A5.D2.3 – Patterns and Integration Schemes Languages (Second Version)

A. Botella, L. Compagna, P. El Khoury, C. Kloukinas, K. Li, A. Maña, A. Muñoz, G. Pujol, A. Saidane, F. Sanchez-Cid, J. Salvador, D. Serrano, G. Spanoudakis, S. Sinha

Document Number	A5.D2.3
Document Title	Patterns and Integration Schemes Languages (Second Version)
Version	1.0
Status	Final
Work Package	WP 5.2
Deliverable Type	Report
Contractual Date of Delivery	31 December 2007
Actual Date of Delivery	15 February 2008
Responsible Unit	UMA
Contributors	SAP, CUL, UTN
Keyword List	
Dissemination level	PU

Change History

Version	Date	Status	Author (Unit)	Description
0.1	09/02/08	Draft	Francisco Sanchez-Cid (UMA)	Added new sections. Document revised
1.0	13/02/08	Final	Francisco Sanchez-Cid (UMA)	Revised from Quality Check

Executive Summary

This document is a precise description of the modelling artefacts used in the description of S&D Solutions. These artefacts range from *S&D Patterns* (and *Integration Schemes*) and *S&D Classes* to *S&D Implementations*. This document describes their conceptual meanings and proposes a structured language for expressing them, along with an XML-based representation for this language. The second version of the language, presented in this document, provide the readers with guidance on how to correctly use the modelling artefacts to describe generic S&D Solutions. Thus, every field in the artefacts is concisely described, in some cases coming with an example of use. Finally, with the aim of building an illustrative guide for those novels to the Serenity approach, the document also introduces some basic concepts of the Serenity Architecture.

This particular release contains some important changes from the Initial Version of the Language. A briefing of the major additions is listed below:

- There is a new proposal for the analysis and specification of “Pre-conditions”: Section 4.1.2. presents the structure and the syntax of the preconditions as well as the guidelines for their creation and later evaluation.
- For the S&D Patterns, it was necessary to formally define the structure and syntax for (i) the declaration of Operations and (ii) the Class Adaptor. After studying several approaches, ASL has been selected as the most suitable one. Action Specification Language (ASL) is a pseudo-language independent of (i) the target platform and (ii) the implementation language. It is strongly related to xUML, but it can be used independently. It provides a simple way to define the operations inside the patterns. There is literature about it [2], but nevertheless, a short description of ASL including the minimum knowledge necessary to codify the Pattern’s has been created and made available for internal use [1]. Sections 4.3. and 4.3.1. cover this issue.
- We have introduced the concept of “role” in the S&D Class definition: Two Patterns belonging to the same Class, can play “different roles” in the application. E.g. Server/Client role. Section 4.2. deals with this issue.

Some of the concepts already presented in First Version have been clarified:

- The “timestamp” field has been formally defined for the three S&D Artefacts (sections 4.2. , 4.3. and 4.4.).
- The “Naming Scheme” for the identification of S&D Classes, Patterns and Implementation is now formalized and standardized (section 4.1.1.).
- The “Creator” field (common to all the three Artefacts) now contains the “Name” of the creator and the “Date” of creation.
- A “Comments” field has been added to the S&D Artefacts.
- “System Patterns” are now known as “Event Observer Patterns”.

Some elements of the language had a vague definition. New and more detailed descriptions have been included and, in some cases, new names had been used to avoid confusion:

- There is no “Executable Implementation” now, but “Executable Component”.

- "Components" of the S&D Patterns are now named as "Parts".
- Initial Version of the language did not provide a clear distinction between Parameters and Components ("Parts" from now on). This issue has been addressed in the revision.
- The "Interface Adaptor" is now a "Class Adaptor", meant to adapt from Class' Calls to Patterns' Operations.

And finally, and as a consequence of the previous changes:

- Section 5. has been carefully revised to include the newly defined fields, to update the name of some elements here and there, etc… This includes: (i) revising figures/tables containing XML Schemes and (ii) revising S&D Artefacts defined in XML language.
- The same applies to section 6. , where (i) all tables representing the structure of the S&D Artefacts have been revised and (ii) the appendix (section Appendix A) now includes full-revised XML Schemes expressed in XML.

Upcoming versions of the language will include detailed descriptions of the following issues:

- Final definition of Integration Schemes: how to create them. Examples of use.
- Final definition of S&D Implementations: references to the *Executable Components*.
- Study of possible Post-Conditions.
- Key Features: structure, representation, and use.
- Definition of Global IDs for S&D Artefacts.
- Parameters: specification, data structure, and use.
- Static Tests Performed: evidences, formal proofs, verification of S&D Artefacts Threat Models considered.
- Trust Mechanisms.
- System Configuration.
- Define a version control system for S&D Artefacts.
- Update of monitoring fields.

Table of Contents

1. Introduction

The modelling and representation of security and dependability solutions (S&D Solutions) is one of the biggest challenges in SERENITY. This representation is strongly related to the SERENITY Conceptual Model [3], to the design of the Runtime Architecture and to the Runtime Monitoring activity. This introduction tries to put it all together by first, presenting the main concepts the user should get familiar with, then introducing the Serenity modelling artefacts, eventually showing how these artefacts fit in the Serenity Runtime and Development time architecture.

1.1. Modelling context

Before we start dealing with the artefacts that we will use for modelling S&D Solutions, we must describe our envisaged scenario and define some basic terms. We must emphasize that our work is focused on the modelling of S&D Solutions for Ambient Intelligence (AmI) scenarios. In fact, the new scenarios of Ambient Intelligence, their underlying pervasive technology, and their notion of mobile services –where the IT environment moulds itself around the user's needs, raise the bar for what is a satisfactory security and dependability solution well beyond standard IT security technology. For this reason we expect our results to be applicable in many other (probably less demanding) scenarios.

The scenarios of Ambient Intelligence introduce a new computing paradigm and set new challenges for the design and engineering of secure and dependable systems. In these scenarios the concepts of system and application as we know them today will disappear, evolving from static architectures with well-defined pieces of hardware, software, communication links, limits and owners, to architectures that will be sensitive, adaptive, context-aware and responsive to users' needs and habits. We will refer to these architectures as *AmI ecosystems*. These AmI ecosystems will offer highly distributed dynamic services in environments that will be heterogeneous, large scale and nomadic, where computing nodes will be omnipresent and communication infrastructures will be dynamically assembled. This is the scenario where our work on modelling security and dependability solutions will be applied. The most important aspects to take into account in this scenario are the highly distributed nature of the computing model and the combination of heterogeneity, dynamism and large number of computing and communication elements, controlled by different entities. All these characteristics make matters worse when it comes to designing and operating the necessary security mechanisms. For this reason, it is essential that these security mechanisms can adapt themselves to the ever-changing AmI context. Consequently, our main goal in the modelling of security and dependability solutions becomes the ability to use the models for *automated selection* and *adaptation* of the security and dependability mechanisms by automated means.

Before we proceed, some terms are defined in order to facilitate subsequent explanations.

- **AmI ecosystem:** We define an AmI ecosystem as the composition of multiple systems controlled by multiple authorities (usually the system owner). In particular, this means that for every system that is part of the ecosystem there is an authority that is responsible for its security and dependability.
- **S&D Authority:** Entity that is responsible for the security and dependability of a system or set of related systems.
- **S&D Realm:** A set of systems controlled by one S&D Authority is called an S&D Realm. In practice it is frequent for an authority to control more than one system. This happens for

instance in the case of a corporate network composed of multiple computing and communication devices. We call SERENITY Realm to a SERENITY-enabled S&D Realm. It is possible for a realm to have nested realms.

- **S&D Property:** An S&D Property is a quality of a system that enhances its security or dependability in some way.
- **S&D Requirement:** An S&D Requirement is the expression of the need for an S&D Property to hold on a system or part of it.
- **S&D Solution:** An S&D Solution is defined as a mechanism that is used to realize some S&D Requirement.

Figure 1 shows a graphical representation of the concepts defined above. It depicts a fictional AmI environment composed by six realms. *S&D Realm 1* is composed by four systems—managed by *S&D Authority 1*, and other two realms: *S&D Realm 4* and *5* –managed by different authorities. Considering “Computing Department” as *S&D Realm 1*, we can think of *S&D Realm 4* as a laptop owned by a lecturer. Although the laptop remains inside the Computing Department, the lecturer is the one with administrative privileges on his own laptop and, consequently, the lecturer is also the S&D Authority for what concerns the laptop (i.e. *S&D Realm 4*). The lecturer –as S&D Authority–, must comply with the policies imposed by *S&D Authority 1*, but to any extent the lecturer is the unique authority with capacity to manage the *SERENITY Runtime Framework (SRF)* of the *S&D Realm 4*.

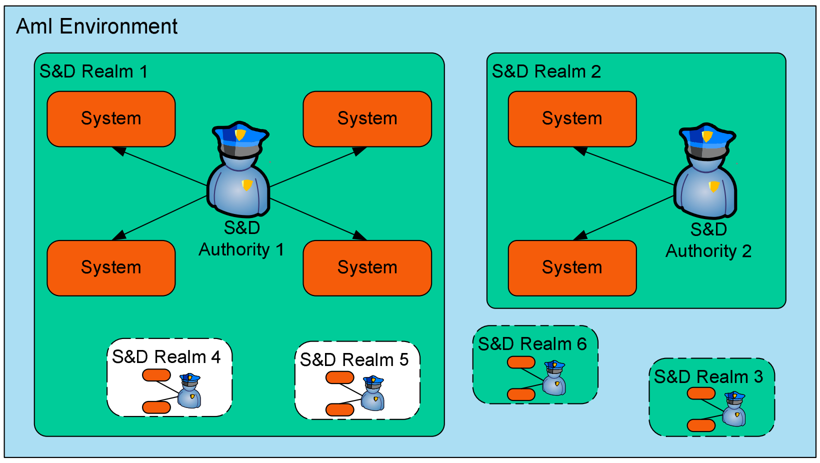

Figure 1 – Relations between the modelling artefacts

1.2. Artefacts for modelling S&D Solutions

The representation of S&D Solutions in SERENITY is supported by three main artefacts: S&D Classes, S&D Patterns and S&D Implementations. In this section we will define them, describe them in detail, and justify their structure and usefulness. Before continuing, the three main concepts must be introduced:

- **S&D Patterns** represent abstract S&D solutions. These solutions are well-defined mechanisms that provide one or more S&D Properties. There is a special type of S&D Pattern that represents the combination of several S&D Patterns. This type of S&D Patterns is called

Integration Schemes. The popular Needham-Schroeder public key protocol is an example of an S&D solution that can be represented as an **S&D Pattern.** One important aspect of the solutions represented as **S&D Patterns and Integration Schemes** is that they can be statically analysed using *S&D Engineering Tools* (in particular, Activities 1, 2 and 3 of the SERENITY project will produce such tools). However, the limitations of the static analysis tools introduce the need to support the dynamic validation of the behaviour of the described solutions by means of monitoring mechanisms.

— **S&D Classes** represent abstractions of a set of **S&D Patterns** characterized for providing the same S&D Properties and complying with a common interface. This artefact is mainly used at development time by the *SERENITY Development Tools*, as will be described in section 2. of this document. An example of an S&D Pattern Class is the *Confidentiality* Class[1], which defines an interface that includes the *SendConfidential(Data, Recipient)* abstract method. S&D Patterns and Integration Schemes that belong to an S&D Class can have different interfaces, but they must describe how these specific interfaces map into the S&D Class interface. The way to express this correspondence is sections 4.3. and 4.3.1. later in this document. The main purpose of introducing this artefact is to facilitate the dynamic substitution of the S&D mechanisms at runtime. This is a basic pillar behind the idea of the Artefacts: first, select an abstract definition at development time (i.e. abstract methods from Classes); second, have several patterns complying to this definition (by means of their Class Adaptor); and third, at runtime, the patterns will be selectable and interchangeable because (though having different interfaces) they all comply with the same abstract one. Given that interoperability is a key issue at this level, with this approach it is possible to create an application bound to S&D Class, as this artefact defines the high-level interface (i.e. the set of functions, calls, or methods that form the functionality offered by an artefact).

Thus, given that artefacts in an S&D Library have a reference to the higher level artefact they belong to, it is always possible to track back from an Executable Component to its S&D Class in three backward steps maximum. In conclusion, all S&D Patterns (and their respective S&D Implementations) belonging to an S&D Class will be selectable by the framework at runtime.

— **S&D Implementations** represent working S&D Solutions. It is important to note that the expression "working solutions" refers here to any final solution (e.g. component, web service, library, etc.) that has been implemented and tested. These solutions are made accessible to applications thanks to the *SERENITY Runtime Framework (SRF)*. The description of either a specific dynamic library providing encryption services or a web service providing timestamping services (both including a reference to its corresponding Executable Component), are examples of S&D Implementations. At this stage, it is important to note that the physical implementation (either software or hardware) of an S&D Patterns corresponds to an *Executable Component* pointed by an S&D Implementation, and not to the S&D Implementation itself. In fact, an S&D Implementation describes not just an implementation of the S&D Solution, but describes an implementation of an S&D Pattern. This means that all S&D Implementations of an S&D Pattern must conform directly to the interface, monitoring capabilities, and any other characteristic described in the S&D Pattern. However, they may have differences, such as the specific context conditions that must be met before applying one specific S&D Implementation,

[1] This class is described in detail in section 0of this document

their performance, target platform, programming language or any other feature not fixed at pattern's level.

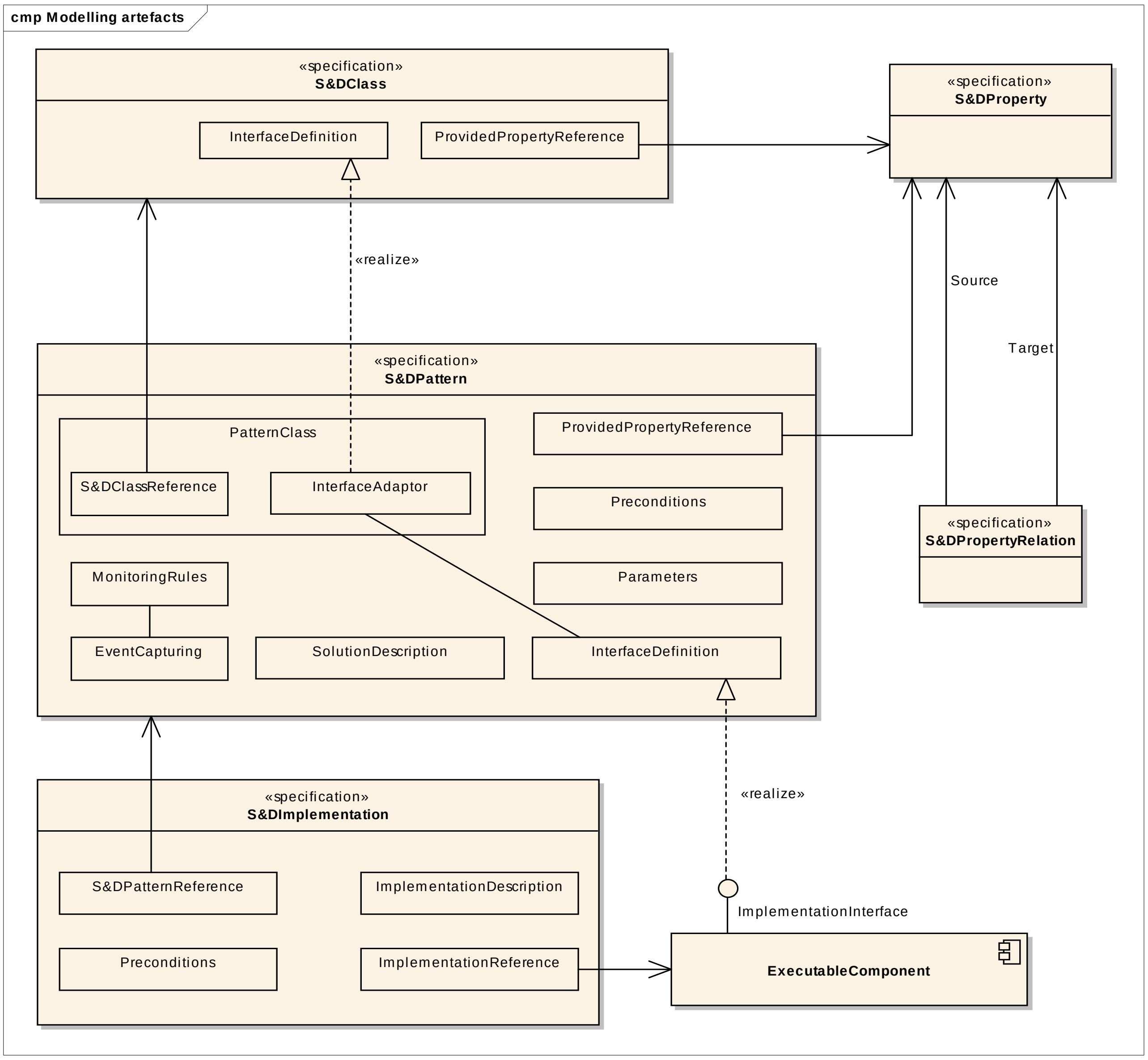

Figure 2 – Relations between the modelling artefacts

All these artefacts are represented in Figure 2, along with their composing elements and their interrelations. The rationale for introducing these three artefacts is based on the following reasons:

— S&D Patterns *can be verified* using the *SERENITY S&D Engineering Tools*, while S&D Classes and S&D Implementations cannot. Therefore it is wise to separate their definitions, since all information referring to the provided properties and the available proofs concern only the abstract solution (i.e. the S&D Pattern) and not the interface (i.e. S&D Class) or the specific implementation (i.e. S&D Implementation).

— S&D Patterns are verified by S&D experts (usually by means of formal methods) while the S&D Implementations are tested by their producers. In opposition to the case of S&D Patterns, which will be frequently produced by people who did not created the S&D Solutions described in such S&D Patterns, the creators of S&D Implementations will frequently be the creators of the corresponding Executable Components. Finally, S&D Classes are mainly interface definitions that are meant to facilitate application development.

— S&D Classes will be defined by entities mainly interested in interoperability (e.g. industry associations, standardization bodies). S&D Patterns will be produced by independent entities interested in security and dependability (e.g. S&D Companies and Experts, but maybe standardization bodies as well). However, patterns will not only enhance security and dependability, but also interoperability, as all implementations of an S&D Pattern will be required to conform to the pattern specification. Finally S&D Implementations will be produced by entities interested in the creation of working solutions (commercial solution providers, open source communities, etc).

All these definitions, concepts and characteristics of modelling artefacts are revised and extended from sections 1.5. to 1.7.

1.3. SERENITY Runtime Model

SERENITY covers all aspects of the lifecycle of S&D Solutions. It addresses both (i) the creation of new solutions and their characterization as S&D Patterns; and (ii) the description of their real executable implementations (i.e. Executable Components) as S&D Implementations. In addition, SERENITY supports the development process of the application assisting developers in the selection and use the most appropriate solution (pattern) fulfilling their requirements (making it clear that is their responsibility to take the final decision).

The dynamic selection and use of S&D Implementations according to the requirements and the context conditions is also part of this model. Thanks to the elicitation of S&D Requirements, S&D Classes can be found that fulfil them. From S&D Classes, the next step is to make a selection from the pool of all available Patterns that belong to these Classes. The purpose of this selection is to discard those Patterns that (despite they belong to a valid Class) are not valid given the requirements specified by developer. This process is based on the features made explicit in the Patterns: the developer specifies the key features he is looking for, so the last remaining artefacts will be those that fulfil both (i) the S&D Requirements and (ii) the features specified by the developer. From this refined list of artefacts, only those whose preconditions hold can be eventually selected and in turn, only the S&D Implementations whose preconditions hold can be eventually deployed. To end with, SERENITY model provides means for monitoring the correct execution of these implementations, which is necessary because of the interaction with external systems that might not be under the control of the local S&D Authority. In this section we will concentrate on the runtime support.

SERENITY anticipates a distributed, dynamic and heterogeneous scenario where systems interact and collaborate forming spontaneously AmI ecosystems. In our scheme S&D Realms have a component that is responsible for the enforcement of their security and dependability requirements. We call these realms *SERENITY Realms*, and the inner component the *SERENITY Runtime Framework* (SRF). To be precise, what SERENITY Realms integrate is an instance of the SERENITY Runtime Framework. Given the diversity of devices that may have SRF Instances, these instances must be platform-specific implementations of the generic SRF, some of them explicitly designed for mobile phones, some others for web servers, and so on. For the sake of simplification, both the abstract framework and its instances will be referred as SERENITY Runtime Framework now on.

We must note, however, that it is not mandatory for a system or S&D Realm to contain an SRF instance. In other words, because the SRF has well-defined interfaces (a Negotiation and a Monitoring interface, both described in the SERENITY architecture), it is possible for other non SERENITY-enabled systems to interact with SERENITY Realms. There is at least one SRF instance in each SERENITY Realm. For simplicity we can work under the assumption that every SERENITY Realm has one and only one SRF.

Each SRF has an S&D Library composed of the S&D Classes, S&D Patterns and S&D Implementations that are available in this particular SRF instance. At runtime, the SRF is responsible for fulfilling S&D Requirements by selecting and using the most appropriate S&D Implementations. In this sense, we call it *activation* referring to the complex process of loading, integrating, initializing and using (including the runtime monitoring of its correct execution) an S&D Implementation. Once an S&D Implementation is activated, the corresponding Executable Component is deployed. Thus, the Executable Component must include everything that is needed to execute the solution, going from the configuration details, to the code for deploying it.

The S&D Authority of the SERENITY Realm is responsible for defining the S&D Configuration of the SRF. This configuration includes different aspects, such as preferences, or system-wide S&D Requirements. This configuration can be considered as the security and dependability *policy* of the realm. In any SRF instance, there are two types of active S&D Patterns: on the one hand, "Event Observer Patterns" are activated as a result of the S&D Configuration; and on the other hand, "Application Patterns" are activated as a result of the S&D Requirements coming from a specific application. For a more detailed description of these elements, see Deliverable A6.D3.1.

1.4. SERENITY Development Time Model

SERENITY supports system developers at development time by (i) helping them to express their S&D Requirements; and (ii) supporting them in the selection and use of S&D Solutions fulfilling those requirements. Precisely, we have introduced the S&D Class artefact in order to support (ii).

When creating a new system, developers build the models of the system. Later, the analysis of these models helps them in the elicitation of the S&D Requirements of the system. The most important questions arise at this point: What are the possible solutions fulfilling the requirements? How should we deploy them? What are their restrictions and limits? Are they applicable in our environment? Furthermore, can all the solutions be applied together avoiding the risk of harmful interactions? These are just a few of the many extremely-hard-to-answer questions they may ask themselves. By having well-defined and precise descriptions of the possible solutions, especially covering details such as the applicability, compatibility or the interoperability, developers will be able to create better systems because they will be able to make informed decisions about the S&D Solutions that they include in their systems.

But what happens when we do not know the possible problems in advance? What happens if we do not know in advance how the system will be or will behave? Maybe these questions seem a bit unrealistic if we focus on traditional systems, but that is precisely the situation in AmI environments. In this case we need something more. And this something is the ability for applications developers to delay the decisions about which are the appropriate S&D solutions to use until the moment when we have enough information to decide correctly. That is, until runtime. Of course, developers need tools to control and restrict the decisions that will be taken by automated means at runtime. For the previous reasons SERENITY needs to be extremely flexible in supporting system developers. The solution we propose is to use three complementary artefacts: S&D Classes, S&D Patterns, and S&D Implementations.

When system developers identify an S&D Requirement, they can decide to leave the selection of the actual solution for runtime. In this case they will use a particular S&D Class providing the S&D Properties that they need in order to fulfil the requirements. S&D Classes fix only the minimum amount of information for developers in order to proceed with the development of their system. In particular, S&D Classes contain "the problem" (that is, the S&D Property provided) and a definition of an interface that must be used by the developers in order to access these services. S&D Classes do not have a defined behaviour, and therefore they do not need to be proven, validated or verified by any means.

All S&D Patterns belonging to an S&D Class need to conform to the class interface. However, each specific S&D Solution, and therefore each specific S&D Pattern, may have a different interface. This is so because interfaces are strongly related to the details of the solution. Therefore, S&D Patterns also contain a specification that allows the SRF to map the abstract calls defined in the S&D Class into the specific calls defined in the S&D Pattern. Thus, the Executable Component can either rigorously follow this interface when implementing its own functionality, or provide –

through its S&D Implementation, a wrapper that maps from the Pattern calls to the Executable Component functions.

In case developers select an S&D Class to fill the requirements, the SRF will be able to select among all the S&D Implementations that correspond to all the S&D Patterns that belong to that class. On the other hand, if developers decide to make the runtime selection process more restrictive, then they can select an S&D Pattern instead of an S&D Class. This way, although the S&D Pattern is fixed at development time, developers are still allowing the SRF to dynamically select among the possible S&D Implementations that point to the selected S&D Pattern. This selection is based on the information taken from the runtime context. Although not all S&D Implementations of an S&D Pattern have exactly the same characteristics and applicability, all of them share exactly the same interface and behaviour.

An S&D Implementation represents a working solution and therefore it contains a reference to the corresponding *Executable Component*. While an S&D Implementation is only a formal description of an implementation, the Executable Component is the actual implementation as an executable code or entity. There is a one to one relation between S&D Implementations (the descriptions of the working solutions) and Executable Components (the real working solutions), so that no S&D Implementation is possible without an Executable Component associated. Therefore, it is also possible for developers to choose a specific S&D Implementation for their system. In this case the advantages of dynamism are reduced, but not completely absent. In fact, the SRF will still be able to monitor the behaviour of the Executable Component corresponding to that S&D Implementation even if it cannot be changed.

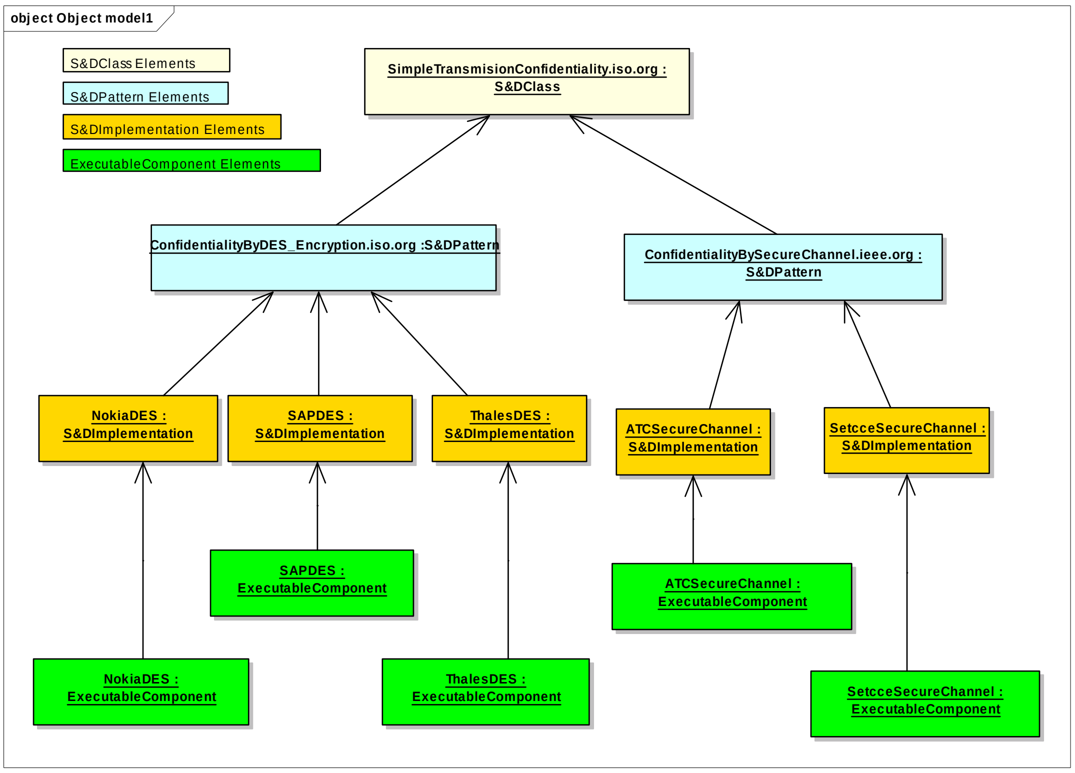
object Object model1
S&DClass Elements
S&DPattern Elements
S&DImplementation Elements
ExecutableComponent Elements
SimpleTransmisionConfidentiality.iso.org : S&DClass
ConfidentialityByDES_Encryption.iso.org :S&DPattern
ConfidentialityBySecureChannel.ieee.org : S&DPattern
NokiaDES : S&DImplementation
SAPDES : S&DImplementation
ThalesDES : S&DImplementation
ATCSecureChannel : S&DImplementation
SetcceSecureChannel : S&DImplementation
SAPDES : ExecutableComponent
ATCSecureChannel : ExecutableComponent
NokiaDES : ExecutableComponent
ThalesDES : ExecutableComponent
SetcceSecureChannel : ExecutableComponent

Figure 3 – Example of related S&D Classes, S&D Patterns and S&D Implementations

Summarizing, the developers have been supported at (i) development time selection of the most appropriate solution and at (ii) runtime monitoring of the correct operation of the Executable

Components. Figure 3 depicts an object diagram showing an example of the relations between S&D Classes, S&D Patterns, S&D Implementations (and their corresponding Executable Components).

Each SERENITY Framework instance will incorporate an S&D Library composed of different types of artefacts (S&D Classes, S&D Patterns and S&D Implementations) that will enhance the correct selection and use of the available working solutions (i.e. Executable Components).

It is important to remark that two S&D Pattern's instantiating the same S&D Class can play different roles in a system. For instance, an S&D Pattern for secure transmission over untrusted networks can play on client or on server sides while the associated functionality is notably different. Consequently, as the use of the interface depends on the role that an S&D Pattern plays in the system, it is necessary for each possible role to explicitly describe its functionality. The S&D Class includes a description of each possible role than can play an S&D Pattern that belongs to that S&D Class.

The Interface Definition at Class level, clearly distinguish the functionality offered by the different roles. This info can be extrapolated at Pattern level, using the Class Adaptor. Using it, we know the Pattern methods that belong to a particular role. As an Executable Components rigorously implements the interface of the Pattern, the functionality of each the role is perfectly available when using Executable Components.

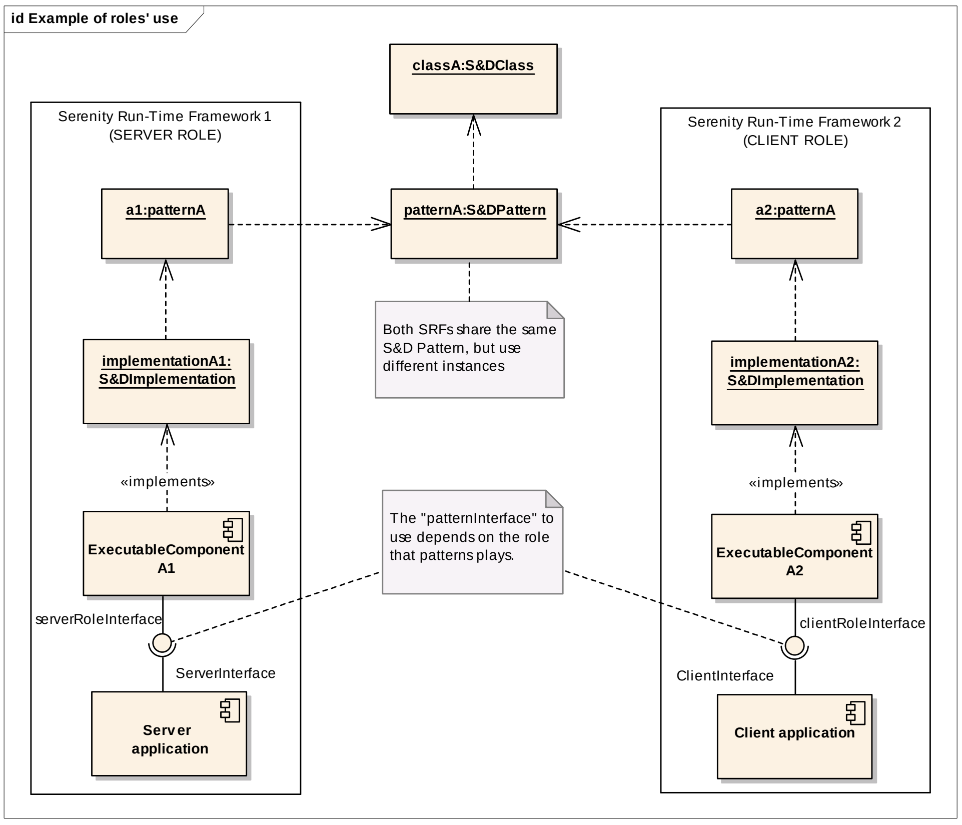
id Example of roles' use
classA:S&DClass
Serenity Run-Time Framework 1 (SERVER ROLE)
Serenity Run-Time Framework 2 (CLIENT ROLE)
a1:patternA
patternA:S&DPattern
a2:patternA
Both SRFs share the same S&D Pattern, but use different instances
implementationA1: S&DImplementation
implementationA2: S&DImplementation
«implements»
«implements»
ExecutableComponent A1
ExecutableComponent A2
The "patternInterface" to use depends on the role that patterns plays.
serverRoleInterface
ServerInterface
clientRoleInterface
ClientInterface
Server application
Client application

Figure 4 – The S&D Pattern "patternA" plays two different roles in each SRF

Figure 4 shows an example of the S&D Pattern behaviour based on the roles they play in the system. In our example, two SRFs have their own instance (*a1* and *a2*) of the same S&D Pattern (*patternA*). For pattern *a1* in SRF 1, the S&D Implementation *A1* is applied and the *ExecutableComponent A1* is running under a server role. For pattern *a2* in SRF 2, the S&D

Implementation *A2* is applied and the *ExecutableComponent A2* is running under a client role. Thus, each side in the communication channel is playing a different role: the SRF 1, on the left hand side, is providing an S&D Solution to a server application while the SRF 2, on the right hand side, is providing the same S&D Solution to a client application. Following this scheme, both applications are using the same S&D Solution (represented by the same S&D Pattern), but with sensitively different functionalities.

Given that each instance of the S&D Pattern is playing a different role, it is necessary to equip the S&D Pattern with the means to distinguish between the two different functionalities. This feature is provided by means of the *roles' section*. Included in the S&D Class definition and applicable at development time, it makes it possible the guidance for programmers during the development phase of Serenity-enabled applications. When an S&D Pattern is selected and then applied, the appropriate role is selected.

This *roles' section* shows explicitly, in the S&D Class definition, what functions are available for each role identified. For example, the use of certain functionality may not be necessary for one role, while it may be strictly necessary for another one. Moreover, given two roles that share some function *calls* (e.g. both encrypt/descript the information using the same *call* in the S&D Pattern), the sequence of those actions depends on the side of the communication channel where the S&D Pattern is being used.

In absence of an explicit role's section, developers had to "manually" separate the functionality associated to the role of interest, and apply the corresponding *calls* on their own discretion. Basically, this means that without the roles' section, we would have to use the pattern's interface as we use, for instance, a Java Lib: we read the documentation, and then we learn which Class and Functions to apply for my "hello world" application.

1.5. S&D Patterns and Integration Schemes

S&D Patterns are detailed descriptions of abstract S&D Solutions. These descriptions must contain all the information necessary for the selection, instantiation and adaptation, and dynamic application of the solution represented in the S&D Pattern. Just as one S&D Solution provides one or more properties, also one S&D Pattern refers to one or more S&D Properties.

A special type of S&D Pattern is called Integration Scheme. An Integration Scheme is an S&D Pattern that describes a complex S&D Solution. While S&D Patterns are independent or atomic descriptions of S&D Solutions, Integration Schemes describe solutions for complex S&D Requirements achieved by the combination of some S&D Solutions.

Note that the difference rests on the description, not on the solution itself. Therefore a complex S&D Solution can be represented as an S&D Pattern if it is described in an atomic or independent way (i.e. it does not refer to other descriptions). On the other hand, if we describe the same solution by making references to the S&D Patterns that are combined to achieve the complex property, or combination of properties, then we are representing the solution as an Integration Scheme.

In general, Integration Schemes are more difficult to analyse and to model, but in return they are more flexible and have better properties regarding the dynamic application. Let us consider the following example: one solution that provides Attestation Identification keys using a TPM. An S&D Pattern would require a TPM module as a precondition. Otherwise, the solution would be not applicable. On the other hand, an Integration Scheme representing the same solution would have no preconditions: it will combine both the Pattern for creating the Attestation Identification Keys and the Pattern for accessing and managing the TPM module. In this example, the main difference rests in the preconditions: while the S&D Pattern has the TPM as a precondition, the Integration Scheme has no precondition, given that it provides the TPM by itself.

From the point of view of the SRF, an Integration Schemes plays the role of an application. This is to say that once the Integration Scheme has been activated and deployed, it acts as an application, asking the SRF for the activation of the S&D Patterns needed. Figure 5 shows the sequence of activation of an Integration Scheme:

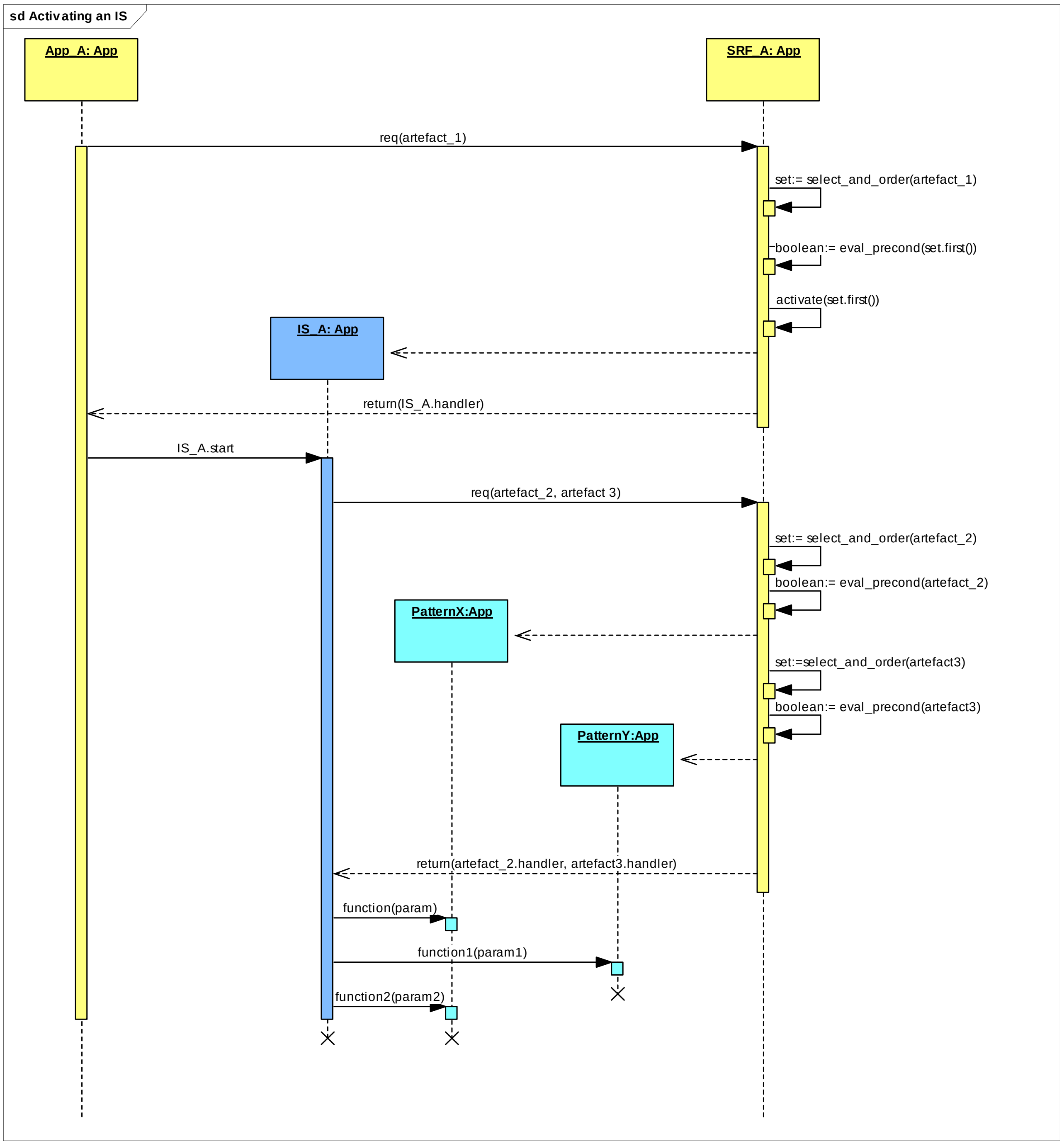
sd Activating an IS
App_A: App
SRF_A: App
req(artefact_1)
set:= select_and_order(artefact_1)
boolean:= eval_precond(set.first())
activate(set.first())
IS_A: App
return(IS_A.handler)
IS_A.start
req(artefact_2, artefact 3)
set:= select_and_order(artefact_2)
boolean:= eval_precond(artefact_2)
PatternX:App
set:=select_and_order(artefact3)
boolean:= eval_precond(artefact3)
PatternY:App
return(artefact_2.handler, artefact3.handler)
function(param)
function1(param1)
function2(param2)

Figure 5 – Activating an Integration Scheme

In the example, "*App_A*" request the SRF "*artefact_1*". The SRF creates a list of the possible Patterns belonging to this artefact, extracts the first of them, and then evaluates its preconditions. At this point, it is important to remark that the SRF must check the preconditions of the S&D Pattern and, in case they hold, check the preconditions of the selected S&D Implementation. If both hold, then, the corresponding Executable Component can be deployed and its handler put at "*App_A*" disposal.

In the example, the first Pattern of the list is applicable and it turns to be an Integration Scheme. The IS is activated (named as "*IS_A*") and the SRF returns the IS handler to "*App_A*" (message "*return(IS_A.handler)*"). As stated before, now that the IS has been activated, it starts acting as an application, both for being accessed from "*App_A*" and to access the SRF by its own.

Now that application "*App_A*" has fully access to the IS functionality, it calls the IS to start (message "*IS_A.start()*"). The IS must activate its Patterns, so it asks the SRF for a couple of artefacts to be activated and deployed: "*artefact_2*" and "*artefact_3*". When the process of selecting and activating the artefacts is finished, the SRF comes back to "*IS_A*" with the handlers of the Executable Components of "*artefact_2*" and "*artefact_3*", respectively. Now the IS accesses these artefacts using "*artefact_2.handler*" and "*artefact_3.handler*".

Eventually, the definitions for the previous concepts state as follows:

S&D Pattern: A self-contained description of an S&D Solution, meaning that it does not refer to (or depends on) other S&D Solutions.

Integration Scheme: A description of a composed S&D Solution that refers to (or depends on) other S&D Solutions. In some cases, Integration Schemes will be used to represent ways of correctly combining S&D Solutions with the objective of avoiding that they badly interfere.

The description of the S&D Pattern contains many different elements. The most important are:

S&D Pattern: A self-contained description of an S&D Solution, meaning that it does not refer to (or depends on) other S&D Solutions.

Integration Scheme: A description of a composed S&D Solution that refers to (or depends on) other S&D Solutions. In some cases, Integration Schemes will be used to represent ways of correctly combining S&D Solutions with the objective of avoiding that they badly interfere.

The description of the S&D Pattern contains many different elements. The most important are:

— *ProvidedProperties*. This element is used to point to the descriptions of the S&D Properties provided by the S&D Pattern. One S&D Pattern can provide one or more properties. It is natural for one Pattern to provide several Properties, given that the Pattern can belong to more than one Class.

— *Preconditions*. Every S&D Pattern represents a specific S&D Solution. For this reason, we assume that they are not universally applicable. This element contains the specification of the conditions under which the S&D Pattern is able to provide the mentioned properties.

— *MonitoringRules*. Because S&D Patterns are not expected to represent perfect solutions, and because the solutions will frequently depend on the behaviour of external components that will not be under our control, the solution must be monitored during its execution in order to guarantee that it works correctly. This element contains instructions for an external monitoring mechanism to perform this activity. We assume that every solution is responsible for capturing the events that are necessary for monitoring it. Therefore, this element declares this events and how to capture them.

- *Parameters*. This element allows us to build more generic solutions. Parameters (for instance, the length of the keys in an encryption algorithm) can change without affecting the general behaviour of the solution. They can always be represented by a 2-tuple with a name and a value.
- *PartDescription*. Sometimes a solution makes use of external elements that can be replaced, but that need to comply with some conditions. *Parts* (for instance, a camera in a surveillance system) are a special type of parameters that represent working parts of the solution. They can be replaced as long as the new *Part* conforms to the conditions expressed in this element. A *Part*, in contrast to a simple parameter, does not represent a single value, but a component that: (i) is a piece of the solution and (ii) have an associated behaviour and specific characteristics.
- *Tests Performed*. Every S&D Pattern represents a proven solution. Therefore, this element is used to specify the proofs that have been applied in order to claim that the pattern description is sound.
- *SolutionDescription*. This element is used to represent the solution.
- *InterfaceDefinition*. This element describes the native interface of the S&D Solution described by the S&D Pattern. More specifically, it allows to: (i) adapt the native interface coming from the Class to the interface of the S&D Solution and (ii) precisely describing the interface of the S&D Solution.
- *PatternClass*. This element represents references to the classes where the pattern belongs. It is divided into two components: an *S&DClassReference* is the reference itself; and a *Class Adaptor* is the description of the adaptation of the pattern interface in order to conform to the class interface.

1.6. S&D Classes

S&D Classes are introduced to solve the need of system developers of knowing at development time the way to access the services related to the desired S&D Property, while maintaining the maximum flexibility in the dynamic selection of the specific S&D Solution (in this case S&D Implementation).

An application developer needs a minimum amount of information about the S&D Solutions (in the form of S&D Implementations) that will be used to fulfil its S&D Requirements. Therefore, this artefact is designed to provide this minimum amount of information, while maximizing the flexibility and the number of possible solutions that can be selected and applied at runtime.

The description of an S&D Class contains:

- *ProvidedProperties*. This element points to the descriptions of the S&D Properties provided by the S&D Patterns that belong to this S&D Class. Note that the S&D Class does not provide properties. One S&D Class can point to one or more properties.
- *InterfaceDefinition*. This element describes the native interface of the S&D Class. This interface must be designed in order to be simple and generic enough for many solutions to be able to comply with it.
- Roles' definition. The previous interface, which defines a set of available operations, is refined into one or several sequences of operations. Each sequence is defined for the different roles that an S&D Class can play when refined as an S&D Pattern. The benefit of

this distinction is straigthforward. Let us consider a Pattern providing Confidential Transmision. All functionality is already defined. Now consider a sender and a receptor using that Pattern. Both are using the same functionality, but in a different way. While one encrypts, the other decrypts. Each side (that is, each role) must have a clear vision of its functionality defined first at Class level, and then refined at Pattern level.

1.7. S&D Implementations

S&D Implementations describe executable mechanisms that conform to an S&D Pattern. In other words, and S&D Implementation precisely depicts an implementation of the S&D Pattern, and not the abstract S&D Solution represented by the pattern. The description of an S&D Implementation includes:

— *ImplementationDescription*. This element is used to represent the implementation details.

— *ImplementationReference*. This element points to the actual Executable Component.

— *Preconditions*. Frequently, an implementation will have some specific preconditions that join the pattern preconditions making more restrictive (but also more precise) the process of selecting the most suitable implementation.

— *S&DPatternReference*. This element is a reference to the pattern that the S&D Implementation implements.

We must highlight that there is no specification of the S&D Implementation interface because all S&D Implementations of a given S&D Pattern must have exactly the same interface that the pattern has.

2. Conceptual Model

The objective of this section is to formally represent the conceptual elements that are used in SERENITY. For the Use Case view, the reader can refer section 2.2 from the previous version of the deliverable [4].

The following class diagram shows the different conceptual elements that are used in SERENITY as well as their relations.

We can observe that S&D Patterns and Integration Schemes (*S&DPatterns* in Figure 6) refer to solutions (*S&DSolutions*) and contain the semantics (*S&DSolutionSemantics*) that describe such solution. The semantics are described in terms of the semantics (*S&DPropertySemantics*) of the particular properties (*S&DProperty*) provided by the solution. Solutions *(S&DSolutions)* can be monitored by the monitor service *(MonitorService*). Solutions Semantics provide a monitoring specification *(MonitoringSpecification)* that describes politics and events involved in monitoring tasks. Solutions *(S&DSolutions)* may have different implementations *(S&DImplementations)*.

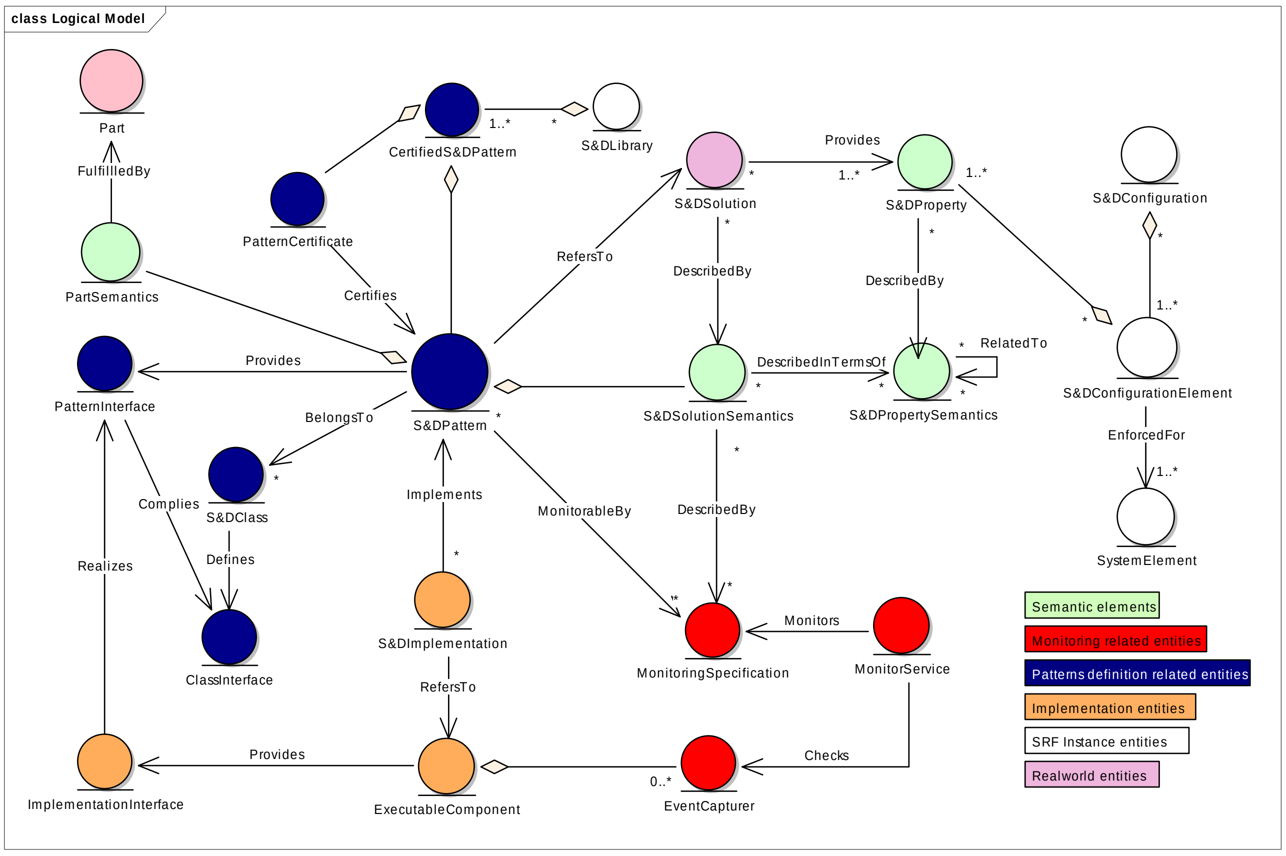

Figure 6 – Logical model

An S&D Implementation is a description of an implementation that fits a solution. An ExecutableComponent is a tangible element (e.g. a software application or a cryptography library) that supplies a particular implementation. Different S&D Implementations for the same solution are the result of having a number of solutions for the same problem but fitting different context conditions or requirements. Each ExecutableComponent provides a particular Interface

(ImplementationInterface). ImplementationInterfaces must realize the whole PatternInterface. PatternInterface helps to maintain similar (but not equal) interfaces for all ExecutableComponent. For monitoring purposes ExecutableComponents provide EventCapturers. Serenity Framework will use the S&D Implementation in order to choose the correct one among all the possible implementations of a specific pattern. The description of a pattern should be a more general definition than implementation description is.

S&D Patterns and Integrations Schemes are certified by a special type of digital certificate (*PatternCertificate*). The library of S&D Artefacs (*S&DLibrary*) is composed of S&D Patterns and Integration Schemes that hold a certificate, the so-called certified patterns (*CertifiedS&DPattern*).

S&D Patterns provide *interfaces* (*PatternInterfaces*) that are used by Serenity Runtime Framework in order to establish the criteria for pattern's use. All implementations (*S&DImplementations*) of a pattern must comply with the interface of the implemented pattern. It is possible to have more than one implementation for each pattern. S&D Classes also provide *interfaces*, named as *ClassInterfaces*. ClassInterfaces are not the same than PatternInterfaces, given that PatternInterfaces must comply with the ClassInterfaces definition. *S&DClasses* are used to group a set of *S&DPatterns*. All patterns that define the same interface come under the same umbrella: an *S&DClass*. At some extent, the concept of *S&DClass* is close to the concept of class in orient object programming.

Finally, users define the security and dependability requirements (*S&DConfiguration*) for their systems, grouping a set of specific requirements (*S&DConfigurationElement*). Each specific requirement is specified by means of a set of properties (*S&DProperty*) that must be enforced for a particular element of the system (*SystemElement*). All this elements are shown in Figure 6.

3. Architectural Model

The aim of this section is to depict and describe the components of the architectural view of SERENITY, as well as the relations established among them. As every single SERENITY-aware device will run this multifaceted architecture, it is important to underline the main components, their role, and criticalness in the whole process of securing a device. Among all the components, one of them rises as the core one since it holds the knowledge and the experience of security experts in form of S&D Patterns: the SERENITY library. This component is described in section 3.1. , just before the description of the whole picture given in section 3.2. .

3.1. SERENITY Library

SERENITY Library is the result of the effort to represent, in a general and machine-readable format, the solutions developed by security experts for a wide range of security problems. It contains patterns that describe, at different levels of abstraction, security solutions that solve specific security problems. However, the patterns not only hold the description of the solution but also how to use it, the conditions needed for its application and how to monitor the correctness of the process.

Obviously, from an AmI point of view, every single device has different security needs and is surrounded by a different working context that obligates SERENITY to instantiate the library for every particular situation. Given an instance offering concrete solutions, the correctness for the concrete device and problems is assured; however, this correctness can not be assured in the event of a change in the application context. As some of the applied solutions can be no longer valid in the new context conditions, the library offers the channel for SERENITY Framework to dynamically react and update/change the existing solutions in order to fit with the new applicability conditions. For the time being, if some change arises in the context that makes a solution no longer valid, this solution is deactivated. Then, a search process starts that looks for the most suitable solution from those available and, if found, activates it.

Today devices offer a variety of internally complex but, on the other hand, easy-to-use applications coming with different hardware/system requirements. In AmI context, applications will also come along with security requirements expressed by means of security properties to provide in order to safely achieve the intended functionality. At this stage we can use the information we usually get from the manufacturer; for instance, an example of these requirements might go like: “the use of my brand-new chat application is fully secure when used among ACME devices, but no confidentiality is assured if any of the parties in chat is not using an ACME device”. This assertion makes clear that in case you really need confidentiality, some further functionality has to be added to the original application. However, frequently at this point the developer has not enough information to select the most appropriate S&D Solution. Therefore, the developer takes the final decision, assisted by the SDF (Serenity Development Framework) using generic solutions to their requirements, represented by S&D Classes.

S&D Classes are abstract classes that group several *S&D Patterns* with one common trait: all of them offer a solution for the problem specified in an *S&D Class*. Reasoning the previous example, the *S&D Class* to look for is the one talking about the problem of *confidentiality* when communicating two principals. Obviously, a number of solutions –each one with some peculiarities, advantages and drawbacks, have been offered in the literature to provide this property. For each one of these solutions, an *S&D Pattern* is the appropriate artefact to represent and describe them in a machine-readable way.

As a solution for a concrete problem, an *S&D Pattern* refers to an *S&D Class*. As several solutions can be proposed for the same problem, several *S&D Patterns* can refer to the same *S&D Class*. In this way we can symbolize SSL and TLS as different but appropriate solutions for the problem of confidentiality. Apart from the reference to the abstract *class*, each *S&D Pattern* includes information about the context in which it can be applied, a description of the solution, and some useful information about monitoring that can be used to monitor the execution of the pattern during its lifetime. Figure 7 represents all the elements described in this section as part of the SERENITY Library.

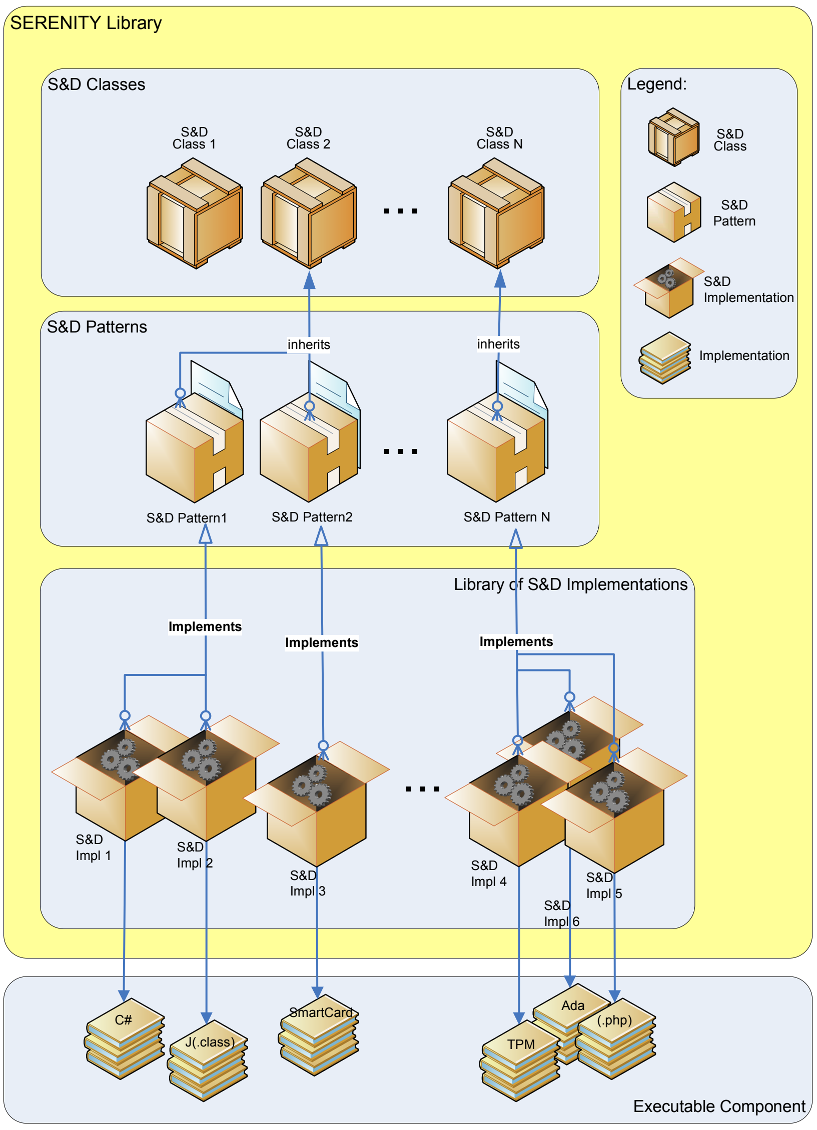

Figure 7 – Representation of SERENITY Library

Not only each problem can have different solutions but also each solution may have different implementations. As a mere example, SSL or TLS describe a protocol that has different implementations depending on the provider (e.g. OpenSSL from BSD and JSSE from Java are just two of the most popular implementations of SSL).

For each available implementation, a different *S&D Implementation* document is included in the library, describing: the specific system requirements, the necessary interface to use when calling the implementation, and the location of the *Executable Component*. Each *S&D Implementation* refers to one concrete implementation –its own Executable Component, so that once a solution is selected by the framework, a handler for that component is made available for the application. A solution can be not only a software solution but also include hardware elements such as a TPM (Trusted Platform Module) or a SmartCard. In any case, it makes no difference from the application point of view. For instance, if the Executable Component works with a TPM, the SRF will return to the application not a direct link to the TMP, but rather the handler of the driver that manages the TPM.

In our discussion, the path from the problem to the concrete solution is the path that goes through the library and includes the *S&D Class* for confidentiality, the *S&D Pattern* offering SSL for secure communications, and the *S&D Implementation* describing the interface and the specific mechanisms of the concrete Executable Component, such as OpenSSL. Finally, once a pattern is found and selected as the most suitable one, it is activated (included among the *Active Patterns*) and used by the *Serenity Framework*. From an architectural point of view, a pattern coming from the *S&D Library* and subsequently activated is known as an *Application Pattern*. The concept of *Active Pattern* and *Application Pattern* will be more extendedly described in next section.

3.2. Architecture Description

As the S&D Library represents the static knowledge extracted from security experts, the architecture as a whole represents the dynamic reasoning that takes the knowledge and makes it available to the final user/application. Figure 8 depicts the main architectural elements as well as the interactions among them.

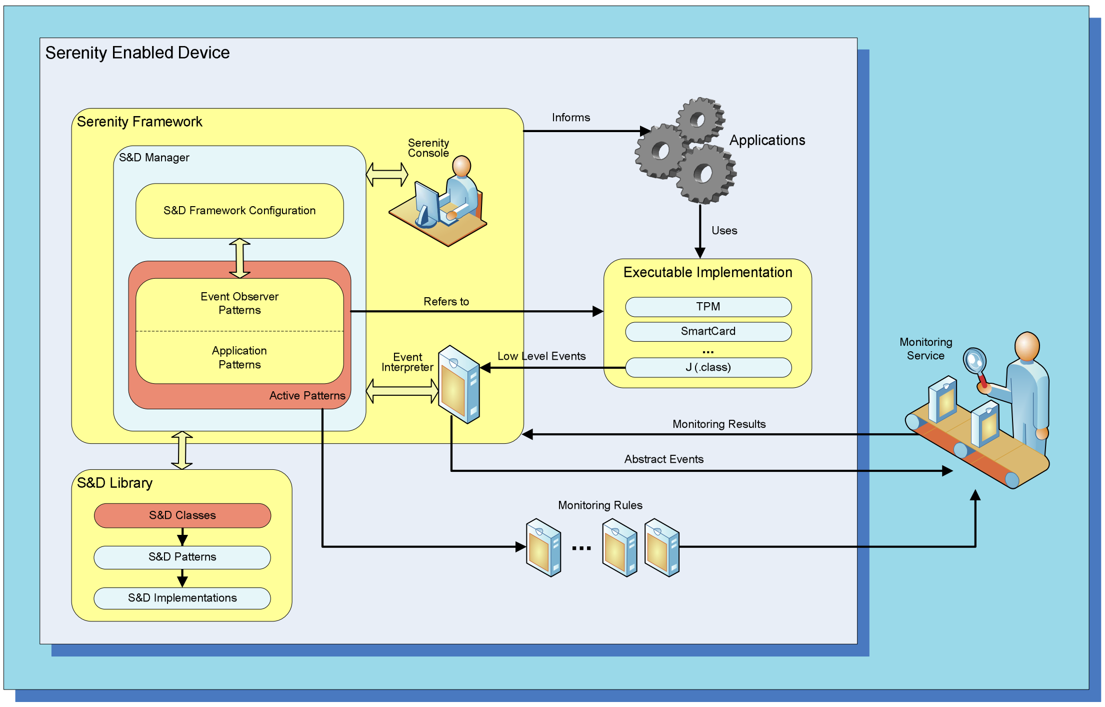

Figure 8 – Main elements of the SERENITY Architecture

3.2.1. *Internal Elements*

Along with the *S&D Library* –described in the previous section, the *Serenity Framework* is the other basic pillar of the architecture embedded inside a *Serenity Enabled Device*. The framework is composed of three elements, namely:

- S&D Manager: this is the core of the framework. Among other duties, it has to (i) manage all the parameters concerning the user configuration; (ii) deal with the patterns specifically designed for the device (embedded patterns installed independently of the applications running on top); and (iii) interact with the S&D Library, the applications running on the device, and the Monitoring Service. Inside the S&D Manager, two artefacts coexist:
  - Active Patterns: it contains the set of Patterns already working in the system, along with data about the date of activation, the foresee date of deactivation, the application that is using the Pattern, and so on.
  - S&D Framework Configuration: in order to grant some flexibility to the user, some degree of configuration is permitted. For instance, taking into account that the monitoring service may consume resources from the device (possibly degrading the performance), the user may prefer to switch off the monitoring of certain rules in specific contexts. E.g. if the user considers that the office environment is sure enough to trust on the underlying connection, some monitoring mechanisms can be obviated.
- Serenity Console: this element acts as the man-in-the-middle between the Serenity Framework and the user. The information that flows between both parties is bidirectional. On one side, whenever the user has to deal with the framework configuration and specify some preference or configuration parameter, the information is retrieved through the console and sent to the S&D Framework Configuration element. On the other side, whenever the framework has to send some warning or indicate some relevant event to the user, the information is presented throughout the console. For instance, if one of the solutions is no longer valid due to an unforeseen change in the context, apart from starting a series of reactions, the user is alerted of the incident and some of the subsequent decisions will depend on his elections. All this input/output process is made by means of the Serenity Console.
- Event Interpreter: it receives all the low level events generated by the patterns (i.e. the implementation of the patterns). The Monitoring Service should receive these events from the Framework in order to analyse them and send the monitoring results back. However, the Monitoring Service is not well suited for low level events, so that the Serenity Framework offers this interpreter in order to translate them into abstract events, appropriate for the service to check them against the monitoring rules.

Some devices come with specific security needs that have nothing to do with the application layer but with the underlying hardware, OS or the environment in which the device is used. We can not rely on the availability of third party applications to capture and monitor some relevant information such as the connection to a trusted/untrusted network. For instance, we can not assume the existence of an application running on every smart phone able to monitor whether the device is connected to a European GSM network or an American CDMA network. This environment-related information has some important security implications that can change the assumptions made on the basis of the external context. These assumptions are basic for the definition and later evaluation of Artefacts' Preconditions and consequently, if these assumptions vary, the applicability of the existing patterns should be revised. In this sense, the SRF provides a set of Event Observer Patterns to cover these relevant events that are not covered by the general-purpose, application-specific Patterns.

The *S&D Library* contains all the artefacts made available for the SRF, while the *Active Patterns* is a list of the Patterns already active. For each active Pattern, there is an Executable Component that has been installed and deployed. In order for the application to use the Executable Components, they need the handler that points to them. This handler is available in the *Active Patterns* list, and it can point either to a web service, a programming module, and applet, and so on, making it transparent for the application. There is no restriction regarding the implementation mechanism, as far as it is in accordance with the interface and the functionality described in the corresponding *S&D Implementation* document. Consequently, the language and the technology used in each implementation may be different from the others.

As every implementation has a well-defined interface, applications running in the device make use of them by means of simple calls, following the same fashion used in Web Services technology. The *Serenity Framework* is the one in charge of informing the applications about the interface they have to use as well as the correct sequence of steps to follow when using the interface. Apart from that, the *Serenity Framework* keeps information about the context to ensure the correctness and validity of the implementations that are in use. If any of the patterns (and thereby the corresponding implementation) is not valid in a new context, the application is informed and the framework provides a new solution (if applicable) or a warning message for the user if no solution is available at the moment. As stated in previous paragraphs, any communication to or from the user is conducted through the *Serenity Console*.

3.2.2. External Elements

All the elements described above are integrated in the user device, namely: *S&D Library* –where the knowledge in security is stored; *Serenity Framework* –where this knowledge is analysed and put at applications' disposal; the *Executable Component* –offering the functionality formally described in the patterns; and finally the applications that takes advantages of the whole infrastructure.

Nevertheless, in order to fully understand the performance of the Serenity Architecture it is necessary to add one new element, peripheral to the user's device: the *Monitoring Service*. When an *S&D Pattern* is activated, particular monitoring information included in the pattern specification is sent to the Monitoring Service. This information, sent from the *Active Patterns* artefact, includes: what to monitor –in form of abstract events, and how to monitor it –in form of monitoring rules.

A rule is part of a monitor and is used to detect certain events that relate to the values of a monitor. In case of a ping monitor, used to test whether a particular host is reachable across an IP network, the rule states the predefined critical value of the response time. If this critical value is exceeded, action is required. There are plenty of situations where this can be applied. For instance, a web server having access problems can be easily detected this way without constant surveillance. When the critical value for the response time is exceeded, an alert will be displayed.

Without detailed knowledge of how to use a rule, it is quite difficult to specify a rule properly. That is the reason why all this knowledge is embedded in the pattern specification and sent to the monitoring service when required. As soon as the rules are triggered, the following actions can be defined: popup message, e-mail message, pager/SMS, log event, execute command line, SNMP trap, start/stop services, terminate process, and shutdown the device. In any case all these reactions are sent from the *Monitoring Service* and received by the *S&D Manager*, who redirects the message to the *Serenity Console* in order for the user to get informed.

4. A Language for Describing S&D Solutions

In this section the reader will find a precise description of the elements that internally compose the different S&D Artefacts, as well as a set of considerations common to all the three S&D Artefacs.

4.1. Common Considerations

4.1.1. Naming scheme

In order to standardize the naming method for the modelling artefacts we define a simple syntax similar to the URL syntax for Internet protocols. Already described in the document, three are the artefacts present in SERENITY architecture: S&D Classes, S&D Patterns and S&D Implementations. In all the three cases, the naming scheme states as follows:

```
<artefactName>.<issuerName>
```

,where each element follows Backus Naur Form (BNF[4]) notation, defined as follows (Table 1):

```
artefactName  =  alphadigit | alphadigit *[ alphadigit | "-" | "_"] alphadigit
issuerName    =  1*[ domainlabel "." ] toplabel
domainlabel   =  alphadigit | alphadigit *[ alphadigit | "-" | "_"] alphadigit
toplabel      =  alpha | alpha *[ alphadigit | "-" | "_"] alphadigit

alpha        =  lowalpha | hialpha

digit        =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |"8" | "9"

alphadigit  =  alpha | digit

lowalpha  =  "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
             "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
             "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
             "y" | "z"

hialpha  =  "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
            "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
            "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
```

Table 1 – Name Scheme in BNF notation

Note that in this BNF notation the character *"|"* is used to designate alternatives, and brackets *[]* are used to indicate optional or repeated elements. Some other considerations are: literals are quoted with *""*; optional elements are enclosed in brackets *[]*, and elements may be preceded with *<n>** to designate *n* or *more* repetitions of the element that follows; n defaults to 0 (see RFC 1738 for more details).
Some examples are defined below (Table 2) in order to facilitate the understanding of the notation:

Class Name	=	SimpleTransmissionConfidentiality.iso.org
<artefactName>	=	SimpleTransmissionConfidentiality
<issuerName>	=	iso.org
domainlabel	=	iso
toplabel	=	Org

Pattern Name	=	ConfidentialityByDES_Encryption.rsa-labs.com
<artefactName>	=	ConfidentialityByDES_Encryption
<issuerName>	=	rsa-labs.com
domainlabel	=	rsa-labs
toplabel	=	com

Implementation Name	=	CryptoJ_BSafeDES.rsa.com
<artefactName>	=	CryptoJ_BSafeDES
<issuerName>	=	rsa.com
domainlabel	=	rsa
toplabel	=	com

Table 2 – Examples of use of the naming scheme

4.1.2. Study of Preconditions

Preconditions are classified depending on the check process performed, falling into two different categories: *SRF context preconditions* and *External preconditions*.

SRF context preconditions. This group includes all preconditions related to the information collected by the Context Manager of the SRF. Therefore, the basic facts and events related with these preconditions are observed and captured by the SRF. In particular, it deals with information related to the following elements:

— *Pattern History*: During the lifetime of an SRF instance, artefacts are deployed, activated and deactivated as the context and the S&D requirements evolve. The Context Manager records the information relative to these activations and deactivations of S&D Implementations, along with additional information like the parameters used, the reason for deactivation, etc. In particular, it keeps track of S&D Implementations that are currently active, including the applications that are using it. Thanks to this information we can, for instance, check the preconditions of a particular artefact that is incompatible with some specific artefact. In some other case, an artefact is not applicable unless one particular pattern had been applied before. Both cases require the specification of preconditions that refer to the *Pattern History*.

— *Event history*: This element of the Context Manager stores the list of relevant events occurred in the past under the supervision of the SRF. These events can refer to a wide variety of incidents or circumstances, and are of great use when expressing the pattern preconditions. For instance, a precondition may ask for an implementation of that pattern not been deactivated in the past as a result of an attack.

— *SRF Configuration elements*. The SRF Configuration is also stored in the SRF Context Manager. This is probably the most heterogeneous part of the Context Manager, because it is highly dependent on the specific characteristics of the SRF instance considered. It stores information about the Operating System, the type of platform of the device, and so on. In general, the SRF Configuration is not very relevant at the level of S&D Pattern's preconditions, but it is of great use when defining the preconditions of S&D Implementations. One configuration element can be named as "*SRF.library.is_dynamic*", which defines whether it is allowed for the SRF to download and install new S&D Solutions in the library on demand at runtime.

External preconditions. This group includes all preconditions that refer to aspects that are not under the direct control of the SRF. The basic facts for these conditions are not known by the framework; otherwise these would become SRF context preconditions. Typical examples of this category are those solutions that rely on some hardware element that must be present (or active) in the system (e.g. "the TMP device must be active"). Other example might be a pattern requiring a wired connection along with a battery charge not under 30%.

4.1.2.1. Structure and Sintaxis of Preconditions

As suggested in previopus paragraphs, preconditions are expressed both for S&D Patterns and for S&D Implementations. In the former case, preconditions include queries on the *Pattern History*, the *Event History* and the events recorded for the *external preconditions*. In the later case, preconditions are far more focussed on implementation and target system details, so that most of the queries target the *SRF Configuration* elements.

In both cases, Xquery [6] is the language proposed to elicit the Preconditions. XQuery is a query language developed by W3C and used (in short) for XML information retrieval. It relies on XPath and XML Schema Datatypes for finding and extracting elements and attributes from XML documents. Under the scope of the language of preconditions, XQuery is used to query the SRF database and extract the information of the events that must be checked prior to the deployment of an S&D Artefact.

Let us consider the following example: an S&D Pattern called *SecCript* is selected to achieve secure encryption in my system. This solution needs to interact with a TPM. This TMP is controled by a Pattern called *TMPManager*, so previous to the application of *SecCript*, it asks for the *TMPManager* to be active. Otherwise, *SecCript* will not be applicable. As stated before, the SRF stores the information about the active Patterns inside the *Context Manager* in a table called *Active Patterns List*. The following aims to be an illustrative example of a possible instance of *ActivePatternsList* table, filled with fictional data:

```
<activepatternslist>
    <execComponentId>0</execComponentId>
    <SDpattern>TPMManager</SDpattern>
    <SDImplementation>implementation_sample</SDImplementation>
    <activationDate>2008-01-30</activationDate>
    <activationTime>13:00:00</activationTime>
    <deactivationDate>2008-08-01</deactivationDate>
    <deactivationTime>00:00:00</deactivationTime>
    <handler>http://url_sample.com</handler>
    <isActive>true</isActive>
</activepatternslist>
```

Table 3 – Example of ActivePatternList tuple

Important elements in previous example tuple are `<SDpattern>TPMManager</SDpattern>` where the Pattern ID is stored, and `<isActive>true</isActive>` where the value remains *true* if the Pattern is active and *false* otherwise.

The precondition in *SecCript* would include a simple XQuery like the one below, where the query accesses the *ActivePatternList* table and looks for a tuple in which *SDPattern* column has the value "*TPMManager*" and the *isActive* column containing the value "*true*". The XQuery returns whether a tuple like this exists or not (i.e. *true* or *false*):

```
xquery version "1.0" encoding "UTF-8";

for
        $w in //contextmanager
let
        $prec1 := $w/activepatternslist/SDpattern='TPMManager' and $w/activepatternslist/isActive='true'
return
        <result>
                {$prec1}
        </result>
```

Table 4 – Xquery example

If the XQuery is executed against the example *Context Manager*, it would answer "*true*", returning the following structure:

```
<result>
     true
</result>
```

Table 5 – Xquery result example

The previous example was a simple precondition, since it launched a query on a single tuple of the *ActivePatternList* table. Thus, all the logic operators for the precondition where located in the "let" statement.

Now, let us suppose that in addition to the need of having "*TPMManager*" active, our Pattern is incompatible with the "*SuperSecCript*" Pattern, launched by an industry competitor. The preconditions would go like this:

```
for
        $w in //contextmanager,
        $r  in //contextmanager
let
        $prec1 := $w/activepatternslist/SDpattern='TPMManager' and $w/activepatternslist/isActive='true',
        $prec2 := $w/activepatternslist/SDpattern='SuperSecCript'
return
        <result>
          {$prec1 and not($prec2)}
        </result>
```

Table 6 – Xquery example: two simple preconditions

In this particular case, some additional logics appear in the "return" statement. That is because the final precondition is in fact the logical *and* of two simple preconditions: the first one asking for "*TPMManager*"; and the second one asking for the absence of "*SuperSecCript*".

We can even go one step futher and make the precondition as logically complex as it is needed. Let us consider the table "*EventsHistory*", in which the SRF stores the events that are being monitored, such as the availability of network conection, the low battery charge, and so on. This information is represented as shown below (Table 7):

```
<eventshistory>
   <eventId>lowBatteryCharge</eventId>
   <execComponentId>batteryMonitor</execComponentId>
   <ruleId>R1</ruleId>
   <violationDate>2008-02-04 15:08:53</violationDate>
   <meaning>The battery charge of the device is below 30%</meaning>

</eventshistory>
```

Table 7– Example of EventsHistory tuple

For instance, the following precondition joins three simple preconditions into a single expression:

```
for
  $w in //contextmanager,
  $t in //contextmanager,
  $u in //contextmanager
let
  $prec1 := $w/eventshistory/eventId= lowBatteryCharge and $w/eventshistory/execComponentId= batteryObserver,
  $prec2 := $u/eventshistory/eventId=2
  $prec3 := $t/eventshistory/eventId=3 and not($t/eventshistory/execComponentId=29)
return
<result>
  {(not($prec1) and $prec2) or not($prec3)}

</result>
```

Table 8– Xquery example: three preconditions

In the particular case depicted in Table 8, we can appreciate three preconditions expressed as follows:

```
$prec1 := $w/eventshistory/eventId= lowBatteryCharge and $w/eventshistory/execComponentId= batteryObserver,
```

Here, we ask for an event called "*lowBatteryCharge*" that has been produced by the Executable Component called "*batteryObserver*". Then, we ask for event "2" and event "3" iff it has not been produced by Executable Component "29". Finally, we logically join these single preconditions in a common formula:

```
{(not($prec1) and $prec3) or not($prec2)}
```

This formula will be true if precondition 2 does not hold, *or* if precondition 3 holds *and* precondition 1 does not.

To summarize with:

— Each single precondition is expressed in the "let" statement.

— Two or more preconditions can be combined in the "return" statement using the logic operators considered in XQuery, namely: AND, OR, and NOT, in their usual meaning.

Valid examples of combining simple preconditions are:

`((Precond. **and** Precond.) **and** Precond.)`
`(**Not** (Precond.))`
`((Precond. **or** Precond.) **or** (Precond. **and** Precond.))`
`(Precond. **and** ((Precond. and Precond.) **or** (Precond. **and** Precond.)))`

Table 9 – Examples of Preconditions' definition

4.1.2.2. Creation and Evaluation of Preconditions

Although the preconditions in previous examples are hand made created, it is obvious that the process for creating preconditions is much more agile and simple with the assistance of the editing tools developed in Activity 6, and more specifically PSMT tool.

Regarding the creation of Preconditions, we must emphasize that the field "Preconditions" in both S&D Patterns and S&D Implementations is composed by zero or several simple preconditions. A simple precondition queries a single event, and it may ask for any information on that event. As a matter of fact, this event can refer to virtually any possible situation that can be captured and monitored using a computing system. Summarizing, each simple precondition relates to one event of the system, wheter it be about the activation of a pattern, the charge of the battery, or the presence of an human being in front of the computer.

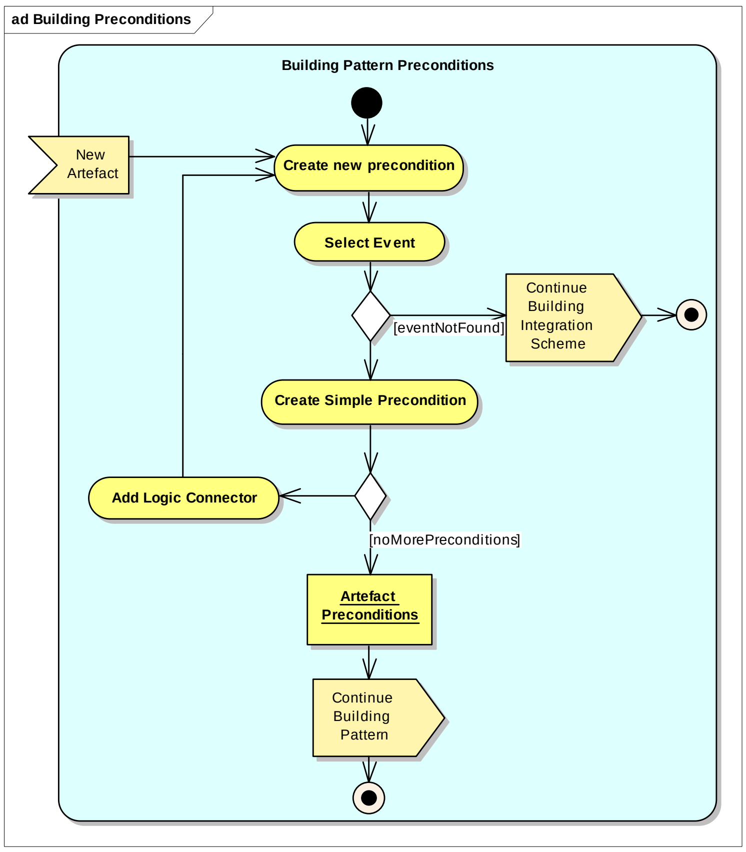
ad Building Preconditions
Building Pattern Preconditions
New Artefact
Create new precondition
Select Event
[eventNotFound]
Continue Building Integration Scheme
Create Simple Precondition
Add Logic Connector
[noMorePreconditions]
Artefact Preconditions
Continue Building Pattern

Figure 9 – Building Pattern Preconditions

The process for specifying preconditions of a given pattern (see Figure 9), starts with a simple precondition, continues with the addition of the simple preconditions needed and eventually, it ends up with the combination of them through logic operators. When the security expert has defined all the simple preconditions that constitute the *Preconditions*, the process of building an Artefact continues to the next step.

There is a special situation that can arise when creating preconditions that is worth mentioning here, given that it is closely related with the definition of Integration Schemes. When creating a precondition, a list of all possible events to consider is made available to the security expert. He can navigate through them and look for the most appropriate given his plan for the precondition. In some cases, no event is found that fulfils the requirements of the security expert. In these situations, there is no way out but creating your own *event observer* that will capture, monitor and trigger the specific event you need. Once the necessary *event observer* is created, we can consider it as a brand new Artefact. Thus, the only way to link the functionality of my artefact with the functionality of my new *event observer* is by means of an Integration Scheme. However, the discussion on the creation and functioning of Integration Schemes is out of the scope of this section, and the reader will find additional info in subsequent sections of this same document.

Concerning the process of evaluating preconditions, it consists of three steps (see Figure 10). First, when an S&D Artefact is found to be of interest (given the provided *S&D Properties* and the Artefact *Features*), the system extracts the XQuery expression that holds the specification of the Preconditions. Once the SRF has extracted the Xquery from the artefact, it launchs the query and the result is captured. The result coming from the execution of the XQuery must be a Boolean, taking the value *true* when the preconditions holds and *false* in any other case. If the result of the evaluation is *true*, then the Artefact is ready for deployment. Otherwise, Preconditions do not hold and the artefact is discarded.

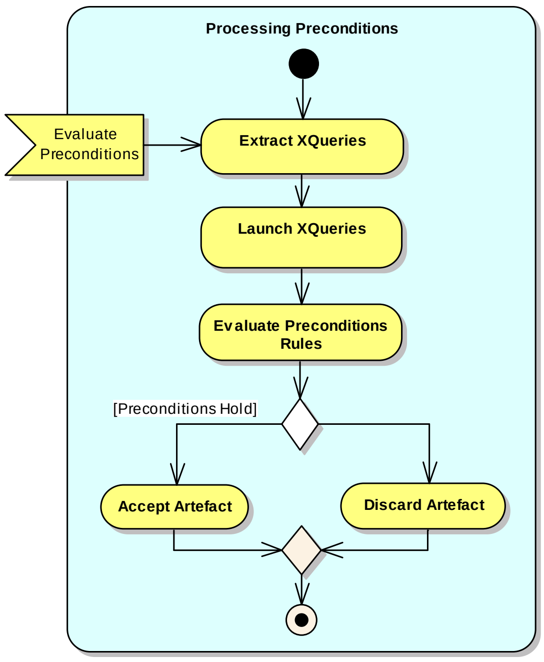

Figure 10 – Processing Preconditions

4.2. Detailed description of S&D Classes

As a first approach for the language, we have chosen to reduce this description as much as possible limiting it to the essential elements: the specification of the security properties provided and the definition of the interface to use the solution. Section 6.1.1.1. describes an example with values for each one of the fields in Table 10:

S&D Class	
1	Creator
1.1	Name
1.2	Date
2	Timestamp
3	TrustMechanisms
4	Provided Properties
4.1	Property
4.1.1	ID
4.1.2	Timestamp
5	SolutionFeatures
5.1	Feature
6	Interface
6.1	Calls
6.2	Sequence
7	Roles
7.1	Role
7.1.1	RoleName
7.1.2	Functionality
7.1.2.1	CallName
8	Comments

Table 10 – High-level data structure for an S&D Class

1. **Creator:** This field identifies the creator/provider of the pattern. It includes *Name* and *Date* fields to specify the creator of the class description and the date of creation, using the following format: yyyy-mm-dd.

2. **Timestamp:** this field represents a "digital proof that objectively enables to detect the creation time of certain data". The data stored is this field are the milliseconds spent since January 1, 1970, 00:00:00 GMT. A negative number indicates a date prior to January 1, 1970, 00:00:00 GMT.

   One of the requirements is to create a library of artefacts in which you can trust. Together with signatures, the time-stamp is a valuable proof that gives us some degree of trustworthiness by proving the time of creation of the S&D Class/Pattern/Implementation. Thus, the goal of the timestamp is to allow the users of an S&D Class/Pattern/Implementation to check not only that the document is authentic or has not been modified since its creation but also that they are working with the correct version of the document. The same definition for *TimeStamp* property applies to sections 4.3. and 4.4. .

3. **Trust mechanisms:** It is a digital signature meant to guarantee that the class description has been produced indeed by the creator, and that no modification has been done to the original Class.

4. **Provided Properties**: The main idea is to offer the user a solution for its security problems taking as input the security properties that one wants to achieve. Extracted from the application/device requirements, the security properties reduce the search for an

appropriated S&D Class. Once found, the S&D Class provides the interface to use its functionality (*Interface* field in S&D Class definition).

5. **Solution Features**: a list of the main features of the solution chosen to accomplish a security property. This is a valuable hint for the serenity user to select the most appropriated class to solve his/her problem at development time.
6. **Interface**: In this field the class developer can include the operations offered by the class (i.e. the *calls*) and the recommended *sequence* to use these calls.
7. **Roles:** When applications make use of Patterns, the used functionality is strongly dependent on the communication side in which they are applied. Then, it is important for the S&D Classes to provide the S&D Patterns with the guidelines on how to apply the functionality depending on the role they play (e.g. Server/Client). This information is necessary for developers when creating Serenity-aware applications. For instance, an IDE would be able to identify the available operations for an S&D Pattern for each role and thus, show them up in order to simplify the development process. This proposal takes shape in the *roles* section of the S&D Class definition. This section is composed by a set of roles. For each role in the set, a sequence of the available operations is available. The main goal of this section is to provide developers guidance to facilitate their work.

   Note that this section is included as a part of the S&D Class definition and not as part of the S&D Pattern. As an S&D Pattern always belongs to (at least) one S&D Class, the relation between Pattern's functionality and the role it plays, can be automatically derived from the S&D Class.

   A developer can follow two approaches to apply security solutions in a Serenity-aware application. On the one hand, the developer can apply an S&D Class to model the solution. Following this approach the roles are clearly defined in the S&D Class roles section so that at runtime, the role is selected and applied when the S&D Pattern is instantiated. On the other hand, if the developer decides to use an S&D Pattern at development time to model the solution, when it comes to runtime, the information on the role's functionality will not be directly available. Instead, as the relation between an S&D Pattern and its S&D Class was made explicit when creating the S&D Pattern, the necessary information can be easily derived from the S&D Class.
8. **Comments:** Here the creator can include any relevant information regarding the Class definition, the functionality, the expected behaviour, applicability, etc.

4.3. Detailed description of S&D Patterns

The language used in order to describe an S&D Pattern with the objective of being used by automated means in dynamic environments requires of different aspects to be included. All of them are enumerated in Table 11, and the most relevant ones are detailed afterwards:

S&D Pattern	
1	**Creator**
1.1 **1.2**	**Name** **Date**
2	**Timestamp**
3	**TrustMechanisms**
4	**PatternFeatures**
4.1	**Feature**
5	**Provided Properties**

5.1	**Property**
5.1.1	**ID**
5.1.2	**Timestamp**
6	**Interface**
6.1	**Operations**
6.1.1	**Operation**
7	**ClassAdaptors**
7.1	**Class**
7.1.1	**Adaptor**
7.1.2	**Description**
8	**Parts**
8.1	**Part**
9	**Parameters**
9.1	**Parameter**
10	**Pre-Conditions**
10.1	**SRFContext-pre-conditions**
10.1.1	**SRFContext pre-condition**
10.2	**External pre-conditions**
10.2.1	**External pre-condition**
11	**Static Tests Performed**
11.1	**Test**
11.1.1	**Conditions of test**
11.1.2	**Attack models considered**
12	**System Configuration**
13	**Monitoring**
13.1	**Monitor**
13.1.2	**Type**
13.2	**Monitoring Formulae**
13.2.1	**Rule-1**
13.2.1.1	**Event**
14	**Comments**

Table 11 – High-level data structure for an S&D Pattern

1. **Creator:** Identity of the creator/provider of the pattern. It includes *Name* and *Date* fields to specify the creator of the pattern description and the date of creation, using the following format: yyyy-mm-dd.

2. **Timestamp:** this field represents a "digital proof that objectively enables to detect the creation time of certain data". The data stored is this field are the milliseconds spent since January 1, 1970, 00:00:00 GMT. A negative number indicates a date prior to January 1, 1970, 00:00:00 GMT.

3. **Trust mechanisms:** Digital signatures and other mechanisms to guarantee that the pattern description corresponds to the pattern/solution, that it has been produced by the creator, and that it has not been modified.

4. **Pattern Features**: In this field when can find a list of the main features of the pattern. This is a hint to help to software developers to select a pattern of the library once he has selected the class to get the needed functionality. Furthermore, it will be very valuable for the SRF to choose, at runtime, the most appropiated pattern according to the enviroment.

5. **Provided Properties:** Reference to the properties provided by the pattern. Properties have a timestamp and refer to descriptions provided by the entity that defines the property (this can be the creator itself, an independent certification entity or even the SERENITY post-project organization). For the moment it is important to emphasize that these descriptions contain formal descriptions given by security experts and describe the relations between different

properties, therefore enabling the interoperation of systems referring to properties defined by different sources.

6. **Interface**: It includes every operation that integrates the interface of the pattern. This interface describes the public functionality of the S&D Pattern; in other words, the functionality made available for the applications to use the S&D Pattern's *Operations* (element 6.1 in Table 11). All the semantics, parameters and types of the S&D Pattern interface are defined in this element. The syntax used to specify the operations will follow the one from ASL language, given that it is a platform independent and easy to use language. Section 4.3.1. exposes the rationale for the adoption of ASL.

7. **Class Adaptor.** It is also necessary to count on a mechanism to map from the original high-level interface –coming the S&D Class, to the medium-level interface –used in the S&D Pattern. The translation is not direct since it is possible for a single operation at S&D Class level, to be mapped into a sequence of operations at S&D Pattern level, and thus, we have to provide some mapping from one interface to another. Moreover, as it is feasible for an S&D Pattern to belong to more than one S&D Class, then it is possible to find several *Adaptors* for the same S&D Pattern's Interface (each adaptor linked to the S&D Class that adapts). Consequently, each *ClassAdaptor* includes a set of *Class*, one for each adapted S&D Class. Each *Class* has a reference to the S&D Class adapted; and a couple of elements that define the interface: (i) the *Adaptor*, where Class calls are mapped to Pattern' operations; and (ii) an optional *Description* of the adaptor. Summarizing, element 7 describes the adequate sequence of S&D Pattern level operations for each one of the S&D Class level operations.

8. **Parts:** in order to achieve its full functionality, some external components may be used by the Pattern. These components (*Parts* from now on) are elements that have specific behaviour and features that complement the S&D Pattern functionality. As a *Part* provides its own functionality and has an associated description, the reader may confuse it with an S&D Pattern. However, there is a crucial distinction between both concepts: while an S&D Patterns do provide specific S&D Properties, *Parts* do not. As an example, in the case study described in the next section, the camera is a *Part*. The requirements for the application of the *Parts* are included in their description.

9. **Parameters:** An S&D Solution has some variables whose values are assigned when the solution is instantiated for particular scenario. These instantiable elements are called S&D Patterns' parameters. At some extent, they allow S&D Patterns to act as overloaded operator in a programming language: their precise behaviour is not known before execution time, and it depends on the types of values given when calling the operator. For instance, the length of a cryptographic key might be defined as a parameter. Note that Parameters can always be defined using a simple tuple (name, value) while *Parts* can not.

   Associated to parameters, there must be also a constraints' field describing the restrictions that an element must meet in order to be used as actual parameter. One important aspect of these constraints is that they are internal to the S&D Solution.

10. **Pre-Conditions:** they describe general conditions that the target system must meet before applying a pattern. A pattern is not necessarily a universal solution. This means that in order for the pattern to be successfully used to provide the declared properties, some pre-conditions must be met. In most cases, these preconditions will be derived from the analysis of the solutions made by security engineers. Preconditions are classified depending on the check process performed, falling in two different categories: *SRF context preconditions* and *External preconditions*.

11. **Static Tests Performed:** Security engineers will be responsible for the static testing of the pattern. This section will describe all relevant information regarding the static tests performed. We foresee that it might be necessary to develop mechanisms for the description of the tests, in a similar way to the description of the properties. This section will be useful for the end user because it will facilitate the selection of the appropriate pattern and for the monitoring mechanism because some monitoring rules can be derived from it. It is important to note that the monitoring activity might have an impact on the S&D Solution. Therefore, the static test should explicitly consider this interaction.
12. **System Configuration:** In addition to the instantiation and integration of the pattern in the system it will be sometimes necessary to perform some actions prior to the integration of the pattern in the system. Likewise, when the pattern is to be removed, some actions may also be necessary. We will use the term *activate* to refer to the process of instantiating the pattern, integrating it in the running system and initializing it, so that it is ready to provide the properties declared. Similarly, the term *deactivate* will be used to refer to the process of removing the pattern from the system, which may require some "closing-up" procedure. In summary, the system configuration section of the description will describe the initialization and closing up processes, along with any other relevant system-specific information. This other system-specific information includes, for instance, the type of connections used. An important aspect to be considered in the system configuration set-up is *when* the monitor should be initialized. In this sense, the administrator of the system could set up a *monitoring priority policy*. N.B. that in this section we do not include the description of the monitor and the monitoring rules.
13. **Monitoring:** This row describes all information necessary for the monitoring of the pattern. In particular, it must include which monitor to use, and the configuration of such monitor (events to monitor, rules, reactions, etc.). Section 4.4. of this document gives a complete study on the monitoring information to be included in this row.
14. **Comments:** Here the creator can include any relevant information regarding the Pattern definition, the functionality, the expected behaviour, applicability, etc.

4.3.1. Rationale for ASL Adoption

At S&D Pattern's level, no information about the software execution platform, hardware or programming language is available. Consequently, the interface specification may be considered as a Platform Independent Model (PIM) in the sense of OMG's Model Driven Architecture, given that:

— The S&D Pattern that specify the S&D Solution behaviour can be ported without change even if the target platform changes,

— All the solution features that are unique to the target platform must be declared at S&D Implementation level,

— The translation from the S&D Pattern's level (PIM) to the S&D Implementation's level (Platform Specific Model, or PSM), should be straightforward.

For those not familiar with MDA Models, MDA defines two primary sets of model, the Platform Independent Model and the Platform Specific Model. Here the term *platform* is used to refer to technology and engineering details that are irrelevant to the fundamental functionality of the software. These model types are a key concept in the MDA architecture; it mandates the separation-of-concerns of analysis (the PIM) from its realization on a particular computing platform and technology (the PSM) and recognizes that the refinement relationship between the two types of model should be achieved by applying a *mapping*. The parallelism with our concepts of S&D Pattern, S&D Implementation and Executable Component is apparent and reinforced by the need of a platform independent language to express the functionality of the S&D Pattern (the definition of the *Operations*). This brings us to the next element in the *Interface* definition, which exposes the need of an implementation independent language for specifying processing within the context of the S&D Patterns: the *Class Adaptor.*

At this stage, it is mandatory the selection of an appropriate language for expressing the Interface *Operations* definition as well as the *Class Adaptor*. Following the parallelism with MDA, there is an emerging technology that merges the Unified Modelling Language and the concepts of PIM and PSM: eXecutable UML (xUML). The idea was simple; for UML to be executable, we must have rules that define the dynamic semantics of the specification. That is when xUML snaps into action. Executable UML is designed to produce a comprehensive and comprehensible model of a solution without making decisions about the organization of the software implementation. And to do that, xUML is supported by a UML compliant Action Language: the ASL or Action Specification Language.

The ASL definition is independent of any particular implementation and can be freely used by modellers and developers. It provides an unambiguous, concise and readable definition of the processing to be carried out by an object-oriented system within the context of an Executable UML (xUML) model, and it is easily applicable to the definition of the S&D Pattern's *Interface* and the *Class Adaptors*. In addition, different techniques have been developed for mapping the ASL into the chosen software architecture and implementation language. This means that the translation from an Class Adaptor definition to the Executable Component that realizes that functionality can be semi-automatic. The translation techniques range from fully automatic generation to manual coding using a defined set of rules. Target languages have included c, c++, Objective c, Ada, Java, Fortran, and SQL.

4.4. Detailed description of S&D Implementations

At this point, we foresee the following components of this description (Table 12):

S&DImplementation	
1	**Creator**
1.1	**Name**
1.2	**Date**
2	**TimeStamping**
3	**TrustMechanisms**
4	**SandDPatternReference**
5	**Preconditions**
5.1	**Precondition**
6	**ImplementationDescription**
7	**ImplementationReference**
7.1	**Reference**
8	**ComplianceProofs**
8.1	**Proof**
9	**Comments**

Table 12 – High-level data structure for an S&D Implementation

1. **Creator:** Identity of the creator/provider of the S&D Implementation. It includes *Name* and *Date* fields to specify the creator of the pattern description and the date of creation, using the following format: yyyy-mm-dd.
2. **Time-stamp:** Analogous to the one used in the S&D Patterns, it stores the milliseconds spent since January 1, 1970, 00:00:00 GMT. A negative number indicates a date prior to January 1, 1970, 00:00:00 GMT.
3. **Trust mechanisms:** These are analogous to the ones used in the S&D Patterns.
4. **Reference to the S&D Pattern Implemented:** Each implementation references the S&D Pattern it implements.
5. **Particular Preconditions of this implementation:** In addition to the preconditions related to the solution (S&D Pattern), each implementation may have some additional preconditions derived from the implementation details.
6. **Description of the Implementation:** This description is meant to be useful for the selection of a particular implementation.
7. **Reference to the actual implementation:** There must be a secure (probably cryptographic) reference to the actual implementation, in order to avoid this description to be erroneously associated to a different implementation.
8. **Compliance Proofs:** Opposed to the S&D Patterns, where the formal analysis and other validation tools are very useful, in the case of implementations, the important aspect is to have proofs of the compliance of the implementation to the S&D Pattern description.
9. **Comments:** Here the creator can include any relevant information regarding the Implementation definition, the functionality, the expected behaviour, applicability, etc.

4.5. Specifying Monitoring Rules in S&D Patterns

In this section we describe the language used for expressing the monitoring rules within the S&D Patterns. The exact position within the S&D Patterns where these rules will be described is under the *Monitoring Formulae* clause that is part of the more general *Monitoring* clause (see pattern description example in section 7.2.). This language is an extension of EC-Assertion – an event calculus (EC [4]) based language defined by an XML. EC-Assertion has been developed at City University to support the specification of general functional and quality requirements that should be monitored during the execution of service based systems as part of the SECSE project [8][9]. For the purposes of SERENITY, we have introduced certain extensions to this language and generated a new version of EC-Assertion that we describe below.

The extensions that we have introduced to *EC-Assertion* in order to support the specification of security and dependability properties that could be monitored at runtime are:

- The introduction of a generic scheme for specifying different types of monitorable *events*
- The introduction of a generic scheme for the specification of fluents (i.e. conditions about the state of a system) including fluents signifying the authentication and authorisation of agents to issue and accept events requesting the execution of operations or responding to operation calls

The extended version of *EC-Assertion* has been defined as an XML schema [5] in order to provide a standard way of expressing the event calculus (EC) formulas that will be monitored. This schema is described in Section 1.1. of this report and its full definition is provided in Appendix A. In the following, we describe the extended form of EC-Assertion and give an example of using it to express a rule for monitoring a security property. This description follows an overview of Event Calculus that provides the logic based foundation of our language.

4.5.1 Specification of Monitoring Rules in Event Calculus

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

An event in EC is something that occurs at a specific instance of time (e.g., invocation of an operation) and may change the state of a system. Fluents are conditions regarding the state of a system which are initiated and terminated by events. A fluent may, for example, signify that a specific system variable has a particular value at a specific instance of time or that a specific relation between two objects holds.

The occurrence of an event is represented by the predicate $Happens(e,t,\Re(t_1,t_2))$. This predicate signifies that an instantaneous event e occurs at some time t within the time range $\Re(t_1,t_2)$. The boundaries of $\Re(t_1,t_2)$ can be specified by using either time constants or arithmetic expressions over the time variables of other predicates in an EC formula. The initiation of a fluent is signified by the EC predicate $Initiates(e,f,t)$ whose meaning is that a fluent f starts to hold after the event e at time t. The termination of a fluent is signified by the EC predicate $Terminates(e,f,t)$ whose meaning is that a fluent f ceases to hold after the event e occurs at time t. An EC formula may also use the predicates $Initially(f)$ and $HoldsAt(f,t)$ to signify that a fluent f holds at the start of the operation of a system and that f holds at time t, respectively. An EC formula can also specify additional constraints about the time variables of predicates using the predicates < and =. For example, t1 < t2 is true if t1 occurred at a time instance before t2; and t1=t2 is true if t1 occurred at the same time instance as t2.

Our EC based language uses special types of events and fluents to specify monitorable properties of systems. More specifically, fluents can be defined by the user as relations between objects as follows:

relation(Object$_1$, …, Object$_n$) (I)

,where *relation* is the name of the relation that takes as arguments *n* objects (Object$_1$, …, Object$_n$) that can be fluents or terms. A pre-defined relation for fluents that is commonly used is:

valueOf(variable, value_exp) (II)

whose meaning is *variable* has the value *value_exp*. In (II), *variable* denotes a typed variable or a list of typed variables which may be:

— *System variables* – A system variable is a variable of the system that is being monitored whose value can be captured at any time during the monitoring process, or

— *Monitoring variables* – A monitoring variable is introduced by the users of the monitoring framework to represent the deduced states of the system at runtime (i.e. states which the system itself might not be aware of but the monitor of the system uses in order to reason about the system).

— *value_exp* is a term that either represents an EC variable/value or signifies a call to an operation that returns an object of the same type as the variable. This operation may be a built-in operation of the monitoring engine (e.g. an operation that computes the average of a set of values) or an operation that is invoked in an external party. When *value_exp* is an operation call, then effectively the return value of the operation becomes the value of *variable*.

Events in our framework represent exchanges of messages between the agents that constitute a system. A message can invoke an operation in an agent or return results following the execution of an operation. Events are described in EC by terms that have the following generic form:

event(_id, _sender, _receiver, _status, _oper, _source) (III)

where:

— *_ID* is a unique identifier of the event

— *_sender* is the identifier of the agent that sends the message.

— *_receiver* is the identifier of the agent 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 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.

— *_oper* is the signature of operation that the event invokes or reports the results of.

— *_source* is the name of the agent that provided information about the event.

5. XML Representation of the language

5.1.1. XML Schema for S&D Classes

The structure of an S&D Class is defined by the complex XML type called *S_and_DClass*. Graphically represented in Figure 11, *S_and_DClass* has no attributes but includes the following child elements:

1. One *creator* element, which is of type *creatorType*, a complex type that consists of the following child elements:

   I. *Name*: String used to specify information about the author of the S&D Class. The creator can be a person, a software company or organization, etc.

   II. *Date*: String used to store information about the date when the S&D Class was created. The format used to specify that date will be: yyyy-mm-dd.

2. One *timestamping* element, stored as proof to detect the creation time of the S&D Class. The format used will be of type long, where the user can specify the milliseconds spent since January 1, 1970, 00:00:00 GMT. A negative number indicates a date prior to January 1, 1970, 00:00:00 GMT.

3. One *trustMechanisms* element. This element is used in order to store digital signatures or any other trust mechanism well suited to guarantee that the pattern description (i) really corresponds to the pattern/solution it describes, (ii) has been produced by the defined creator, and (iii) has not been modified during its lifecycle. It is of type *trustMechanismsType*, a complex type that consists of the following sequence of child elements:

   I. *SignatureType:* It is of type *String*, and it is used to define the sign algorithm, the parameters necessary to verify it, and any other element related with the type of signature scheme used.

   II. *Signer:* It is used to define the entity that has signed this *Class*.

4. One *providedProperties* element. ProvidedProperties Element is of type propertiesType and is meant to hold the security properties offered by the adoption of this S&D Class. The type propertiesType consists of the following elements:

   I. *Property:* This element is of type *propertyType*. *PropertyType* is a complex type that consists of the following sequence of elements:

      i. *Id:* This is the identification for a concrete security property. The user can map from this ID to the complete description of the security property provided by the S&D Class.

      ii. *timestamp:* the timestamp for this property.

5. One *solutionFeatures* element. This element is of type solutionFeaturesType and is meant to hold the main features to describe the solution proposed for the S&D Class. SolutionFeaturesType is a complex type that consists of feature elements of string type.

6. One *interface* element. This element is used to describe the interface provided by this S&D Class. All the S&D Patterns have to comply with the interface of the S&D Class from which they inherit. It is of type *interfaceType*. The *interfaceType* type is a complex type that consists of the following elements:

    I. *Calls*: This element is of type *callsType*, described later in this paragraph. The aim of *calls* element is to provide the way in which the S&D Class should be invoked. In some sense, is a set of interfaces corresponding to high level functions available to interact with the S&D Class.

        i. *callsType*: callsType Type consists of one element called call (of type String), which includes the format (i.e. name, parameters) of a concrete call.

    II. *Sequence:* This element is of type *sequenceType*. It allows the user to specify the correct sequence of callings when invoking the S&D Class.

        i. *SequenceType:* It is a complex type that stores *sequence* elements, described below.

        ii. *Sequence: Sequence* elements are composed by a set of *step* Elements. It allows the creator to specify the sequence to use the *calls*.

    III. *Constrains*: This element exposes the contraints of the sequence to take on account when the developer is using the *calls* at the development time.

7. One *roles* element. This element is of type *rolesType* and it is used to describe the sequence of *calls* of the interface definition set for each role. This complex type is composed by a sequence of elements of type *roleType*. This type contains the two following fields:

    I. *roleName:* It is of type String and it defines the name of the role.

    II. *funcionality:* It is of type *functionalityType* and represents a sequence of *functionName* elements, each one of type String.. These *functionName* elements refer to the functions available for the role represented by the *roleName* field.

8. One *comments* element. This element can be used to specify any general comment regarding the S&D Class specification.

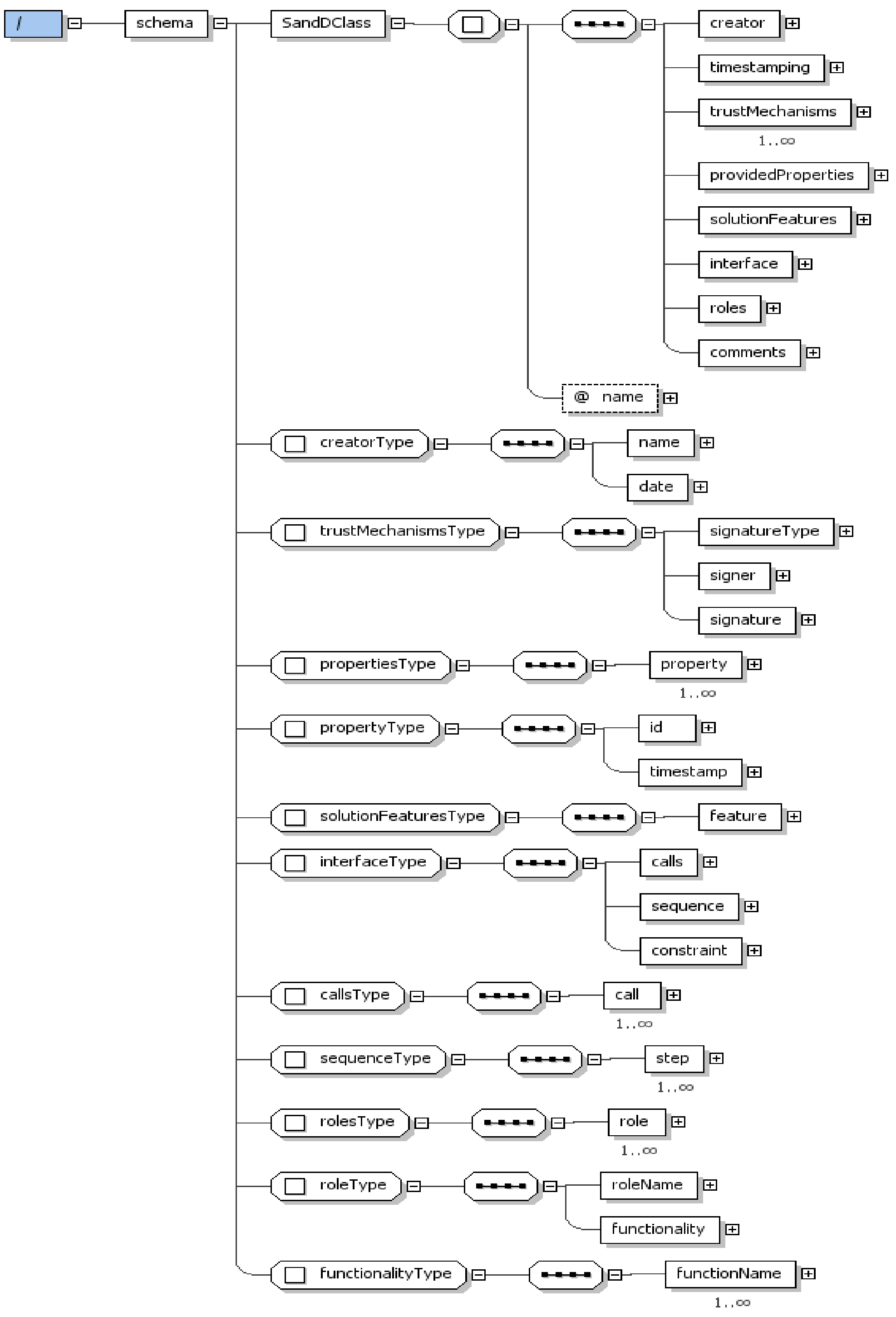

Figure 11 – Representation of XML Schema for S&D Classes

5.1.2. *XML Schema for S&D Patterns*

The structure of an S&D Pattern, as shown in Figure 12, is defined by the complex XML type called *S_and_DPattern*. It has one attribute: *name*, a String used to store the name of the pattern. The *S_and_DPattern* is composed of the following child elements:

1. One *creator* element, whick is of type *creatorType*, a complex type that consists of the following child elements:
    a. *Name*: String used to specify information about the author of the S&D Pattern. The creator can be a person, a software company or organization, etc.
    b. *Date*: String used to store information about the date when the S&D Pattern was created. The format used to specify that date will be: yyyy-mm-dd.
2. One timestamping element, stored as proof to detect the creation time of the S&D Pattern. The format used will be of type long, where the user can specify the milliseconds spent since January 1, 1970, 00:00:00 GMT. A negative number indicates a date prior to January 1, 1970, 00:00:00 GMT.
3. *TrustMechanisms* elements. Apart from sharing the same name, this element plays the same role in S&D Patterns that trustMechanisms element plays in S&D Classes. It is of complex type trustMechanismsType. The trustMechanismsType is a complex type composed of the following child elements:
    a. One or more *sign* elements. This element is of type *signType*, that is a complex type composed by:
        i. One *signatureType*, of type String.
        ii. A *signer* that is used to store, in a String, the signer.
        iii. This type will probably have some more elements but this issue stills in discussion.
4. One *patternFeatures* element. This element is of type patternFeaturesType and is meant to hold the main features to describe the S&D Pattern. PatternFeaturesType is a complex type that consists of feature elements of String type. Each one of this features will be decisive to select an appropriate S&D Pattern among all the patterns that comply with the S&D Requirements.
5. One *ProvidedProperties* element. This element plays the same role that ProvidedProperties element plays for S&D Classes. Several Properties can be defined. Properties element is of type propertiesType. The PropertiesType complex type is composed by a set of property elements, which is an element of type propertyType, composed of:
    a. An *ID* element of type String. It is used to store the identification of the property.
    b. A *timestamp* element. It is of type String.
6. One *Parts* element suited for describing the Parts that are used by the S&D Pattern. It is of complex type partsType, which is a set of part elements of complex type partType, composed of the following attributes and elements.
    a. An *id* attribute of type String to store the identification of the represented *Part*.
    b. A *url* attribute of type String, used to reference the URL of the Part.

   c. A *type* attribute. It is a String for defining the type of the Part.

   d. A *description* element of type String. Describes the *part* element itself.

7. One interface element that describes all the functionality of the S&D Pattern. The interface element is of the type InterfaceType. The InterfaceType is a complex type composed of the following child elements:

   a. One *operations* element, of type *operationsType*. This *operationsType* is a complex type that contains one or many *operation* element, each one of the type *operationType*. The *operationType*, is a complex type that contains:

      i. One attribute called *name*, of the type *String*, to specify the name of the operation.

      ii. A *definition* element to describe the operation interface using the ASL syntax.

   b. One *classAdaptors* element to describe the adaptation from SandD_Class operations to SandD Pattern operations. This element is of *classAdaptorsType* type that contains one or many *adaptor* elements. This is because a SandD Pattern can offer an adaptor for different SandD Classes. The adaptor element is of type *adaptorType*, that contains:

      i. An attribute, *classReference*, to specify the SandD Class adapted. It is of type *String*.

      ii. An *operation* element for describing, using ASL syntax, the adaptation. The type of the *operation* is the *operationType* described above.

8. One *Parameters* element. This element is used to store data about the parameters of the pattern. These parameters are especially relevant when the pattern is instantiated, as some concrete value has to be assigned to them at instantiating time. Several Parameters can be defined. It is of type parametersType, a complex type that consists of a set of following child element:

   a. *Parameter* element is of type String.

9. *Preconditions* element. It is planned to have a Preconditions element for each pattern precondition. Several Preconditions can be defined. It is of type preconditionsType, a complex type that consists of a sequence the following elements:

   a. *ParameterPreconditions* is of complex type *parameterPreconditionsType* that is a sequence of *parameterPrecondition* elements. The *parameterPrecondition* is of type *parameterPreconditionType*. This type is not defined yet.

   b. *SolutionPreconditions* is of complex type *solutionPreconditionsType* that is a sequence of *solutionPrecondition* elements. The *solutionPrecondition* is of type *solutionPreconditionType*. This type is not defined yet.

10. *StaticTestsPerformed* element. This is the element specially suited for representing the static test performed to probe the solution described by the pattern. The StaticTestsPerformed element is of complex type staticTestsPerformedType that is a sequence of test elements. A Test element is of complex type testType that has an attribute called name that is a String. Test elements consist of a sequence of the following elements:

    a. *conditionsTest*, a String that is used in order to describe test conditions.

   b. *attackModels* for representing attack models. This element is of type String.

11. SystemConfiguration element is used to describe the system configuration of the target system. It includes a textual description with the technical details. The reader can consult section 4.3. (Detailed description of S&D Patterns) for more information about system configuration issues. It is of type systemConfigurationType. This complex type is not defined yet in this version of the pattern definition language.
12. Monitoring elements, these elements are intended for monitoring purposes. Section 4.5. of this document gives a complete study on the monitoring information to be included in this row.
13. One comments element. This element can be used to specify any general comment regarding the S&D Pattern specification.

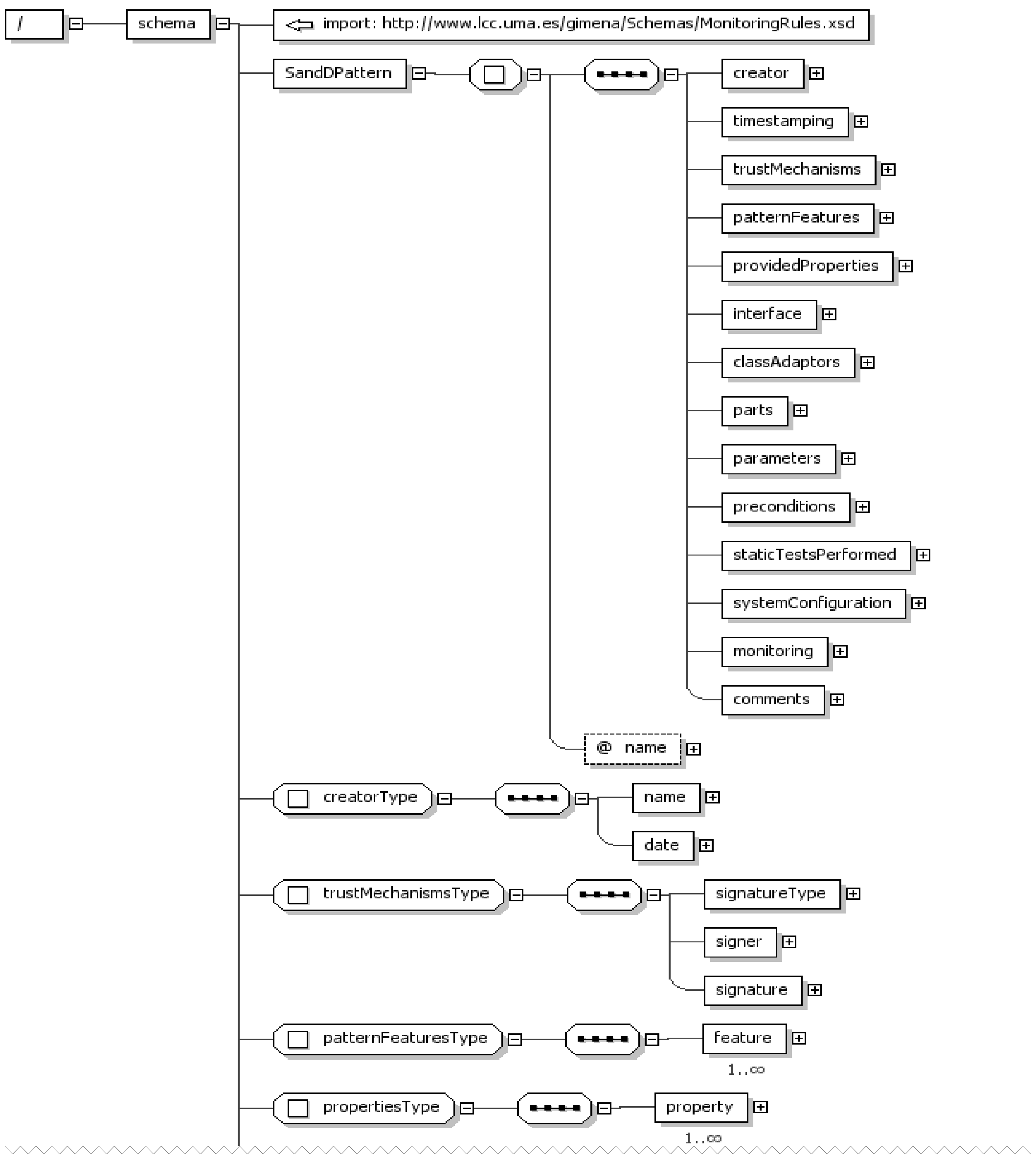

Figure 12 – Partial representation of XML Schema for S&D Patterns (I)

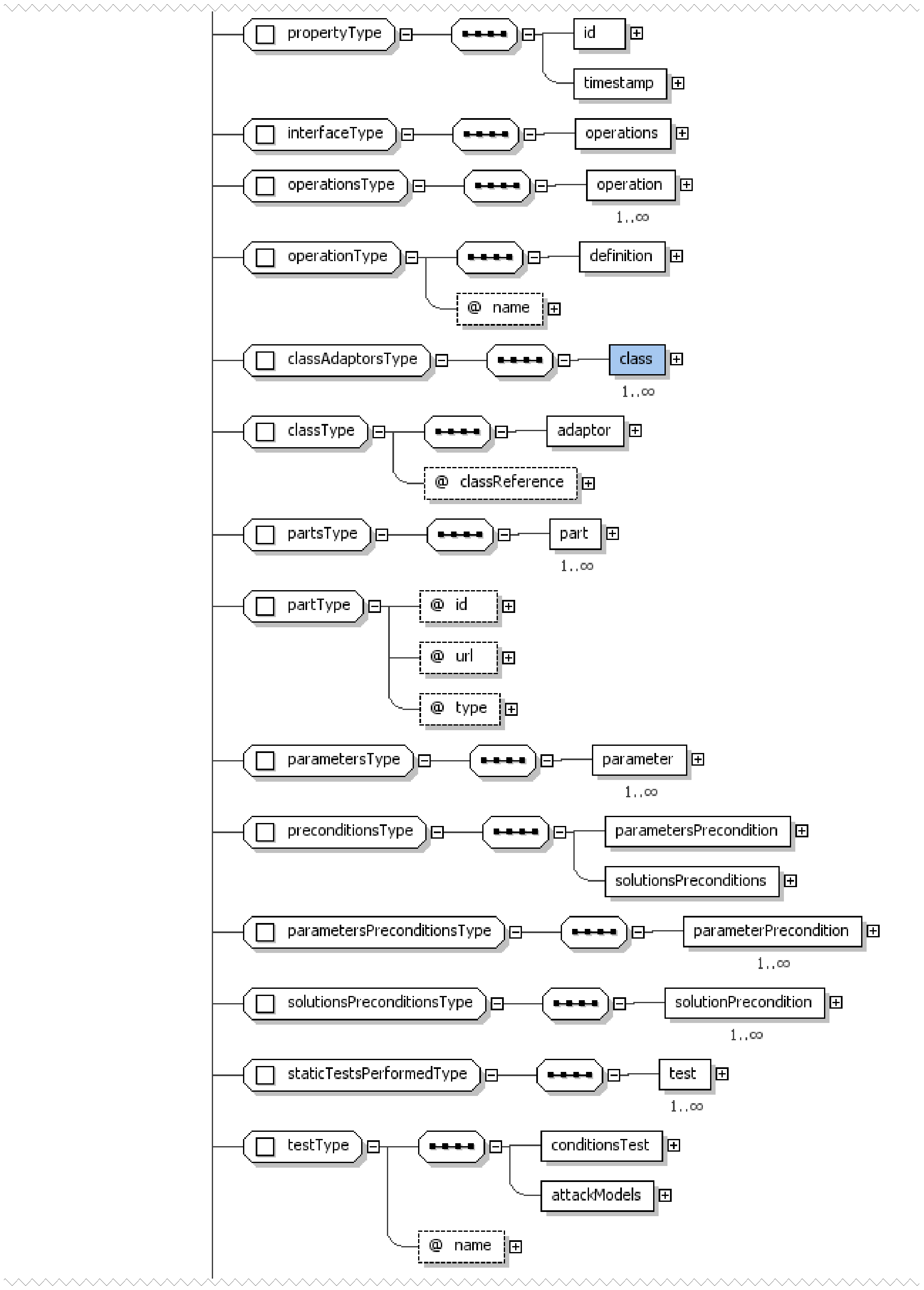

Figure 13 – Partial representation of XML Schema for S&D Patterns (II)

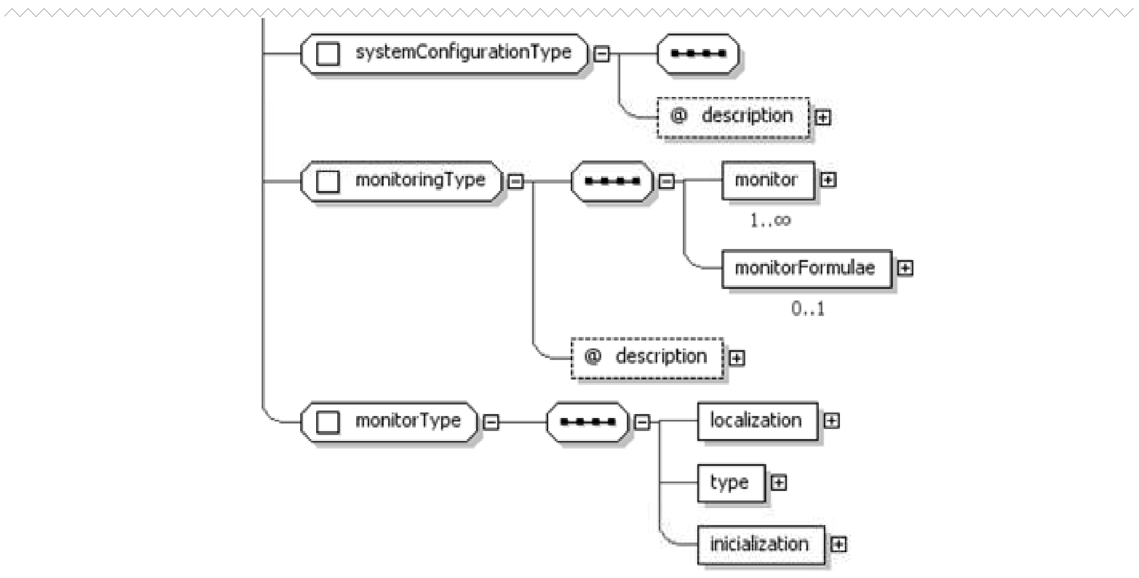

Figure 14 – Partial representation of XML Schema for S&D Patterns (and III)

5.1.3. *XML Schema for S&D Implementations*

The structure of an S&D Implementation is defined by the complex XML type called *S_and_DImplementation* and represented in Figure 15. *S_and_DImplementation* consists of the following child elements:

1. One *creator* element, which is of type *creatorType*, a complex type that consists of the following child elements:
    a. *Name*: String used to specify information about the author of the S&D Implementation. The creator can be a person, a software company or organization, etc.
    b. *Date*: String used to store information about the creation date of the S&D Implementation. The format used to specify that date is: yyyy-mm-dd.
2. One timestamping element, stored as proof to detect the creation time of the S&D Implementation. The format used will be of type long, where the user can specify the milliseconds spent since January 1, 1970, 00:00:00 GMT. A negative number indicates a date prior to January 1, 1970, 00:00:00 GMT.
3. TrustMechanisms element. Similar TrustMechanisms elements are described in the two previous sections and incorporated into their XML Schemas.
4. An S_and_DPatternReference element refers to the S&D Pattern implemented by this S&D Implementation. It is of type S_and_DPatternReferenceType, a complex type that consists of the following child elements:
    a. *Id*: Identification for the corresponding S&D Pattern.
    b. *Signature*: it includes a signature to verify the authenticity of this reference.

5. Preconditions, It is planned to have a precondition element for each implementation precondition. The preconditions element is of a complex type, which consists of a sequence of one or more precondition elements.
    a. The *precondition* element is of type String and will to store data about preconditions.
6. ImplementationDescription element is used to store the description of the Executable Component pointed by the S&D Implementation. It is composed by a complex type that has only one element:
7. Description: a String where the textual description is stored.
8. An implementationReference element that is of type implementationReferenceType. This element is used to establish a reference to the actual implementation, in order to avoid this description to be erroneously associated to a different implementation. The implementationReferenceType is a complex type that consists of a sequence of property elements. The property complex type consists of the following elements:
    a. *url*: which stores a String
    b. *signature*: to check the validity of this element.
9. One or more complianceProofs elements. They are aimed to store proofs of the compliance of the implementation to the corresponding S&D Pattern. This element has a complex type with only one element: proof. This element is of type String.
10. One comments element. This element can be used to specify any general comment regarding the S&D Implementation specification.

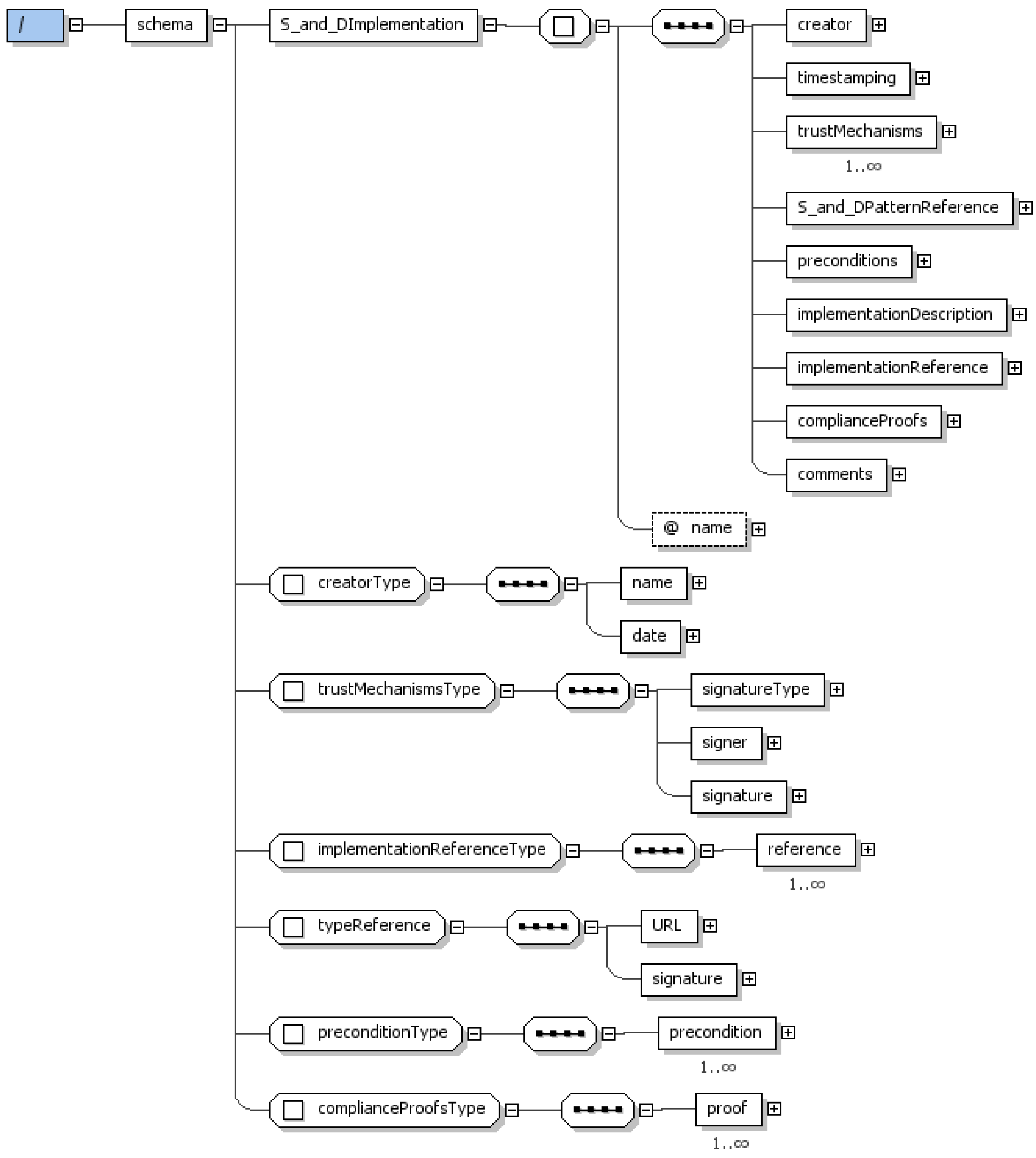

Figure 15 – Representation of XML Schema for S&D Implementations

5.1.4. *XML Schema for Monitoring Rules*

The structure of a monitoring rule is defined by the complex XML type called *formulaType.* It has two attributes: *formulaid* for identifying the formula, and *forChecking*, a Boolean used to distinguish between assumptions and rules. *formulaType* consists of the following child elements:

1. At least one quantification element, which is used to specify the quantification of variables in an EC formula. It is of type quantificationType, which is a complex type and consists of a quantifier element, to represent the quantifier (i.e. existential or universal), and a choice of variables that can be quantified, i.e. regularVariable (all other variables except for time variables) or timeVariables.
2. Zero or one body element, which specifies the expression on the Right Hand Side (RHS) of the implication (if any), i.e. the body of the formula. It is of type bodyHeadType, a complex type that consists of the following sequence of child elements:
    a. *A predicate* element that is used to define the predicate in the formula and whose type is *predicateType*. *predicateType* is a complex type that has two attributes: *negated*, a Boolean used to indicate if a predicate is negated and whose default value is false, and *unconstrained,* a Boolean that is true if the predicate is unconstrained and whose default value is false. It also consists of the following child elements:
        i. *happens*, which is of complex type *happensType*, that consists of the following sequence of elements:
            - *event* that is of type *eventType* for representing the event. This type is a complex type and consists of the following child elements: *eventID* of type String for identifying uniquely the event, *sender* of type *variableType* for specifying the agent that sends the message, *receiver* of *variableType* for specifying the agent that receives the message, *status* of type *String* for representing the processing status of an event, *oper* of type *operationType* for representing the operation signature that the event invokes or reports the result of and *source* of type *String* for specifying the agent that provided information about the event. The complex type *variableType* is explained later (see 8th bullet point). The complex type *operationType* consists of the following sequence of child elements: *opName* of type *String* for defining the name of the operation and zero or one *op_args* of type *String* for defining the possible argument of an operation. See Figure 16.
            - *timeVar* is of complex type *timevariableType* that represents the time variable. The type *timevariableType* consists of the following child elements: *varName* for specifying the name of the variable, *varType* for specifying the type of the variable, and zero or one *value* element for specifying the value of the variable.
            - *fromTime* is of type *TimeExpression* and represents the starting time of the time range within which the formula should hold. TimeExpression consists of: a *time* element that is of type *timevariableType* that has been described above; and a choice of time operators, namely *plusTime* that is of type *timevariableType*, *minusTime* that is of type *timevariableType*, *plus* and *minus* which are both of *decimal* type.

- *toTime* is of type *TimeExpression* and represent the finishing time of the time range within which the formula should hold. *TimeExpression* has been described in detail above.

ii. *initiates,* which is of complex type *initiatesType* that consists of the following child elements:

- *event* that is of type *eventType* for representing the event, as described above.
- *fluent* is of type *fluentType* and it distinguishes between the different types of fluents that can be described in the formula. *fluentType* is a complex type and has the following child elements:
  - *author* that is of type *authorisationFluentType* and is used to represent that an authorised agent (*authorisedAgent*) has been authorised by an authorising agent (authorisingAgent) to receive and process an event or to send an event;
  - *exp* that is of type *exposesFluentType* and is used to represent that the response generated from the execution of an operation (*event*) will disclose an information term (*infoTerm*) which belongs to the agent owner.
  - *authen* that is of type *authenticationFluentType* and that is used to represent that an agent (*agent*) is authenticated when a specific event (*event*) has been processed.
  - *valueof* that is of type *valueofType.* This represents a predefined relation for fluents where a variable that is given at the *target* (i.e. the first argument) is updated with the value or either a variable at the *source* (i.e. the second argument) or with the return value of an operation that is called. The complex type *valueofType*, therefore consists of: a *target* and a *source* element. The types of these elements consequently consist of a *variable* element, and in the case of the source, or an *operationCall* element.
- *timeVar* is of complex type *timevariableType* that represents the time variable.

iii. *holdsAt* is of type *holdsatType* that consists of the following sequence of elements:

- *fluent* that is of type *fluentType* (as described for the initiated predicate).
- *timeVar* that is of type *timevariableType*(as described for the initiated predicate).

iv. *initially* is of type *holdsatType,* which is described above.

v. *terminates* is of type *terminatesType* that is a complex type that consists of the following child elements:

- *event* is of type *eventType* that has been previously described.
- *fluent* is of type *fluentType* that has been previously described.
- *timeVar* is of type *timevariableType* that has been previously described.

b. *relationalPredicate* is of complex type *relationalPredicateType* that specifies the possible relations between two variables in the formula. This type has the following child elements:

i. a choice of the following elements:

- *equalto*
- *notEqualTo*
- *lessThan*
- *greaterThan*
- *lessThanEqualTo*
- *greaterThanEqualTo*

which are all of complex type *varRelationType* that consists of two elements: *operand1* and *operand2* of type *operandType.* The complex type *operandType* consists of the following choice of elements (only one of these elements will be represented):

- *operationCall* that is of type *operationCallType* that has a sequence of child elements: *name* of type *String*, zero or one *partner* of type *String* and zero or more (unbounded) *variable* elements of type *variableType*, which is described below.
- *variable* that is of type *variableType.* This type is a complex type that has two attributes: *persistent* that indicates whether the value of the variable is the same throughout all instances (like static variables in Java) and *forMatching* that distinguishes between internal and external variables (i.e. its value is false for internal variables). Also, the type consists of the following child elements: *varName* that is of type *String*, and either a *varType* and *value* element, both of type *String*, or an *array* element of type *arrayType* with elements that describe the array structure: a *type* accompanied by zero or one *index*, both of type *String*, and zero or more *value* elements of type *arrayValueType*.
- *constant* that is of type *constantType* for describing constants. This type consists of two elements: *name* and *value* elements which are both of type String.

ii. *timeVar* is of type *timevariableType* that has been previously described.

c. a possible sequence of an *operator* and a choice of either:

i. a *predicate* that is of type *predicateType* that has been explained earlier,

ii. a *timePredicate* that is of type *timepredicateType*. This element is used to express a relation between two time variables in the formula. It has a choice of the following child elements: *timeEqualTo, timeNotEqualTo, timeLessThan, timeGreaterThan, timeLessThanEqualTo, timeGreaterThanEqualTo,* all of complex type *TimeRelation* that consist of two elements: *timeVar1* and *timeVar2* of type *TimeExpression* that has been described earlier. Or

iii. a *relationPredicate* that is of type *relationPredicateType* that has been explained earlier.

3. A head element which is of type bodyHeadType, which is described above.

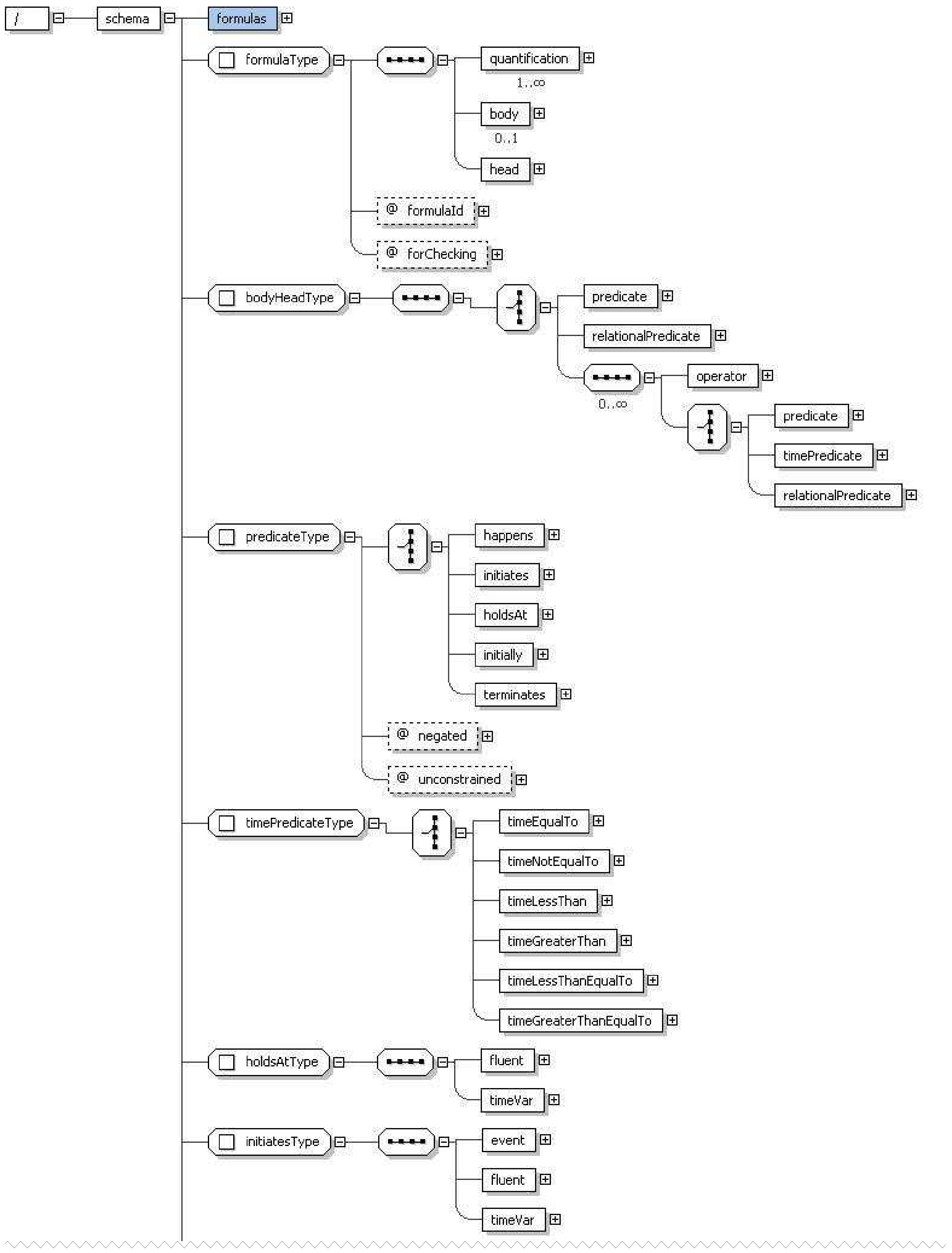

Figure 16 – XML Formula Representation Schema (I)

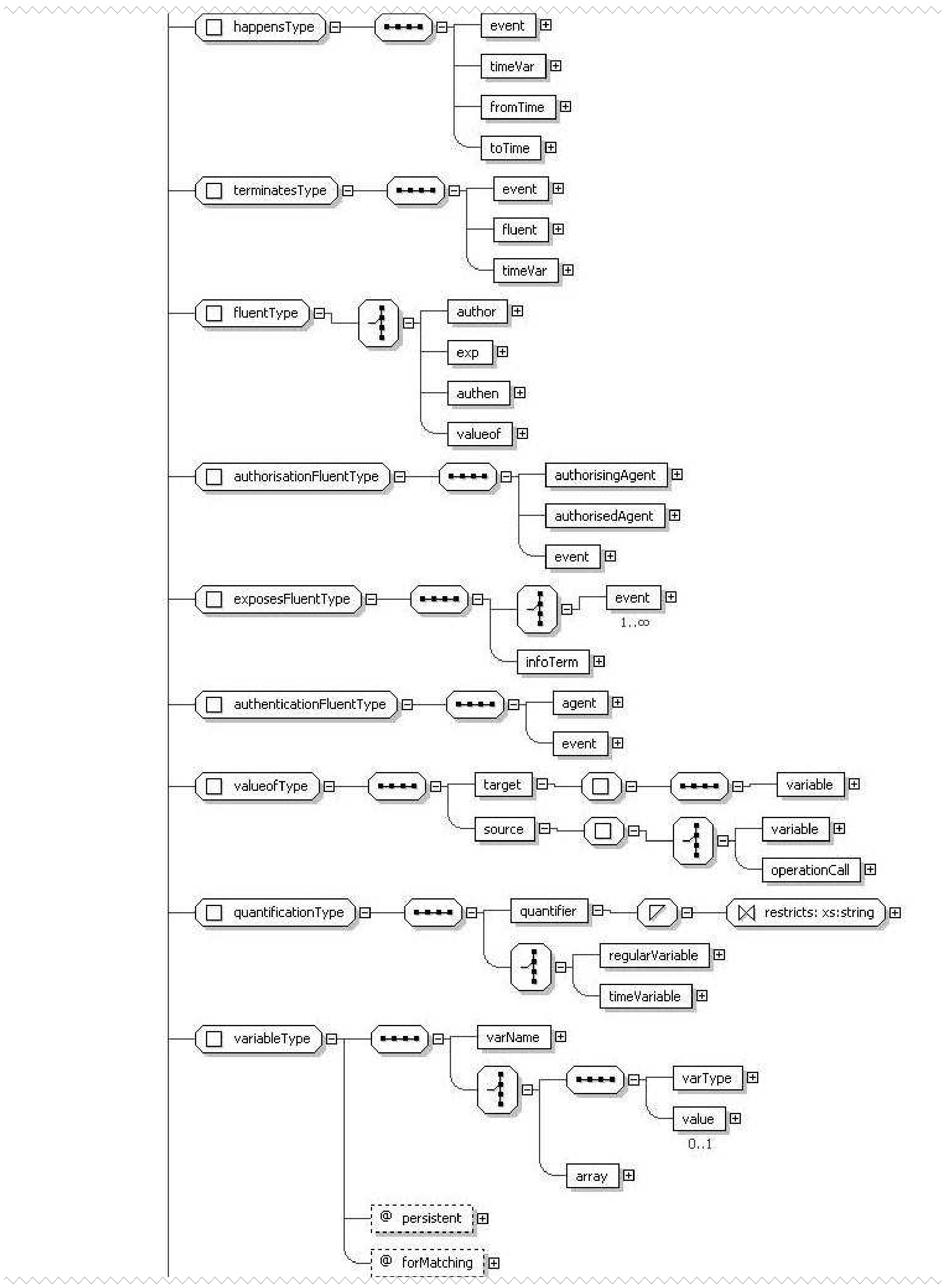

Figure 17 – XML Formula Representation Schema (II)

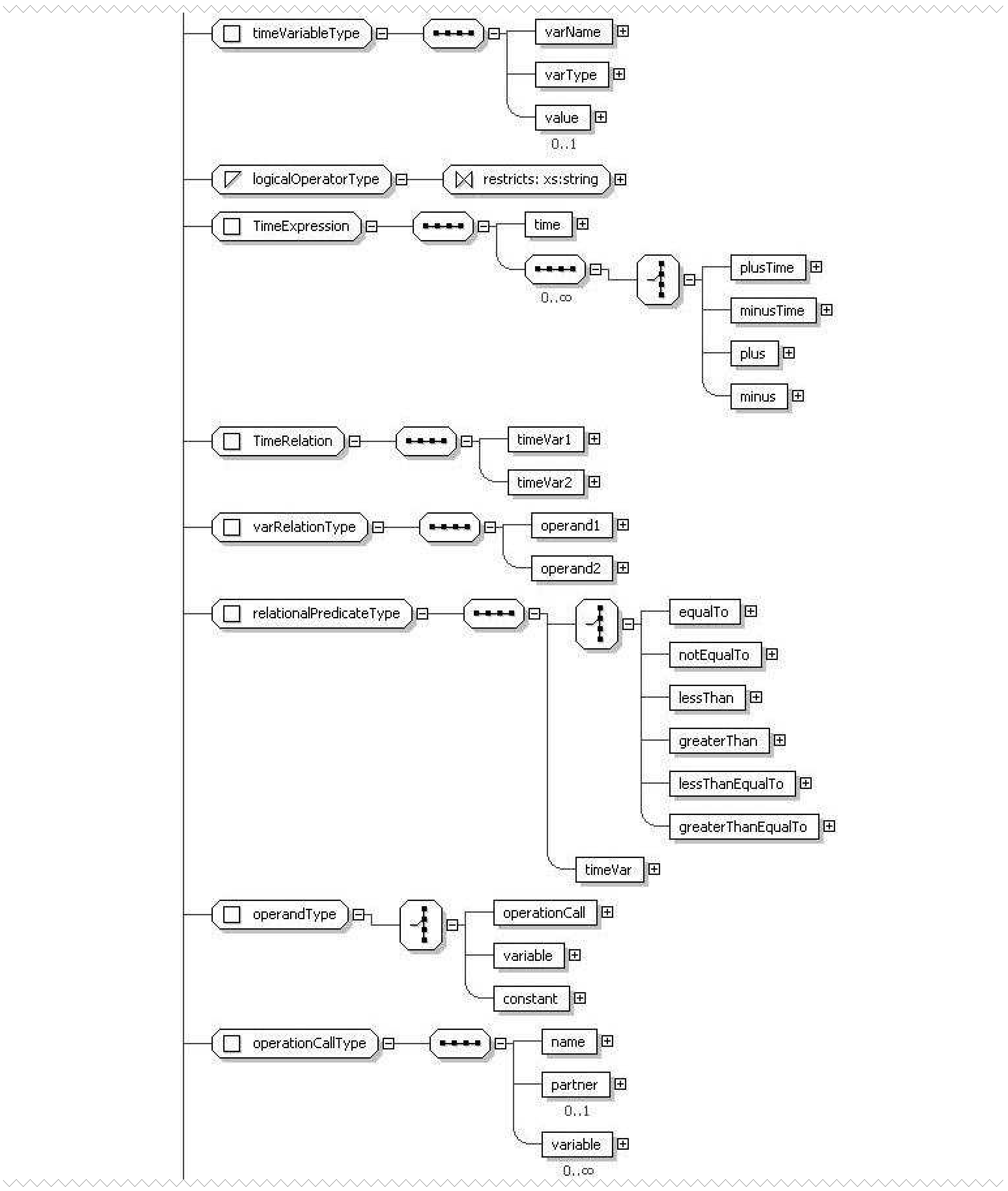

Figure 18 – XML Formula Representation Schema (III)

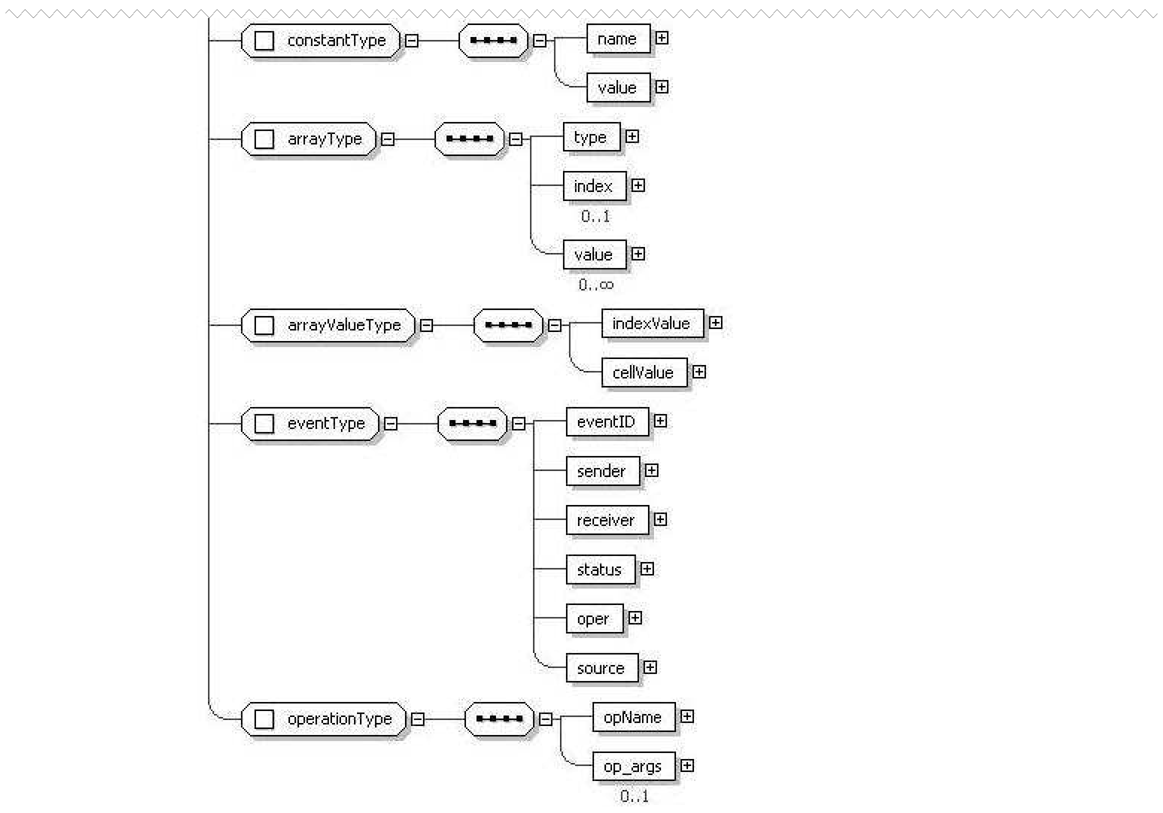

Figure 19 – XML Formula Representation Schema (and IV)

6. Examples of descriptions

In this section we present an example that is designed to provide a global vision of the modelling artefacts in practice. The example shows some related S&D Classes, S&D Patterns and S&D Implementations. We also include in a separate subsection an example of monitoring rules.

As shown in Figure 20 below, the example includes only one S&D Class *(SimpleTransmisionConfidentiality.iso.org).* Then we propose two patterns called *ConfidentialityByDES_Encryption.iso.org* and *ConfidentialityByDES_Encryption.rsa-labs.com*, which belong to the mentioned S&D Class. Each S&D Pattern provides a description allowing an automatic mechanism to make the transformation from the interface declared in the S&D Class to the native interface provided by the S&D Pattern. The interface declared by an S&D Implementation must realize (match exactly) the interface provided by the corresponding S&D Pattern.

In this example, there are three S&D Implementations. Two of them *(UMA_Crypt.gisum.uma.es* and *TPMDES.infieon.com)* are implementations of the *ConfidentialityByDES_Encryption.iso.org* pattern, while the last one (*CryptoJ_BSafeDES.rsa.com*) realizes the *ConfidentialityByDES_Encryption.rsa-labs.com* pattern.

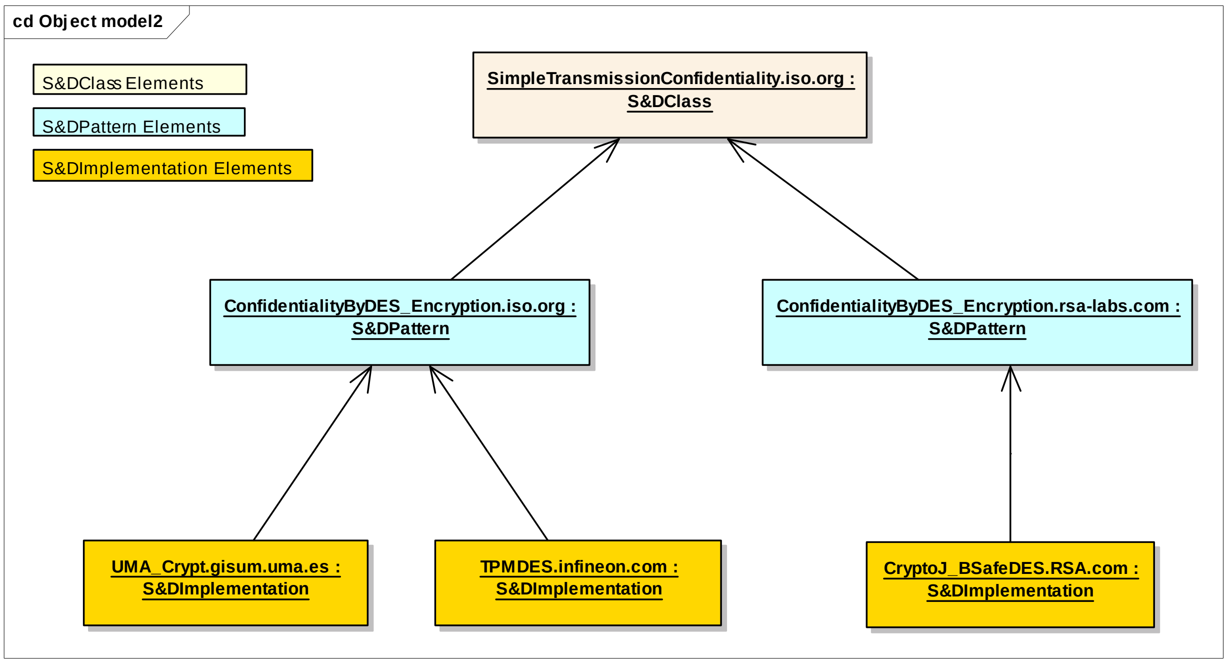
cd Object model2
S&DClass Elements
S&DPattern Elements
S&DImplementation Elements
SimpleTransmissionConfidentiality.iso.org : S&DClass
ConfidentialityByDES_Encryption.iso.org : S&DPattern
ConfidentialityByDES_Encryption.rsa-labs.com : S&DPattern
UMA_Crypt.gisum.uma.es : S&DImplementation
TPMDES.infineon.com : S&DImplementation
CryptoJ_BSafeDES.RSA.com : S&DImplementation

Figure 20 – Relation between the elements in the example

6.1.1. Confidential Transmission

6.1.1.1. S&D Class: SimpleTransmissionConfidentiality.iso.org

S&D Class: SimpleTransmissionConfidentiality.iso.org	
1	**Creator**
	Name: iso.org **Date:** 2007-05-04
2	**Timestamp:** 1178307611
3	**TrustMechanisms:** *...signature...*
4	**Provided Properties**
	Property **ID:** TransmissionConfidentiality.iso.org **Timestamp:** 1146771611
5	**SolutionFeatures**
	Feature: Shared key
6	**Interface**

	Calls
	SendConfidential(Conf_data:raw; Recipient:raw)
	ReceiveConfidential(Conf_data:raw; Sender: raw)
	Sequence
	Sender.SendConfidential(x1,Receiver)
	Receiver.ReceiveConfidential(x1,Sender)

7	**Roles**
	Role **RoleName:** Sender **Functionality** **CallName:** SendConfidential **CallName:** ReceiveConfidential **Role** **RoleName:** Receiver **Functionality** **CallName:** SendConfidential **CallName:** ReceiveConfidential
8	**Comments** The sender starts the transmission.,encrypts some data and sends it The receiver waits for the sender to send the data. After reception, decrypts the data.

Table 13 – Definition of S&D Class SimpleTransmissionConfidentiality

This is a very simple class that shows the simplest interface for confidential communication, composed of two calls:

- SendConfidential(Conf_data:raw; Recipient:raw); and
- ReceiveConfidential(Conf_data:raw; Sender: raw);

It also shows that it is possible to specify the correct sequence of calls.

6.1.1.2. S&D Pattern: TransmissionConfidentialityByDES_Encryption.iso.org

S&D Pattern: TransmissionConfidentialityByDES_Encryption.iso.org	
1	**Creator**
	Name: iso.org **Date:** 2007-05-07
2	**Timestamp:** 1178521503
3	**TrustMechanisms:** *…signature…*
4	**PatternFeatures**
	Feature: encryption **Feature:** DES
5	**Provided Properties**
	Property **ID:** TransmissionConfidentiality.iso.org **Timestamp:** 1146985503
6	**Interface**
	Operations **Operation:** encrypt **Definition:** **define function** encrypt **input** plainData**:**text, key:text **output** encryptedData:text #returns the plainData encrypted with the key **enddefine** **Operation:** decrypt **Definition:** **define function** decrypt **input** encryptedData:text, key:text **output** plainData:text #returns the cypheredData decrypted with the key **enddefine** **Operation:** getKey **Definition:** **define function** getKey **input** userID:text, **output** key:text #returns the key requested by the user **enddefine** **Operation:** send **Definition:** **define function** send **input** recipient:text, encryptedData:text **output** sentOK:boolean #sends some encrypted data to the user #"recipient" parameter. #returns TRUE if everything goes right. FALSE #otherwise **enddefine** **Operation:**receive **Definition:** **define function** receive **input** sender:text, encryptedData:text **output** receiveOK:boolean #receives some encrypted data from the user specified in #"sender" parameter #returns TRUE if everything goes right. FALSE otherwise **enddefine**

Table 14 – Definition of S&D Pattern Confidentiality by DES (I)

	ClassAdaptors **Class:** SimpleTransmissionConfidentiality.iso.org **Adaptor:** **define function** sendConfidential **input** data:text, recipient:text **output** sentOK:Boolean key:text encrypted:text result:boolean key = getKey [recipient] encryptedData = encrypt [data, key] result = send [encryptedData, recipient] If !result then #log the event and possible cause endif return result **enddefine** **define function** receiveConfidential **input** encryptedData:text, sender:text **output** receptionOK:Boolean key:text plainData:text result:boolean result = receive [encryptedData, sender] If result then key = getKey [sender] plainData = decrypt [encryptedData, key] else #log the event and possible cause Endif return result **enddefine**
7	**Parts**
	Part: CommunicationNetwork
8	**Parameters**
	Parameter: User_A **Parameter:** User_B **Parameter:** Key **Parameter:** Data **Parameter:** ClearTextType **Parameter:** CipherTextType **Parameter:** KeyType **Parameter:** UserIDType
9	**Pre-Conditions**
	Parameter pre-conditions **Parameter pre-condition:** *Key is known and confidential for User_A and User_B* **Solution pre-conditions** **Solution pre-condition:** …
10	**Static Tests Performed**
	Test **Conditions of test:** … **Attack models considered:** …
11	**System Configuration:** *A description based on BPEL, UML… It should include all necessary **initializations** of the parts, framework, initialization of the monitor, etc.*

Table 15 – Definition of S&D Pattern Confidentiality by DES (II)

12	**Monitoring** **Monitor:** *(constraints, or even explicit reference)* **Type:** Asynchronous **Monitoring Formulae:** **Rule-1:** **Event:**
13	**Comments: ...**

Table 16 – Definition of S&D Pattern Confidentiality by DES (and III)

The previous description corresponds to an S&D Pattern that belongs to the previous described S&D Class (SimpleTransmissionConfidentiality.iso.org). The third field of the S&D Pattern includes information about the properties that are fulfilled by the solution represented in this pattern.

Two important fields in Table 16 are the Interface and the Class Adaptor. *Interface* defines functions that this pattern provides. The *Class Adaptor* contains the rules for automatic translation between calls to S&D Class *interface* into calls to S&D Pattern *interface*.

Usually S&D Pattern interfaces are closer to solution details than S&D Class interfaces because this S&D Patterns interfaces include lower level functions. Field number three of S&D Pattern includes information about the Properties that fulfils this Pattern. The S&D Pattern shows also information about the *Parts* required. In this case, a communication network is required.

Parameters include some variable data from one instance of the pattern to other. This Pattern requires information about the transmission source and target, the key, the data and the data types used in the Parameters. At last, preconditions say that the key used in the transmission must be shared by the two principals and confidential.

6.1.1.3. S&D Implementation: UMA_Crypt.gisum.uma.es

S&DImplementation: UMA_Crypt.gisum.uma.es	
1	**Creator**
	Name: uma.es **Date:** 2007-05-09
2	**TimeStamping:** 1178535559
3	**Trust mechanisms:** *signed by rsa.com*
4	**S&DPatternReference:** TransmissionConfidentialityByDES_Encryption.iso.org
5	**Preconditions**
	Precondition: KeyType = 64_Bit_DES_Key_Type **Precondition:**:*JDK (Sun) v1.4 or later installed* **Precondition:** *Valid Platforms (WIN32, Solaris 10, RedHat 7.0)* **Precondition:**ConfidentialityByDESEncryption.iso.org/CommunicationNetwork/ access_method= TCP/IP
6	**ImplementationDecription**
	Description: *Fullfils FIPS140-2* **Description:** *Software Implemented* **Description:** *Only suitable for short-term storage keys*
7	**ImplementationReference**
	Reference: *uma-crypt.jar + Hash of the code*
8	**ComplianceProofs**
	Proof: *validated and signed by cmvp.csrc.nist.gov*
9	**Comments:...**

Table 17 – Definition of S&D Implementation UMA_Crypt.gisum.uma.es

6.1.1.4. S&D Implementation: TPMDES.infineon.com

S&DImplementation: TPMDES.infineon.com	
1	**Creator**
	Name: infineon.com **Date:** 2007-05-09
2	**TimeStamping:** 1178536658
3	**Trust mechanisms:** *signed by infineon.com*
4	**S&DPatternReference:** TransmissionConfidentialityByDES_Encryption.iso.org
5	**Preconditions**
	Precondition: KeyType = 64_Bit_DES_Key_Type **Precondition:** *TPM v1.1 or newer installed*
6	**ImplementationDecription**
	Description: *Fullfils FIPS46-3* **Description:** *Hardware + Software Implemented*
7	**ImplementationReference**
	Reference: *Infineon_TPM_Manager.exe + Hash of the code*
8	**ComplianceProofs**
	Proof: *validated and signed by iacs.cesg.gov.uk*
9	**Comments:...**

Table 18 – Definition of S&D Implementation TPMDES.infineon.com

The S&D Implementation of TransmissionConfidentialityByDES_Encryption is shown in Table 18 and named as TPMDES. Concerning preconditions two of them are declared. First precondition refers to the key needed in DES algorithm. This is a 64 bits length. Second one refers to a TPM v1.1 or higher is needed to be installed. Third point is related to the description of the implementation.

Two descriptions are declared, first one describes that this S&D Implementation fulfils with FIPS46-3 description [7]. Second description describes that this Implementation is a combination of hardware and software solution. The reference of Implementation executable file is Infineon_TPM_Manager.exe plus a Hash of the code in order to test the integrity of this executable code. Some Compliance proofs have been performed such as validated and signed by iacs.cesg.gov.uk [8]. Finally the last element refers to Trust mechanisms and describes that is signed by infineon.com [9].

6.1.2. *Confidentiality by DES*

6.1.2.1. S&D Pattern: ConfidentialityByDES_Encryption.rsa-labs.com

S&D Pattern: ConfidentialityByDES_Encryption.rsa-labs.com	
1	**Creator:**
	Name: rsa-labs.com **Date:** 2007-05-10
2	**Timestamp:** 1178537748
3	**Trust Mechanisms:** *signed by rsa-labs.com*
4	**Pattern Features**
	Feature: Confidentiality **Feature:** Encription **Feature:** DES
3	**Provided Properties**
	Property: **ID:** TransmissionConfidentiality.iso.org **Timestamp:** 20060621100230
4	**Interface**
	Operations **Operation:** Session **Definition:** **define function** Session **input** userId:userIdType **output** session: sessionType #The function receives an userId parameter and stablish a session among both #users **enddefine** **Operation:** KeyAgree **Definition:** **define function** KeyAgree **input** session: sessionType **output** key: KeyType #returns the key that sender and receiver will use to encrypt the communications **enddefine** **Operation:** SymetricCipher **Definition:** **define function** SymetricCipher **input** cleartext:ClearTextType , key: KeyType **output** ciphertext: CipherTextType #The function gets a key and a plain text and generates cipher text using that key **enddefine** **Operation:** SymetricDecipher **Definition:** **define function** SymetricDecipher **input** ciphertext: CipherTextType, key: KeyType **output** cleartext:ClearTextType # This is the reverse function above, it gets a key and an encrypted text and gets #the plain text ciphered before **enddefine**

Table 19 – Definition of S&D Pattern ConfidentialityByDES_Encryption (I)

```
        Operation:  send
        Definition:
            define function send
            input encryptedData:text, recipient:text
            output sentOK:boolean
            #sends some encrypted data to the user
            #"recipient" parameter.
            #returns TRUE if everything goes right. FALSE
            #otherwise
            enddefine
        Operation : receive
        Definition:
            define function  receive
            input sender:text, encryptedData:text
            output receiveOK:boolean
            #receives some encrypted data from the user specified in
            #"sender" parameter
            #returns TRUE if everything goes right. FALSE otherwise
        Enddefine

ClassAdaptors
        Class: SimpleTransmissionConfidentiality.iso.org
        Adaptor:
            define function sendConfidential
            input data:text, recipient:text
            output sentOK:Boolean
               s:long
               key:text
               msg:text
               result:boolean
               s = Session[recipient]
               key = KeyAgree[s]
               msg = SymmetricCipher[data, key]
               result = Send[msg, s]
               If !result then
                 #log the event and possible cause
               endif
               return result
            enddefine
            define function sendConfidential
            input data:text, sender:text
            output sentOK:Boolean
               s:long
               key: text
               msg: text
               plainText: text
               result: boolean
               s = Session[sender]
               result = receive [msg, sender]
               If !result then
                 #log the event and possible cause
               else
                 key = KeyAgree [sender]
                 plainText = SymetricCipher [msg, k]
               endif
               return result

            enddefine
```

Table 20 – Definition of S&D Pattern ConfidentialityByDES_Encryption (II)

6	**Parts**
	Part: CommunicationNetwork
7	**Parameters**
	Parameter: User_A **Parameter:** User_B **Parameter:** Key **Parameter:** Data **Parameter:** ClearTextType **Parameter:** CipherTextType **Parameter:** KeyType **Parameter:** UserIDType
8	**Pre-Conditions**
	Parameter pre-conditions **Parameter pre-condition:** *Key is agreed once a session is started between the principals, User_A and User_B* **Solution pre-conditions** **Solution pre-condition:** …
9	**Static Tests Performed**
	Test: … **Conditions of test:** **Attack models considered:**
10	**System Configuration:** *A description based on BPEL, UML... It should include all necessary initializations of the parts, framework, initialization of the monitor, etc.*
11	**Monitoring**
	Monitor: *(constraints, or even explicit reference)* **Type:** Asynchronous **Monitoring Formulae:** **Rule-1:** **event:**
13	**Comments: ...**

Table 21 – Definition of S&D Pattern ConfidentialityByDES_Encryption (and III)

The encryption service described by this pattern is aimed to protect data that is sent between hosts across a network. Encryption services, such as DES [10], use a reversible algorithm to convert plain-text data into an unintelligible form, thus protecting data from being used by unauthorized parties, providing confidentiality for hosts.

An acronym for Data Encryption Standard, DES was developed by IBM. The algorithm expands a single message by up to 8 bytes. DES is a block cipher that encrypts data in blocks of 64 bits by using a 56-bit key. Using this algorithm, this pattern provides *Transmission Confidentiality* conform the ISO standard.

The interface of the pattern provides the following calls:

- JSAFE_Session(userID: UserIDType): it starts a session between the two principals. As specified in the pattern *preconditions*, after starting the session a shared key has to be agreed.
- JSAFE_KeyAgree(key: KeyType, userID: UserIDType): once the session is started, the principals can agree on the key they will use to encrypt/decrypt the data
- JSAFE_SymetricCipher(in cleartext:ClearTextType; in key: KeyType; out ciphertext: CipherTextType): it takes the clear text as input and gives the cipher text as output. It uses the key given by *JSAFE_KeyAgree* to encrypt/decrypt the data

- JSAFE_SymetricCipher(in ciphertext: CipherTextType; in key: KeyType; out cleartext:ClearTextType): it generates the clear text from the cipher text

Apart from the cryptographic functions, this S&D Pattern includes two calls focused on communication between two principals. Both *send* and *receive* functions are provided by the *Communication Network* Part, given that the implementation of the Pattern deals with the encryption algorithm and not with the underlying network.

- Send(data: CipherTextType; recipient: UserIDType): it sends the cipher text to the specified recipient
- Receive(data: CipherTextType; Sender: UserIDType): it prepares the recipient to receive the cipher text from the sender

The *Class Adaptor* gives the exact sequence of calls to follow in order to correctly execute *SendConfidential* and *ReceiveConfidential* functions. The parameters specified in these calls are:

- User_A: the sender
- User_B: the recipient
- Key: the key used to encrypt/decrypt the data
- Data: the information exchanged between the users
- ClearTextType: data in clear
- CipherTextType: cipher data
- KeyType: the type of the key, including the key length, validity period, etc.
- UserIDType: the format used to identify the user

6.1.2.2. S&D Implementation: CryptoJ_BSafeDES.RSA.com

S&DImplementation: CryptoJ_BSafeDES.RSA.com	
1	**Creator**
	Name: RSA.com **Date:** 2007-05-11
2	**Timestamping:** 1178536658
3	**Trust mechanisms:** *signed by rsa.com*
4	**S&DPatternReference:** ConfidentialityByDES_Encryption.rsa-labs.com
5	**Preconditions**
	Precondition: KeyType = 64_Bit_DES_Key_Type **Precondition:** *JDK (Sun, HP, IBM) v1.1 or later installed* **Precondition:** *Valid Platforms (WIN,Solaris,HP-UX,RedHat,AIX)* **Precondition:** ConfidentialityByDESEncryption.rsa-labs.com/CommunicationNetwork/access_method= TCP/IP
6	**ImplementationDecription**
	Description: *Fullfils FIPS140-2* **Description:** *Software Implemented* **Description:** *Only suitable for short-term storage keys*
7	**ImplementationReference**
	Reference: *jsafeCEFIPS.jar + Hash of the code* **Reference:** *jsafeFIPS.jar + Hash of the code*
8	**ComplianceProofs**
	Proof: *validated and signed by cmvp.csrc.nist.gov*
9	**Comments:...**

Table 22 – Definition of S&D Implementation CryptoJ_BSafeDES.RSA.com

This *S&D Implementation* refers to the *ConfidentialityByDES_Encryption.rsa-labs.com S&D Pattern*. RSA BSAFE cryptography products [6] are designed to allow state-of-the-art privacy and authentication features to be built into virtually any application for optimized performance. The RSA BSAFE Crypto J Toolkit Module versions 3.5 and 3.5.2 (Crypto J Module) is a non proprietary cryptographic module. It includes a wide range of data encryption and signing algorithms, including DES, Triple-DES, the high-performing RC5, the RSA Public Key Cryptosystem, and more.

The Crypto J Module is software implemented and meets the security requirements of FIPS 140-2. The distribution includes two API interfaces, described in Table 22 as *ImplementationReference*:

- jsafeFIPS.jar JSAFE Application Programmer Interface to the Crypto J Module
- jsafeJCEFIPS.jar JCE Application Programmer Interface to the Crypto J Module.

FIPS 140-2[2] (Federal Information Processing Standards Publication 140-2 – Security Requirements for Cryptographic Modules) details the U.S. Government requirements for cryptographic modules.

As *preconditions*, the Crypto J module requires JDK running on the target device and is only valid for the platforms described in the table above. The DES algorithm for Crypto J requires at least a 64 bit key and is only valid if TCP/IP is the network communication protocol used on the device.

[2] More information about the FIPS 140-2 standard and validation program is available on the NIST website http://csrc.nist.gov/cryptval/.

6.2. S&D Patterns expressed in XML: an example

The usage of patterns within the Serenity framework changes the traditional view of “patterns” from the software engineering perspective. Since Serenity patterns are not directly related with design patterns but with concrete, ready-to-apply solutions, the connection between Serenity patterns and both software and hardware components is tight and quite common. For this reason we needed a new and common syntax, resulting in the definition of a new XML-based language. We chose the definition of some easy-to-understand tags in order to represent all the information described in section 4.3. . The selection of XML as the metalanguage for defining the S&D patterns takes advantage of its ability to perform data migration tasks in an easy and flexible way.

```
<?xml version="1.0" encoding="UTF-8"?>
<SandDPattern xmlns:ns1="http://tempuri.org/ec/formula"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="S%26Dpatterns_monitoringRules.xsd"
  name="TransmissionConfidentialityByDES_Encryption.iso.org">
  <creator>
    <name>iso.org</name>
    <date>2007-05-07</date>
  </creator>
  <timestamping>1178521503</timestamping>
  <trustMechanisms>
    <signatureType>http://www.w3.org/2000/09/xmldsig#sha1</signatureType>
    <signer>iso.org</signer>
    <signature>j6lwx3rvEPO0vKtMup4NbeVu8nk=</signature>
  </trustMechanisms>
  <patternFeatures>
    <feature>confidentiality</feature>
    <feature>encryption</feature>
    <feature>DES</feature>
  </patternFeatures>
  <providedProperties>
    <property>
      <id>TransmissionConfidentiality.iso.org</id>
      <timestamp>1146985503</timestamp>
    </property>
  </providedProperties>
  <interface>
    <operations>
      <operation name="encrypt">
        <definition>
          define function encrypt
          input plainData:text, key:text
          output encryptedData:text
          #returns the plainData encrypted with the key
          enddefine
        </definition>
      </operation>
      <operation name="decrypt">
        <definition>
          define function decrypt
          input encryptedData:text, key:text
          output plainData:text
          #returns the cypheredData decrypted with the key
          enddefine
        </definition>
      </operation>
```

Table 23 – Definition of an S&D Pattern in XML language (I)

```
<?xml version="1.0" encoding="UTF-8"?>
<SandDPattern xmlns:ns1="http://tempuri.org/ec/formula"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="S%26Dpatterns_monitoringRules.xsd"
  name="TransmissionConfidentialityByDES_Encryption.iso.org">
  <creator>
    <name>iso.org</name>
    <date>2007-05-07</date>
  </creator>
  <timestamping>1178521503</timestamping>
  <trustMechanisms>
    <signatureType>http://www.w3.org/2000/09/xmldsig#sha1</signatureType>
    <signer>iso.org</signer>
    <signature>j6lwx3rvEPO0vKtMup4NbeVu8nk=</signature>
  </trustMechanisms>
  <patternFeatures>
    <feature>confidentiality</feature>
    <feature>encryption</feature>
    <feature>DES</feature>
  </patternFeatures>
  <providedProperties>
    <property>
      <id>TransmissionConfidentiality.iso.org</id>
      <timestamp>1146985503</timestamp>
    </property>
  </providedProperties>
  <interface>
    <operations>
      <operation name="encrypt">
        <definition>
          define function encrypt
          input plainData:text, key:text
          output encryptedData:text
          #returns the plainData encrypted with the key
          enddefine
        </definition>
      </operation>
      <operation name="decrypt">
        <definition>
          define function decrypt
          input encryptedData:text, key:text
          output plainData:text
          #returns the cypheredData decrypted with the key
          enddefine
        </definition>
      </operation>
```

Table 24 – Definition of an S&D Pattern in XML language (II)

```
    <parts>
        <part id="id3" url="http://localhost" type="CommunicationNetwork"/>
    </parts>
    <parameters>
        <parameter>User_A</parameter>
        <parameter>User_B</parameter>
        <parameter>Data</parameter>
        <parameter>ClearTextType</parameter>
        <parameter>CipherTextType</parameter>
        <parameter>KeyType</parameter>
        <parameter>UserIDType</parameter>
    </parameters>
    <preconditions>
        <parametersPrecondition>
            <parameterPrecondition>Key is known and confidential for User_A and User_B</parameterPrecondition>
        </parametersPrecondition>
        <solutionsPreconditions>
            <solutionPrecondition> </solutionPrecondition>
        </solutionsPreconditions>
    </preconditions>
    <staticTestsPerformed>
        <test name="TestName">
            <conditionsTest>conditionsTest0</conditionsTest>
            <attackModels>attackModels0</attackModels>
        </test>
    </staticTestsPerformed>
    <systemConfiguration description="description0"/>
    <monitoring>
        <monitor>
            <localization>localization0</localization>
            <type>type3</type>
            <inicialization>inicialization0</inicialization>
        </monitor>
    </monitoring>
    <comments>comments0</comments>
</SandDPattern>
```

Table 25 – Definition of an S&D Pattern in XML language (and III)

6.3. Monitoring rules expressed in XML: an example

In this section, we consider the pattern for a Mechanism for Optimistic Fair Exchange with Trusted Third Party (TTP), which is described in section 7. of this deliverable, and give an example of a rule that can be monitored for this pattern. The monitoring rule is derived from the requirement that TTP must be available. It should consider two cases, i.e. the case when Alice tries to communicate with TTP and the case when Bob tries to communicate with TTP. Therefore, two rules are required. We illustrate the second case, more specifically: if Bob sends a “solve” message to TTP, then TTP should respond with “send_item” message within some time limit (t1+tu where t1 is the time when Bob sent the “solve” message). In event calculus we express this as follows:

```
∀ _eID1, _eID2, Bob_ID, _TTP_ID: String; t1, t2:Time
  Happens(e(_eID1,Bob_ID,TTP_ID,REQ-B,solve((Item_A)Ka1,Item_B)),Bob_ID),t1,  ℜ(t1,t1))
  ⇒
  Happens(e(_eID2,TTP_ID,Bob_ID,RES-A,send_item(((Item_A)Ka1)Ka2),t2, ℜ(t1,t1+tu)
```

Table 26 – Event Calculus example for the Fair Exchange example

The XML document that describes the above mentioned monitoring rule is given in Table 27. Firstly, the quantification of the variables in the formula is represented in lines 7-32. Two types of variables are quantified, namely regular variables (any variable except for time variables) and time variables. Next, the body of the formula is represented in lines 33-78, i.e. the expression on the RHS of the implication. The body consists of the **Happens** predicate and its arguments, i.e. an event, a time variable and a time range. The event is represented in lines 36-55.

The time variable has been specified in lines 56-59 and the time range in lines 60-75. Finally, the head of the formula (i.e. the expression on the LHS of the implication) is represented in lines 79-120. This also consists of a **Happens** predicate with an event, a time variable and a time range, and thus is represented similarly to the body of the formula.

```
<?xml version="1.0" encoding="UTF-8"?>
<formulas xmlns="http://tempuri.org/ec/formula" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://tempuri.org/ec/formula file:/Z:/Serenity/A5%20contribution%20-%20September06/EC-
Assertion6.xsd" formulaId="">
	<quantification>
		<quantifier>universal</quantifier>
		<regularVariable>
			<varName>Bob_ID</varName>
			<varType>String</varType>
		</regularVariable>
		<regularVariable>
			<varName>TTP_ID</varName>
			<varType>String</varType>
		</regularVariable>
		<regularVariable>
			<varName>_eID1</varName>
			<varType>String</varType>
		</regularVariable>
		<regularVariable>
			<varName>_eID2</varName>
			<varType>String</varType>
		</regularVariable>
		<timeVariable>
			<varName>t1</varName>
			<varType>Time</varType>
		</timeVariable>
		<timeVariable>
			<varName>t2</varName>
			<varType>Time</varType>
		</timeVariable>
	</quantification>
	<body>
		<predicate>
			<happens>
				<event>
					<eventID>_eID2</eventID>
					<sender>
						<varName>TTP_ID</varName>
						<varType>String</varType>
					</sender>
					<receiver>
						<varName>Bob_ID</varName>
						<varType>String</varType>
					</receiver>
					<status>RES-A</status>
					<oper>
						<opName>send_item</opName>
						<op_args>
							<varName>(((Item_A)Ka1)Ka2)</varName>
							<varType>String</varType>
						</op_args>
					</oper>
					<source>TTP_ID</source>
				</event>
				<timeVar>
					<varName>t2</varName>
					<varType>Time</varType>

				</timeVar>

				<fromTime>
					<time>
						<varName>t1</varName>
						<varType>Time</varType>
					</time>
				</fromTime>
```

Table 27 – XML document representing the rule that checks the availability of TTP (I)

```
                        <toTime>
                                <time>
                                        <varName>t1</varName>
                                        <varType>Time</varType>
                                </time>
                                <plusTime>
                                        <varName>tu</varName>
                                        <varType>Time</varType>
                                </plusTime>
                        </toTime>
                </happens>
        </predicate>
</body>
<head>
        <predicate>
                <happens>
                        <event>
                                <eventID>_eID1</eventID>
                                <sender>
                                        <varName>Bob_ID</varName>
                                        <varType>String</varType>
                                </sender>
                                <receiver>
                                        <varName>TTP_ID</varName>
                                        <varType>String</varType>
                                </receiver>
                                <status>REQ-B</status>
                                <oper>
                                        <opName>Solve</opName>
                                        <op_args>
                                                <varName>((Item_A)Ka1,Item_B))</varName>
                                                <varType>String</varType>
                                        </op_args>
                                </oper>
                                <source>Bob_ID</source>
                        </event>
                        <timeVar>
                                <varName>t1</varName>
                                <varType>Time</varType>
                        </timeVar>
                        <fromTime>
                                <time>
                                        <varName>t1</varName>
                                        <varType>Time</varType>
                                </time>
                        </fromTime>
                        <toTime>
                                <time>
                                        <varName>t1</varName>
                                        <varType>Time</varType>
                                </time>
                        </toTime>
                </happens>
        </predicate>
</head>
</formulas>
```

Table 28 – XML document representing the rule that checks the availability of TTP (and II)

7. Applying the language

7.1. A Pattern for Fair Exchange

A complete scenario with the guidelines for specifying of S&D Solutions and the construction of Artefacts using this can be found in section 7 in [17]. Based on this scenario, what follows is a revision of the previous work.

Intuitively, a fair exchange mechanism allows two parties to exchange items in a fair way, so that either each party gets the other's item, or neither party does. A typical way to solve the fair exchange problem is to introduce a semi-trusted arbitrator (Charlie) to the model. Alice will first register her key with Charlie. This registration is performed only once and, as a result, Charlie may possibly learn some part of Alice's secret. Upon the completion of the one-time registration process, Alice can perform many fair exchanges with different merchants.

In any such exchange, Alice and Bob want to exchange two pieces of information σ and τ.

Alice first issues some verifiable "partial signature" σ' to Bob. Bob verifies the validity of this partial signature and fulfils his obligation by sending Alice the required information τ, after which Alice sends her "full signature" σ to complete the transaction. Thus, if no problem occurs, Charlie does not participate in the protocol (such protocols are called optimistic). However, if Alice refuses to send her full signature σ at the end, Bob will send σ' to Charlie (and a proof of fulfilling his obligation, including the information *I* that should be sent to Alice), and Charlie will convert σ' into σ, sending σ to Bob and *I* to Alice. Informally, we wish to achieve the following security guarantees:

— Alice should not be able to produce a valid partial signature σ' which Charlie cannot convert into a full signature σ.

— Bob should not be able to produce a valid partial signature σ' which he did not get from Alice.

— Bob should not be able to produce a valid full signature σ which he did not get from Alice (or Charlie provided Bob possesses σ').

— Charlie should not be able to produce a valid full signature σ without seeing a valid partial signature σ' computed by Alice.

While the first three properties are clearly important to prevent parties from cheating, the last property is equally crucial: we do not want the arbitrator Charlie to make signatures without Alice's consent. Indeed, otherwise Charlie would have to be completely trusted. Moreover, if one is willing to have a completely trusted arbitrator, then the problem becomes technically trivial, and no elaborate protocols are needed at all: Alice may use any signature scheme and simply give Charlie her entire secret key during registration. Figure 21 represents the whole process as a collaboration diagram.

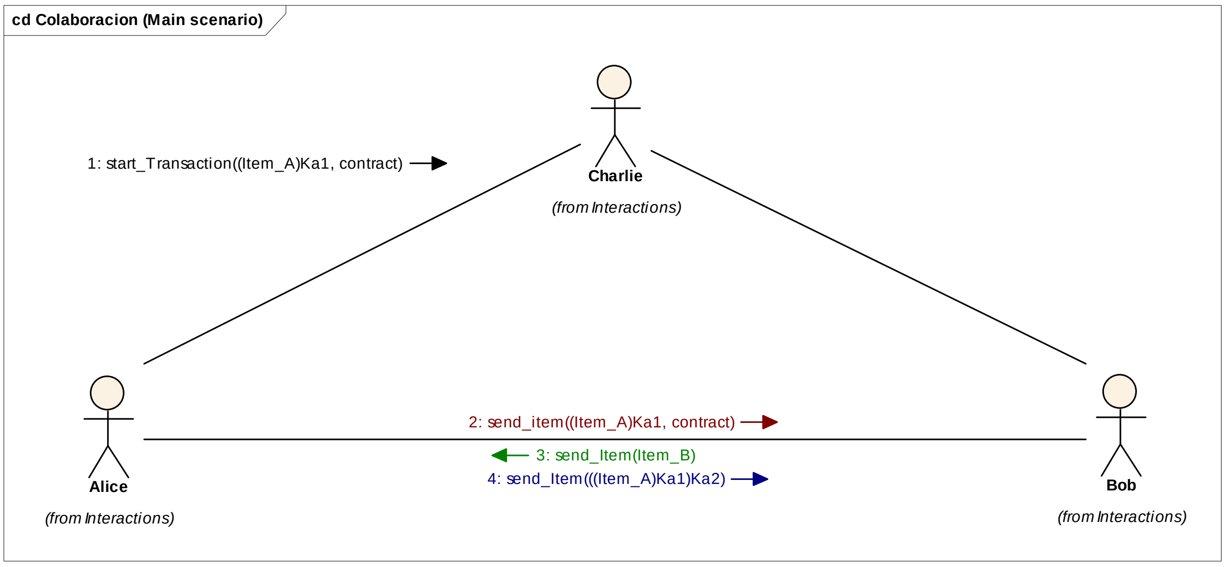
cd Colaboracion (Main scenario)
1: start_Transaction((Item_A)Ka1, contract)
Charlie
(from Interactions)
Alice
(from Interactions)
Bob
(from Interactions)
2: send_item((Item_A)Ka1, contract)
3: send_Item(Item_B)
4: send_Item(((Item_A)Ka1)Ka2)

Figure 21 – Collaboration diagram of Fair Exchange protocol

Next Sequence Diagram (Figure 22) is an extension of the previous collaboration diagram. It represents the case in which Alice tries to cheat Bob. Bob is waiting the reception of the item until a timeout exception is triggered. Bob sends a request to the arbitrator (*solve request*), and Charlie (as arbitrator) sends back the Item to Bob, after checking Bob's request.

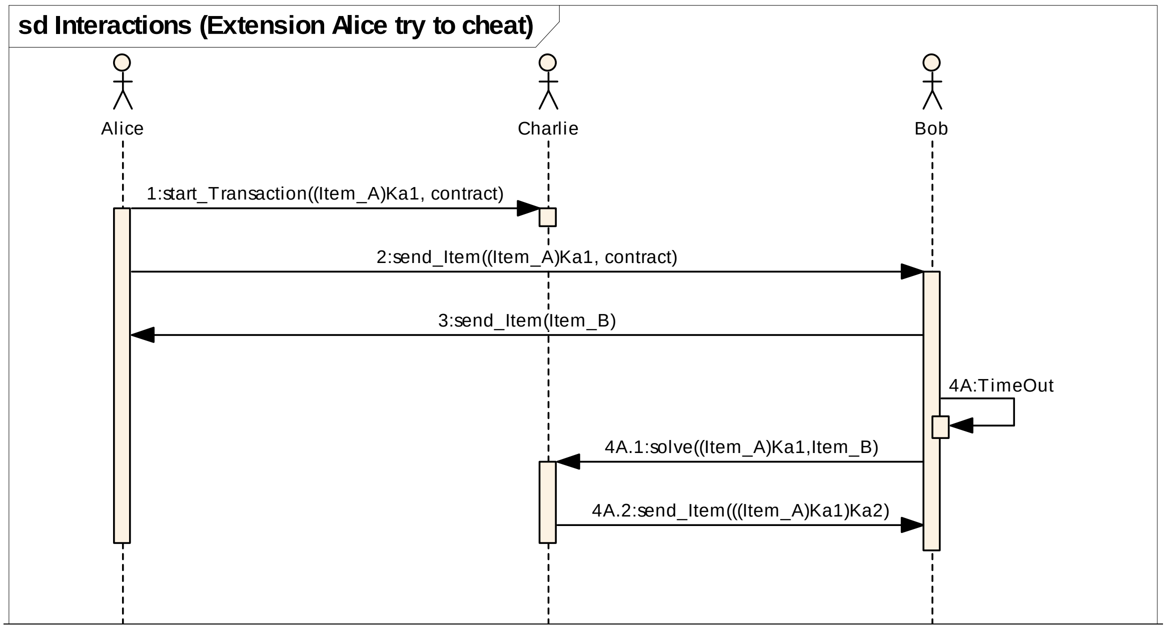
sd Interactions (Extension Alice try to cheat)
Alice
Charlie
Bob
1:start_Transaction((Item_A)Ka1, contract)
2:send_Item((Item_A)Ka1, contract)
3:send_Item(Item_B)
4A:TimeOut
4A.1:solve((Item_A)Ka1,Item_B)
4A.2:send_Item(((Item_A)Ka1)Ka2)

Figure 22 – Sequence diagram of Fair Exchange: Alice tries to cheat

Figure 23 represents Bob trying to cheat Alice. Bob receives Alice's item but he does not sent his item in response. Alice asks Charlie (Trusted Third Party) to solve the situation so that Charlie ends up sending Alice the un-received item.

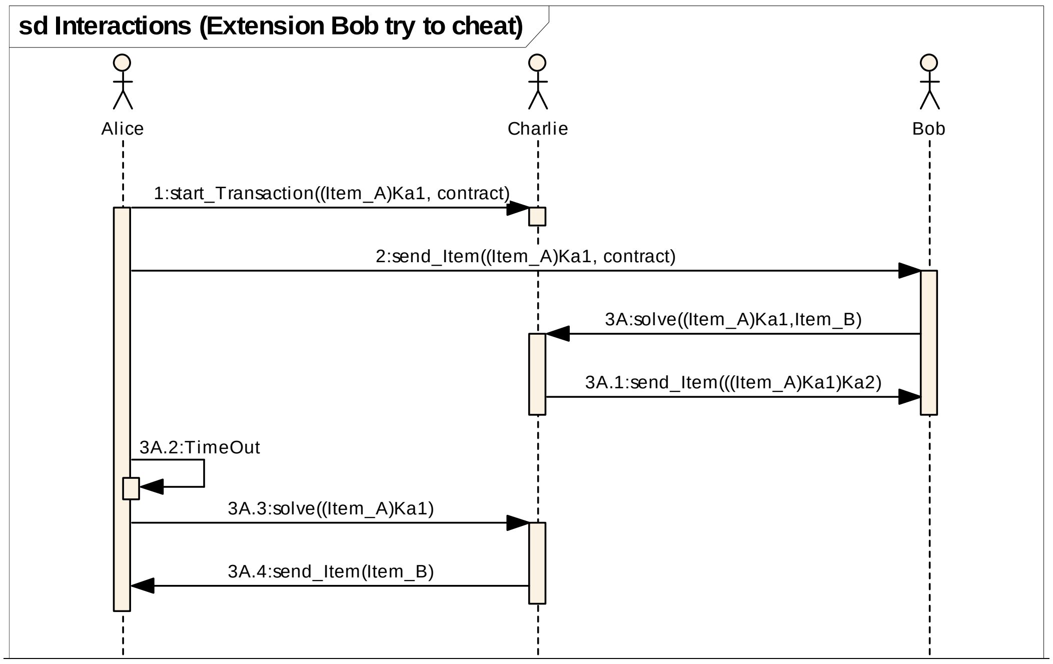
sd Interactions (Extension Bob try to cheat)
Alice
Charlie
Bob
1:start_Transaction((Item_A)Ka1, contract)
2:send_Item((Item_A)Ka1, contract)
3A:solve((Item_A)Ka1,Item_B)
3A.1:send_Item(((Item_A)Ka1)Ka2)
3A.2:TimeOut
3A.3:solve((Item_A)Ka1)
3A.4:send_Item(Item_B)

Figure 23 – Sequence diagram of Fair Exchange: Bob tries to cheat

7.2. Pattern Description Example

In this section an example of pattern description is presented. This example consists on a Mechanism for Optimistic Fair Exchange with Trusted Third Party, expressed as an S&D Pattern. This means that for instance, the digital signature operations are embedded into the fair exchange mechanism. Note that it does provide fair exchange but not confidentiality.

S&D Pattern: TTPOptimisticFairExchange.acme.com	
1	**Creator:**
	Name: acme.com
	Date:2007-05-20
2	**TimeStamp:** 1178676437
3	**Trust Mechanisms:** *signed by acme.com*
4	**Pattern Features**
	Feature:...
5	**Provided Properties**
	Property:
	ID: fair_exchange.acme.com
	Timestamp: 20060621100230
6	**Interface**
	Operations
	Operation:...
	Definition:...
	ClassAdaptors
	Class:...
	Adaptor:..
7	**Parts**
	Part: TTP
	Part: CommunicationNetwork
8	**Parameters:**
	Parameter: User_A
	Parameter: User_B
	Parameter: Item_A
	Parameter: Item_B
	Parameter: Contract
9	**Pre-Conditions:**
	Parameter pre-conditions:
	Parameter pre-condition: *User_A is registered with TTP (has a partial signature key...)*
	Parameter pre-condition: *User_B recognises TTP (has the public key of TTP...)*
	Parameter pre-condition: *User_B has the public key of User_A*
	Solution pre-conditions:
	Solution pre-condition: *The validity of Item_A can be verified with the contents of Contract*
	Solution pre-condition: *The validity of Item_B can be verified with the contents of Contract*
10	**Static Tests Performed:**
	Test: APA-Based_FormalTest.sit.fraunhofer.de
	Conditions of test:
	Attack models considered:
	Test: SDL-Based_FormalTest.lcc.uma.es
	Conditions of test:
	Attack models considered:

11	**System Configuration:** *A description based on BPEL, UML… It should include all necessary **initializations** of the parts, framework, initialization of the monitor, etc.*
12	**Monitoring:**
	Monitor *(constraints, or even explicit reference)* **Location:** localhost/SERENITY/async-mon **Type:** Asynchronous **Monitoring Formulae** **Rule-1:** *TTP registers contract* **event:** Registered contract: *intercepted from TTP* **Rule-2:** *TTP is available* **event:** TTP available: *requested from TTP*
13	**Comments:…**

Table 29 – S&D Pattern definition for TTP example

Appendix A. XML Schemas

A.1. XML Schema of S&D Classes

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <!-- ************************************************************ -->
        <xsd:element name="SandDClass">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element name="creator" type="creatorType"/>
                                <xsd:element name="timestamping" type="xsd:long"/>
                                <xsd:element name="trustMechanisms" type="trustMechanismsType" maxOccurs="unbounded"/>
                                <xsd:element name="providedProperties" type="propertiesType"/>
                                <xsd:element name="solutionFeatures" type="solutionFeaturesType"/>
                                <xsd:element name="interface" type="interfaceType"/>
                                <xsd:element name="roles" type="rolesType"/>
                                <xsd:element name="comments" type="xsd:string"/>
                        </xsd:sequence>
                        <xsd:attribute name="name" type="xsd:string" use="required"/>
                </xsd:complexType>
        </xsd:element>
        <!-- ************************************************************ -->
        <xsd:complexType name="creatorType">
                <xsd:sequence>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:element name="date" type="xsd:string"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ************************************************************ -->
        <xsd:complexType name="trustMechanismsType">
                <xsd:sequence>
                        <xsd:element name="signatureType" type="xsd:string"/>
                        <xsd:element name="signer" type="xsd:string"/>
                        <xsd:element name="signature" type="xsd:string"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ************************************************************ -->
        <xsd:complexType name="propertiesType">
                <xsd:sequence>
                        <xsd:element name="property" type="propertyType" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="propertyType">
                <xsd:sequence>
                        <xsd:element name="id" type="xsd:string"/>
                        <xsd:element name="timestamp" type="xsd:string"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ************************************************************ -->
        <xsd:complexType name="solutionFeaturesType">
                <xsd:sequence>
                        <xsd:element name="feature" type="xsd:string"/>
                </xsd:sequence>
        </xsd:complexType>

        <!-- ************************************************************ -->
        <xsd:complexType name="interfaceType">
                <xsd:sequence>
                        <xsd:element name="calls" type="callsType"/>
                        <xsd:element name="sequence" type="sequenceType"/>
                        <xsd:element name="constraint" type="xsd:string"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ************************************************************ -->
```

Table 30 – XML Schema proposal for S&D Classes (I)

```
        <!-- ***************************************************************** -->
        <xsd:complexType name="callsType">
                <xsd:sequence>
                        <xsd:element name="call" type="xsd:string" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ***************************************************************** -->
        <xsd:complexType name="sequenceType">
                <xsd:sequence>
                        <xsd:element name="step" type="xsd:string" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ***************************************************************** -->
        <xsd:complexType name="rolesType">
                <xsd:sequence>
                        <xsd:element name="role" type="roleType" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ***************************************************************** -->
        <xsd:complexType name="roleType">
                <xsd:sequence>
                        <xsd:element name="roleName" type="xsd:string" />
                        <xsd:element name="functionality" type="functionalityType" />
                </xsd:sequence>
        </xsd:complexType>
        <!-- ***************************************************************** -->
        <xsd:complexType name="functionalityType">
                <xsd:sequence>
                        <xsd:element name="functionName" type="xsd:string" maxOccurs="unbounded" />
                </xsd:sequence>
        </xsd:complexType>
        <!-- ***************************************************************** -->

</xsd:schema>
```

Table 31 – XML Schema proposal for S&D Classes (and II)

A.2. XML Schema of S&D Patterns

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:MonitoringRule="http://tempuri.org/ec/formula"
elementFormDefault="qualified">
        <xsd:import namespace="http://tempuri.org/ec/formula"
schemaLocation="http://www.lcc.uma.es/gimena/Schemas/MonitoringRules.xsd" id="MonitoringRule"/>
        <xsd:element name="SandDPattern">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element name="creator" type="creatorType"/>
                                <xsd:element name="timestamping" type="xsd:long"/>
                                <xsd:element name="trustMechanisms" type="trustMechanismsType"/>
                                <xsd:element name="patternFeatures" type="patternFeaturesType"/>
                                <xsd:element name="providedProperties" type="propertiesType"/>
                                <xsd:element name="interface" type="interfaceType"/>
                                <xsd:element name="parts" type="partsType"/>
                                <xsd:element name="parameters" type="parametersType"/>
                                <xsd:element name="preconditions" type="preconditionsType"/>
                                <xsd:element name="staticTestsPerformed" type="staticTestsPerformedType"/>
                                <xsd:element name="systemConfiguration" type="systemConfigurationType"/>
                                <xsd:element name="monitoring" type="monitoringType"/>
                                <xsd:element name="comments" type="xsd:string"/>
                        </xsd:sequence>
                        <xsd:attribute name="name" type="xsd:string" use="required"/>
                </xsd:complexType>
        </xsd:element>
        <!-- ************************************************************************ -->
        <xsd:complexType name="creatorType">
                <xsd:sequence>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:element name="date" type="xsd:string"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ************************************************************************ -->
        <xsd:complexType name="trustMechanismsType">
                <xsd:sequence>
                        <xsd:element name="signatureType" type="xsd:string"/>
                        <xsd:element name="signer" type="xsd:string"/>
                        <xsd:element name="signature" type="xsd:string"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ************************************************************************ -->
        <xsd:complexType name="patternFeaturesType">
                <xsd:sequence>
                        <xsd:element name="feature" type="xsd:string" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ************************************************************************ -->
        <xsd:complexType name="propertiesType">
                <xsd:sequence>
                        <xsd:element name="property" type="propertyType" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="propertyType">
                <xsd:sequence>
                        <xsd:element name="id" type="xsd:string"/>
                        <xsd:element name="timestamp" type="xsd:string"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ************************************************************************ -->
```

Table 32 – XML Schema proposal for S&D Patterns (I)

```
<xsd:complexType name="interfaceType">
        <xsd:sequence>
                <xsd:element name="operations" type="operationsType" />
                <xsd:element name="interfaceAdaptors" type="interfaceAdaptorsType" />
        </xsd:sequence>
</xsd:complexType>
<!-- ***************************************************************** -->
<xsd:complexType name="operationsType">
        <xsd:sequence>
                <xsd:element name="operation" type="operationType"  maxOccurs="unbounded"/>
        </xsd:sequence>
</xsd:complexType>
<!-- ***************************************************************** -->
<xsd:complexType name="operationType">
        <xsd:sequence>
                <xsd:element name="definition" type="xsd:string" />
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
<!-- ***************************************************************** -->
<xsd:complexType name="interfaceAdaptorsType">
        <xsd:sequence>
                <xsd:element name="adaptor" type="adaptorType" maxOccurs="unbounded" />
        </xsd:sequence>
</xsd:complexType>
<!-- ***************************************************************** -->
<xsd:complexType name="adaptorType">
        <xsd:sequence>
                <xsd:element name="operation" type="operationType" maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="classReference" type="xsd:string" use="required"/>
</xsd:complexType>
<!-- ***************************************************************** -->
<xsd:complexType name="partsType">
        <xsd:sequence>
                <xsd:element name="part" type="partType" maxOccurs="unbounded"/>
        </xsd:sequence>
</xsd:complexType>
<!-- ***************************************************************** -->
<xsd:complexType name="partType">

        <xsd:attribute name="id" type="xsd:string" use="required"/>
        <xsd:attribute name="url" type="xsd:string" use="required"/>
        <xsd:attribute name="type" type="xsd:string" use="required"/>
</xsd:complexType>
<!-- ***************************************************************** -->
<xsd:complexType name="parametersType">
        <xsd:sequence>
                <xsd:element name="parameter" type="xsd:string" maxOccurs="unbounded"/>
        </xsd:sequence>
</xsd:complexType>
<!-- ***************************************************************** -->
<xsd:complexType name="preconditionsType">
        <xsd:sequence>
                <xsd:element name="parametersPrecondition" type="parametersPreconditionsType"/>
                <xsd:element name="solutionsPreconditions" type="solutionsPreconditionsType"/>
        </xsd:sequence>
</xsd:complexType>
<!-- ***************************************************************** -->
```

Table 33 – XML Schema proposal for S&D Patterns (II)

```
        <xsd:complexType name="parametersPreconditionsType">
                <xsd:sequence>
                        <xsd:element name="parameterPrecondition" type="xsd:string" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="solutionsPreconditionsType">
                <xsd:sequence>
                        <xsd:element name="solutionPrecondition" type="xsd:string" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ************************************************************** -->
        <xsd:complexType name="staticTestsPerformedType">
                <xsd:sequence>
                        <xsd:element name="test" type="testType" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ************************************************************** -->
        <xsd:complexType name="testType">
                <xsd:sequence>
                        <xsd:element name="conditionsTest" type="xsd:string"/>
                        <xsd:element name="attackModels" type="xsd:string"/>
                </xsd:sequence>
                <xsd:attribute name="name" type="xsd:string" use="required"/>
        </xsd:complexType>
        <!-- ************************************************************** -->
        <xsd:complexType name="systemConfigurationType">
                <xsd:sequence>
                        <!-- include all necesary inicialiation of the components, frameworks,etc.) -->
                </xsd:sequence>
                <xsd:attribute name="description" type="xsd:string" use="required"/>
        </xsd:complexType>
        <!-- ************************************************************** -->
        <xsd:complexType name="monitoringType">
                <xsd:sequence>
                        <xsd:element name="monitor" type="monitorType" maxOccurs="unbounded"/>
                        <xsd:element name="monitorFormulae" type="MonitoringRule:formulaType" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="description" type="xsd:string"/>
        </xsd:complexType>
        <!-- ************************************************************** -->
        <xsd:complexType name="monitorType">
                <xsd:sequence>
                        <xsd:element name="localization" type="xsd:string"/>
                        <xsd:element name="type" type="xsd:string"/>
                        <xsd:element name="inicialization" type="xsd:string"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- ************************************************************** -->
</xsd:schema>
```

Table 34 – XML Schema proposal for S&D Patterns (and III)

A.3. XML Schema of S&D Implementations

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
	<!-- **************************************************************** -->
	<xsd:element name="S_and_DImplementation">
		<xsd:complexType>
			<xsd:sequence>
				<xsd:element name="creator" type="creatorType"/>
				<xsd:element name="timestamping" type="xsd:long"/>
				<xsd:element name="trustMechanisms" type="trustMechanismsType" maxOccurs="unbounded"/>
				<xsd:element name="S_and_DPatternReference" type="xsd:string" />
				<xsd:element name="preconditions" type="preconditionType"/>
				<xsd:element name="implementationDescription" type="xsd:string"/>
				<xsd:element name="implementationReference" type="implementationReferenceType"/>
				<xsd:element name="complianceProofs" type="complianceProofsType"/>
				<xsd:element name="comments" type="xsd:string"/>
			</xsd:sequence>
			<xsd:attribute name="name" type="xsd:string" use="required"/>
		</xsd:complexType>
	</xsd:element>
	<!-- **************************************************************** -->
	<xsd:complexType name="creatorType">
		<xsd:sequence>
			<xsd:element name="name" type="xsd:string"/>
			<xsd:element name="date" type="xsd:string"/>
		</xsd:sequence>
	</xsd:complexType>

	<!-- **************************************************************** -->
	<xsd:complexType name="trustMechanismsType">
		<xsd:sequence>
			<xsd:element name="signatureType" type="xsd:string"/>
			<xsd:element name="signer" type="xsd:string"/>
			<xsd:element name="signature" type="xsd:string"/>
		</xsd:sequence>
	</xsd:complexType>

	<!-- **************************************************************** -->
	<xsd:complexType name="implementationReferenceType">
		<xsd:sequence>
			<xsd:element name="reference" type="typeReference" maxOccurs="unbounded"/>
		</xsd:sequence>
	</xsd:complexType>
	<xsd:complexType name="typeReference">
		<xsd:sequence>
			<xsd:element name="URL" type="xsd:string"/>
			<xsd:element name="signature" type="xsd:string"/>
		</xsd:sequence>
	</xsd:complexType>
	<!-- **************************************************************** -->
	<xsd:complexType name="preconditionType">
		<xsd:sequence>
			<xsd:element name="precondition" type="xsd:string" maxOccurs="unbounded"/>
		</xsd:sequence>
	</xsd:complexType>
	<!-- **************************************************************** -->
	<xsd:complexType name="complianceProofsType">
		<xsd:sequence>
			<xsd:element name="proof" type="xsd:string" maxOccurs="unbounded"/>
		</xsd:sequence>
	</xsd:complexType>
	<!-- **************************************************************** -->
</xsd:schema>
```

Table 35 – XML Schema proposal for S&D Implementations

A.4. XML Schema of EC-Assertion

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://tempuri.org/ec/formula" xmlns="http://tempuri.org/ec/formula"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
        <!-- define formulas -->
        <xs:element name="formulas" type="formulaType"/>
        <!-- definition of complex types -->
        <xs:complexType name="formulaType">
                <xs:sequence>
                       <xs:element name="quantification" type="quantificationType" minOccurs="1" maxOccurs="unbounded"/>
                        <xs:element name="body" type="bodyHeadType" minOccurs="0"/>
                        <xs:element name="head" type="bodyHeadType"/>
                </xs:sequence>
                <xs:attribute name="formulaId" type="xs:string" use="required"/>
                <xs:attribute name="forChecking" type="xs:boolean" default="true"/>
        </xs:complexType>
        <xs:complexType name="bodyHeadType">
                <xs:sequence>
                        <xs:choice>
                                <xs:element name="predicate" type="predicateType"/>
                                <xs:element name="relationalPredicate" type="relationalPredicateType"/>
                                <xs:sequence minOccurs="0" maxOccurs="unbounded">
                                        <xs:element name="operator" type="logicalOperatorType"/>
                                        <xs:choice>
                                                <xs:element name="predicate" type="predicateType"/>
                                                <xs:element name="timePredicate" type="timePredicateType"/>
                                               <xs:element name="relationalPredicate" type="relationalPredicateType"/>
                                        </xs:choice>
                                </xs:sequence>
                        </xs:choice>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="predicateType">
                <xs:choice>
                        <xs:element name="happens" type="happensType"/>
                        <xs:element name="initiates" type="initiatesType"/>
                        <xs:element name="holdsAt" type="holdsAtType"/>
                        <xs:element name="initially" type="holdsAtType"/>
                        <xs:element name="terminates" type="terminatesType"/>
                </xs:choice>
                <xs:attribute name="negated" type="xs:boolean" default="false"/>
                <xs:attribute name="unconstrained" type="xs:boolean" default="false"/>
        </xs:complexType>
        <xs:complexType name="timePredicateType">
                <xs:choice>
                        <xs:element name="timeEqualTo" type="TimeRelation"/>
                        <xs:element name="timeNotEqualTo" type="TimeRelation"/>
                        <xs:element name="timeLessThan" type="TimeRelation"/>
                        <xs:element name="timeGreaterThan" type="TimeRelation"/>
                        <xs:element name="timeLessThanEqualTo" type="TimeRelation"/>
                        <xs:element name="timeGreaterThanEqualTo" type="TimeRelation"/>
                </xs:choice>
        </xs:complexType>
        <xs:complexType name="holdsAtType">
                <xs:sequence>
                        <xs:element name="fluent" type="fluentType">          </xs:element>
                        <xs:element name="timeVar" type="timeVariableType">      </xs:element>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="initiatesType">
                <xs:sequence>
                        <xs:element name="event" type="eventType">      </xs:element>
                        <xs:element name="fluent" type="fluentType"/>
                        <xs:element name="timeVar" type="timeVariableType"/>
                </xs:sequence>
        </xs:complexType>
```

Table 36 – XML Schema proposal for EC-Assertion (I)

```
<xs:complexType name="happensType">
        <xs:sequence>
                <xs:element name="event" type="eventType">      </xs:element>
                <xs:element name="timeVar" type="timeVariableType"/>
                <xs:element name="fromTime" type="TimeExpression"/>
                <xs:element name="toTime" type="TimeExpression"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="terminatesType">
        <xs:sequence>
                <xs:element name="event" type="eventType">      </xs:element>
                <xs:element name="fluent" type="fluentType"/>
                <xs:element name="timeVar" type="timeVariableType"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="fluentType">
        <xs:choice>
                <xs:element name="author" type="authorisationFluentType">                </xs:element>
                <xs:element name="exp" type="exposesFluentType">          </xs:element>
                <xs:element name="authen" type="authenticationFluentType">             </xs:element>
                <xs:element name="valueof" type="valueofType">       </xs:element>
        </xs:choice>
</xs:complexType>
<xs:complexType name="authorisationFluentType">
        <xs:sequence>
                <xs:element name="authorisingAgent" type="variableType">                </xs:element>
                <xs:element name="authorisedAgent" type="variableType">                 </xs:element>
                <xs:element name="event" type="eventType">               </xs:element>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="exposesFluentType">
        <xs:sequence>
            <xs:choice>
                 <xs:element name="event" type="eventType" minOccurs="1" maxOccurs="unbounded"></xs:element>
            </xs:choice>
            <xs:element name="infoTerm" type="variableType">         </xs:element>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="authenticationFluentType">
        <xs:sequence>
                <xs:element name="agent" type="variableType">                  </xs:element>
                <xs:element name="event" type="eventType">       </xs:element>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="valueofType">
        <xs:sequence>
                <xs:element name="target">
                        <xs:complexType>
                                <xs:sequence>
                                        <xs:element name="variable" type="variableType"/>
                                </xs:sequence>
                        </xs:complexType>
                </xs:element>
                <xs:element name="source">
                        <xs:complexType>
                                <xs:choice>
                                        <xs:element name="variable" type="variableType"/>
                                        <xs:element name="operationCall" type="operationCallType"/>
                                </xs:choice>
                        </xs:complexType>
                </xs:element>
        </xs:sequence>
</xs:complexType>
```

Table 37 – XML Schema proposal for EC-Assertion (II)

```
<xs:complexType name="quantificationType">
	<xs:sequence>
		<xs:element name="quantifier">
			<xs:simpleType>
				<xs:restriction base="xs:string">
					<xs:enumeration value="forall"/>
					<xs:enumeration value="existential"/>
				</xs:restriction>
			</xs:simpleType>
		</xs:element>
		<xs:choice>
			<xs:element name="regularVariable" type="variableType"/>
			<xs:element name="timeVariable" type="timeVariableType"/>
		</xs:choice>
	</xs:sequence>
</xs:complexType>
<xs:complexType name="variableType">
	<xs:sequence>
		<xs:element name="varName" type="xs:string"/>
		<xs:choice>
			<xs:sequence>
				<xs:element name="varType" type="xs:string"/>
				<xs:element name="value" type="xs:string" minOccurs="0"/>
			</xs:sequence>
			<xs:element name="array" type="arrayType"/>
		</xs:choice>
	</xs:sequence>
	<xs:attribute name="persistent" type="xs:boolean" default="false"/>
	<xs:attribute name="forMatching" type="xs:boolean" default="true"/>
</xs:complexType>
<xs:complexType name="timeVariableType">
	<xs:sequence>
		<xs:element name="varName" type="xs:string"/>
		<xs:element name="varType" type="xs:string" fixed="TimeVariable"/>
		<xs:element name="value" type="xs:string" minOccurs="0"/>
	</xs:sequence>
</xs:complexType>
<xs:simpleType name="logicalOperatorType">
	<xs:restriction base="xs:string">
		<xs:enumeration value="and"/>
		<xs:enumeration value="or"/>
	</xs:restriction>
</xs:simpleType>
<xs:complexType name="TimeExpression">
	<xs:sequence>
		<xs:element name="time" type="timeVariableType"/>
		<xs:sequence minOccurs="0" maxOccurs="unbounded">
			<xs:choice>
				<xs:element name="plusTime" type="timeVariableType"/>
				<xs:element name="minusTime" type="timeVariableType"/>
				<xs:element name="plus" type="xs:decimal"/>
				<xs:element name="minus" type="xs:decimal"/>
			</xs:choice>
		</xs:sequence>
	</xs:sequence>
</xs:complexType>
<xs:complexType name="TimeRelation">
	<xs:sequence>
		<xs:element name="timeVar1" type="TimeExpression"/>
		<xs:element name="timeVar2" type="TimeExpression"/>
	</xs:sequence>
</xs:complexType>
<xs:complexType name="varRelationType">
	<xs:sequence>
		<xs:element name="operand1" type="operandType"/>
		<xs:element name="operand2" type="operandType"/>
	</xs:sequence>
</xs:complexType>
```

Table 38 – XML Schema proposal for EC-Assertion (III)

```
        <xs:complexType name="relationalPredicateType">
                <xs:sequence>
                        <xs:choice>
                                <xs:element name="equalTo" type="varRelationType"/>
                                <xs:element name="notEqualTo" type="varRelationType"/>
                                <xs:element name="lessThan" type="varRelationType"/>
                                <xs:element name="greaterThan" type="varRelationType"/>
                                <xs:element name="lessThanEqualTo" type="varRelationType"/>
                                <xs:element name="greaterThanEqualTo" type="varRelationType"/>
                        </xs:choice>
                        <xs:element name="timeVar" type="timeVariableType"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="operandType">
                <xs:choice>
                        <xs:element name="operationCall" type="operationCallType"/>
                        <xs:element name="variable" type="variableType"/>
                        <xs:element name="constant" type="constantType"/>
                </xs:choice>
        </xs:complexType>
        <xs:complexType name="operationCallType">
                <xs:sequence>
                        <xs:element name="name" type="xs:string"/>
                        <xs:element name="partner" type="xs:string" minOccurs="0"/>
                        <xs:element name="variable" type="variableType" minOccurs="0" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="constantType">
                <xs:sequence>
                        <xs:element name="name" type="xs:string"/>
                        <xs:element name="value" type="xs:string"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="arrayType">
                <xs:sequence>
                        <xs:element name="type" type="xs:string"/>
                        <xs:element name="index" type="xs:string" minOccurs="0"/>
                        <xs:element name="value" type="arrayValueType" minOccurs="0" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="arrayValueType">
                <xs:sequence>
                        <xs:element name="indexValue" type="xs:string"/>
                        <xs:element name="cellValue" type="xs:string"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="eventType">
                <xs:sequence>
                        <xs:element name="eventID" minOccurs="1" maxOccurs="1" type="xs:string"/>
                        <xs:element name="sender" type="variableType">              </xs:element>
                        <xs:element name="receiver" type="variableType">       </xs:element>
                        <xs:element name="status" type="xs:string">                </xs:element>
                        <xs:element name="oper" type="operationType">              </xs:element>
                        <xs:element name="source" type="xs:string">              </xs:element>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="operationType">
                <xs:sequence>
                        <xs:element name="opName" type="xs:string">                 </xs:element>
                        <xs:element name="op_args" minOccurs="0" maxOccurs="1" type="variableType">      </xs:element>
                </xs:sequence>
        </xs:complexType>
</xs:schema>
```

Table 39 – XML Schema proposal for EC-Assertion (and IV)

References

[1] Botella, A. and Maña, A. An Introduction to Action Specification Language (ASL) for Serenity. https://bscw.sit.fraunhofer.de/bscw/bscw.cgi/d904220/Introduction%20to%20ASL%20for%20SERENITY.pdf. SERENITY Project. Internal Report. 2007.

[2] Wilkie I., King A., Clarke M., Weaver C., Raistrick C. and Francis P. The UML Action Specification Language Reference Guide. December, 2006.

[3] Maña A., Muñoz A., Sanchez-Cid F., Serrano D., 2006: Deliverable A5.D0.1: SERENITY Conceptual Model. SERENITY Project. Internal Report. April.

[4] Maña A., Presenza D., Piñuela A., Serrano D., Soria P.,and Sotiriou D. Deliverable A6.D3.1 – Specification of SERENITY Architecture. SERENITY Project.31 December 2006.

[5] Marcotty M. and Ledgard H., The World of Programming Languages, Springer-Verlag, Berlin 1986., pages 41 and following.

[6] W3C XML Query (XQuery). See http://www.w3.org/XML/Query/.

[7] Shanahan, M.P., 1999: The Event Calculus Explained, in Artificial Intelligence Today, LNAI no. 1600:409-430, Springer

[8] Spanoudakis G. Mahbub K, 2006: Non Intrusive Monitoring of Service Based Systems , International Journal of Cooperative Information Systems, Vol. 15, No. 3, 325-358

[9] Mahbub K., Spanoudakis G., November 2004: A Framework for Requirements Monitoring of Service Based Systems, 2nd International Conference on Service Oriented Computing, New York

[10] W3C. XML Schema Reference from the XML Schema Working Group. http://www.w3.org/XML/Schema.html.

[11] RSA Security. Information about RSA BSAFE® Encryption, Signature and Privacy solutions available at http://www.rsasecurity.com/node.asp?id=1202.

[12] NIST Computer Security: Federal Information Processing Standards (FIPS) page: http://csrc.nist.gov/publications/fips/index.html.

[13] Infosec Assurance and Certification Services (IACS). http://www.cesg.gov.uk/site/iacs/index.cfm.

[14] Infineon Technologies. http://www.infineon.com/.

[15] Data Encryption Standard (DES) conforming with FIPS 46-3. http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[16] W3C. XML Schema Reference from the XML Schema Working Group. http://www.w3.org/XML/Schema.html.

[17] Maña A., Muñoz A., Sanchez-Cid F., Serrano D., Spanoudakis G., Androutsopoulos K., Compagna L.. Deliverable A5.D2.1 – Patterns and Integration Schemes Languages (Initial Version). SERENITY Project. 30 September 2006.