A5.D2.5 – Patterns and Integration Schemes Languages (Final 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. K. 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 September 2008
Actual Date of Delivery	15 November 2008
Responsible Unit	UMA
Contributors	SAP, CUL, UTN
Keyword List	
Dissemination level	PU

Change History

Version	Date	Status	Author (Unit)	Description
0.1	10/10/08	Draft	Daniel Serrano, Gimena Pujol, José Ruiz, Rajesh Harjani, Antonio Maña	Integration of A5.D2.3 (previous version of this deliverable), A5.R8 (artefact language 1.0), and A5.R9 (preconditions language).
0.2	11/10/08	Draft	Daniel Serrano, Gimena Pujol, José Ruiz, Rajesh Harjani, Antonio Maña	Review and updated of A5.D2.3 sections.
Final	12/11/08	Draft	Paul El Khoury, Keqin Li, Smriti K. Sinha	Reviewed and updated A5.D2.5

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:

- New S&D Artefact languages (called version 1.0).
- There is a new language for the specification of "Preconditions". This deliverable presents the structure and the syntax of the preconditions as well as the guidelines for their creation.

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 [1], 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*.

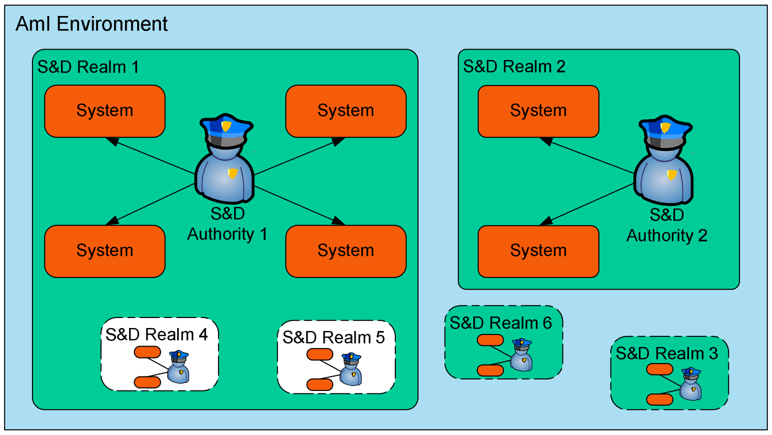


Figure 1 – Relations between the modelling artefacts

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 3 of 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 ffmeans 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 time-stamping 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, their performance, target platform, programming language or any other feature not fixed at pattern's level.

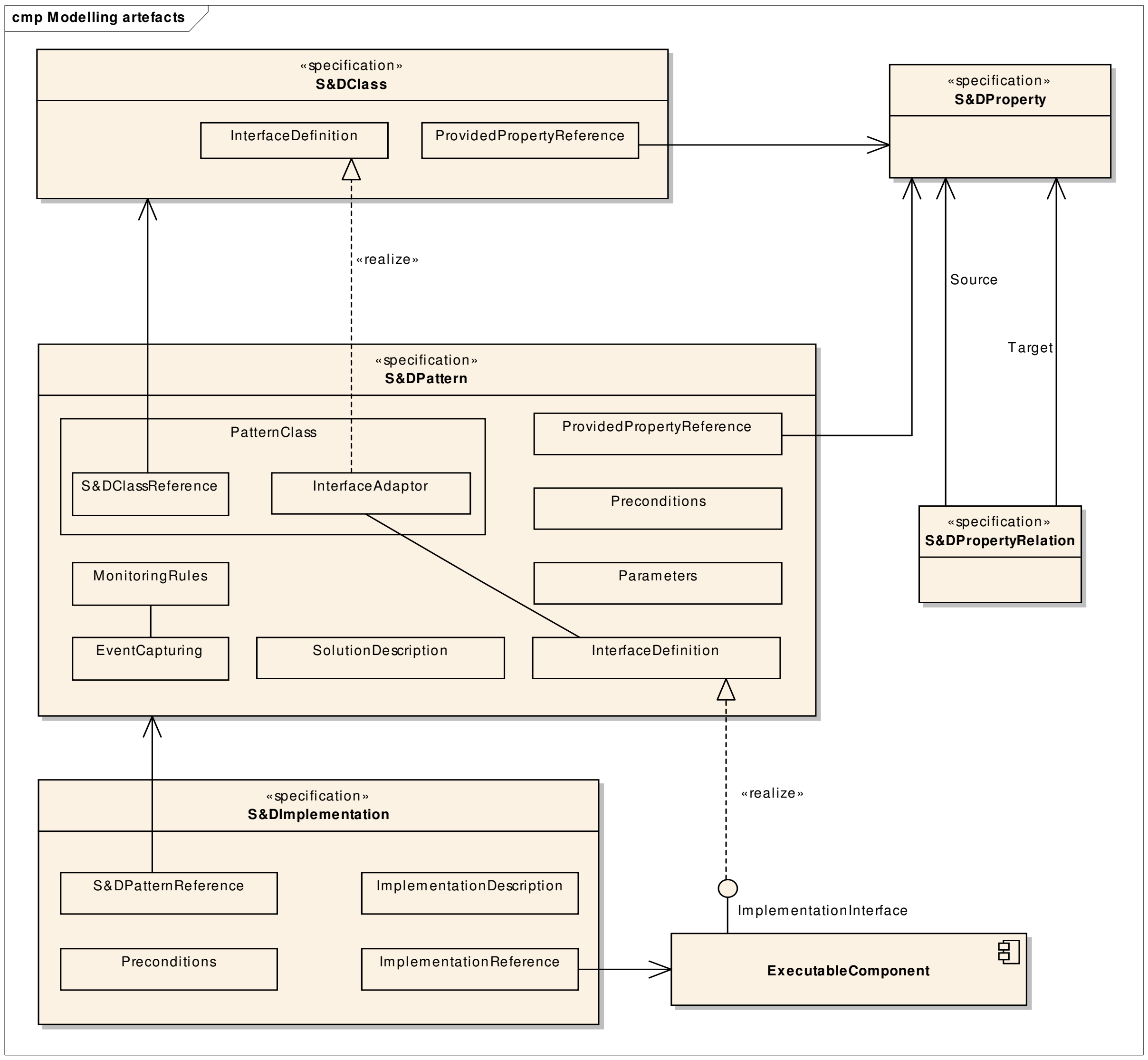


Figure 2 – Relations between the modelling artefacts

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 be. 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 in sections 2, 3 and 4.

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? May be 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.

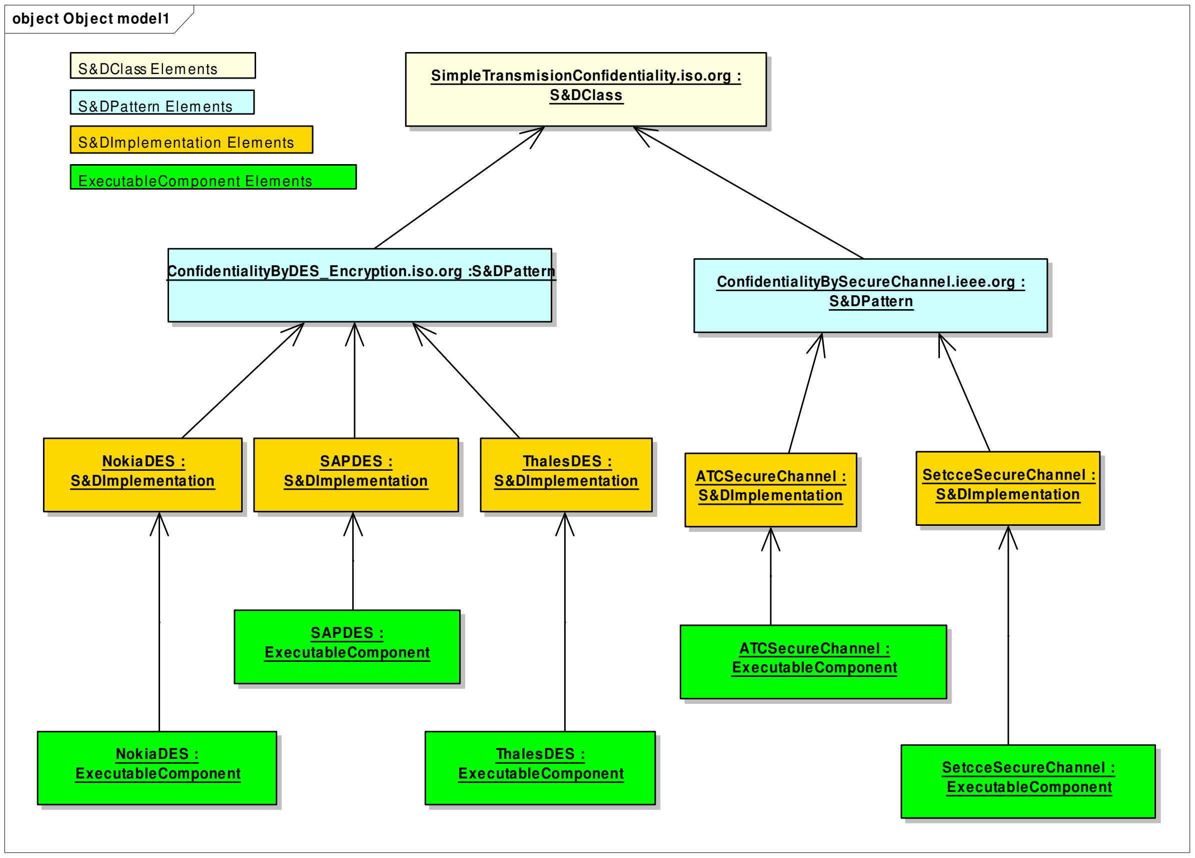

object Object model1
S&DClass Elements
S&DPattern Elements
S&DImplementation Elements
ExecutableComponent Elements
SimpleTransmisionConfidentiality.iso.org : S&DClass
ConfidentialityByDES_Encryption.iso.org :S&DPattern
ConfidentialityBySecureChannel.ieee.org : S&DPattern
NokiaDES : S&DImplementation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

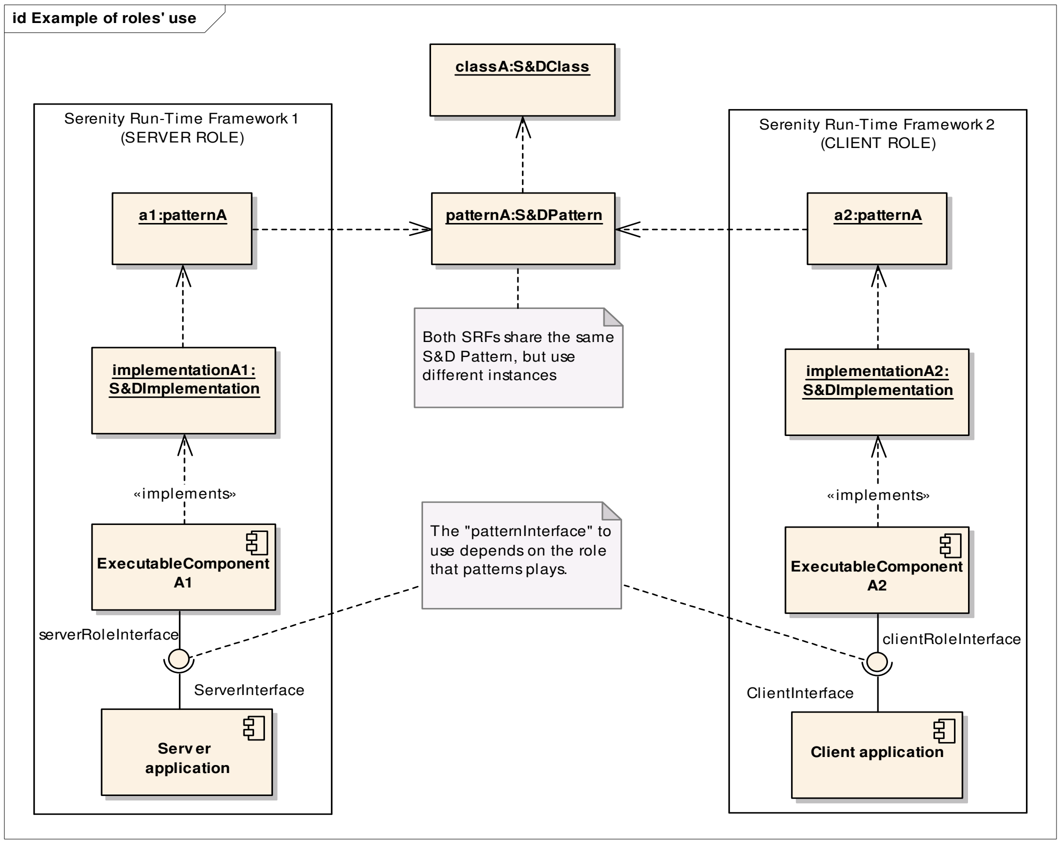

id Example of roles' use
classA:S&DClass
Serenity Run-Time Framework 1 (SERVER ROLE)
Serenity Run-Time Framework 2 (CLIENT ROLE)
a1:patternA
patternA:S&DPattern
a2:patternA
Both SRFs share the same S&D Pattern, but use different instances
implementationA1: S&DImplementation
implementationA2: S&DImplementation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

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:

— *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&D ClassReference* 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&D PatternReference*. 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 first version of the deliverable [3].

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&D Patterns* in Figure 5) refer to solutions (*S&D Solutions*) and contain the semantics (*S&D SolutionSemantics*) that describe such solution. The semantics are described in terms of the semantics (*S&D PropertySemantics*) of the particular properties (*S&D Property*) provided by the solution. Solutions *(S&D Solutions)* 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&D Solutions)* may have different implementations *(S&D Implementations)*.

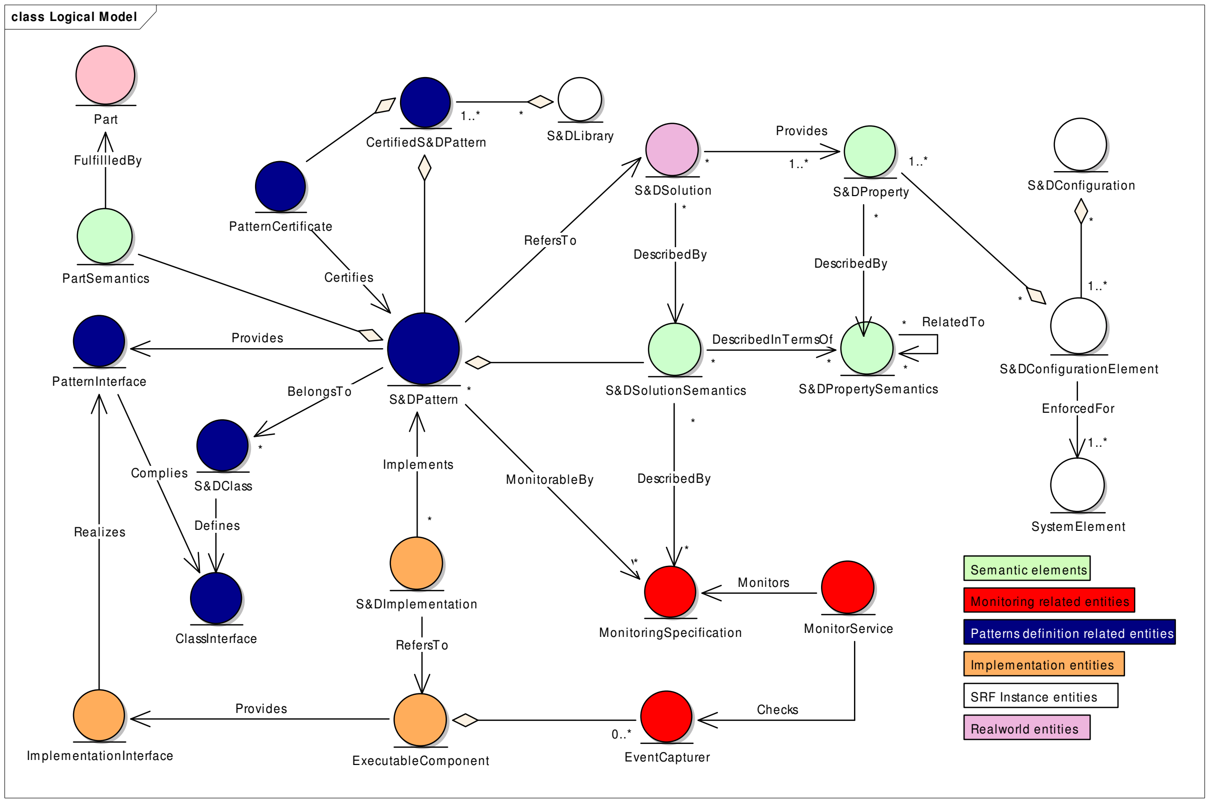


Figure 5 – Logical model

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&D Library*) is composed of S&D Patterns and Integration Schemes that hold a certificate, the so-called certified patterns (*CertifiedS&D Pattern*).

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&D Implementations*) 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&D Classes* are used to group a set of *S&D Patterns*. All patterns that define the same interface come under the same umbrella: an *S&D Class*. At some extent, the concept of *S&D Class* is close to the concept of class in orient object programming.

Finally, users define the security and dependability requirements (*S&D Configuration*) for their systems, grouping a set of specific requirements (*S&D ConfigurationElement*). Each specific requirement is specified by means of a set of properties (*S&D Property*) that must be enforced for a particular element of the system (*SystemElement*). All this elements are shown in Figure 5.

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. There are two different implementations of the SERENITY Library.

Firstly, the SERENITY Development-time Framework (*SDF*) S&D Library is oriented to support development processes of S&D Solutions and secure applications. This S&D Library has the following features: Large-scale database of S&D Solutions, supporting development-oriented queries, accessible through WebServices-based application (client tools for using and managing on-line S&D libraries). The SDF includes tools for S&D Patterns developers and tools for application developers.

Secondly, the SERENITY Run-time Framework (*SRF*) S&D Library supports the run-time S&D Solution selection process. The SRF S&D Library is a Small-scale (device specific implementation), optimized to context-oriented queries, and it is an internal component of the SRF being accessible only from the SRF through an integrated interface.

The main reason for having a particular implementation of the SERENITY Library in the SRF is that 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, by means of the SRF S&D 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 SRF S&D Library offers the channel for the SRF 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 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 6 represents all the elements described in this section as part of the SERENITY Library.

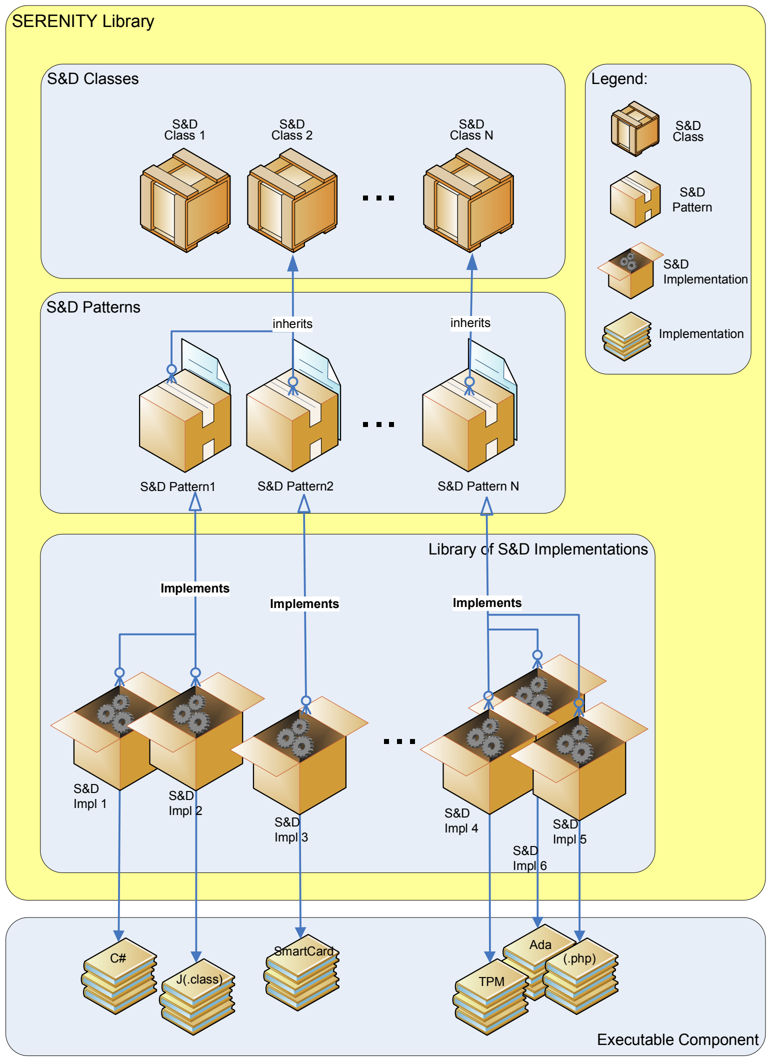


Figure 6 – Representation of SERENITY Library

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 SERENITY Library (SDF S&D Library and SRF 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 7 depicts the main architectural elements as well as the interactions among them. The architecture of the SRF can be consult in [3].

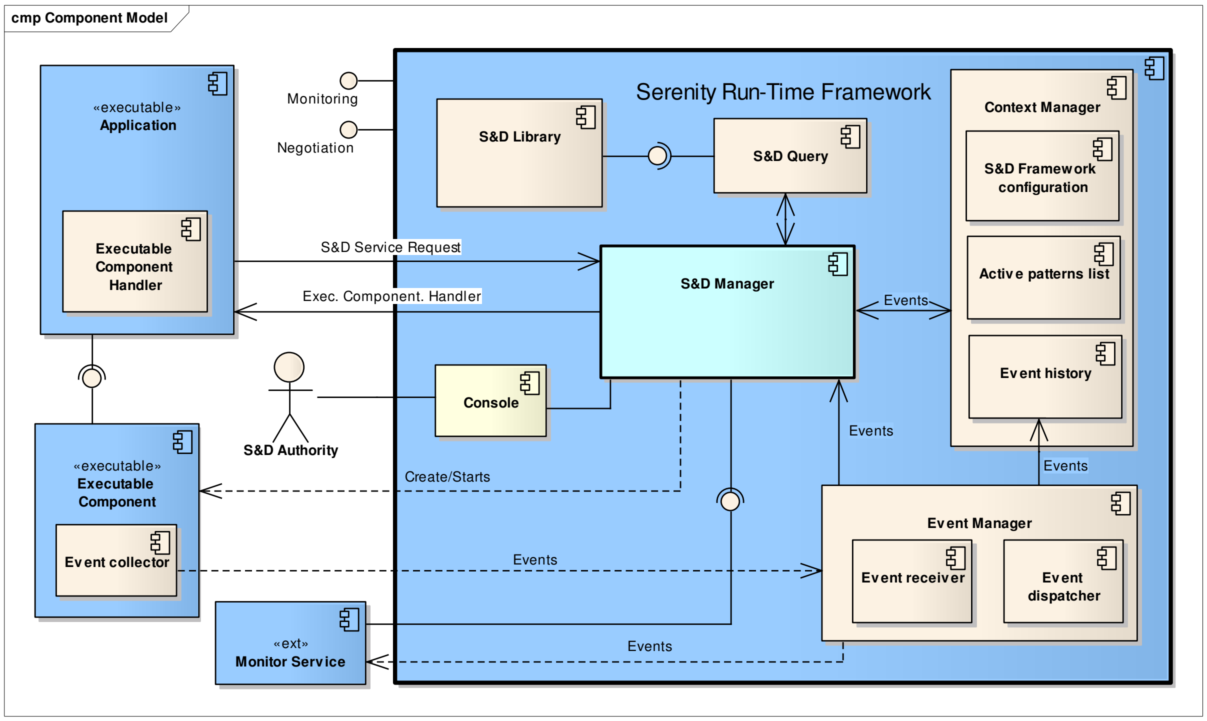


Figure 7 – Main elements of the SERENITY Architecture

3.2.1. Internal Elements

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

— The S&D Manager is the core of the system. It controls the rest of the components of the system. It processes SDrequests from Serenity-aware applications returning back ExecutableComponents. It is responsible for the activation and deactivation of the ExecutableComponents; it also takes the necessary recovery actions when it is informed by the monitoring services of a monitoring rule violation. The S&D Manager component manages monitoring services by sending the monitoring rules to the Monitoring Service.

— The Context Manager Component records the data related to the context (S&D Framework configuration, active patterns, events history). The context is used by the S&D Manager to select the most appropriate S&D Solution for a given scenario. Inside the Context 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.

Event history: This element keeps the history of monitoring rules satisfied. Monitoring Services sends this information as result of events and monitoring rules analysis.

— The Console is the main interface through which the S&D Manager contacts with the SRF administrator and vice versa. Different implementations of this console provide ways of defining, viewing and changing the S&D Configuration through a user-friendly interface.

— Event Manager: it receives all the events generated by the patterns (these events are generated by the Executable Components implementing patterns), and it sends them to Monitoring Services. Monitoring Services should receive these events from the SRF in order to analyse them and send the monitoring results back.

The SRF *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. Deliverable [4] introduces how applications interact with Executable Components. The *SRF* is the one in charge of keeping 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.

3.2.2. External Elements

From the outside point of view, every instance of the SRF, as a system, provides interfaces in order to allow interaction with other systems, or other SRFs instances. The SRF provides two main interfaces. Begin with the negotiation interface. It is used in order to establish the configuration of the interacting SRFs. This interaction makes sense when two applications supported by different

SRFs need to agree on the use of the same S&D Solution. Secondly, SRFs offer a monitoring interface. External systems interacting with an instance of the SRF are able to monitor that the behaviour of the framework is correct. These interfaces provide support for the dynamic supervision and adaptation of the system's security to the transformations in ever-changing AmI ecosystems.

Outside of the SRF, but very related to it, we find Serenity-aware applications, Executable Components and Monitoring Services. Every time an application needs a security service (provided by an S&D Solution) it sends an SDrequest to the SRF. The SRF responds with a ready to use Executable Component, if and only if there is an S&D Solution meeting the SDrequest. Monitoring services are external entities having the responsibility of performing dynamic analysis of the properly operation of S&D Solutions. They count on monitoring rules associated to each S&D Solution. Monitoring services operation is based on event calculus [5], the events are collected from events collectors implemented as part of the ExecutableComponents. The SRF reacts performing recovery task taking into account the information provided by monitoring services.

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 Artefacts. This deliverable introduces a new version of the S&D Artefact languages. This new version of the language introduces several changes worth to be highlighted. One of the minor changes is a new field corresponding to the version of the artefact language used is added. This version number does not describe the artefact version but the artefact language used to describe it.

One of the most radical changes is the separation of almost all fields in Operational and Informational sets of fields. Besides, we have included a new field to indicate the validity periods of time for artefacts. In this version roles appear grouped and containing all role-related data. As aforementioned, all S&D artefacts information has been split into two sets (operational and informational), we have included one section devoted to roles in each one language part (note that not always is applicable). Other remarcable change consists on new fields devoted to Monitoring services and monitoring services polling times. Finally, we have merged "ExternalPreconditions" and "ContextPreconditions" fields into a field called "preconditions".

4.1. Informational vs. Operational content

One of the most important enhancements is to categorize all fields in the artefacts' descriptions into two different sets: the *Informational* part and the *Operational* one. On the one hand, the informational part includes all fields containing information meant to be used by system developers. On the other hand, the operational part is intended to contain all fields directly used at run-time. Figure 8 presents how in previous S&D artefacts languages all information was non categorized and how the version 1.0 of the language categorizes all information into "Informational" and "Operational" parts.

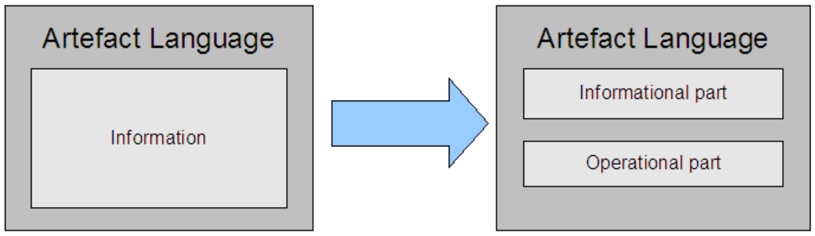


Figure 8: Separation of Operational and Informational components.

4.2. Enhancements to Monitoring representation

This language version includes a section devoted to describe all monitoring services used by the pattern (due to the current state of S&D artefacts research, monitoring services are applicable only for S&D Patterns and for Integration Schemes). The new *monitoring services declaration* section includes the list of monitoring services related with this artefact. Indeed, this section is used to link an identifier to each monitoring service, as well as some information about the monitoring service, such as type and optionally location. This identifier is intended to be used in the monitoring rules sections for linking each monitoring rule to a monitoring service.

4.3. Representing Roles

In order to manage the role representation, we propose the structure presented in Figure 9. This approach is based on grouping the elements related to each role in a role section. Every role section contains the role declaration (with fields such as, role name, cardinality, etc.) plus all elements that are related to that role (interface, class adaptors, preconditions, monitoring rules, etc.). In the case of S&D Classes the role section has no sense in the operational part. Consequently, a role section appears only in the informational part. On the contrary, in the case of S&D Patterns, there is a role section in both the operational and the informational parts. Besides, only in the case of S&D Patterns, the role declaration section includes one or more class adaptors.

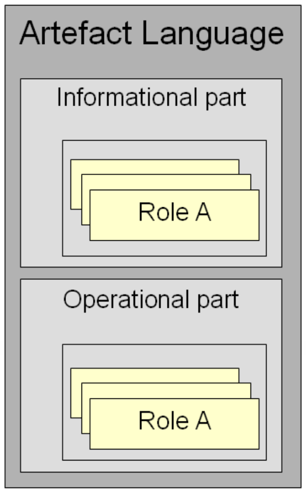


Figure 9: Each part includes a role subsection.

In order to provide some examples, Figure 10 presents a scenario composed by one S&D Pattern (*P1*) and three S&D Classes (*C1, C2 and C3*). The P1 S&D Pattern belongs to C1, C2 and C3 S&D Classes.

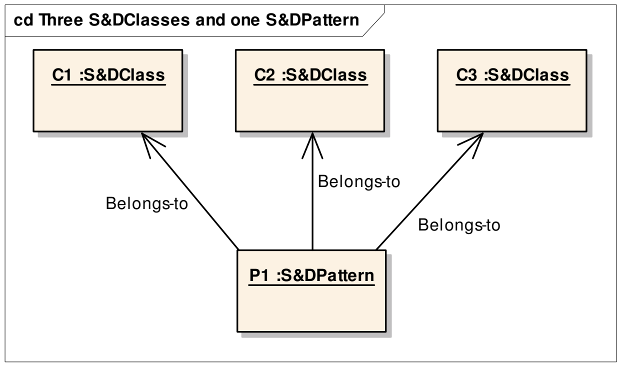


Figure 10: One S&D Pattern belonging to more than one S&D Class.

The Class Adaptor sections (informational part → role section) are used in order to create associations between S&D Patterns and S&D Classes. That is, class adaptors do not adapt S&D Classes to S&D Patterns, but they adapt S&D Classes roles to S&D Patterns roles.

Figure 11 presents the relations between P1, C1 and C2. This diagram shows that P1 has two roles (PR1 and PR2). The C1 S&D Class has one role CR1, and C2 has to roles CR2 and CR3. The adaptation between PR1 and CR1 is done by means of the Class Adator CA1-1. Next S&D Class, C2, has two roles (CR2 and CR3) that are implemented by roles PR1 and PR2 in the P1 S&D Pattern. Because of this, PR1 has a Class Adator (CA1-2) that adapts PR1 interface to CR2 interface, and PR2 has a Class Adaptor (CA2-3) adapting it to CR3.

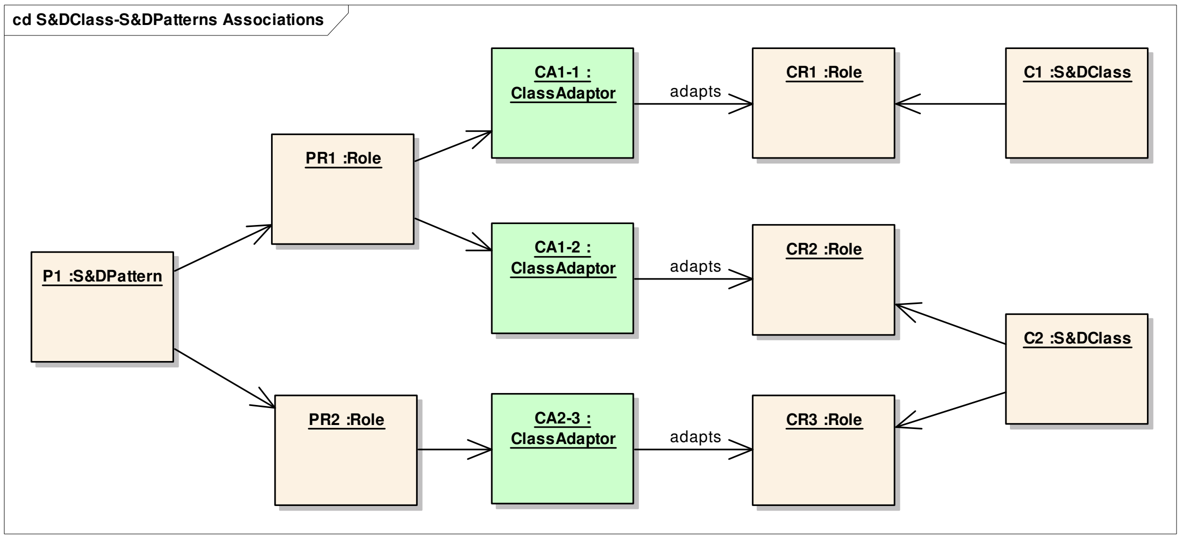


Figure 11: One S&D Pattern belonging to more than one S&D Class.

For those cases, in which two (or more) S&D Class roles are implemented by means of only one S&D Pattern role, a new role at the level of the S&D Pattern needs to be developed. In these cases, the new role contains a classAdaptor performing the adaptation of S&D Class roles. The S&D Class C3 has two roles (CR4 and CR5) to be adapted to only one P1 role (PR3). That is to say, the PR3 role implements both CR4 and CR5 functionallities. This scenario is presented in Figure 12, where the role PR3 has been added in order to implement both CR4 and CR5 role of the C3 S&D Class. The PR3 role mix PR1 and PR2 functinallities. An example of this scenario could be a S&D Class representing a security solution based on an asymmetric communication (for instance, client-server) implemented by a S&D Pattern implementing for sides of the communication.

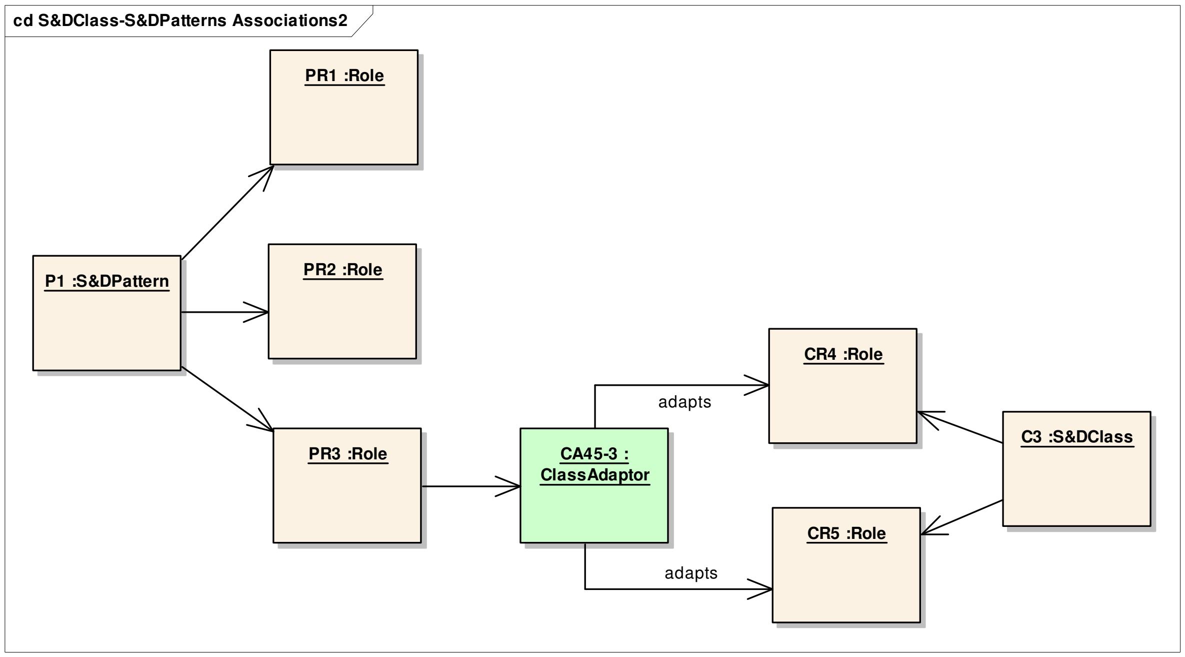


Figure 12: An S&D Pattern role implementing the funcionallities of two S&D Classes roles.

4.4. 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 **Erreur ! Source du renvoi introuvable.**) 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).

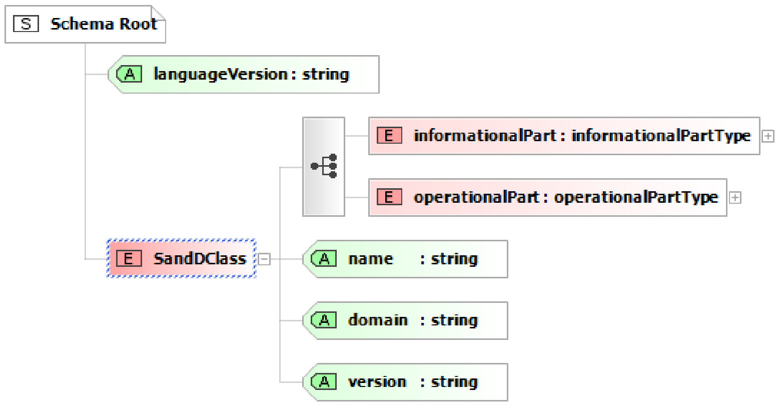
Some examples are defined below (Table 2) in order to facilitate the understanding of the notation:

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

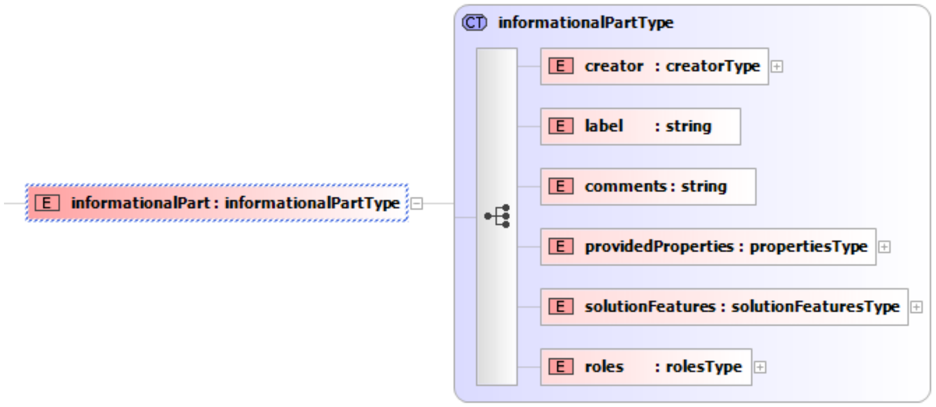
Table 2 – Examples of use of the naming scheme

4.5. Detailed description of S&D Classes

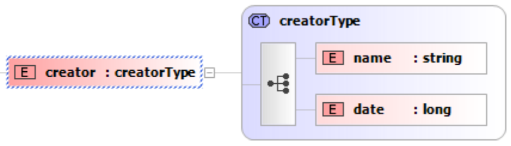
This section details the XML structure for the language used for representing S&D Classes. Among the most important changes of this new version of the S&D Class language we highlight the separation of the all fields in two sections; informational and operational parts. All fields belonging to operational part are suitable to be used at runtime. However, fields belonging to the informational part will not be used at runtime. Next figure depicts a general view of the structure of the S&D Class language. At this level, an S&D Class is composed by name, domain, version, informationalPart and operationalPart.


S Schema Root
A languageVersion : string
E informationalPart : informationalPartType
E operationalPart : operationalPartType
E SandDClass
A name : string
A domain : string
A version : string

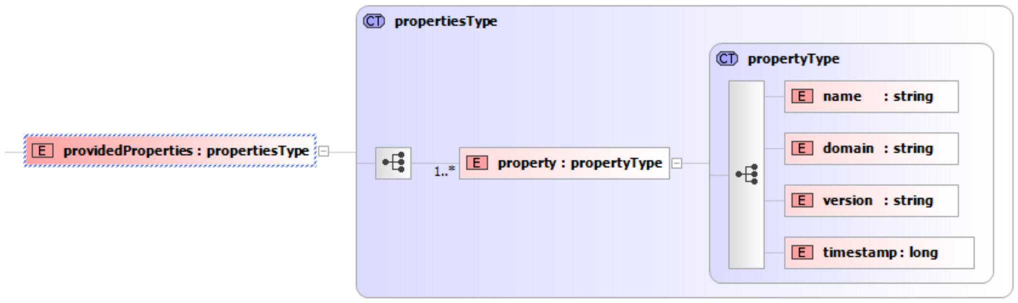
Next picture shows the informationalPart elements of the S&D Class language. Concretely, we highlight the creator field. A field is dedicated to describe the provided properties. Finally, there is a field describing the different roles related to the S&D Solution represented in the S&D Class.


CT informationalPartType
E creator : creatorType
E label : string
E comments : string
E providedProperties : propertiesType
E solutionFeatures : solutionFeaturesType
E roles : rolesType
E informationalPart : informationalPartType

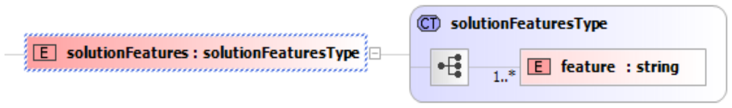
Next figure depicts in detail the creator field; it is composed by the creator name and the date of creation.



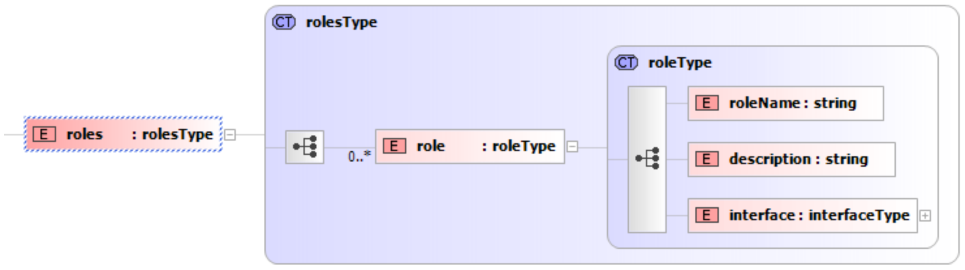
The "providedProperties" field stores all information regarding to the properties provided by the S&D Class. It is composed by several "property" elements. The structure "property" elements can be check in the following figure.



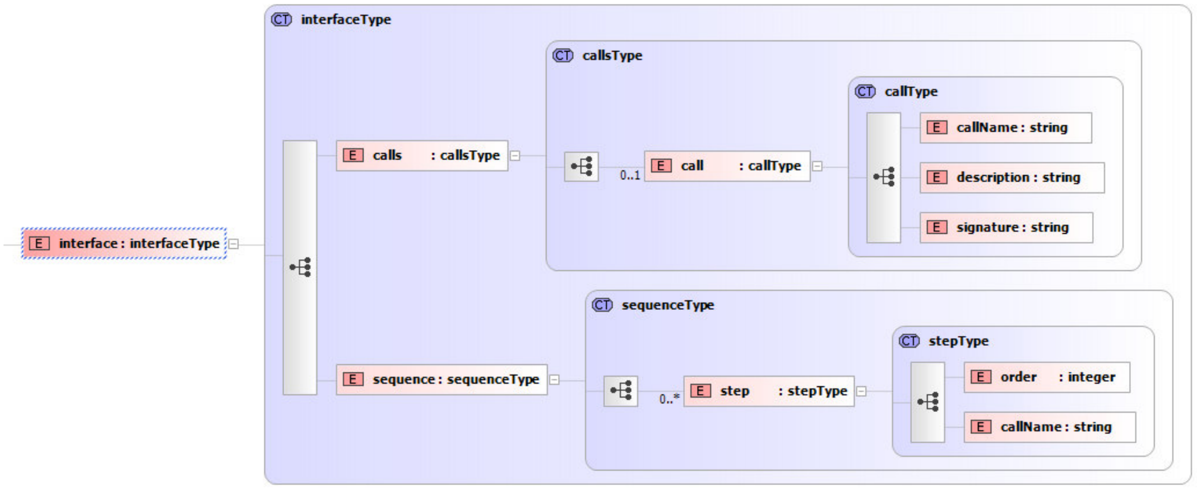
The "solutionFeatures" elements represents the S&D Solution features, it is composed by one or more "feature" elements of type String.



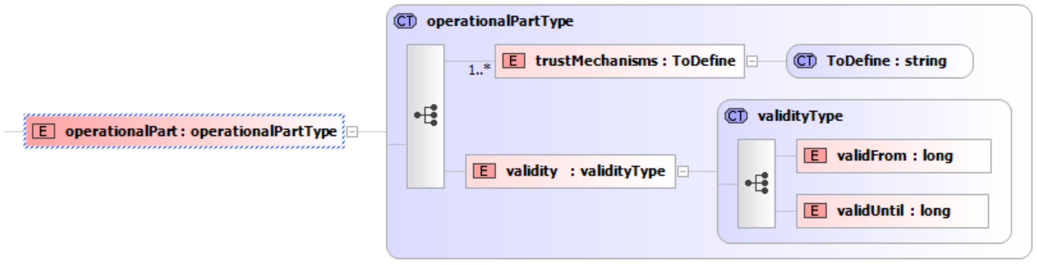
Concerning the roles, next figure depicts in detail the structure of the role representation in the informationalPart. As aforementioned, the informationalPart is not used at run-time, at least directly. This role section is composed by several (0 or more) "role" elements of type "roleType". Every "role" element is composed by the name of the role, a description of it and the interface of this role. This interface is composed by the calls provided and the sequence required to be invoked.



The role interface is of "interfaceType". The following figure details the structure of the "interfaceType" type.

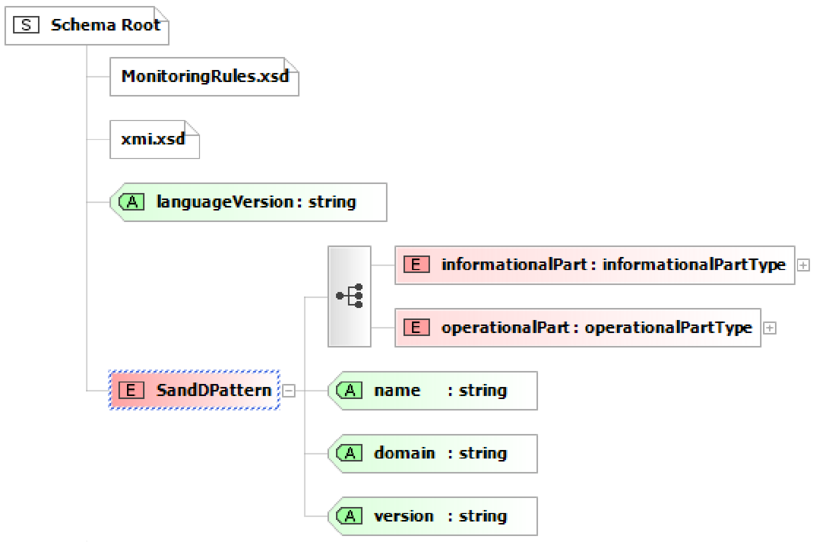


Similarly to the previous description of the informationalPart, the rest of this subsection presents the operationalPart of the S&D Class language. The operationalPart section is composed by two elements a list of trustmechanisms, and the artefact validity period of time. Currently, the trustMechanisms structure is not defined. The "validity" element is of type "validityType", which is composed by two elements of type String: "validFrom" and "validUntil".

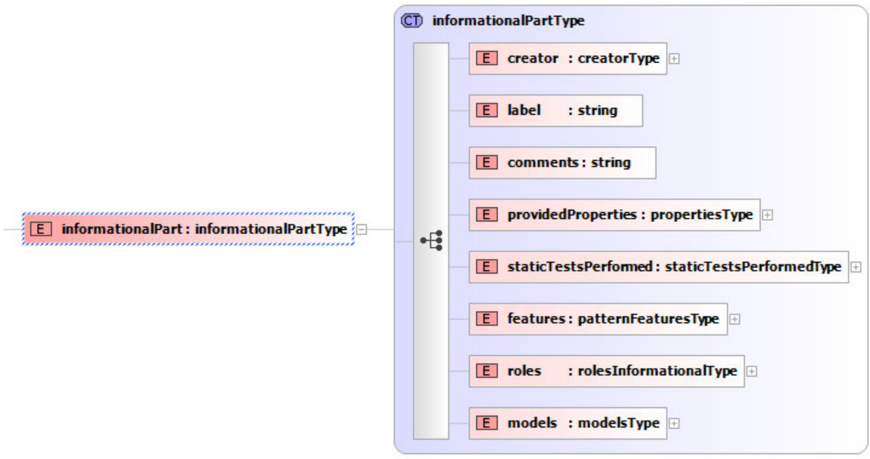


4.6. Detailed description of S&D Patterns

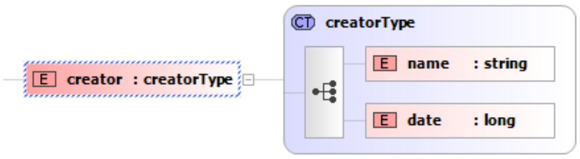
This section presents the XML language to be used for S&D Patterns. Similarly to the S&D Class language, in this case there is a separation of fields in two different sets: the operational part and the informational part. Next figure depics a global view of S&D Pattern language showing the fields at a first glance. It presents a simplified view showing the name, domain and version of the S&D Pattern. And, two more sections sections one of them it is the "informationalPart" element, used to store the informational part, and the other it is the "operationalPart" element.


Schema Root
MonitoringRules.xsd
xmi.xsd
A languageVersion : string
E informationalPart : informationalPartType
E operationalPart : operationalPartType
E SandDPattern
A name : string
A domain : string
A version : string

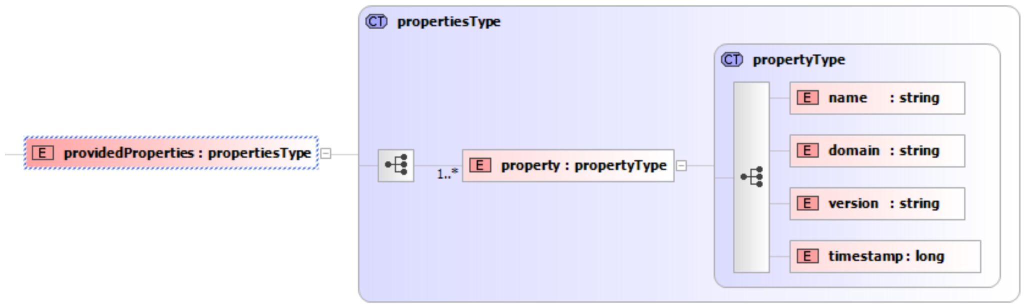
The following figure presents the informational part elements. The informational part is composed by the information related with the creation of the S&D Pattern. Also, a label field and a comments field are available. Other needed fields are the related with the provided properties, the field related with the static tests performed, the field concerning with the roles and an element storing the model of the S&D Pattern.


informationalPartType
E informationalPart : informationalPartType
E creator : creatorType
E label : string
E comments : string
E providedProperties : propertiesType
E staticTestsPerformed : staticTestsPerformedType
E features : patternFeaturesType
E roles : rolesInformationalType
E models : modelsType

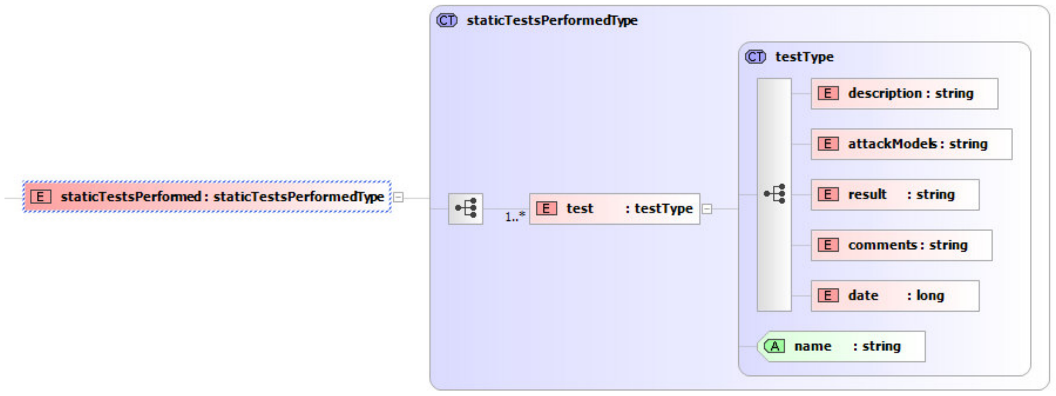
Next figure depicts in detail the "creator" field. The "creator" is of type "creatorType" and this type is composed by two elements: the name of the creator and the date of creation.



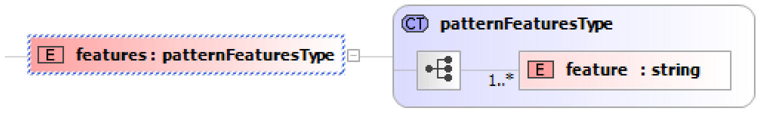
Next figure shows how providedProperties field is composed by a list from 1 to "more than 1" property elements. Every "property" element is composed by name, domain, version and a timestamp.



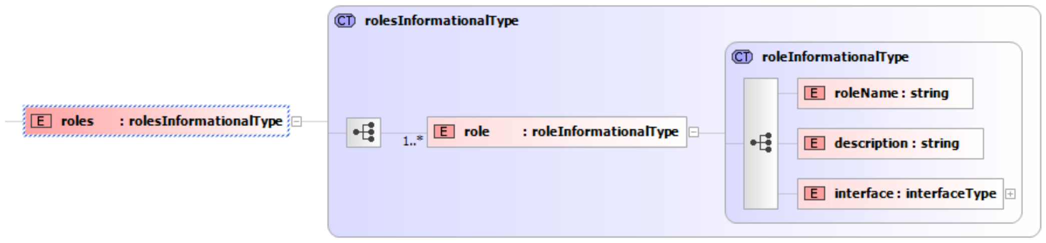
The "StaticTestPerformed" field is depicted in next figure. Reader can appreciate that these are composed by a list of "tests". These tests are composed by the name, the description, the attackModels, the result, the comments and the date of the test.



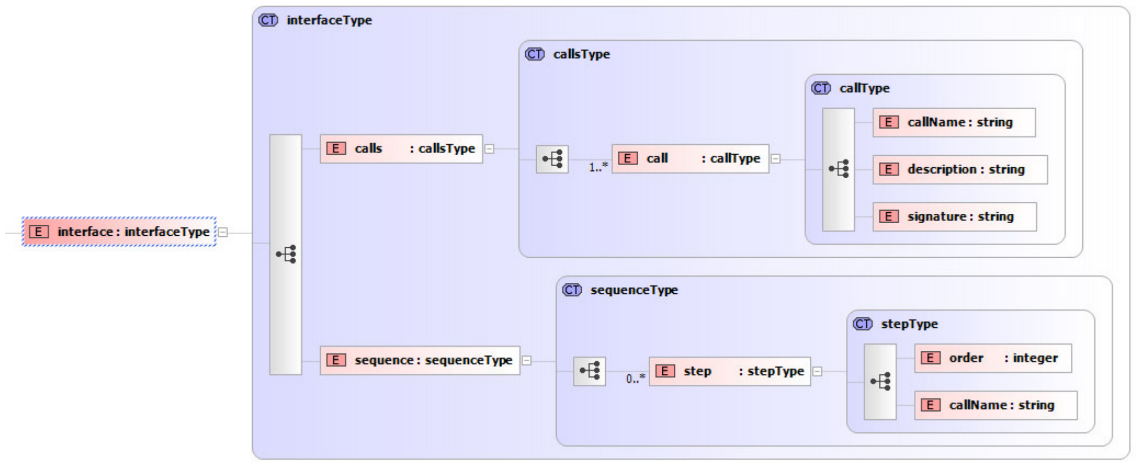
Next figure depicts the "features". This element is composed by a list of features, at least of feature is needed.



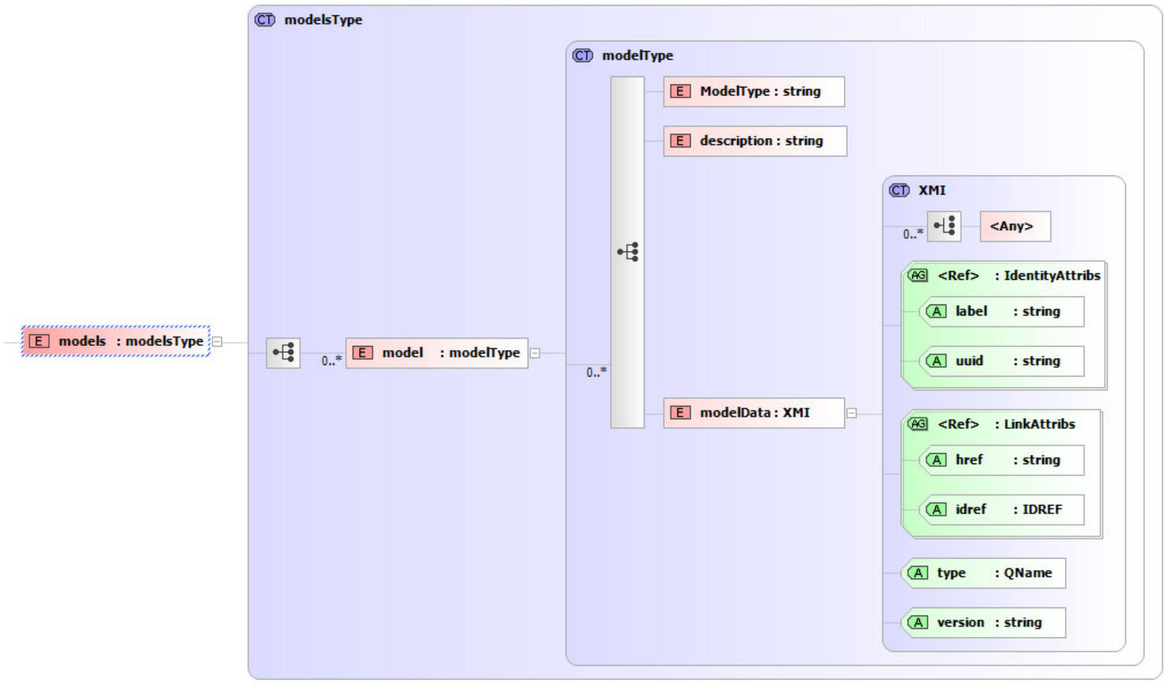
The following figure presents the roles subsection in the informational part section. This element is of type "rolesInformationalType", and it is composed of a list of role elements. Role elements are composed of roleName, description and the interface.



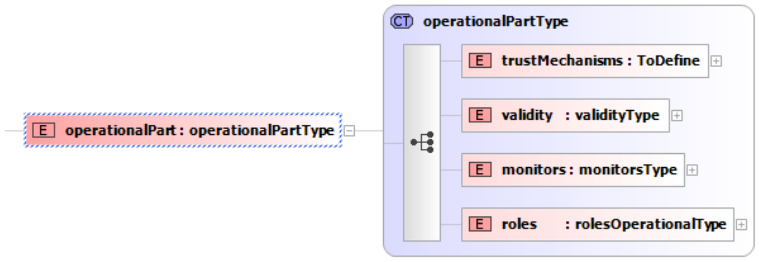
Next figure depicts in detail the "interface" field in the "role" elements. Interface is of type "interfaceType" and it is composed of a list of calls and a sequence element. Every "call" is composed of callName, description and signature. The signature element describes the call by expressing it using the Java standard format. The sequence element is composed of a list of steps. Every step is composed of the order and the callName.



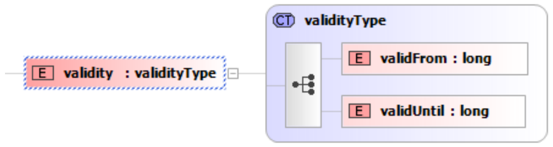
Next figure depicts in detail the model field from the informational part of S&D Pattern language. This field is intended to store information about a TROPOS or a UML model of the S&D Pattern. The "ModelType" element is of type String, but can contain only the strings "UML" or "TROPOS". The element "modelData" is used in order to store the model; it is of type XMI model. The XML Metadata Interchange (XMI) is an OMG standard for exchanging metadata information via Extensible Markup Language (XML). It can be used for any metadata whose metamodel can be expressed in Meta-Object Facility (MOF). The most common use of XMI is as an interchange format for UML models, although it can also be used for serialization of models of other languages (i.e. TROPOS).



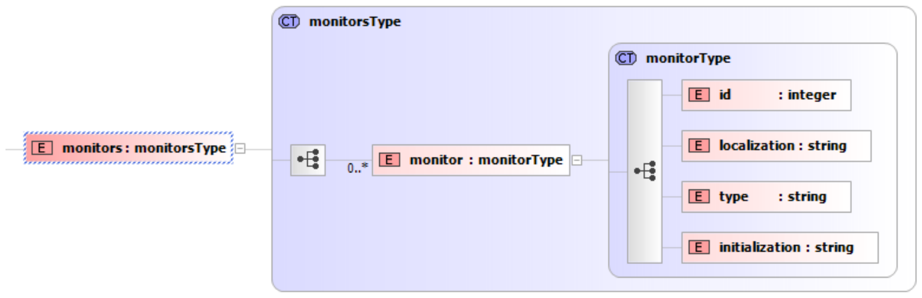
Next figure presents a brief description of the operational part of the S&D Pattern language. This depicts that the operational part is composed by a field for trust mechanisms, other field for the validity period, and a new field for monitors. Finally, there is a section devoted to roles operational information.


operationalPartType
trustMechanisms : ToDefine
validity : validityType
operationalPart : operationalPartType
monitors : monitorsType
roles : rolesOperationalType

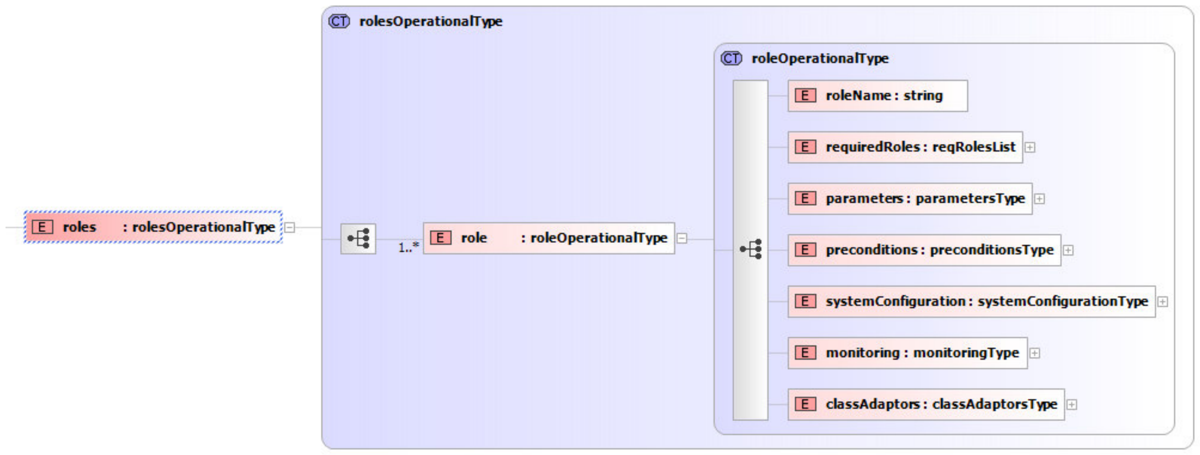
As in the S&D Class language, the validity of S&D Patterns is represented using two elements indicating the period of time in which the S&D Pattern is valid.



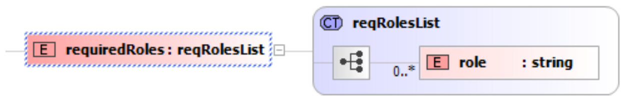
The definition of monitors is located in the operational part of the S&D Pattern language. We can see how a list of monitors is declared. These monitors are composed by an identifier, a location, the type of monitor and some initialization data.



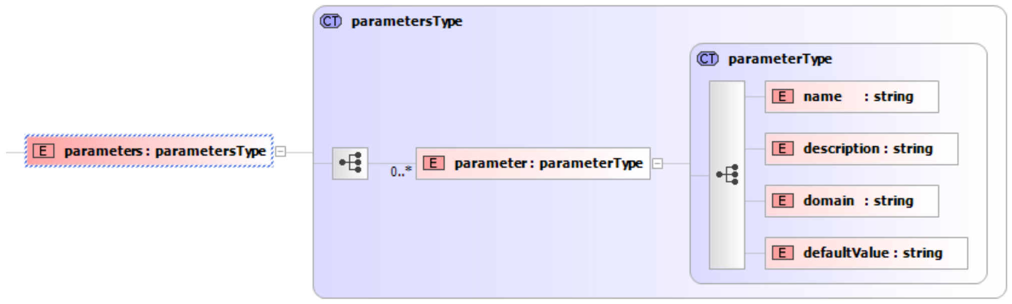
Next figure presents a complete description of the role in the operational part section. This field is composed by the name, a list of required roles, a list of parameters, a list of preconditions to be checked, a list for system configuration, a list of "monitoring" elements (for storing the monitoring rules associated to the role), and finally a list of class adaptors.


roles : rolesOperationalType
rolesOperationalType
1..*
role : roleOperationalType
roleOperationalType
roleName : string
requiredRoles : reqRolesList
parameters : parametersType
preconditions : preconditionsType
systemConfiguration : systemConfigurationType
monitoring : monitoringType
classAdaptors : classAdaptorsType

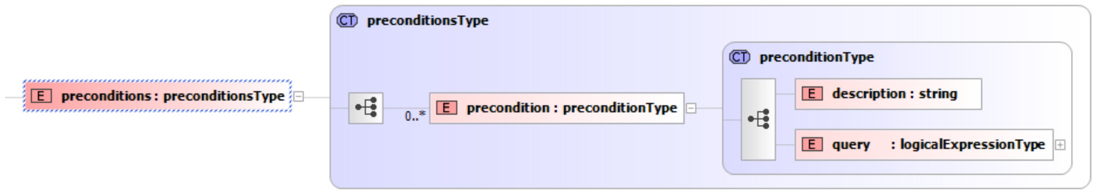
The "requiredRoles" element is of type "reqRolesList". This type is a list of "role" elements; in this case a role is a string. The requiredRoles element is used to describe the complementary roles that need the application of this specific role. For instance, the role "client" in a particular S&D Pattern may require the role "server" in order to operate. Note that from a S&D Pattern role you can require roles from the same S&D Pattern, since an S&D Pattern can only interact with a different instance of itself.



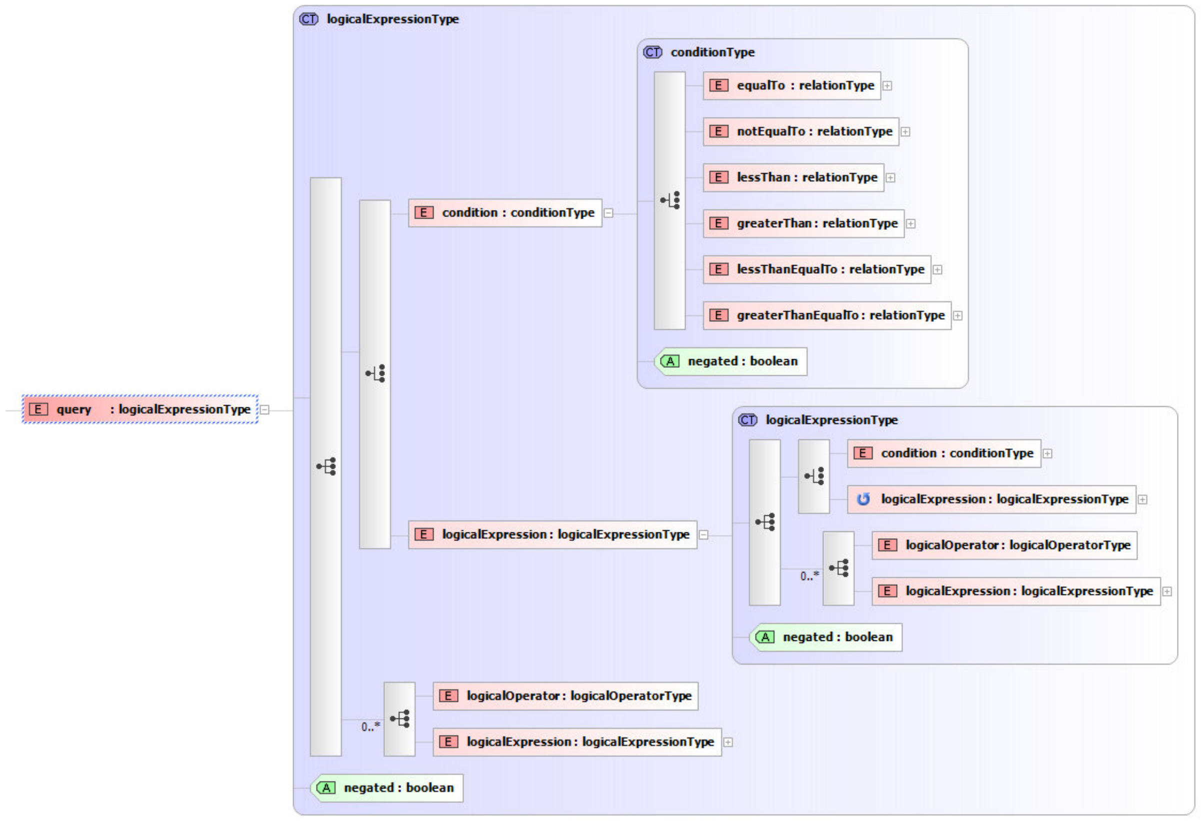
Next figure depicts the parameters field. This is composed by a list from 0 to more than one parameter elements. These parameters are name, description, domain and defaultValue. These elements are used in order to represent the parameters that a S&D Pattern accepts at instantiation time.

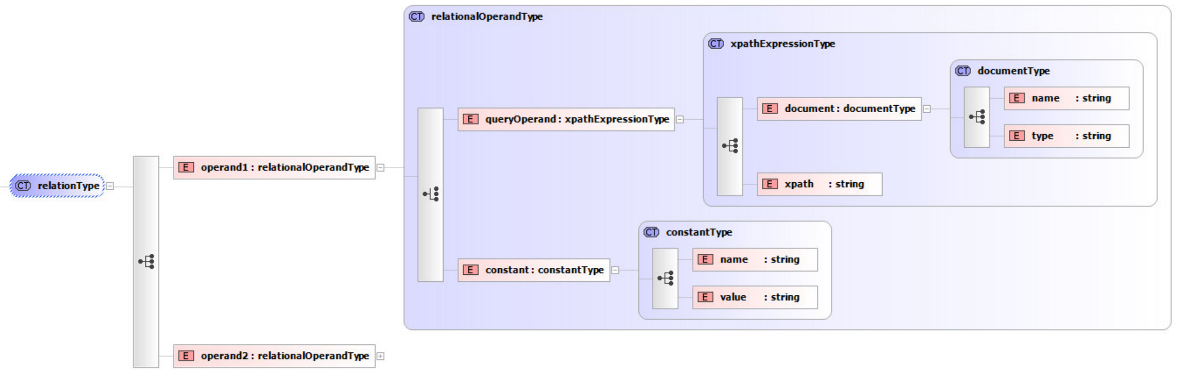


Next figure details the preconditions associated to an S&D Pattern role. Preconditions are composed of a list of 0 to more than 1 precondition elements. Every precondition is composed of a description and a query.

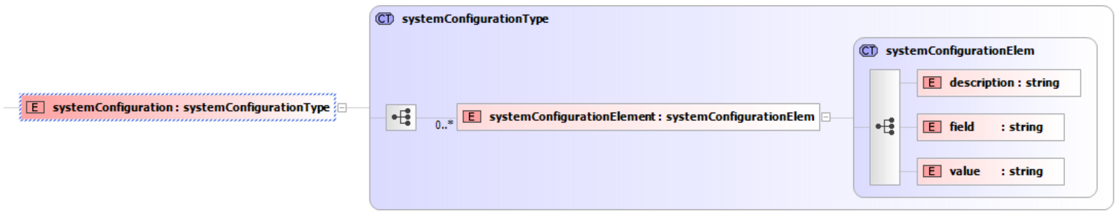

CT preconditionsType
CT preconditionType
E preconditions : preconditionsType
0..*
E precondition : preconditionType
E description : string
E query : logicalExpressionType

Next figure shows a detailed description of queries. Because of queries description is the same that monitoring formulaes reaction conditions, a further description of queries can be found in “A4.R6 - Draft Schema for Recovery Actions to be Taken Upon Potential or Definite Violations of Monitoring Rules”.

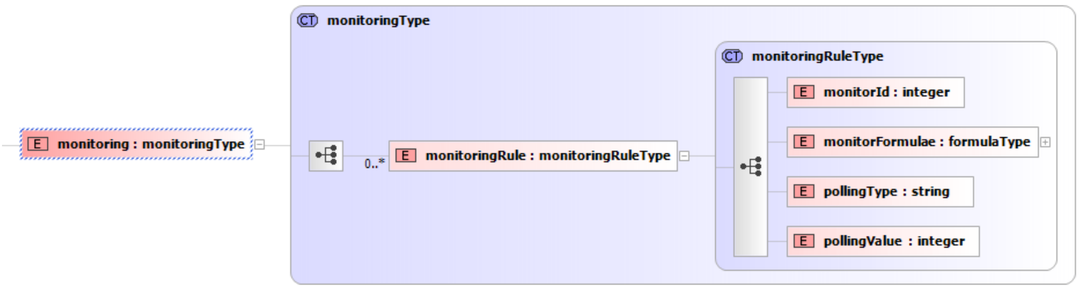

logicalExpressionType
conditionType
equalTo : relationType
notEqualTo : relationType
lessThan : relationType
greaterThan : relationType
lessThanEqualTo : relationType
greaterThanEqualTo : relationType
negated : boolean
query : logicalExpressionType
condition : conditionType
logicalExpressionType
condition : conditionType
logicalExpression : logicalExpressionType
logicalOperator : logicalOperatorType
logicalExpression : logicalExpressionType
logicalExpression : logicalExpressionType
negated : boolean
0..*
logicalOperator : logicalOperatorType
logicalExpression : logicalExpressionType
negated : boolean



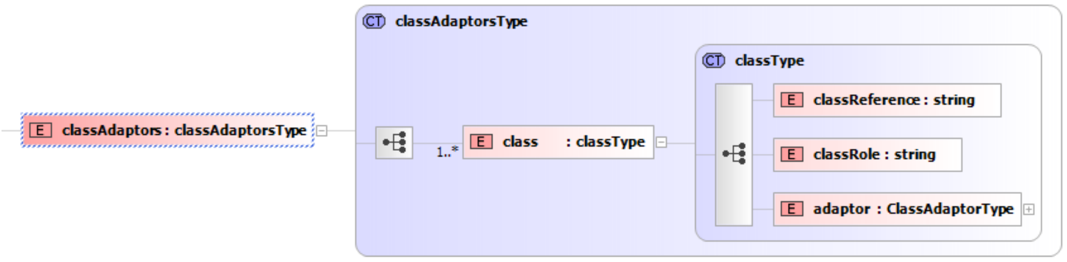
SystemConfiguration field is shown in next figure. These are composed by a list from 0 to more than 1 of systemConfigurationElements. These elements are composed by a description element and a value element. The SystemConfiguration elements allow S&D Patterns developers to store some configuration options in the SRF configuration table.



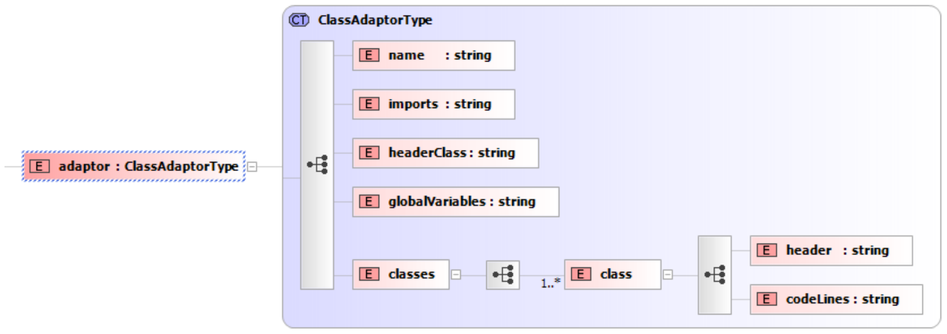
Next figure depicts the monitoring field in detail. This field is composed by a list from 0 to * of monitoringRules. A monitoringRule is a monitorId, a monitorFormulae, a pollingType and a pollingValue. A detailed description of the formulaType can be found in the "A4.R6 - Draft Schema for Recovery Actions to be Taken Upon Potential or Definite Violations of Monitoring Rules" report.



The classAdaptor element is composed by a list of at least one "class" element. A "class" element is composed by a "classReference", a "classRole" and an "adaptor" element.



The adaptor element is composed by the following fields: name, imports, headerClass, globalVariables a classes. Classes are composed by a list of classes, at least one is required. A class field is a header and a codeLines field. The adaptor element is intended to describe the Java code implementing the translation between the S&D Class interface to the S&D Pattern interface. We have included tags in order to separate this Java code.



4.6.1. Representing Integration Schemes

Integrations Schemes are a special type of S&D Pattern. Integration Schemes are used in order to describe 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.

From S&D Solution developer point of view, the development of an Integration Scheme follows the same process than the development of S&D Patterns. This is since both artefacts use the same XML schema definition. At the implementation level of Integration Schemes (and consequently, in the executable component implementing it) is where the composition of solutions S&D Is performed. Doing this, from the point of view of the SRF, an executable component implementing an integration Scheme 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, requesting to the SRF for the S&D Patterns needed. These executable components implement both the requests to the SRF and the use of the corresponding S&D Patterns. On one hand, they offer to applications the integration Scheme interface, and in the other hand, they make use of the interfaces provided by the S&D Patterns that they are composing. It is important to remark that an integration scheme may combine S&D Patterns, S&D Classes and even Integration Schemes. Figure 5 presents a sequence diagram showing the activation of an integracion scheme that composes two patterns.

Firstly, the application, in the left side of the diagram, sends an SDRequest to the SRF. Note, that in this case the application has requested an artefacts of type pattern (P:). This is because, applications ask for S&D Solutions but they are not aware if the S&D Solutions are implemented by a S&D Pattern or by an Integration Scheme. In any case, from both, SRF and application, point of view S&D Patterns and Integration Schemes have the same behaviour. Secondly, the SRF analyses the SDRequest and performs the selection algorithm, this results in the activation of an Executable Component implementing the Integration Scheme. Once that the Integration Scheme implementing Executable is running, it sends SDRequests when it needs to use the S&D Patterns under combination. This issue is depicted in the diagram by two SDRequests sent by the Integration Scheme Executable Component to the SRF. Each of the SDRequests, results in the activation of an Executable Component implementing an S&D Pattern. Finally, the Integration Scheme Executable Component accesses to the S&D Patterns functionalities.

There are three points worth to be highlighted. First, the Integration Scheme presented in the diagram shown in Figure 13 starts its execution by requesting the two patterns that it combines. As a general rule, Integration Schemes request S&D Patterns when they need them. Second, S&D Patterns preconditions are checked when they are requested. Consequently, it is possible that an Integration Scheme tries to use an S&D Pattern which is not applicable for the given context conditions. Doing this, there is a clear separation between Integration Scheme preconditions, which corresponds to preconditions of the S&D Solution they represent and S&D Patterns under combination preconditions. These latter ones need to be fulfilled by the atomic components of the S&D Solutions represented by the Integration Scheme. And third, Integration Schemes could be used to create S&D Solutions combining S&D Classes, S&D Patterns, S&D Implementations and event Integration Schemes.

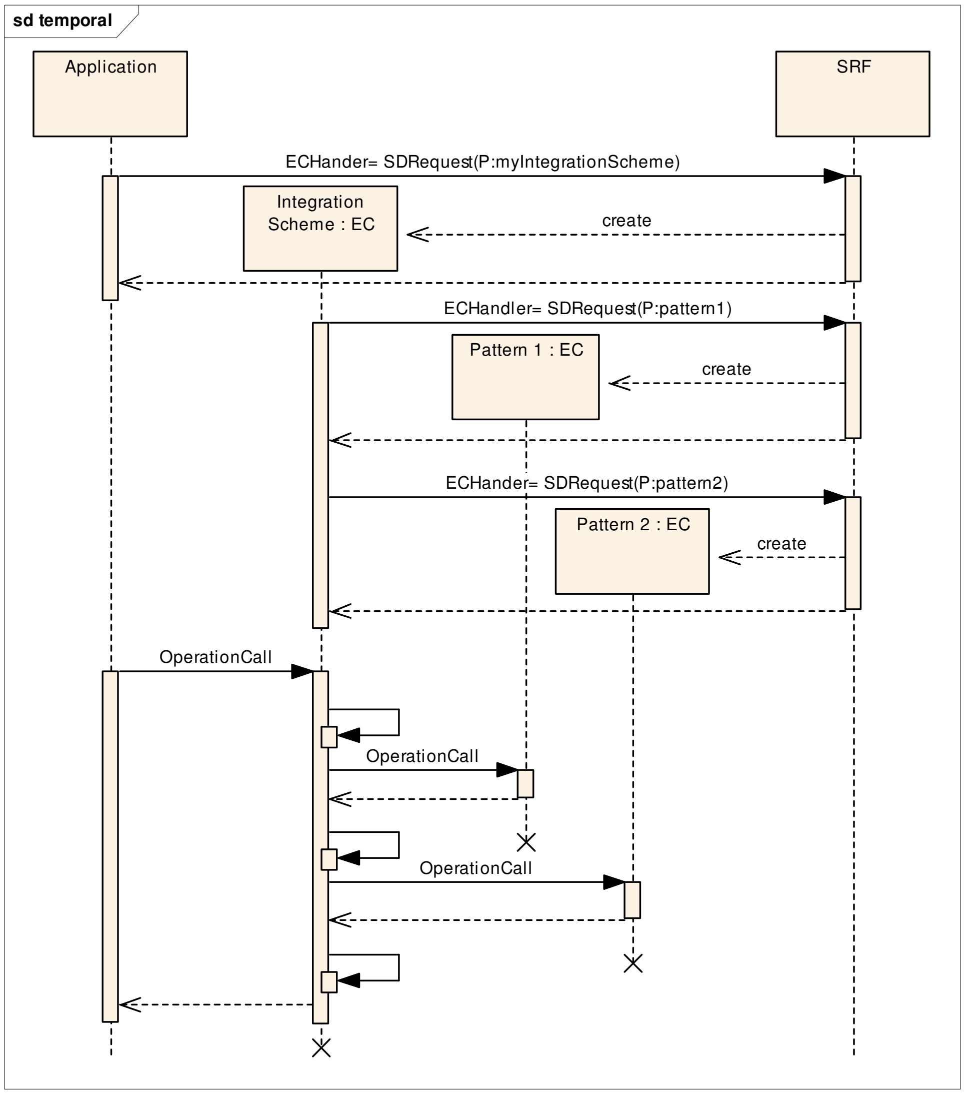


Figure 13: Sequence diagram showing an Integration Scheme that combines two S&D Patterns (Pattern 1 and Pattern 2)

4.6.2. 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 5). This language is an extension of EC-Assertion – an event calculus (EC **Erreur ! Source du renvoi introuvable.**) 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 **Erreur ! Source du renvoi introuvable.** and **Erreur ! Source du renvoi introuvable.**. 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 [10] in order to provide a standard way of expressing the event calculus (EC) formulas that will be monitored. This schema is described in this section 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.

4.6.3. 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:

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

,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.

4.6.4. 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 14.
            - *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.

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

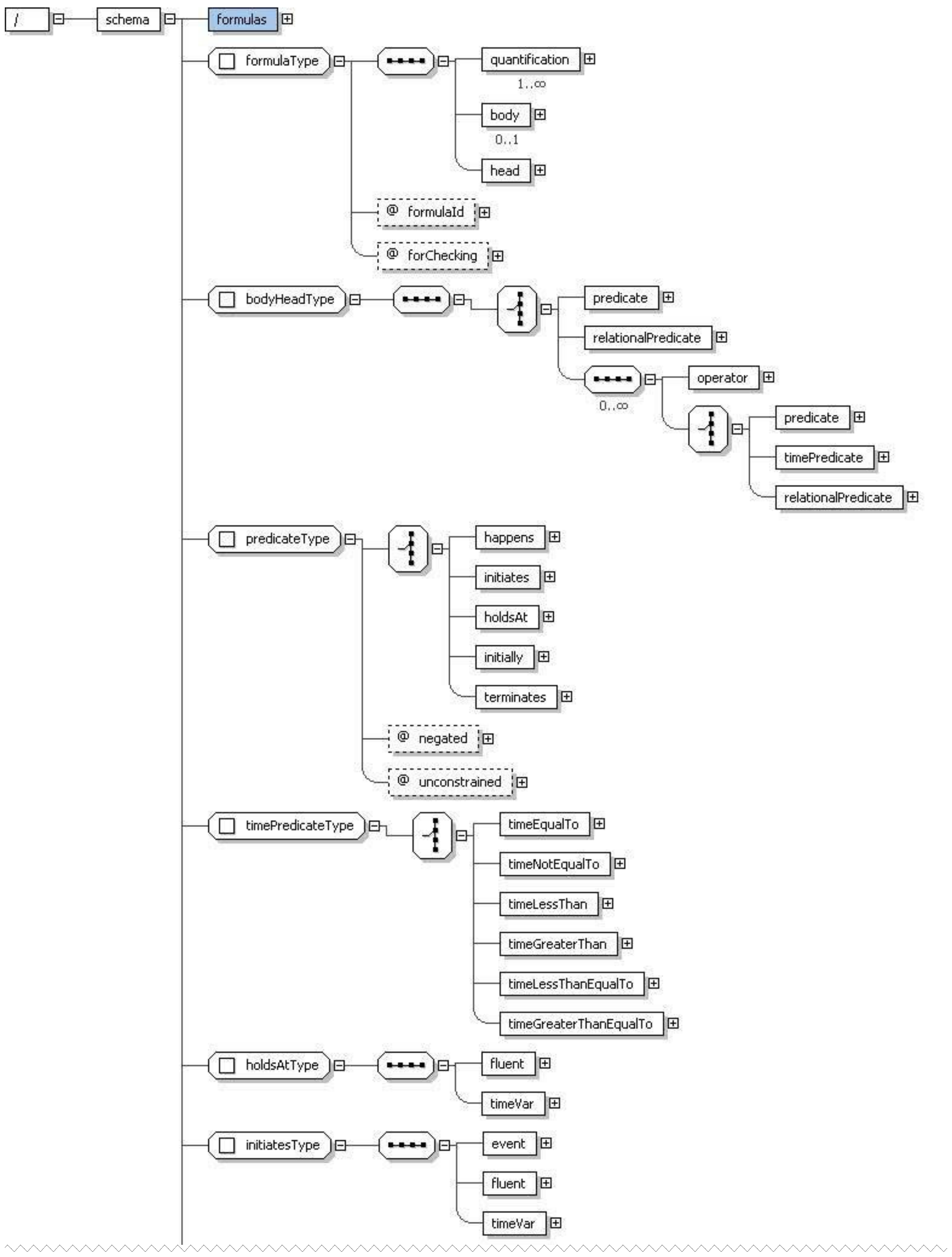


Figure 14 – XML Formula Representation Schema (I)

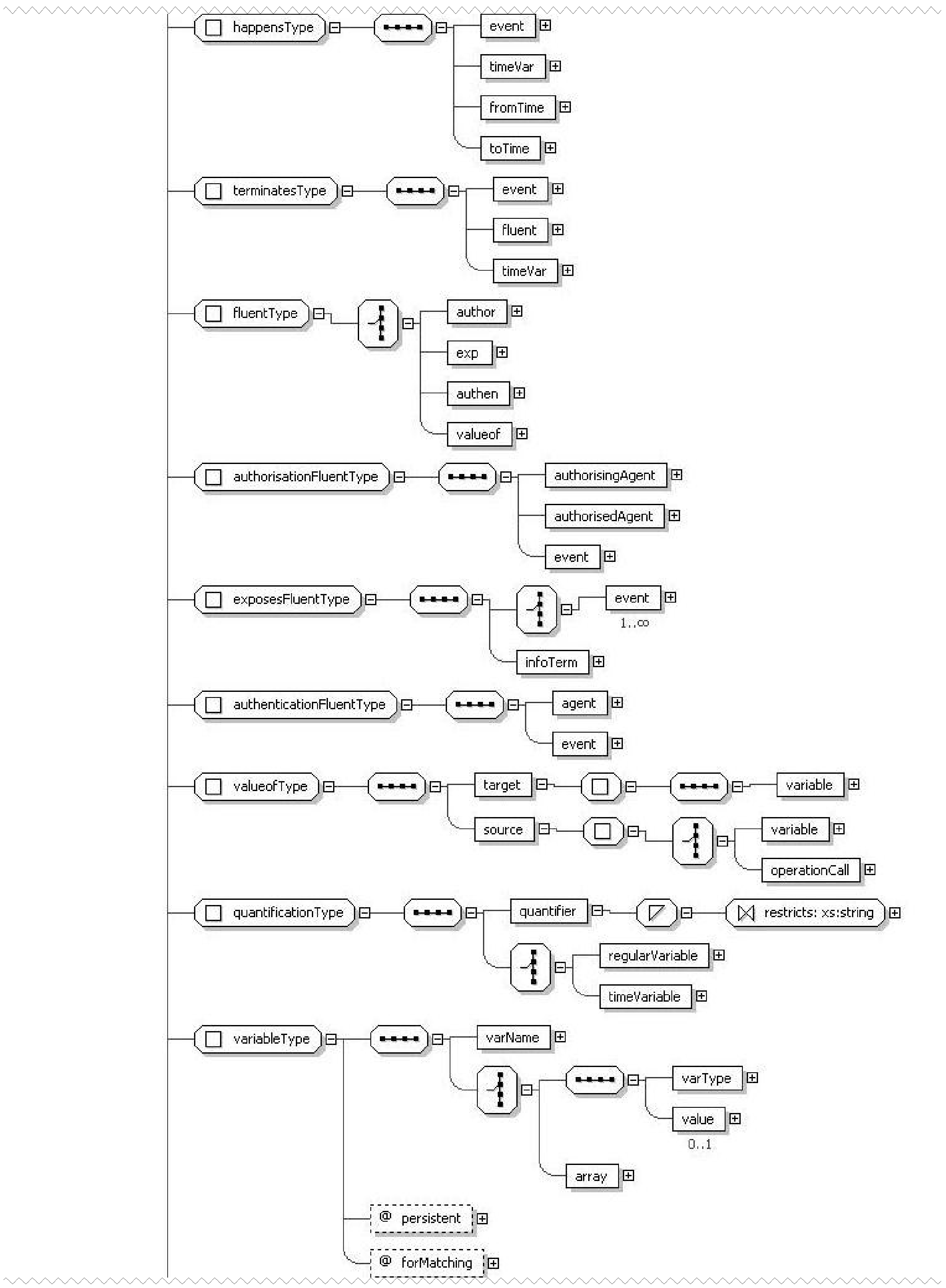


Figure 15 – XML Formula Representation Schema (II)

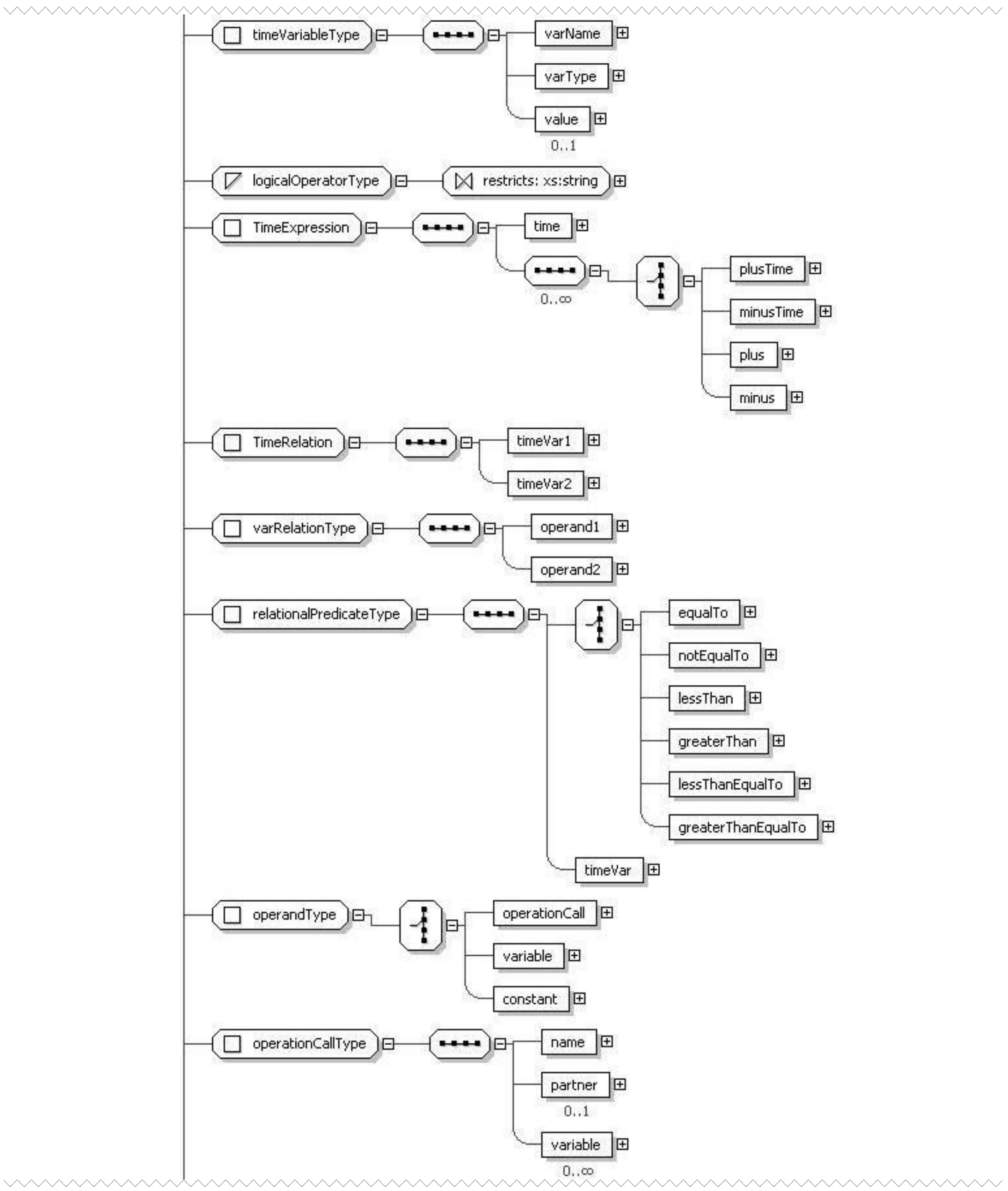


Figure 16 – XML Formula Representation Schema (III)

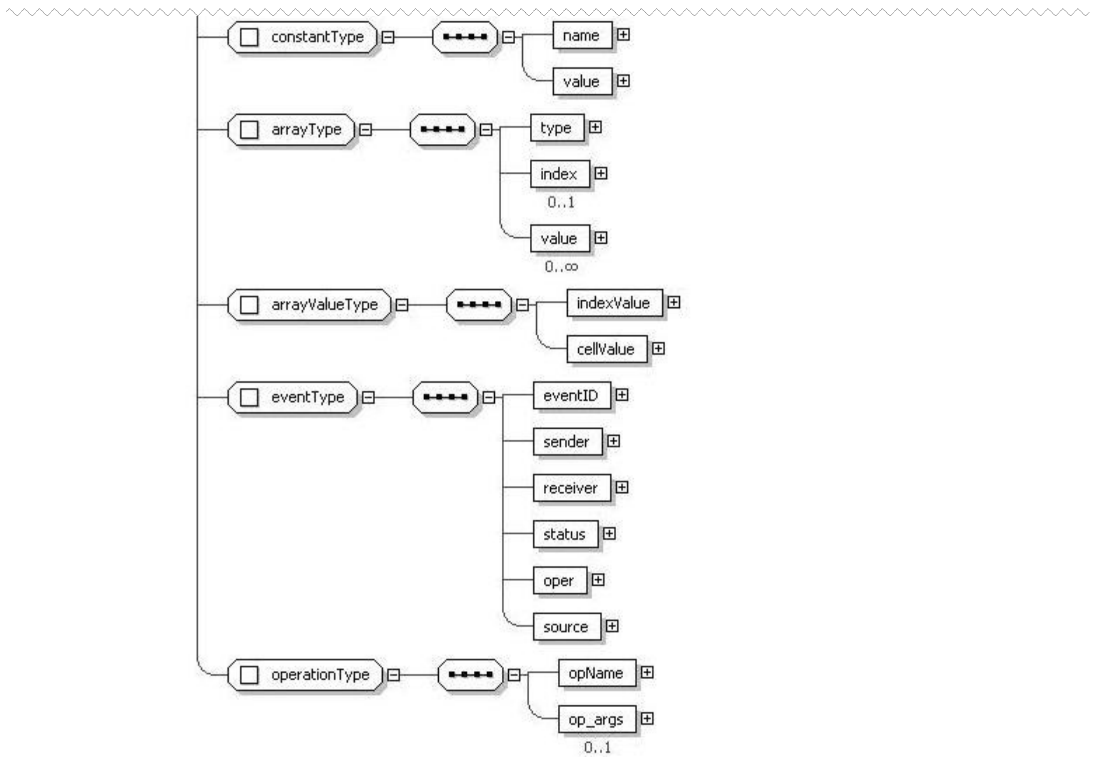
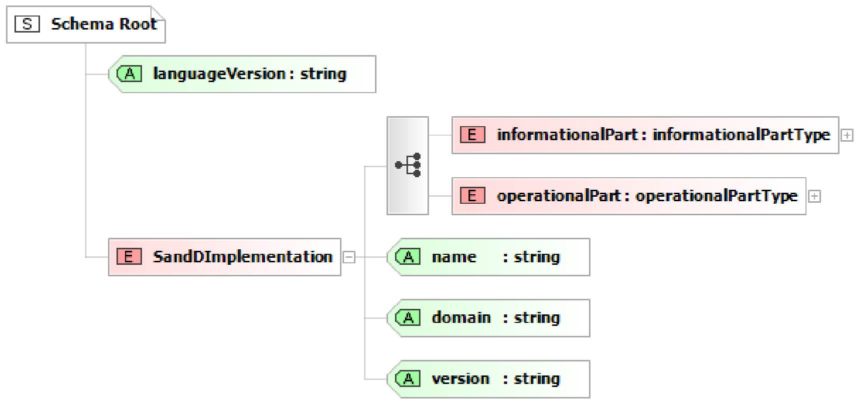


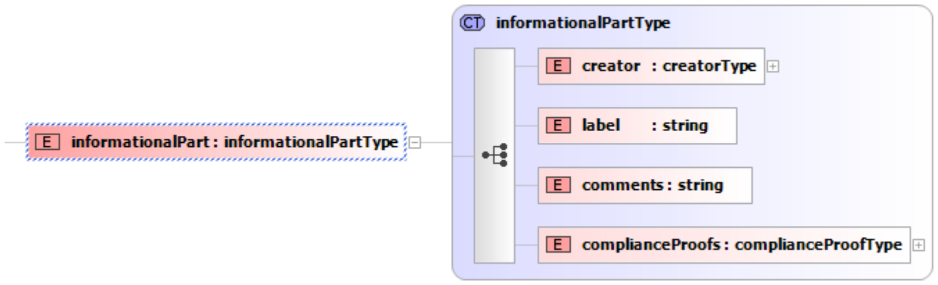
Figure 17 – XML Formula Representation Schema (and IV)

4.7. Detailed description of S&D Implementations

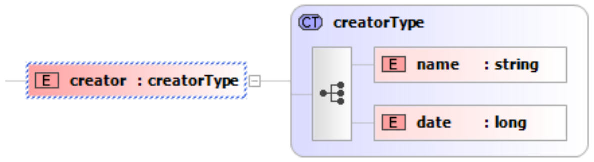
This section presents the S&D Implementation language. As in the rest of artefact languages this language is structured in two sets of fields: the informational part and the operational part. At a first glance the S&D Implementation language is composed by name, domain and version. The rest of fields are separated in the aforementioned two parts.


S Schema Root
A languageVersion : string
E informationalPart : informationalPartType
E operationalPart : operationalPartType
E SandDImplementation
A name : string
A domain : string
A version : string

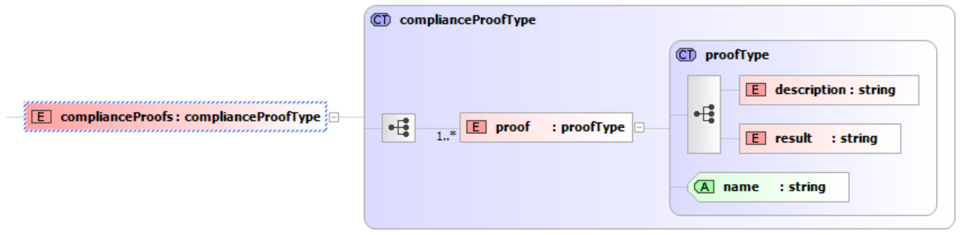
A global view of the informational part is in the next figure. Thus, it shows that this field is composed by the creator, label, comments and a field for complianceProofs.


informationalPart : informationalPartType
informationalPartType
creator : creatorType
label : string
comments : string
complianceProofs : complianceProofType

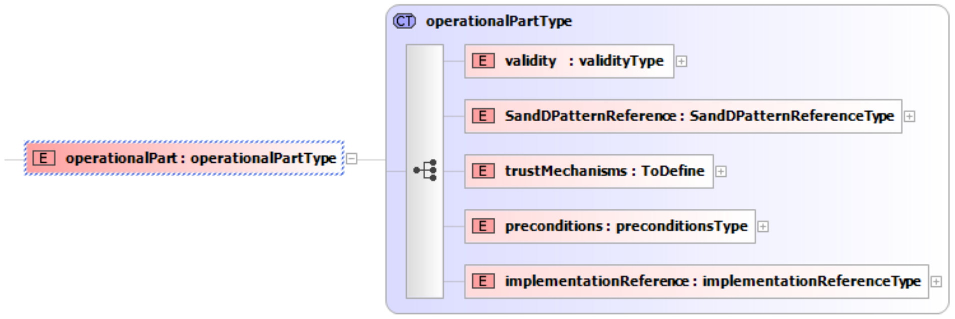
Next figure depicts in detail the creator field.



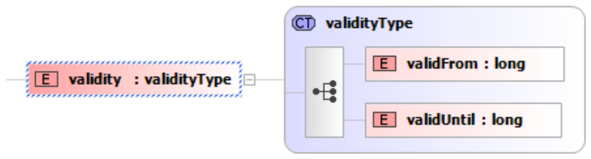
The following picture shows the complianceProofs field composition.



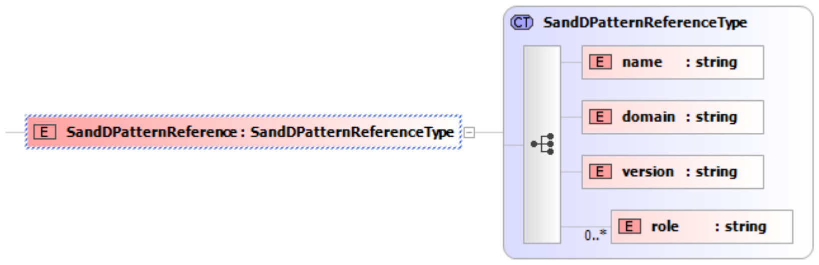
The next figure presents an overview of the operational part of the S&D Implementation language. It is composed of the following elements: validity, SandDPatternReference, trustMechanisms, preconditions, and implementation reference.


operationalPart : operationalPartType
operationalPartType
validity : validityType
SandDPatternReference : SandDPatternReferenceType
trustMechanisms : ToDefine
preconditions : preconditionsType
implementationReference : implementationReferenceType

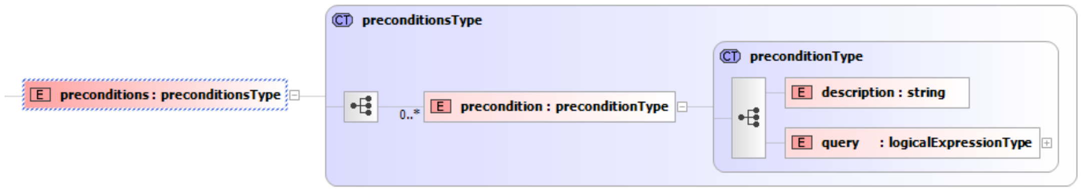
The validity of the S&D Implementation is stored in the validity element. As in the rest of the artefact language this element is composed of two elements.



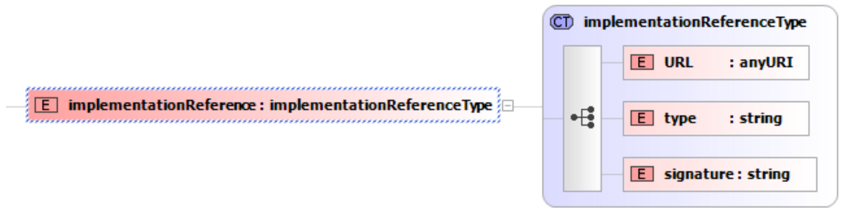
The SandDPatternReference element is used in roder to reference to the pattern implemented by this S&D Implementation. It contains the name, the domain, and the version of the pattern, and a list of the implemented roles.


SandDPatternReference : SandDPatternReferenceType
SandDPatternReferenceType
name : string
domain : string
version : string
0..*
role : string

Next figure presents the preconditions field. It has the same meaning and the same structure than in the S&D Pattern language.


preconditions : preconditionsType
preconditionsType
0..*
precondition : preconditionType
preconditionType
description : string
query : logicalExpressionType

Finally, the next figure presents the "implementationReference" element. This element is used in order to reference the ExecutableComponent implementing this S&D Implementation.


implementationReference : implementationReferenceType
implementationReferenceType
URL : anyURI
type : string
signature : string

4.8. Language used for expressing preconditions

Preconditions contain the specification of the context conditions under which an S&D artefact (S&D Patterns and S&D Implementations) is applicable or not. Consequently, preconditions express particular conditions that can be check by evaluating the content of the SRF's context manager database. S&D Pattern & S&D Implementation languages include fields for expressing preconditions. Since preconditions are evaluated at run-time by the SRF, this field is part of the operational information in the artefacts language, its structure is the following:

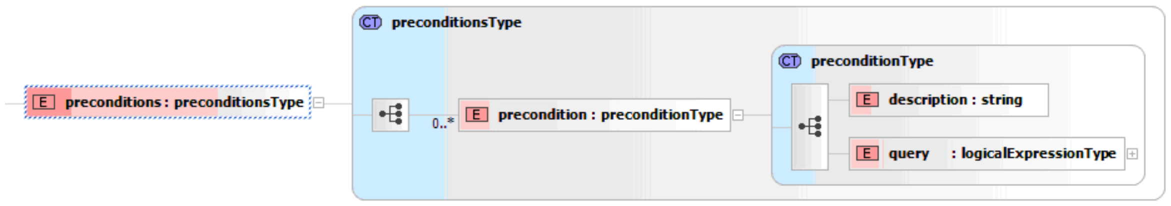

preconditionsType
preconditionType
preconditions : preconditionsType
0..*
precondition : preconditionType
description : string
query : logicalExpressionType

Figure 18: The preconditions field is a list of preconditions.

The precondition field is a of type preconditionsType, this type is a list of elements of type

preconditionType. The preconditionType type has to elements. On the one hand, description of type string and containing a description of the precondition (just to make it more human-readable). On the other hand, query of type logicalExpressionType, containing the actual precondition.

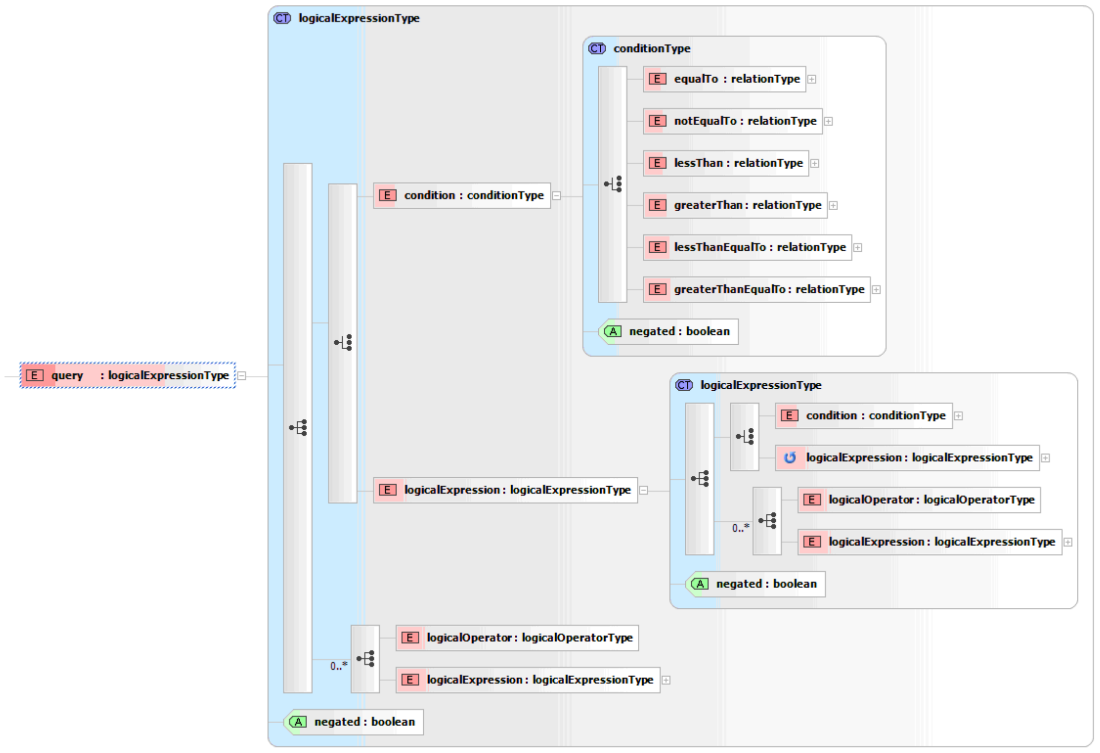

logicalExpressionType
conditionType
equalTo : relationType
notEqualTo : relationType
lessThan : relationType
greaterThan : relationType
lessThanEqualTo : relationType
greaterThanEqualTo : relationType
negated : boolean
condition : conditionType
query : logicalExpressionType
logicalExpressionType
condition : conditionType
logicalExpression : logicalExpressionType
logicalOperator : logicalOperatorType
logicalExpression : logicalExpressionType
0..*
negated : boolean
logicalExpression : logicalExpressionType
logicalOperator : logicalOperatorType
logicalExpression : logicalExpressionType
0..*
negated : boolean

Figure 19: LogicalExpressionType type structure.

The logicalExpressionType type has an attribute expressing if the query is negated or not. It is of type list containing one element condition (of conditionType type) or logicalExpression (of logicalExpressionType type) plus a list of zero or more pairs composed by a logicalOperatior (of logicalOperatorType) and a logicalExpression (of logicalExpressionType) Figure 2 presents the structure of the conditionType type. The structure of the relationType type is presented in figure 3.

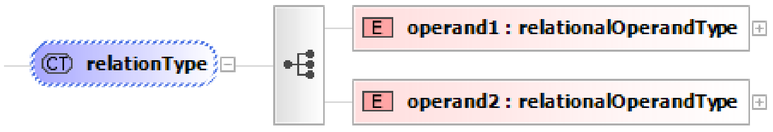

CT relationType
E operand1 : relationalOperandType
E operand2 : relationalOperandType

Figure 20: relationType type.

The relationalType type is presented in Figure 4. The document element of documentType type is reserved for future uses. The type logicalExpressionType is also used for the guard conditions of monitoring rules specification, [11] presents a more detailed description of it. It is important to note that the name element of the constantType represents the type of the constant, its possible values are: "NUMERICAL", "STRING", "BOOLEAN", and "UNDEFINED".

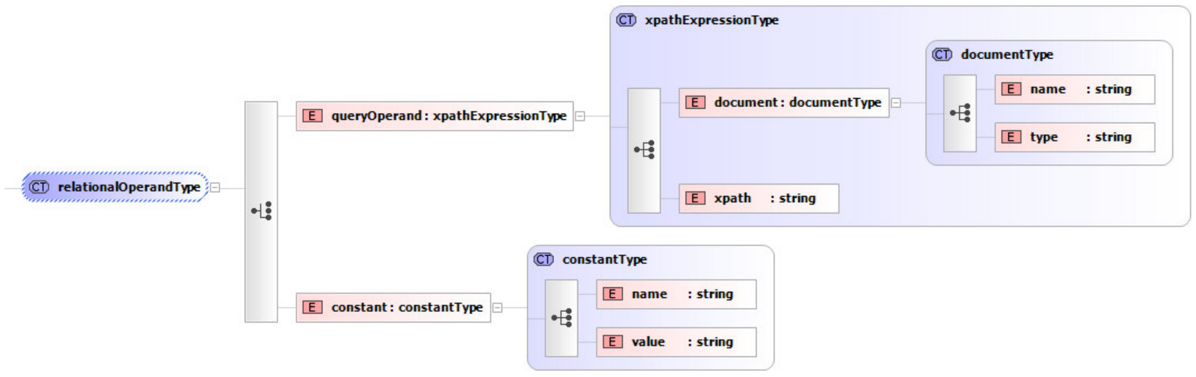


Figure 21: relationalOperandType type.

As aforementioned, preconditions have a close relation to the Context Manager database. Figure 5 presents this database structure. It is composed of seven tables, but the most important ones, from the preconditions point of view, are the ActivePatternList and the EventsHistory tables. The ActivePatternList stores the list of current active S&D Patterns, it also stores S&D Patterns activated in the past. The EventsHistory table keeps a record of the monitoring rules that have been violated. Following the relation between the EventHistory table and the ActivePatternList is possible to derive which S&D Pattern has generated every monitoring rule violation. S&D Pattern developers may use the rest of the tables in their preconditions. Consequently, they can create as complex preconditions as they need.

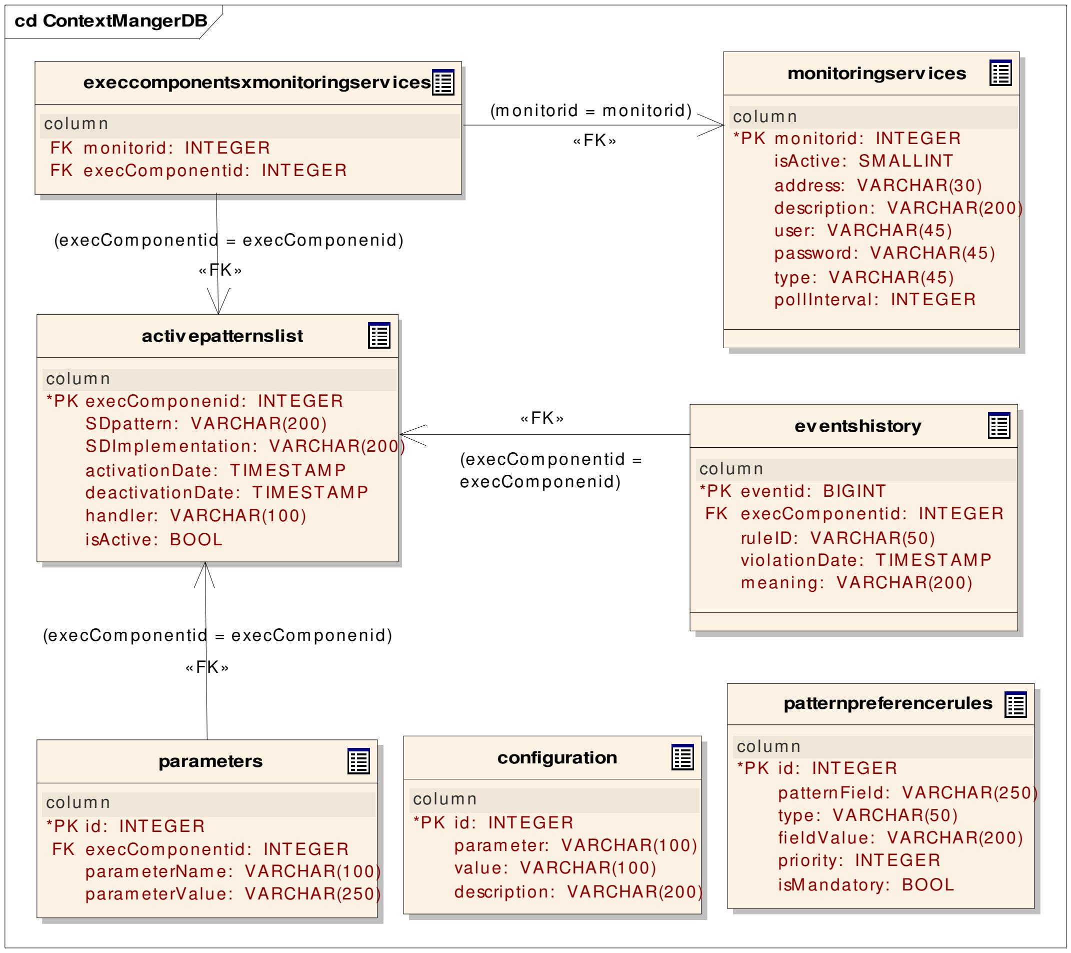


Figure 22: Logical model of the Context Manager database.

According to S&D artefact languages preconditions are expressed using xpath syntax. The Serenity Run-time Framework uses the following XML view of the context manager database. Because of this xpath queries in preconditions may be compliant to this XMLShema (figures 6 and 7).

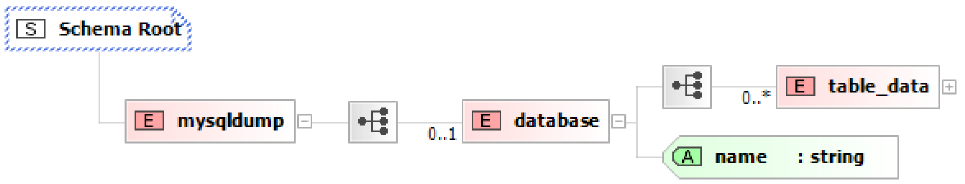


Figure 23: XML schema definition of context manager database (part 1 of 2).

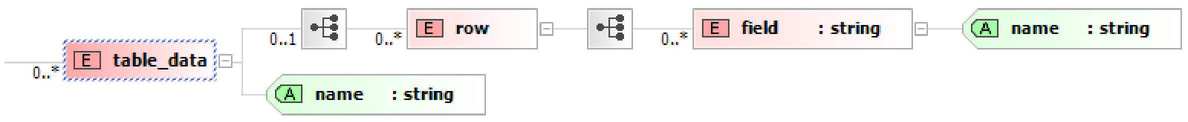


Figure 24: XML schema definition of context manager (part 2 of 2).

In these XML files, table_data elements have an attribute which is the name of the table represented by the node. Possible values for this attribute can be consulted in Figure 5. Also, this figure includes possible values for the name attribute in row elements.

The following code presents an example XML representing a partial view of a context manager.

```
<?xml version="1.0"?>
<mysqldump xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<database name="contextmanager">
        <table_data name="activepatternslist">
        <row>
                <field name="execComponentId">41</field>
                <field name="SDpattern">TextAuthenticationJava</field>
                <field name="SDImplementation">TextAuthentication</field>
                <field name="activationDate">2008-03-05 18:24:50</field>
                <field name="deactivationDate">2002-12-31 23:00:00</field>
                <field name="handler"></field>
                <field name="isActive">1</field>
        </row>
        <row>
                <field name="execComponentId">466</field>
                <field name="SDpattern">TripleDESEncryptJava.uma.es</field>
                <field name="SDImplementation">TripleDESEncrypt.uma.es</field>
                <field name="activationDate">2008-06-26 13:08:08</field>
                <field name="deactivationDate">2002-12-31 23:00:00</field>
                <field name="handler"></field>
                <field name="isActive">1</field>
        </row>
        </table_data>
        <table_data name="eventshistory">
        <row>
                <field name="eventId">32</field>
                <field name="execComponentId">812</field>
                <field name="ruleId">1111222</field>
                <field name="violationDate">2008-09-23 03:08:08</field>
                <field name="meaning">SmartCardConnected</field>
        </row>
        </table_data>
</database>
</mysqldump>
```

The rest of this section introduces some examples. Firstly, we provide some examples of Xpath queries. Secondly, we show how these Xpath queries are integrated into XML formatted preconditions.

— 1) A rule with the ruleId id has been violated.

— //table_data[@name="eventshistory"]/row/field[@name="ruleId"]/text()="ruleId"

— 2) This query returns true or false depending on the violation of the rule ruleId with the STATUS status.

— //table_data[@name="eventshistory"]/row[field[@name="ruleId"]/text()="ruleId"]/field[@name="meaning"]="status"

- 3) This query returns the STATUS resulting in the last violation of rule ruleId.
- //table_data[@name="eventshistory"]/row[field[@name="ruleId"]/text()="ruleId"][last()]/field[@name="meaning"]/text()

- 4) This query returns the ECid of the Executable component responsible of the violation of the monitoring rule ruleId.
- //table_data[@name="eventshistory"]/row[field[@name="ruleId"]/text()="ruleId"][last()]/field[@name="execComponentId"]/text()

- 5) This query return the SDPattern name given a Executable Componend Ecid.
- //table_data[@name="activepatternslist"]/row[field[@name="execComponentId"]/text()="ECid"]/field[@name="SDpattern"]/text()

- 6) This query return the SDImplementation name given a Executable Componend Ecid.
- //table_data[@name="activepatternslist"]/row[field[@name="execComponentId"]/text()="ECid"]/field[@name="SDImplementation"]/text()

- 7) This query return true if the given SDPattern name is active.
- //table_data[@name="activepatternslist"]/row[field[@name="SDpattern"]/text()="SDPatternName"]/field[@name="isActive"]/text()=1

- 8) This query return true if the given SDImplementation name is active.
- //table_data[@name="activepatternslist"]/row[field[@name="SDImplementation"]/text()="SDImplementation"]/field[@name="isActive"]/text()=1

- 9) This query returns true if the given SDPattern was active in the past.
- //table_data[@name="activepatternslist"]/row[field[@name="SDpattern"]/text()="SDPatternName"][last()]/field[@name="isActive"]="0"

- 10) This query returns true if the given SDImplementation was active in the past.
- //table_data[@name="activepatternslist"]/row[field[@name="SDImplementation"]/text()="SDImplementationName"][last()]/field[@name="isActive"]="0"

These xpath queries are integrated into XML formatted preconditions as presented in the following example:

```
          <precondition>
            <description>
                         This precondition is satisfied if there is no instances of the
```

```
                          smartCardConnected.uma.es pattern running in the system.
             </description>
             <query negated="1">
               <condition negated="0">
                 <equalTo>
                   <operand1>
                     <constant>
                       <name></name>
                       <value>true</value>
                     </constant>
                   </operand1>
                   <operand2>
                     <queryOperand>
                       <document>
                         <name>NA</name>
                         <type>NA</type>
                       </document>
                       <xpath>
                             //table_data[@name="activepatternslist"]/row[field[@name="SDpattern"]
                             /text()="smartCardConnected.uma.es"]/field[@name="isActive"]/text()=1
                       </xpath>
                     </queryOperand>
                   </operand2>
                 </equalTo>
               </condition>
             </query>
           </precondition>
```

5. Example

This section presents as example some artefacts developed using the languages introduced in this deliverable. Figure 25 depicts some S&D artefacts implementing user authentications S&D solutions. The example proposed in this section is based on the development of the left side artefacts (those painted in blue).

The first level, on top of the figure, is composed by two S&D Classes. Firstly the "UserAuthentication" S&D Class represents an abstract S&D Solution for user authentication. Secondly, the "Observer" S&D Class represent a special S&D Solution that does not provide any security service but accountability. This S&D Class is used to group all S&D Patterns performing only context checking tasks. All S&D Patterns send events during its execution, but observers S&D Patterns only send events and they do not perform more actions. They are used in order to check particular context conditions.

The second level is composed by four entities, three S&D Patterns and an Integration Scheme. There are two S&D Patterns that represent particular implementations of the S&D Solution represented by the "UserAuthentication" S&D Class ("TextAuthentication" and "smartCardAuthentication"). These S&D Patterns provide user authentication by means of different ways. "TextAuthentication" S&D Pattern requests login and password from user to be authenticated by means of a small GUI window. Nevertheless, "SmartCardAuthentication" S&D Pattern performs a smartcard based authentication. In this case, a Personal Identification Number (PIN) is requested to the user and then the smartcard processes the authentication. The "smartCardConnected" S&D Pattern represents a S&D Solution consisting on checking whether a smartcard is properly connected to the system, and it belongs to the "Observers" S&D Class. This pattern is able to generate two different events; concretely they are called "SmartCardConnected" and "SmartCardDisconnected". Finally, the smartCardAuthenticationIS Integration Scheme belongs to the "UserAuthentication" S&D Class, since it implements the same S&D Solution, but in this case by means of the combination of two S&D Patterns. This Integration Scheme combines the "smartCardConnected" and "SmartCardAuthentication" S&D Patterns, doing this we guarantee that these S&D Patterns will run together just in those cases where the SRF does not run the "smartCardConnected" S&D Pattern automatically (S&D Patterns can be ran automatically by SRFs depending on the SRF configuration).

The rest of the figure presents an S&D Implementation for every S&D Pattern and an Executable Component for every S&D Implementation. This section presents an implementation by means of tables of the "UserAuthentication" S&D Class, "TextAuthentication" S&D Pattern, and "TextAuthenticationJava" S&D Implementation.

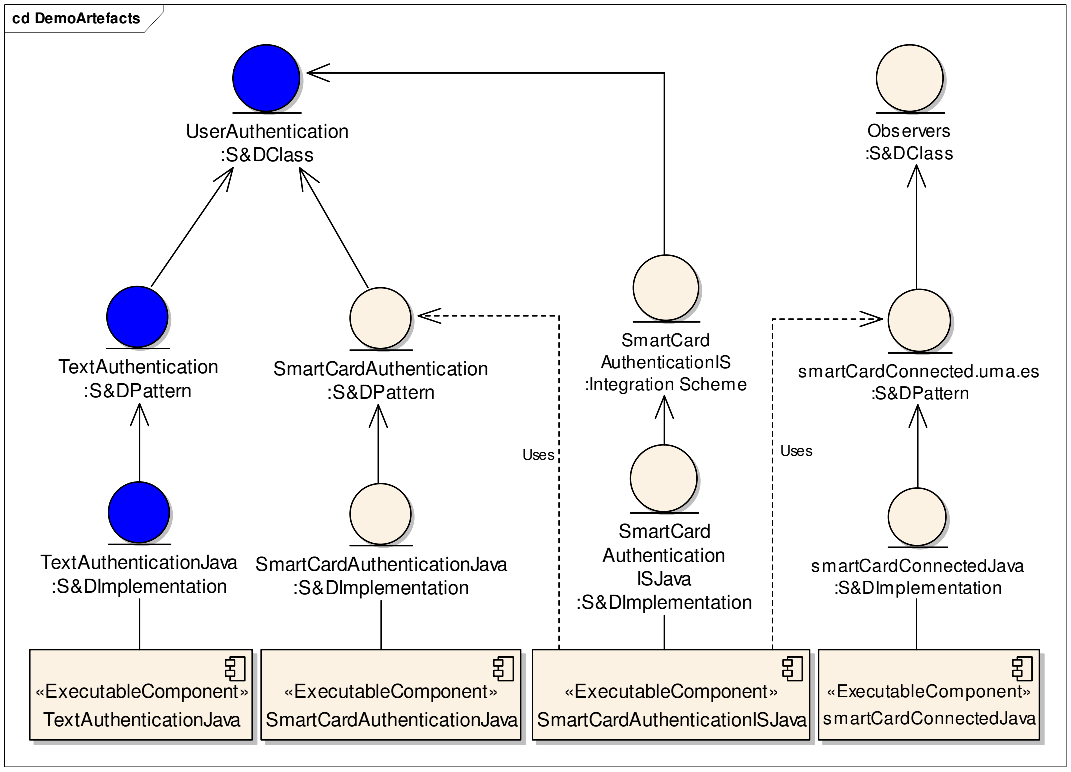

cd DemoArtefacts
UserAuthentication :S&DClass
Observers :S&DClass
TextAuthentication :S&DPattern
SmartCardAuthentication :S&DPattern
SmartCard AuthenticationIS :Integration Scheme
smartCardConnected.uma.es :S&DPattern
Uses
Uses
TextAuthenticationJava :S&DImplementation
SmartCardAuthenticationJava :S&DImplementation
SmartCard Authentication ISJava :S&DImplementation
smartCardConnectedJava :S&DImplementation
«ExecutableComponent» TextAuthenticationJava
«ExecutableComponent» SmartCardAuthenticationJava
«ExecutableComponent» SmartCardAuthenticationISJava
«ExecutableComponent» smartCardConnectedJava

Figure 25: Set of artefacts developed for the examples section.

5.1. S&D Class

In this section a complete example of an S&D Class is presented with the new version of the language. Next table describes the structure of the S&D Class representing an authentication S&D solution. The "*UserAuthentication.uma.es*" S&D Class represents the authentication security mechanism providing the *authentication.uma.es* S&D property. This S&D Class offers an interface with only one call: *"Authentication"*.

S&D Class: UserAuthentication.uma.es	
1	**Name:** UserAuthentication
2	**Domain:**uma.es
3	**Version:**2.0
4	**InformationalPart**
4.1	**Creator**
4.1.1 **4.1.2**	**Name:** uma.es **Date:** 1214750275
4.2	**Label:** This class provides a simple way to authenticate users, it has only one function that results in a Boolean value indicating if the authentication was successfully or not.
4.3	**Comments:**TODO, text explaining more additional information
4.4	**Provided Properties**
4.4.1 **4.4.1.1** **4.4.1.2** **4.4.1.3**	**Property** **Name:** authentication **Domain:** uma.es **Version:**1.0
4.4.1.4	**Timestamp:** 1214750275
4.5	**Solution Features**
4.5.1	**Feature:**Authentication
4.6	**Roles**
4.6.1 **4.6.1.1** **4.6.1.2** **4.6.1.3**	**Role** **Name:** authentication **Description:** uma.es **Interface**
4.6.1.3.1	**Calls**
4.6.1.3.1.1	**Call**
4.6.1.3.1.1.1	**callName:**authentication
4.6.1.3.1.1.2	**Signature:**boolean authentication(void);
4.6.1.3.1.1.3	**Description:**--
4.6.1.3.2	**Sequence**
4.6.1.3.2.1	**step**
4.6.1.3.2.1.1	**Order**:1
4.6.1.3.2.1.2	**callName:**authentication
5	**OperationalPart**
5.1	**trustMechanisms:**TOBE defined
5.2	**validity**
5.2.1	**validFrom:**1214750275
5.2.2	**validUntil:**1449550800

Table 3 –S&D Classes.

5.2. S&D Pattern

This Section presents the *TextAuthentication.uma.es* S&D Pattern. This pattern consists on a simple authentication by means of a login process.

S&D Pattern: TextAuthentication.uma.es	
1	**Name:** TextAuthentication
2	**Domain:** uma.es
3	**Version:** 2.0
4	**InformationalPart**
4.1	**Creator:**
4.1.1 **4.1.2**	**Name:** uma.es **Date:** 1214752024
4.2	**Label:** This pattern provides a simple way to authenticate users by means of a login method.
4.3	**comments: --**
4.4	**Provided Properties**
4.4.1	**property:** Strong authentication
4.4.1.1	**Name:** authentication
4.4.1.2	**Domain:** uma.es
4.4.1.3	**Version:**1.0
4.4.1.4	**Timestamp:** 1214752496
4.5	**Static Tests Performed**
4.5.1	**Test**
4.5.1.1	**Description:--**
4.5.1.2	**attackModels:**man in the middle attack
4.5.1.3	**Result:** ok
4.5.1.4	**Date:** 1214752740
4.5.1.5	**Comments: --**
4.6	**Roles**
4.6.1 **4.6.1.1**	**Role** **roleName:**authenticator
4.6.1.2	**Description:--**
4.6.1.3	**Interface:**
4.6.1.3.1	**Calls**
4.6.1.3.1.1	**Call**
4.6.1.3.1.1.1	**CallName:**authentication
4.6.1.3.1.1.2	**Description:--**
4.6.1.3.1.1.3	**Signature:**boolean authentication(void);
4.6.1.3.2	**Sequence**
4.6.1.3.2.1	**Step**
4.6.1.3.2.1.1	**Order:**1
4.6.1.3.2.1.2	**callName:**authentication
4.7	**Models**
4.7.1	**Model**
4.7.1.1	**ModelType**
4.7.1.2	**Description**
4.7.1.3	**ModelData:**XML description

Table 4 –S&D Pattern I/II.

4.8 4.8.1 5	**Features** **Feature:** Authentication **OperationalPart**
5.1	**Trustmechanisms**
5.2	**validity:**
5.2.1	**validFrom:**1214750275
5.2.2	**validUntil:**1449550800
5.3	**Monitors**
5.3.1	**Monitor**
5.3.1.1	**Id:1**
5.3.1.2	**Localization:**localhost:5050
5.3.1.3	**Type:**syncronous
5.3.1.4	**Initialization:**--
5.4	**Roles**
5.4.1 5.4.1.1	**Role** **roleName:** authenticator
5.4.1.2	**requiredRoles**
5.4.1.2.1	**Role:--**
5.4.1.3 5.4.1.3.1 5.4.1.4 5.4.1.4.1 5.4.1.4.1.1	**Parameters** **Parameter:**-- **Preconditions** **Precondition** **Description: --**
5.4.1.4.1.2	**Query:--**
5.4.1.4.1.2.1	**Condition:**
5.4.1.4.1.2.2	**logicalExpression:**
5.4.1.4.1.2.3	**logicalOperator:**
5.4.1.4.1.2.4	**negated:**not
5.4.1.5	**systemConfiguration:** to be discussed
5.4.1.5.1	**systemConfigurationElement**
5.4.1.5.1.1	**Description:**
5.4.1.5.1.2	**field:**
5.4.1.5.1.3	**value:**
5.4.1.6	**Monitoring**
5.4.1.6.1	**MonitoringRule**
5.4.1.6.1.1	**monitorID: 1**
5.4.1.6.1.2	**monitorFormulae:** defined by CITY
5.4.1.6.1.3	**pollingType:**minimum, mandatory
5.4.1.6.1.4	**pollingValue:**2
5.4.1.7	**classAdaptor**
5.4.1.7.1	**Class**
5.4.1.7.1.1	**classReference:** UserAuthentication.uma.es
5.4.1.7.1.2	**classRole:** authenticator

5.4.1.7.1.3	**Adaptor:**
	Boolean Class.authentication () {
	Return pattern.authentication();
	}

Table 5 –S&D Pattern II/II.

5.3. S&D Implementation

This Section presents the *TextAuthenticationJava.uma.es* S&D Implementation. This artefact is an implementation of the previous S&D Pattern.

S&D Implementation TextAuthenticationJava.uma.es	
1	**Name:** TextAuthenticationJava
2	**Domain:** uma.es
3	**Version:** 2.0
4	**InformationalPart**
4.1	**Creator:**
4.1.1 **4.1.2**	**Name:** uma.es **Date:** 1214752024
4.2	**Label:** This pattern provides a simple way to authenticate users by means of a login method.
4.3	**comments: --**
4.4	**ComplianceProofs**
4.4.1	**Proof:**
4.4.1.1	**Name:**
4.4.1.2	**Description:**
4.4.1.3	**Result:**
5	**OperationalPart**
5.1	**Trustmechanisms**
5.2	**validity:**
5.2.1	**validFrom:**1214750275
5.2.2	**validUntil:**1449550800
5.3	**S&DpatternReference:**
5.3.1 **5.3.2** **5.3.3** **5.3.4** **5.4** **5.4.1**	**Name:**InteractiveTextAuthentication **Domain:**uma.es **Version:**2.0 **role:**authenticator **Preconditions** **Description**
5.4.2	**Query:--**
5.4.1.4.1.2.1	**Condition:**
5.4.1.4.1.2.2	**logicalExpression:**
5.4.1.4.1.2.3	**logicalOperator:**
5.4.1.4.1.2.4	**negated:**not
5.5	**implementationReference**
5.5.1	**URL:**/home/anto/Desktop/Enlace hacia DATOS ANTO/Proyectos/Serenity/Lenguaje/nueva version/TextAuthentication.jar
5.5.2	**Type:** java file
5.5.3	**signature:**

Table 6 –S&D Implementation.

6. Apendix A: S&D artefacts XML Schemas Definition

XMS Schema Definition for the representation of S&D Classes

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Created with Liquid XML Studio 1.0.8.0 (http://www.liquid-technologies.com) -->
<xsd:schema elementFormDefault="qualified" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attribute fixed="1" name="languageVersion" type="xsd:string" />
  <xsd:element name="SandDClass">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="informationalPart" type="informationalPartType" />
        <xsd:element name="operationalPart" type="operationalPartType" />
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" />
      <xsd:attribute name="domain" type="xsd:string" />
      <xsd:attribute name="version" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="informationalPartType">
    <xsd:sequence>
      <xsd:element minOccurs="1" maxOccurs="1" name="creator" type="creatorType" />
      <xsd:element minOccurs="1" maxOccurs="1" name="label">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:length value="50" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="comments" type="xsd:string" />
      <xsd:element name="providedProperties" type="propertiesType" />
      <xsd:element name="solutionFeatures" type="solutionFeaturesType" />
      <xsd:element name="roles" type="rolesType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="operationalPartType">
    <xsd:sequence>
      <xsd:element xmlns:q1="http://www.w3.org/2000/09/xmldsig#" minOccurs="1"
                   maxOccurs="unbounded" name="trustMechanisms" type="ToDefine" />
      <xsd:element name="validity" type="validityType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="classInfoType">
    <xsd:sequence>
      <xsd:element minOccurs="1" maxOccurs="1" name="name" type="xsd:string" />
      <xsd:element minOccurs="1" maxOccurs="1" name="domain" type="xsd:string" />
      <xsd:element name="version" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="creatorType">
    <xsd:sequence>
      <xsd:element minOccurs="1" maxOccurs="1" name="name" type="xsd:string" />
      <xsd:element minOccurs="1" maxOccurs="1" name="date" type="xsd:long" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="propertiesType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="property" type="propertyType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="propertyType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="domain" type="xsd:string" />
      <xsd:element name="version" type="xsd:string" />
      <xsd:element name="timestamp" type="xsd:long" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="solutionFeaturesType">
    <xsd:sequence>
```

```
      <xsd:element minOccurs="1" maxOccurs="unbounded" name="feature" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="validityType">
    <xsd:sequence>
      <xsd:element name="validFrom" type="xsd:long" />
      <xsd:element name="validUntil" type="xsd:long" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="rolesType">
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="role" type="roleType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="roleType">
    <xsd:sequence>
      <xsd:element name="roleName" type="xsd:string" />
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="interface" type="interfaceType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="interfaceType">
    <xsd:sequence>
      <xsd:element name="calls" type="callsType" />
      <xsd:element name="sequence" type="sequenceType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="callsType">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="call" type="callType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="callType">
    <xsd:sequence>
      <xsd:element name="callName" type="xsd:string" />
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="signature" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="sequenceType">
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="step" type="stepType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="stepType">
    <xsd:sequence>
      <xsd:element name="order" type="xsd:integer" />
      <xsd:element name="callName" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ToDefine">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string" />
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:schema>
```

XMS Schema Definition for the representation of S&D Patterns

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Created with Liquid XML Studio 1.0.8.0 (http://www.liquid-technologies.com) -->
<xsd:schema xmlns:ns0=http://www.omg.org/XMI
            xmlns:MonitoringRule="http://tempuri.org/ec/formula" elementFormDefault="qualified"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:import schemaLocation="MonitoringRules.xsd" id="MonitoringRule"
            namespace="http://tempuri.org/ec/formula" />
  <xsd:import schemaLocation="xmi.xsd" namespace="http://www.omg.org/XMI" />
  <xsd:attribute fixed="1" name="languageVersion" type="xsd:string" />
  <xsd:element name="SandDPattern">
    <xsd:complexType>
```

```
      <xsd:sequence>
        <xsd:element name="informationalPart" type="informationalPartType" />
        <xsd:element name="operationalPart" type="operationalPartType" />
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" use="required" />
      <xsd:attribute name="domain" type="xsd:string" />
      <xsd:attribute name="version" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="creatorType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="date" type="xsd:long" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="patternFeaturesType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="feature" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="classAdaptorsType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="class" type="classType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="classType">
    <xsd:sequence>
      <xsd:element name="classReference" type="xsd:string" />
      <xsd:element name="classRole" type="xsd:string" />
      <xsd:element name="adaptor" type="ClassAdaptorType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="validityType">
    <xsd:sequence>
      <xsd:element name="validFrom" type="xsd:long" />
      <xsd:element name="validUntil" type="xsd:long" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="parametersType">
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="parameter" type="parameterType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="preconditionsType">
    <xsd:sequence>
        <xsd:element minOccurs="0" maxOccurs="unbounded" name="precondition"
                     type="preconditionType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="staticTestsPerformedType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="test" type="testType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="testType">
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="attackModels" type="xsd:string" />
      <xsd:element name="result" type="xsd:string" />
      <xsd:element name="comments" type="xsd:string" />
      <xsd:element name="date" type="xsd:long" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required" />
  </xsd:complexType>
  <xsd:complexType name="systemConfigurationType">
    <xsd:sequence>
      <xsd:annotation>
        <xsd:documentation />
      </xsd:annotation>
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="systemConfigurationElement"
                   type="systemConfigurationElem" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="monitoringType">
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="monitoringRule"
                   type="monitoringRuleType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="monitorType">
    <xsd:sequence>
      <xsd:element name="id" type="xsd:integer" />
      <xsd:element name="localization" type="xsd:string" />
      <xsd:element name="type" type="xsd:string" />
      <xsd:element name="initialization" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="informationalPartType">
    <xsd:sequence>
      <xsd:element name="creator" type="creatorType" />
      <xsd:element name="label">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:length value="50" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="comments" type="xsd:string" />
      <xsd:element name="providedProperties" type="propertiesType" />
      <xsd:element name="staticTestsPerformed" type="staticTestsPerformedType" />
      <xsd:element name="features" type="patternFeaturesType" />
      <xsd:element name="roles" type="rolesInformationalType" />
      <xsd:element name="models" type="modelsType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="operationalPartType">
    <xsd:sequence>
      <xsd:element name="trustMechanisms" type="ToDefine" />
      <xsd:element name="validity" type="validityType" />
      <xsd:element name="monitors" type="monitorsType" />
      <xsd:element name="roles" type="rolesOperationalType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="monitorsType">
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="monitor" type="monitorType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="rolesOperationalType">
    <xsd:sequence>
      <xsd:element minOccurs="1" maxOccurs="unbounded" name="role" type="roleOperationalType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="roleOperationalType">
    <xsd:sequence>
      <xsd:element name="roleName" type="xsd:string" />
      <xsd:element name="requiredRoles" type="reqRolesList" />
      <xsd:element name="parameters" type="parametersType" />
      <xsd:element name="preconditions" type="preconditionsType" />
      <xsd:element name="systemConfiguration" type="systemConfigurationType" />
      <xsd:element name="monitoring" type="monitoringType" />
      <xsd:element name="classAdaptors" type="classAdaptorsType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="interfaceType">
    <xsd:sequence>
      <xsd:element name="calls" type="callsType" />
      <xsd:element name="sequence" type="sequenceType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="callsType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="call" type="callType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="callType">
```

```
    <xsd:sequence>
      <xsd:element name="callName" type="xsd:string" />
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="signature" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="sequenceType">
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="step" type="stepType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="stepType">
    <xsd:sequence>
      <xsd:element name="order" type="xsd:integer" />
      <xsd:element name="callName" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="parameterType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="domain" type="xsd:string" />
      <xsd:element name="defaultValue" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="preconditionType">
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="query" type="logicalExpressionType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="systemConfigurationElem">
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="field" type="xsd:string" />
      <xsd:element name="value" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="monitoringRuleType">
    <xsd:sequence>
      <xsd:element name="monitorId" type="xsd:integer" />
      <xsd:element name="monitorFormulae" type="MonitoringRule:formulaType" />
      <xsd:element name="pollingType" type="xsd:string" />
      <xsd:element name="pollingValue" type="xsd:integer" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ToDefine">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string" />
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="reqRolesList">
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="role" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="rolesInformationalType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="role" type="roleInformationalType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="roleInformationalType">
    <xsd:sequence>
      <xsd:element name="roleName" type="xsd:string" />
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="interface" type="interfaceType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="propertiesType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="property" type="propertyType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="propertyType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="domain" type="xsd:string" />
      <xsd:element name="version" type="xsd:string" />
      <xsd:element name="timestamp" type="xsd:long" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="modelsType">
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="model" type="modelType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="modelType">
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="ModelType">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="UML" />
            <xsd:enumeration value="TROPOS" />
            <xsd:enumeration value="OTHER" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="modelData" type="ns0:XMI" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="logicalExpressionType">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="condition" type="conditionType" />
        <xsd:element name="logicalExpression" type="logicalExpressionType" />
      </xsd:choice>
      <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="logicalOperator" type="logicalOperatorType" />
        <xsd:element name="logicalExpression" type="logicalExpressionType" />
      </xsd:sequence>
    </xsd:sequence>
    <xsd:attribute name="negated" type="xsd:boolean" />
  </xsd:complexType>
  <xsd:complexType name="conditionType">
    <xsd:choice>
      <xsd:element name="equalTo" type="relationType" />
      <xsd:element name="notEqualTo" type="relationType" />
      <xsd:element name="lessThan" type="relationType" />
      <xsd:element name="greaterThan" type="relationType" />
      <xsd:element name="lessThanEqualTo" type="relationType" />
      <xsd:element name="greaterThanEqualTo" type="relationType" />
    </xsd:choice>
    <xsd:attribute name="negated" type="xsd:boolean" />
  </xsd:complexType>
  <xsd:complexType name="relationType">
    <xsd:sequence>
      <xsd:element name="operand1" type="relationalOperandType" />
      <xsd:element name="operand2" type="relationalOperandType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="logicalOperatorType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="and|or" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="relationalOperandType">
    <xsd:choice>
      <xsd:element name="queryOperand" type="xpathExpressionType" />
      <xsd:element name="constant" type="constantType" />
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="xpathExpressionType">
    <xsd:sequence>
      <xsd:element name="document" type="documentType" />
      <xsd:element name="xpath" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="documentType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="type" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="constantType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="value" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ClassAdaptorType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="imports" type="xsd:string" />
      <xsd:element name="headerClass" type="xsd:string" />
      <xsd:element name="globalVariables" type="xsd:string" />
      <xsd:element name="classes">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element maxOccurs="unbounded" name="class">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="header" type="xsd:string" />
                  <xsd:element name="codeLines" type="xsd:string" />
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

XMS Schema Definition for the representation of S&D Implementations

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Created with Liquid XML Studio 1.0.8.0 (http://www.liquid-technologies.com) -->
<xsd:schema elementFormDefault="qualified" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attribute fixed="1" name="languageVersion" type="xsd:string" />
  <xsd:element name="SandDImplementation">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="informationalPart" type="informationalPartType" />
        <xsd:element name="operationalPart" type="operationalPartType" />
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" use="required" />
      <xsd:attribute name="domain" type="xsd:string" />
      <xsd:attribute name="version" type="xsd:string" />
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="creatorType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="date" type="xsd:long" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="validityType">
    <xsd:sequence>
      <xsd:element name="validFrom" type="xsd:long" />
      <xsd:element name="validUntil" type="xsd:long" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="preconditionsType">
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="precondition"
```

```
                 type="preconditionType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="complianceProofType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="proof" type="proofType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="proofType">
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="result" type="xsd:string" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required" />
  </xsd:complexType>
  <xsd:complexType name="informationalPartType">
    <xsd:sequence>
      <xsd:element name="creator" type="creatorType" />
      <xsd:element name="label">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:length value="50" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="comments" type="xsd:string" />
      <xsd:element name="complianceProofs" type="complianceProofType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="operationalPartType">
    <xsd:sequence>
      <xsd:element name="validity" type="validityType" />
      <xsd:element name="SandDPatternReference" type="SandDPatternReferenceType" />
      <xsd:element name="trustMechanisms" type="ToDefine" />
      <xsd:element name="preconditions" type="preconditionsType" />
      <xsd:element name="implementationReference" type="implementationReferenceType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="preconditionType">
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="query" type="logicalExpressionType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="systemConfigurationElem">
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string" />
      <xsd:element name="field" type="xsd:string" />
      <xsd:element name="value" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="monitoringRuleType">
    <xsd:sequence>
      <xsd:element name="monitorId" type="xsd:integer" />
      <xsd:element name="monitorFormulae" />
      <xsd:element name="pollingType" type="xsd:string" />
      <xsd:element name="pollingValue" type="xsd:integer" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ToDefine">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string" />
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="reqRolesList">
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="role" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="rolesInformationalType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="role" type="roleInformationalType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="roleInformationalType">
    <xsd:sequence>
      <xsd:element name="roleName" type="xsd:string" />
      <xsd:element name="description" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="propertiesType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="property" type="propertyType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="propertyType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="domain" type="xsd:string" />
      <xsd:element name="version" type="xsd:string" />
      <xsd:element name="timestamp" type="xsd:long" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="SandDPatternReferenceType">
    <xsd:sequence minOccurs="1" maxOccurs="1">
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="domain" type="xsd:string" />
      <xsd:element name="version" type="xsd:string" />
      <xsd:element minOccurs="0" maxOccurs="unbounded" name="role" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="implementationReferenceType">
    <xsd:sequence>
      <xsd:element name="URL" type="xsd:anyURI" />
      <xsd:element name="type" type="xsd:string" />
      <xsd:element name="signature" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="logicalExpressionType">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="condition" type="conditionType" />
        <xsd:element name="logicalExpression" type="logicalExpressionType" />
      </xsd:choice>
      <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="logicalOperator" type="logicalOperatorType" />
        <xsd:element name="logicalExpression" type="logicalExpressionType" />
      </xsd:sequence>
    </xsd:sequence>
    <xsd:attribute name="negated" type="xsd:boolean" />
  </xsd:complexType>
  <xsd:complexType name="conditionType">
    <xsd:choice>
      <xsd:element name="equalTo" type="relationType" />
      <xsd:element name="notEqualTo" type="relationType" />
      <xsd:element name="lessThan" type="relationType" />
      <xsd:element name="greaterThan" type="relationType" />
      <xsd:element name="lessThanEqualTo" type="relationType" />
      <xsd:element name="greaterThanEqualTo" type="relationType" />
    </xsd:choice>
    <xsd:attribute name="negated" type="xsd:boolean" />
  </xsd:complexType>
  <xsd:complexType name="relationType">
    <xsd:sequence>
      <xsd:element name="operand1" type="relationalOperandType" />
      <xsd:element name="operand2" type="relationalOperandType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="logicalOperatorType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="and|or" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="relationalOperandType">
    <xsd:choice>
      <xsd:element name="queryOperand" type="xpathExpressionType" />
      <xsd:element name="constant" type="constantType" />
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="xpathExpressionType">
    <xsd:sequence>
      <xsd:element name="document" type="documentType" />
      <xsd:element name="xpath" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="documentType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="type" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="constantType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="value" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

XMS Schema Definition for the representation 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>
<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>
<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>
<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>
```

References

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

[2] Antonio Maña, Antonio Muñoz, Francisco Sanchez-Cid, Daniel Serrano, George Spanoudakis, Kelly Androutsopoulos, Luca Compagna. A5.D2.1: "Patterns and Integration Schemes Languages (First Version)". SERENITY Project 2006.

[3] Maña A., Presenza D., Piñuela A., Serrano D., Soria P. Sotiriou D. A6.D3.1: "Specification of SERENITY Architecture". SERENITY Project 2006.

[4] P. El Khoury, B. Gallego-Nicasio, S. K. Sinha, K. Li, A. Maña, A. Muñoz J.F. Ruiz, A. Saidane, D. Serrano. A5.D4.1: "End-user requirements specification language (initial version)". SERENITY Project 2008.

[5] Kloukinas C., Spanoudakis G. "A Pattern-Driven Framework for Monitoring Security and Dependability" In proceedings of 4th International Conference on Trust, Privacy and Security in Digital Business (TrustBus`07).

[6] Richard E. Pattis. "Technical Symposium on Computer Science Education archive". Proceedings of the twenty-fifth SIGCSE symposium on Computer science education (1994). Phoenix, Arizona, United States. Pages: 300 – 303 (ISBN:0-89791-646-8).

[7] K. Androutsopoulos, C.Ballas, C. Kloukinas, K. Mahbub, G. Spanoudakis. A4.D3.1: "V1 of Dynamic Validation Prototype". SERENITY Project 2006.

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

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

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

[11] Christos Kloukinas, George Spanoudakis, Theocharis Tsigritis. A4.R6: "Draft Schema for Recovery Actions to be Taken Upon Potential or Definite Violations of Monitoring Rules". SERENITY Project 2008. https://bscw.sit.fraunhofer.de/bscw/bscw.cgi/1035484.