *C* is authorised according to *F* to invoke that operation. Once *F* checks that *C* is authorised by communicating internally with *Auth*, it forwards the original operation call from *C* (with the encrypted data) to *D* (before doing this *F* encrypts the data using *E2* with the public key of *D*). Subsequently, *D* decrypts the data using *D2* and executes the operation. In this solution, *F* functions as a proxy to *D*, filtering the calls made to it and allowing only the authorised ones.

The sequence diagram in Figure 6 illustrates some of the actions that can occur between the components illustrated in Figure 4. In the case where *C* is not authorised by *F* to invoke an operation of *D*, then nothing is sent to or invoked in *D*. Figure 7 illustrates the sequence of actions in this case.



**Figure 6 – Sequence diagram where F authorises C to invoke operation op1 of D**

In the smart items scenario, when a doctor sends a prescription to a patient the prescription goes to the ERC initially and the ERC, which plays the role of *F* in this case, needs to check that the doctor is authorised to assign the prescription. The prescription is sent by the doctor in an encrypted form (using the public key of ERC) that can be decrypted by the ERC (by using its private key). Once ERC has checked that the doctor is authorised to assign the prescription, it encrypts the prescription with the public key of the patient and sends the encrypted prescription to the patient, who can then decrypt it with his/her private key.

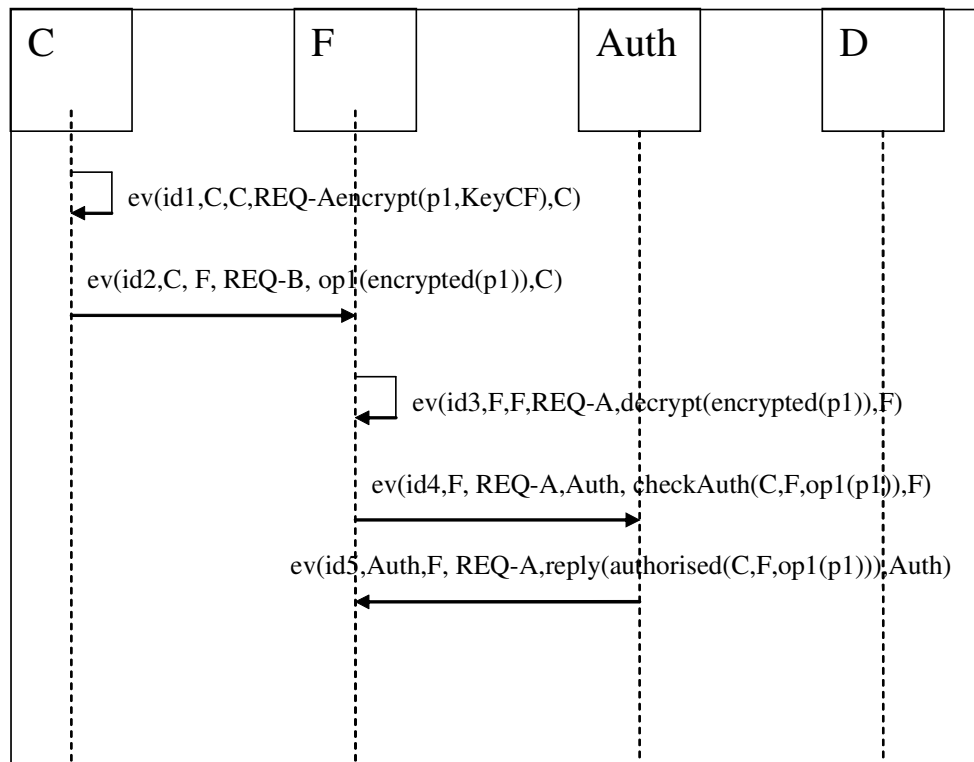


Figure 7 – Sequence diagram where F authorises C to invoke operation op1

## 7.2. S&D Properties

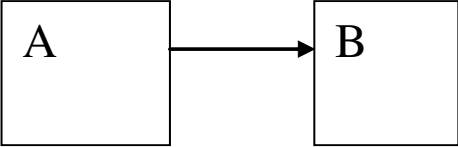
The language for an S&D Property has not yet been defined. In this section, we envision what this language might be and how we can use it to describe the properties that the solution ensures, and discuss any language requirements that we feel are necessary.

Each “type” of security requirement can correspond to more than one different S&D Properties. In the case of integrity, for example, a number of S&D Properties could be described depending on which system architecture integrity is applied to. Thus, we can have integrity of data sent between two components, integrity of data ensured by a third party, integrity of data stored on a single machine etc. Therefore, the language of an S&D property should allow for a description of the general system architecture that the property is applied to.

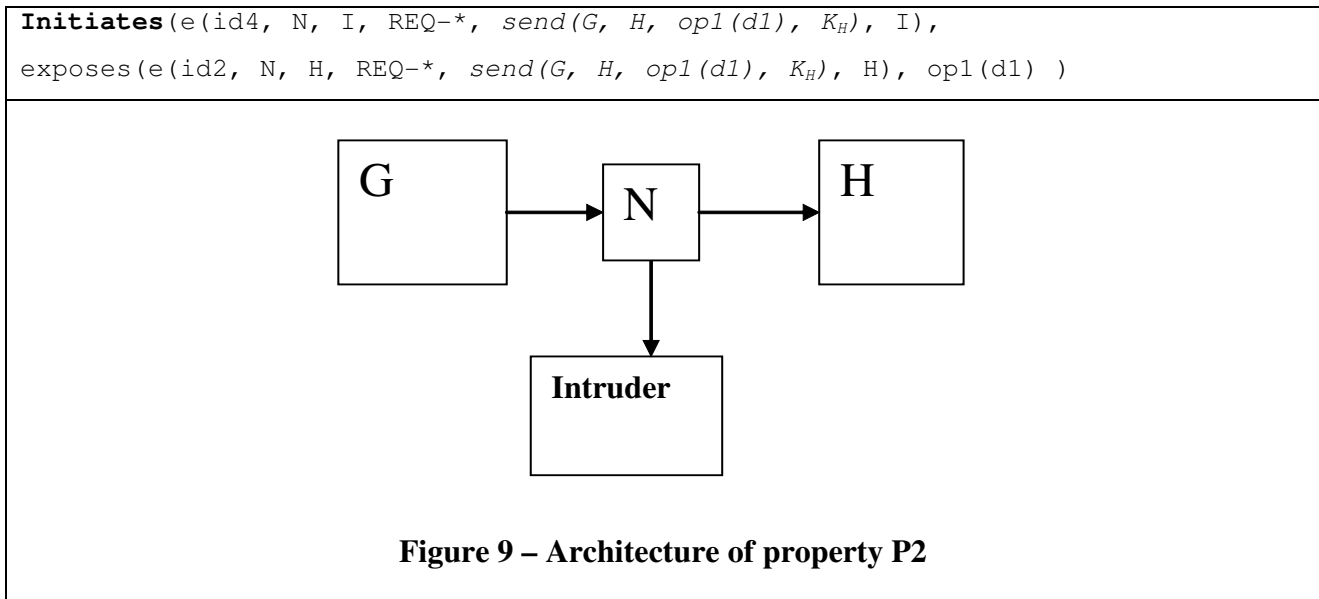
The language of the properties should allow for its description in English and also in a formal notation (to ensure that it is described precisely and unambiguously). The level of formality depends on the types of static and run-time analysis that we wish to perform – for example, in SERENITY we do not expect to see formulae describing the computational complexity of breaking a particular encryption method, since this type of analysis is out of the project’s scope.

In the following, we give examples of two S&D properties: a property expressing the need for authorisation (P1) and a property expressing the need for confidentiality. For both of these properties, we give a description of the property in English and a formalisation in event calculus. Our examples are given to demonstrate the need for incorporating some abstract description of the system architecture which the property refers to and the formal specification of the property. It

should be noted that although our examples use Event Calculus to demonstrate the formal specification of properties, the development of the final property specification language will be determined in the A5 activity in the next phase of SERENITY.

<b>S&amp;D Property : P1</b>
<b>Description (in English):</b> If an operation of B is invoked by agent A, then A must be authorised to invoke that operation in B. (Authorisation)
<b>Formal Description (in event calculus):</b> $(\forall t:\text{Time}; \text{Happens}(\text{ev}(\text{ID}, \text{A}, \text{B}, \text{RES-}^*, \text{op}(\text{d}), \text{B}), t, \mathfrak{R}(t, t)) \Rightarrow \text{HoldsAt}(\text{authorised}(\text{A}, \text{B}, \text{op}(\text{d})), t))$

<b>Figure 8 – Architecture of property P1</b>

<b>S&amp;D Property : P2</b>
<b>Description (in English):</b> The data which are transmitted between two components, <i>G</i> and <i>H</i> , through a network <i>N</i> remain confidential.
<b>Formal Description (in event calculus):</b> <b>Rule:</b> <b>P2.R1:</b> $\text{Happens}(\text{e}(\text{id2}, \text{N}, \text{H}, \text{REQ-}^*, \text{send}(\text{G}, \text{H}, \text{op1}(\text{d1}), \text{K}_\text{H}), \text{H}), t2, \mathfrak{R}(t1, t2)) \Rightarrow \neg \text{HoldsAt}(\text{exposes}(\text{e}(\text{id2}, \text{N}, \text{H}, \text{REQ-}^*, \text{send}(\text{G}, \text{H}, \text{op1}(\text{d1}), \text{K}_\text{H}), \text{H}), \text{op1}(\text{d1}))$  <b>Assumptions:</b> <b>P2.A1:</b> $\text{Happens}(\text{e}(\text{id1}, \text{G}, \text{N}, \text{REQ-}^*, \text{send}(\text{G}, \text{H}, \text{op1}(\text{d1}), \text{K}_\text{H}), \text{G}), t1, \mathfrak{R}(t1, t1)) \wedge \text{Happens}(\text{e}(\text{id2}, \text{N}, \text{H}, \text{REQ-}^*, \text{send}(\text{G}, \text{H}, \text{op1}(\text{d1}), \text{K}_\text{H}), \text{H}), t2, \mathfrak{R}(t1, t2)) \Rightarrow \text{Happens}(\text{e}(\text{id3}, \text{N}, \text{I}, \text{REQ-}^*, \text{send}(\text{G}, \text{H}, \text{op1}(\text{d1}), \text{K}_\text{H}), \text{I}), t2, \mathfrak{R}(t2, t2))$  <b>P2.A2:</b> $\text{Happens}(\text{e}(\text{id4}, \text{N}, \text{I}, \text{REQ-}^*, \text{send}(\text{G}, \text{H}, \text{op1}(\text{d1}), \text{K}_\text{H}), \text{I}), t, \mathfrak{R}(t, t)) \wedge \text{length}(\text{K}_\text{H}) < 100 \Rightarrow$



The specification of P1 makes reference to a simple architecture referring to two communicating agents (A and B) that is necessary in order to express the need for authorisation of one of them in the other. The formal specification of the property in EC states that when an event signifying the invocation of an operation  $op(d)$  by A in B occurs, A must have the appropriate form of authorisation by B.

The specification of P2 demonstrates a more complex example. In the architecture for this property, it is necessary to refer to the network N that exists between two agents G and H and the presence of an intruder (I) against which the confidentiality of the data should be preserved. The formal specification of the property is based on the following reasoning. When G wants to call an operation  $op1(d1)$  in H, it calls operation  $send(G, H, op1(d1), K_H)$  in N. N calls the operation  $send(G, H, op1(d1), K_H)$  in H. At the same time, we may assume that N also calls the same operation on an intruder I (since intruders can read messages which pass through the network). The assumption here is that H knows  $K_H$  but the intruder I doesn't – we do not care whether G knows  $K_H$  since we wish our property to describe both the case of a shared key and the case of a public/private key pair. The Intruder's capabilities are modelled by the fact that a message is revealed to it only if the length of the key  $K_H$  is less than a particular number, let's say 100.

*The formalisation of the attack model already signals a possible monitoring rule for any pattern which will try to offer this property for this attack model, that is, that the keys used for the exchange of messages have a length which is equal to or greater than 100.*

In EC, the property is specified by the rule P2.R1 which states that when G sends a message to H, the message is not exposed to any third party. The assumptions specified for this rule indicate the circumstances under which the property can be violated. P2.A1 expresses the fact that N will always send the encrypted message to an intruder (this is equivalent to assuming an intruder that always has the capacity to catch the encrypted message as discussed earlier). The second assumption (P2.A2) states that the contents of the message are exposed if the encryption key has a length that is less than the assumed threshold.

In this way we can easily model situations where the capabilities of the Intruder range from private persons, able to crack messages with short keys, to big companies, able to crack messages with medium keys, and even to countries, which can break messages with even longer key lengths.

The specifications of these properties including their descriptions in English and Event Calculus as well as the simple architecture models which are required are shown in Figure 9.

### **7.3. S&D Pattern**

In this section, we describe the S&D pattern for a solution of the smart items application using the schema specified in [4] extended with traceability relations that we have introduced in this deliverable.



<b>S&amp;D Pattern: AuthAndConfidentiality</b>	
	<b>Creator:</b> CUL.com
	<b>Trust mechanism:</b> signed by CUL
	<b>Provided Properties:</b>
	<b>Property:</b>
	<b>ID:</b> P1
	<b>Timestamp:</b> 200611161204
	<b>Property:</b>
	<b>ID:</b> P2
	<b>Timestamp:</b> 200611201206
	<b>Interface (this is for the components only):</b>
	<b>Calls:</b> reply(authorise(C,F,op1(p1)) authorise(C, F,op1(p1),result)
	<b>Components:</b>
	<b>Component:</b> Auth
	<b>Component:</b> F'
	<b>Component:</b> E1
	<b>Component:</b> E2
	<b>Component:</b> D1
	<b>Component:</b> D2
	<b>Parameters:</b>
	<b>Parameter:</b> C'
	<b>Parameter:</b> D'
	<b>Parameter:</b> N1
	<b>Parameter:</b> N2

	<b>Parameter:</b> p1
	<b>Parameter:</b> KeyCF
	<b>Parameter:</b> KeyFB
	<p><b>Parameter pre-condition:</b> Public key KeyCF is agreed on and known by C and F before session starts. (C and F trust each other)</p> <p>– Knows(C, KeyCF) <math>\wedge</math> Knows(F, KeyCF)</p>
	<p><b>Parameter pre-condition:</b> Public key KeyFB is agreed on and known by F and B before the session starts. (F and B trust each other)</p> <p>– Knows(F, KeyFB) <math>\wedge</math> Knows(B, KeyFB)</p>
	<p><b>Solution pre-condition:</b> F and B have private keys that are not disclosed.</p> <p>– Knows(F, KeyF) <math>\wedge</math> Knows(B, KeyB) <math>\wedge</math> <math>\neg(\exists I. I \neq F \wedge \text{Knows}(I, \text{KeyF})) \wedge \neg(\exists I. I \neq B \wedge \text{Knows}(I, \text{KeyB}))</math></p>
	<b>Solution pre-condition:</b> C and D cannot communicate with each other directly.
	<p><b>Solution description:</b> Order of events for correct behaviour</p> <p>ev(id1,C, C, REQ-A, encryptedC(p1,KeyCF),C);</p> <p>ev(id2,C,F, REQ-B, op1(encryptedF(p1)),C);</p> <p>ev(id3,F,F, REQ-A, decryptF(encrypted(p1)),F);</p> <p>ev(id4,F, Auth, REQ-A, checkAuth(C,F,op1(p1)),F);</p> <p>ev(id5,Auth,F, REQ-A, reply(authorise(C,F,op1(p1)),Auth);</p> <p>ev(id6, F,F, REQ-A, encryptF(p1,KeyFB),F);</p> <p>ev(id7,F,D, REQ-B, op1(encrypted(p1)),F);</p> <p>ev(id8, D,D, REQ-A, decryptD(encrypted(p1)),D)</p>
	<b>Static Tests Performed:</b>
	<b>Test:</b> ...
	<b>Conditions of test:</b>
	<b>Attack models considered:</b>

	<p><b>System Configuration:</b></p>
	<p><b>UnifiesWithComponent :</b></p> <p>Unification with the S&amp;D Property P1 (note that <math>\rightarrow</math> means unifies in this case):</p> <p>P1.A <math>\rightarrow</math> C P1.B <math>\rightarrow</math> D</p> <p>Unification with the S&amp;D Property P2:</p> <p>P2[1].G <math>\rightarrow</math> C P2[1].H <math>\rightarrow</math> F P2[2].G <math>\rightarrow</math> F P2[2].H <math>\rightarrow</math> D</p> <p>(Note that there are two “instances” of P2 within the pattern – one concerns the communication link between C &amp; F and the other concerns the link between F &amp; D.)</p> <p>-----</p> <p>The decomposition of components C, F, and D are (where <math>\parallel</math> represents parallel composition):</p> <p>C= C' <math>\parallel</math> E1 F= F' <math>\parallel</math> D1 <math>\parallel</math> E2 D= D' <math>\parallel</math> D2</p>
	<p><b>Monitoring:</b></p>
	<p><b>Rule 1:</b></p> <p><math>\forall t1, t2:Time;</math></p> <p><b>Happens</b> (ev (id1, C, F, REQ-A, op1 (encryptedC (p1)), F), t1, <math>\mathfrak{R}(t1, t1)</math>) <math>\wedge</math></p> <p><b>Happens</b> (ev (id2, F, D, REQ-B, op1 (encryptedF (p1)), F), t2, <math>\mathfrak{R}(t1, t2)</math>) <math>\Rightarrow</math></p> <p><b>HoldsAt</b> (authorised (C, F, op1 (encryptedC (p1))), t2)</p> <p>This rule checks whether C is authorised to invoke the operation <i>op1(p1)</i> in F. This rule is compulsory for P1 as we discuss in Section 7.6 below.</p>
	<p><b>Rule 2:</b></p> <p><math>\forall t1, t2:Time;</math></p> <p><b>Happens</b> (ev (id3, F, Auth, REQ-B, checkAuth (C, F, op1 (p1), res), F), t1, <math>\mathfrak{R}(t1, t1)</math>) <math>\Rightarrow</math></p> <p><b>Happens</b> (ev (id4, Auth, F, RES-B, checkAuth (C, F, op1 (p1), res), F),</p> <p>t2, <math>\mathfrak{R}(t1, t1+10)</math>)</p> <p>If F sends a <i>checkAuth</i> message to <i>Auth</i> (to check if C is authorised for F), then <i>Auth</i> must reply within 10 seconds. (This rule checks the availability of <i>Auth</i> based on events captured at F). This rule is recommended for P1 as we discuss in Section 7.6 below.</p>

	<p><b>Assumption 1 (for P1, P2):</b></p> <p><math>\forall t:Time;</math></p> <p><b>Happens</b> (ev(id9, Auth, F, RES-B, checkAuth(C, F, op1(p1), res), F), t, <math>\mathfrak{X}(t, t)</math>) <math>\wedge</math></p> <p><b>HoldsAt</b> (equalTo(res, True), t) <math>\Rightarrow</math></p> <p><b>Initiates</b> (ev(id9, Auth, F, RES-B, checkAuth(C, F, op1(p1), res), F), authorised(C, F, o1(p1)), t)</p> <p>If <i>Auth</i> replies to <i>F</i> with the result of the check for authorisation and the result is true, then the value of fluent <i>authorised(C, F, o1(p1))</i> is initiated.</p>
--	--

The difference between compulsory and recommended monitoring rules is that if a compulsory rule is violated during runtime, then the S&D Property that it ensures is violated too. While, if a recommended rule is violated during runtime, the S&D Property that it refers to is not necessarily violated. Note that Assumption 1 is for enabling the monitoring of S&D Property P1 and P2.

## 7.4. System S&D Configuration

The System S&D Configuration describes the unification of abstract pattern components and parameters with system components and the interfaces of the abstract pattern parameters and components with the interfaces of system components. Once established, these unifications are represented by the traceability relations *UnifiesWithComponent* and *UnifiesWithOperation*, respectively.

In the case of our example scenario and pattern, the unification of components and parameters for consist of the following relations:

*UnifiesWithComponent* (C', Doctor)

*UnifiesWithComponent*(E1, Doctor)

*UnifiesWithComponent* (D', Patient)

*UnifiesWithComponent* (D2, Patient)

*UnifiesWithComponent* (F', ERC)

*UnifiesWithComponent*(D1, ERC)

*UnifiesWithComponent*(E2, ERC)

*UnifiesWithComponent*(Auth, ERC)

*UnifiesWithComponent*(KeyCF, Public Key of ERC)<sup>5</sup>

*UnifiesWithComponent*(KeyFB, Public, Key of Patient)

<sup>5</sup> Some parameters are components, while others are not. Keys, for instance, can be modelled as parameters in S&D patterns and, therefore, they should be unified to concrete artefacts.

Furthermore,

- (i) The unification of the interface of the component *Doctor* consists of the following relations:

*UnifiesWithOperation (encryptC(p1,KeyCF), encrypt(prescription,ERC\_PublicKey))*

*UnifiesWithOperation(op1(encrypted(p1)), assignPrescription(encrypted(prescription)))*

- (ii) The unification of the interface of the component *Patient* consists of the following relations:

*UnifiesWithOperation (decryptD(encrypted(p1)), decrypt(encrypted(prescription)))*

- (iii) The unification of the interface of the component ERC consists of the following relations:

*UnifiesWithOperation (encryptF(p1,FB), encrypt(prescription, Patient\_PublicKey))*

*UnifiesWithOperation (decryptF(encrypted(p1)), decrypt(encrypted(prescription)))*

*UnifiesWithOperation(checkAuth(C,F,op1(p1)),*

*checkAuth(Doctor,ERC,assignPrescription(prescription)))*

## 7.5. Traceability between S&D Properties and S&D Patterns

The *Provides* relation between the S&D Pattern *AuthAndConfidentiality* and the S&D Properties P1 and P2 is shown in Table 6. According to these relations *AuthAndConfidentiality* specifies a solution that achieves the properties P1 and P2 changes. Thus, if any of the definitions of these properties changes than *AuthAndConfidentiality* should be checked for conformance and vice versa.

<i>Provides</i> Relations		<i>SelectedFor</i> Relations	
S&D Pattern	S&D Property	S&D Pattern	S&D Property
AuthAndConfidentiality	P1	AuthAndConfidentiality	P1
AuthAndConfidentiality	P2	AuthAndConfidentiality	P2

**Table 6 – *Provides* and *SelectedFor* relation between *AuthAndConfidentiality* and properties P1 and P2**

*SelectedFor* relations are also defined in S&D Configurations to indicate the S&D properties which an S&D pattern has been selected for. This relation in the case of the *AuthAndConfidentiality* S&D Pattern of our example is also shown in Table 6. According to this table, *AuthAndConfidentiality* was chosen for both the S&D properties P1 and P2 and hence the reference points to both of them.

## 7.6. Traceability between S&D Properties and Monitoring Rules

As defined earlier, the *Satisfies* traceability relation is defined between S&D Properties and monitoring rules to show which property is ensured by the checking of each rule. In the *AuthAndConfidentiality* S&D Pattern, these traceability monitoring rules between monitoring rules and the S&D properties which are provided by the pattern are shown in Table 7.

Properties: Rules:	P1	P2
Rule 1	Compulsory	
Rule 2	Recommended	

**Table 7 – *Satisfies* relations between S&D Properties and monitoring rules**

As specified in this table, *Rule 1* is compulsory for the property P1, since it ensures that no operation execution request that D receives from C through F has been forwarded to it, without F having checked the authorization rights of C first. *Rule 2*, on the other hand, has been only assigned to P1 as a recommended rule. This is because *Rule 2* checks a bounded form of the availability of *Auth* (i.e. whether *Auth* responds to an authorization check request within 10 time units). In this case, whilst the absence of any response to an authorization check request from *Auth* would affect property P1, the absence of a response from *Auth* within the specific time period of 10 time units which is set by *Rule 2* does not mean that *Auth* will not eventually respond. Thus, although Rule 2 specifies a recommended check whose failure can indicate a possible future violation of P1, its violation does not mean that P1 has also been violated or that it will definitely be violated.

## 7.7. Traceability between monitoring rules and events

*Contains* traceability relations between monitoring rules/assumptions and events identify the events that are described in the rules/assumptions. The *Contains* relation for the rules and assumptions of the *AuthAndConfidentiality* S&D Pattern is illustrated in Table 8.

Rule/Assumption	Event
Rule 1	ev(id1,C,F,REQ-A,op1(encryptedC(p1)),F)
Rule 1	ev(id2,F,D,REQ-B,op1(encryptedF(p1)),F)
Rule 2	ev(id3,F,Auth,REQ-B,checkAuth(C,F,op1(p1),res),F)
Rule 2	ev(id4,Auth,F,RES-B,checkAuth(C,F,op1(p1),res),F)
Assumption 1	ev(id9,Auth,F,RES-B,checkAuth(C,F,op1(p1),res),F)

**Table 8 – *Contains* relations between rules/assumptions and the events of *AuthAndConfidentiality* S&D Pattern**

## 7.8. Traceability between events and Components/Parameters

*SourceOf* traceability relations are defined between events and the components where these events will be captured from. In the case of the monitoring rule Rule 1 in our *AuthAndConfidentiality* pattern, both of the events which are referred to in the rule, i.e.  $ev(id1, C, F, REQ-A, op1(encryptedC(p1)), F)$  and  $ev(id2, F, D, REQ-B, op1(encryptedF(p1)), F)$  are to be captured at the component  $F$ .

Table 9 shows the *SourceOf* relations between all the monitoring rule/assumption events and the components/parameters of the *AuthAndConfidentiality* S&D Pattern. As indicated in this example events are not necessarily captured at the point where they were originally emitted. The event  $ev(id4, Auth, F, RES-B, checkAuth(C, F, op1(p1), res), F)$ , for instance, is originally emitted from the component *Auth* in the pattern but is to be captured at the component  $F$  which is the component that receives the event.

Auth	C	D	F
			$ev(id1, C, F, REQ-A, op1(encryptedC(p1)), F)$
			$ev(id2, F, D, REQ-B, op1(encryptedF(p1)), F)$
			$ev(id3, F, Auth, REQ-B, checkAuth(C, F, op1(p1), res), F)$
			$ev(id4, Auth, F, RES-B, checkAuth(C, F, op1(p1), res), F)$
			$ev(id9, Auth, F, RES-B, checkAuth(C, F, op1(p1), res), F)$

**Table 9 – *SourceOf* relation between events and components/parameters**

## 7.9. Traceability between Context and Monitoring Rules

As defined earlier, the context of an S&D pattern is defined as the set of the pre-conditions and the invariants of the pattern and the *CheckedBy* relations that associate context with monitoring rules need to be defined between invariants and monitoring rules. In the case of the *AuthAndConfidentiality* S&D Pattern there are no invariants however. Thus, the *CheckedBy* traceability relation is not defined for this example.

## 7.10. Traceability between Monitoring Rules

In the case of the *AuthAndConfidentiality* S&D Pattern, a *DependOn* traceability relation exists only between *Assumption 1* and *Rule 1*. This dependency is not explicit; it arises implicitly through the following dependencies:

- *DependsOn(Rule 1, axiom EC4)* (EC4 is one of the event calculus axioms which are defined in Table 10)
- *DependsOn(axiom EC4, Assumption 1)*

(EC1) $\text{Clipped}(t1, f, t2) \Leftarrow (\exists e, t) \text{Happens}(e, t, \mathfrak{R}(t1, t2)) \wedge \text{Terminates}(e, f, t)$
(EC2) $\text{Declipped}(t1, f, t2) \Leftarrow (\exists e, t) \text{Happens}(e, t, \mathfrak{R}(t1, t2)) \wedge \text{Initiates}(e, f, t)$
(EC3) $\text{HoldsAt}(f, t) \Leftarrow \text{Initially}_P(f) \wedge \neg \text{Clipped}(0, f, t)$
(EC4) $\text{HoldsAt}(f, t2) \Leftarrow (\exists e, t) \text{Happens}(e, t, \mathfrak{R}(t1, t2)) \wedge \text{Initiates}(e, f, t) \wedge \neg \text{Clipped}(t, f, t2)$
(EC5) $\neg \text{HoldsAt}(f, t2) \Leftarrow (\exists e, t) \text{Happens}(e, t, \mathfrak{R}(t1, t2)) \wedge \text{Terminates}(e, f, t) \wedge \neg \text{Declipped}(t, f, t2)$
(EC6) $\neg \text{HoldsAt}(f, t) \Leftarrow \text{Initially}_N(f) \wedge \neg \text{Declipped}(0, f, t)$
(EC7) $\text{HoldsAt}(f, t2) \Leftarrow \text{HoldsAt}(f, t1) \wedge t1 < t2 \wedge \neg \text{Clipped}(t1, f, t2)$
(EC8) $\neg \text{HoldsAt}(f, t2) \Leftarrow \neg \text{HoldsAt}(f, t1) \wedge (t1 < t2) \wedge \neg \text{Declipped}(t1, f, t2)$
(EC9) $\text{Happens}(e, t, \mathfrak{R}(t1, t2)) \Rightarrow (t1 \leq t2) \wedge (t1 \leq t) \wedge (t \leq t2)$

**Table 10 – Standard axiomatic definition of event calculus**

## 7.11. Traceability between Monitoring Rules and S&D Implementations

As in this document we have not defined the complete solution for the *AuthAndConfidentiality* S&D Pattern including its S&D Implementation, executable implementations and S&D Configuration, it is not possible to illustrate how these elements will be used in traceability relations. However, what would happen in this case is that low-level events will be mapped to high-level events according to a theory. This mapping can occur at three different places and each has advantages and disadvantages. Where this mapping is to occur is a design decision for the SERENITY framework.

We assume that this mapping allows the SERENITY framework to be able to trace back (using the *Mapped* traceability relation) from a high-level event to a low-level event and vice versa.

The *Uses* traceability between event collectors and S&D Implementations for the *AuthAndConfidentiality* S&D Pattern is illustrated in Table 11. This relation is described in the S&D Configuration. According to events given in the monitoring rules, event collectors will be placed at all of the components, i.e. at the ERC, at the Patient and at the Doctor. For the sake of the example, we assume that two executable implementations exist, one implemented in Java (JavaIMP) and one in C++ (C++IMP). Moreover, only the Java implementation is active.

Event collector	Implementations
EC1 at Patient	JavaIMP
EC2 at ERC	JavaIMP
EC3 at Doctor	JavaImp

**Table 11 – The *Uses* traceability relation between event collectors and implementations.**

The *CollectedBy* traceability relation between event collectors and events describes the set of events that can be captured by an event collector. Table 12 illustrates the *CollectedBy* traceability relation for the *AuthAndConfidentiality* S&D Pattern.



Event collector	Events
EC2	$ev(id1, C, F, REQ-A, op1(encryptedC(p1)), F)$
EC2	$ev(id2, F, D, REQ-B, op1(encryptedF(p1)), F)$
EC2	$ev(id3, F, Auth, REQ-B, checkAuth(C, F, op1(p1), res), F)$
EC2	$ev(id4, Auth, F, RES-B, checkAuth(C, F, op1(p1), res), F)$
EC2	$ev(id9, Auth, F, RES-B, checkAuth(C, F, op1(p1), res), F)$
EC1	
EC3	

**Table 12 – The *CollectedBy* traceability relation between event collectors and events**

## 7.12. Traceability between Monitoring Rules and S&D Configuration

The *UnifiesWithComponent* and *UnifiesWithOperation* traceability relations between monitoring rules and S&D Configuration are defined by the unification of components/parameters and operations with the actual system components and operations/interface.

The parameters that appear in the monitoring rules are unified with the actual system component using the *UnifiesWithComponent* traceability relation, as described in Section 5.

The monitoring rules for the *AuthAndConfidentiality* S&D Pattern consist of events with the following IDs: id2, id4, id5, id8, id9. Two of these events occur at the component *Auth* (part of the ERC). For all the events that occur at parameters, we have to trace with *UnifiesWithOperations* their operations to the actual operations defined in the parameter interface that is found in the S&D Configuration. Table 13 illustrates the traceability between operations in events and operations in the parameter interface for the *AuthAndConfidentiality* S&D Pattern.

Event	Operation in parameter interface
$ev(id2, F, D, REQ-B, op1(encryptedF(p1)), F)$	$assignPrescription(encrypted(prescription))$
$ev(id4, F, Auth, REQ-A, checkAuth(C, F, op1(p1), res), F)$	$checkAuth(Doctor, ERC, assignPrescription(prescription))$

**Table 13 – The *UnifiesWithOperation* traceability relation between operations in high-level events and operations in parameter interface**

## 8. Conclusion and Future Work

In this document we have defined a number of horizontal and vertical traceability relations between the S&D modelling artefacts that support the representation of S&D Solutions, i.e. S&D Properties, S&D Patterns, S&D Implementations and S&D Configuration. These traceability relations are of importance for runtime monitoring and we have illustrated how they can be used for activating and deactivating monitoring rules, checking rules and for attaching and detaching event collectors. We propose that the S&D modelling artefacts be extended in order to represent these traceability relations. The S&D modelling artefacts that need to be extended are: S&D Patterns, S&D Configurations and S&D Implementations. Table 14 summarises the traceability relations that we have defined and indicates whether the relation requires the current S&D modelling artefacts to be extended.

Name	Type	Definition	Place of definition	Extension required
Provides	Dependency	Patterns $\rightarrow$ 2 Properties	S&D Patterns	No
SelectedFor	Rationalisation	Patterns $\rightarrow$ 2 Properties	S&D Configuration	No*
Satisfies	Satisfaction	Rules $\times$ {Compulsory, Recommended} $\times$ Properties	S&D Patterns	Yes
Contains	Dependency	Rules $\times$ Events	S&D Patterns	No
SourceOf	Dependency	Components $\times$ Events	S&D Patterns	No
CheckedBy	Generalisation	Invariants $\rightarrow$ 2 <sup>Rules</sup>	S&D Patterns	No (because it can be represented by the extension for Satisfies)
DependsOn	Dependency	Rules $\rightarrow$ 2 <sup>Rules <math>\cup</math> Assumptions</sup>	S&D Patterns	Yes

Mapped	Evolution	High-level events × Low-level events	Extra theory in the S&D Implementation or implicitly in the executable of the S&D Implementation.	No*
Uses	Dependency	Implementations × Event Collectors	S&D Implementation	No*
CollectedBy	Dependency	Event → Event collector ∪ {nil}	S&D Implementation	No*
UnifiesWithComponent	Satisfaction	(Patterns × Parameters) → Components	S&D Configuration	No*
UnifiesWithOperation	Satisfaction	(Patterns × Abstract Operations) × Concrete Operations	S&D Configuration	No*

**Table 14 – Summary of traceability relations**

Note that “No\*” in Table 14 means that the S&D modelling artefact has not yet been developed and hence the traceability relation can be assumed to be part of the representation language. Therefore, the embodiment of the traceability relation does not require an extension of an existing language representing the S&D modelling artefact.

## 8.1. Summary of Required Extensions

As shown in Table 14, certain traceability relations either require extensions to current S&D artefacts or require particular information to be included in artefacts which have not yet been fully developed, e.g., S&D Properties. In this section we will summarise these extra requirements by examining each artefact in turn.

- **S&D Properties:** These need to be described in a language *similar to S&D Patterns*, providing (abstract) interfaces for the components which participate in the description of the relevant property.
- **S&D Patterns:** These need to be extended with an *internal configuration*, which unifies the components of the provided properties with parameters/components of the pattern. This unification is needed both for the static verification of the properties, as well as, by the run-time monitoring of the assumptions and rules expressed in the S&D Properties that the S&D Pattern is providing. For the same reasons, the S&D Property operations must be unified as well to the operations used in the S&D Pattern.

S&D Patterns also need to include information for the *Satisfies relation*, i.e., which S&D Properties are satisfied by which rules and whether the latter are compulsory or recommended for each S&D Property.

The description of the components/parameters may need to be extended to include the *component's realm*, so as to help in the analysis of the S&D Pattern. Another possibility would be to allow *composite components* like C, which is the parallel composition of C' and E1 in the S&D Pattern of Section 7.3, and assuming that a composite component also represents a particular realm. We consider this assumption a bit weak, since the two notions – realm and composition – are not the same, so overloading composition would probably cause confusion later on. We most probably need to support both.

Finally, we believe that it would be greatly beneficial if S&D Patterns would include an *architectural description of the system*, i.e., how the different components/parameters are connected, through what connectors, etc.

- **S&D Implementations:** These must include information for the *Mapped relation*, if the mapping of low-level to high-level events is to be performed by the monitor engine, as well as information for the *Uses* and *CollectedBy relations*.
- **S&D Configuration:** Probably the most important extension is the requirement for introducing *local configurations in the S&D Patterns* (see corresponding item above) for unifying the S&D Pattern artefacts with those of the S&D Properties it provides.

Along with this change, S&D Configurations need to include information for the *SelectedFor relation*, which is used to document the intent of the system designer and to make possible the selection of the run-time rules, and for the *UnifiesWithComponent* and *UnifiesWithOperation relations*, which are used to instantiate the S&D Pattern parameters to exact artefacts of the final system.

Since the language for describing the S&D modelling artefacts is still under development we expect that changes will need to be introduced to the traceability relations discussed in this deliverable, once the language is finalised. Moreover, with increased experience of applying these traceability relations, we might find that some changes are required, that further traceability relations must be defined or that certain relations/relation information are not really needed, e.g., because they are encoded implicitly otherwise or because the goal for which they have been introduced can be reached through other mechanisms. Finally, we still need to investigate the traceability relations between the S&D modelling artefacts described in this deliverable with the S&D Integration Schemes. All these will be investigated and discussed in the next traceability deliverable.

## References

- [1] Kloukinas C., Spanoudakis G., Ballas C., Presenza, D. (2006) A4.D2.2 - Basic set of Information Collection Mechanisms for Run-Time S&D Monitoring
- [2] Lindval M., Sandahl K., (1996) Practical Implications of Traceability, Software Practice and Experience, vol. 26, no. 10, pp 1161-1180
- [3] Mahbub K., Spanoudakis G. (2004) A Framework for Requirements Monitoring of Service Based Systems , 2nd International Conference on Service Oriented Computing, New York. . Also available from <http://www soi.city.ac.uk/~gespan/icsoc04.pdf>
- [4] Mana A., Munoz A., Sanchez F., Serrano D. (2006) A5.D2.1 – Patterns and Integration Schemes Languages (First Version)
- [5] Shanahan M. P. (1999) The Event Calculus Explained, in Artificial Intelligence Today, ed. M.J.Wooldridge and M.Veloso, Springer Lecture Notes in Artificial Intelligence no. 1600, ), pages 409-430, Springer. Also available from <http://www.doc.ic.ac.uk/~mpsha/ECEexplained.pdf>
- [6] Spanoudakis G. Mahbub K, (2006) Non Intrusive Monitoring of Service Based Systems , International Journal of Cooperative Information Systems, Vol. 15, No. 3, 325-358. Also available from <http://www soi.city.ac.uk/~gespan/ijcis06.pdf>
- [7] Spanoudakis G., Zisman A., (2005) Software Traceability: A Roadmap, in "Handbook of Software Engineering and Knowledge Engineering - Vol. 3 Recent Advances", Shi-Kuo Chang (ed.), p. 395-428, World Scientific Publishing Co., ISBN 981-256-273-7.
- [8] Campadello S., Compagna L., Gidoin D., Giorgini P., Holtmanns S., Latanicki J., Meduri V., Pazzaglia J-C., Seguran M., Thomas R., Zanone N., (2006) A7.D2.1 – S&D Requirements specification.
- [9] Melton R. and Garlan D. (1997) Architectural Unification, in “CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research”, p. 18, Toronto, Ontario, Canada, IBM Press. Also available from <http://citeseer.ist.psu.edu/melton97architectural.html>
- [10] Spanoudakis G., Kloukinas C., Androutsopoulos K. (2007), Towards Security Monitoring Patterns, Proceedings of the 22<sup>nd</sup> Annual ACM Symposium on Applied Computing, Technical Track on Software Verification (to appear)
- [11] Mahbub K. (2007), Requirements Monitoring of Service Based Systems, PhD Dissertation, Department of Computing, City University.