Chapter 5
Representation of Security and Dependability Solutions

Francisco Sánchez-Cid, Antonio Maña, George Spanoudakis, Christos Kloukinas, Daniel Serrano, and Antonio Muñoz

Abstract AmI considerations lead us to argue that it is essential for Security and Dependability (S&D) mechanisms to be able to adapt themselves to renewable context conditions in order to be applied to the ever-changing AmI scenarios. The key for this dynamic adaptation relies on the ability to capture the expertise of S&D engineers in such a way that it can be selected, adapted, used and monitored at runtime by automated means. S&D Artefacts proposed in this chapter represent the core of author's approach to precisely model such expertise in form of semantic descriptions. They adopt an integral methodology covering the complete system life cycle going from S&D Classes, mostly used at development time, to S&D Patterns and S&D Implementations, perfectly suited for deployment and runtime use. This chapter traces the foundations and internals of S&D Artefacts, describing how they

Francisco Sánchez-Cid
Computer Science Department, University of Malaga,
Campus de Teatinos. 29071 Málaga. e-mail: cid@lcc.uma.es

Antonio Maña
Computer Science Department, University of Malaga,
Campus de Teatinos. 29071 Málaga. e-mail: amg@lcc.uma.es

George Spanoudakis
Department of Computing, City University of London,
Northampton Square, London, EC1V 0HB, UK. e-mail: gespan@soi.city.ac.uk

Christos Kloukinas
Department of Computing, City University of London,
Northampton Square, London, EC1V 0HB, UK. e-mail: C.Kloukinas@soi.city.ac.uk

Daniel Serrano
Computer Science Department, University of Malaga,
Campus de Teatinos. 29071 Málaga. e-mail: serrano@lcc.uma.es

Antonio Muñoz
Computer Science Department, University of Malaga,
Campus de Teatinos. 29071 Málaga. e-mail: amunoz@lcc.uma.es

are defined and structured, and how they are categorized and grouped to form an exhaustive library of sound S&D Solutions.

5.1 Introduction

It is now widely accepted that end to end security should be adopted and accomplished as part of the early application design and development process. If addressed at the end of the deployment phase or even considered just before the system testing in a pre-production environment, options for reactive or post-mortem security fixes are very limited.

Moreover, as nowadays threats quickly evolve and make existing solutions obsolete, what was considered a secure application design might be insecure in short term. Thus, the challenge faced by our work is to create a quality application security design to ensure that (i) all aspects of application security are considered during the early design stages; (ii) the security and dependability requirements are correctly translated from the design to the implementation; and that (iii) once the system has been deployed, it adopts a reactive approach, ensuring service continuity and recovery in case of a security breach or malicious attack. For this challenge to be accomplished, existing Security and Dependability (S&D) Solutions must be analysed and then described in such a way that allows them to be selected, adapted, used and monitored at runtime by automated means.

S&D Solutions refer to any security or dependability mechanism (e.g. the Apache Rampart mechanism for automatic encryption and decryption of messages exchanged between web services [1]) that realizes an S&D Requirement (e.g. authentication, integrity...). Following SERENITY approach, S&D Experts study, analyse and eventually come out with a sound description of the functionality of S&D Solutions. They design S&D Solutions as blocks or services and use the language of S&D Artefacts to represent them either as S&D Classes or S&D Patterns, depending on the abstraction level considered. They should also have expertise in verification, validation and certification of S&D Solutions as well as in legal issues related to information technologies. This approach is specially interesting for ambient intelligence environments (AmI) because its particular S&D requirements.

Once a solution is described in terms of the Language of S&D Patterns, engineers acting as Component Developers are able to implement it following the strategies defined in the language. As all security mechanisms studied by S&D Experts are described as abstract solutions, both independent of the target platform and implementation language, component developers can create a variety of implementations of the same solution for different platforms and programming languages. The resulting product is an Executable Component which, in turn, is described as an S&D Implementation using the language of S&D Artefacts to facilitate its dynamic application at deployment time.

Each artefact in SERENITY is conceived to represent a solution that provides certain security and dependability requirements. Thus, one of the main aspects in

S&D Patterns' specification is the elicitation of the S&D Properties they provide (i.e. the requirements they fulfil). The definition of these properties along with their formal description is task of the S&D Engineers.

5.2 Existing Approaches for Modeling S&D Solutions

Existing approaches for modeling security and dependability aspects in Ambient Intelligence ecosystems go from approaches based on middleware, through the adaption of Frameworks and Agents technologies, to the enhanced concept of Pattern.

Approaches based on *Middleware*, capture expertise of S&D Engineers in the form of standard interfaces & components that provide applications with a simpler facade to access this set of specialized, powerful and complex capabilities. Though useful, an important problem with middleware–based approaches is that the computational cost of middleware components is far too high for computing devices with limited capabilities. In addition, the security infrastructure of middleware systems is traditionally restricted to authorization and access control issues [2, 3].

Application *Frameworks* [4, 5] have emerged as a powerful technology for developing and reusing middleware and application software. However, this approach does not support secure interoperation with external (and not trusted) elements, and given that frameworks are application templates, they are not well suited to cope with scenarios with high degrees of heterogeneity, dynamism and unpredictability.

In this sense, *Agent* paradigm emerges as a well-suited solution for such highly distributed environments where independent components from different owners coexist and interact. However, minor work has been done in the field of agent-based systems, built on the basis of properties like autonomy, interaction, context-awareness and goal-oriented nature [6]. Authors present in [7] a methodology that takes advantage of the organizational structures of agent systems in order to deal with simple access control and interaction patterns. However, they show limitations when modeling complex security aspects [8]: an agent is an independent entity by definition and many security solutions like a complete access control model (e.g. XACML) can not be represented. Other approaches [9] provide developers with the guidelines for the selection of security patterns, but use natural language and thus do not allow their automatic selection and posterior deployment.

Approaches to simplify the development of a complex system include those based in *Aspects*. Aspects "isolate" the different aspects that must be considered during development of a system (functionality, performance, reliability, security, etc.) managing security solutions as independent modules. Unfortunately, aspects are mainly an implementation technique and not suitable to provide and manage S&D solutions as a whole [10, 11].

5.2.1 Components

Components capture expertise in the form of reusable software elements that solve a certain problem in a certain context, having a set of well-defined interfaces and an associated description of their behavior [12, 13]. The main interest of component composition is to build new systems from their requirements by systematically composing reusable components. In general, this concept is not appropriate to represent S&D solutions because security mechanisms can not always be represented as units that can be connected to the rest of the system by means of well defined interfaces [14].

Nevertheless, components technology has presented advances on several issues relevant to SERENITY. The first refers to interface discovery and analysis, removal architectural mismatches or architecture selection and system composition. In [15], authors assume the use of previously-unknown components that are discovered at runtime but are used without further guarantees of origin or behavior. This is unacceptable from the point of view of security, so SERENITY proposes a framework for the rigorous treatment of security and dependability components introducing trust mechanisms for the component descriptions.

Some authors address the characterization and semantic definitions of components for their selection at development time. One possibility is the addition of semantic descriptions to CORBA in the form of protocols and roles [16], which is not far way from our approach in the sense of enriching the semantic descriptions of S&D Patterns with a set of semantic information. Other approaches first represent components using a well-known structure such as Trees [17], and then use these representations to reason on them and build component classifications. Although Trees offer a semi-formal technique for component description, security properties are not well suited for tree-based classification.

The dynamic analysis of component compatibility, with the objective of adapting components and synthesizing suitable software architectures [18], has been studied as well. Several component-based security models have been proposed in the literature [19, 20] but, unfortunately, these proposals have been based on oversimplified views of security, like those based on security levels [21], not applicable in AmI. These models are useful in specific systems like military organizations but they are just not applicable in more open and heterogeneous environments.

Finally, the issue of trust is usually addressed on the basis of component certification and secure composition [22]. Component Security Certification (CSC) pipeline [23] is an approach for certifying security of software components that provides security-oriented testing processes for software component. SERENITY follows a similar approach in the sense that all S&D components have a certified description, but we also support the runtime management of these components and provide a complementary mechanism for monitoring the components' behavior.

5.2.2 Patterns

The concept of security pattern was first introduced to support system engineers in selecting appropriate security and dependability solutions. However, most security patterns were expressed in textual form as informal indications on how to solve some (usually organizational) security problem [24, 25, 26]. Some of them do use more precise representations based on UML diagrams, but these patterns do not include sufficient semantic descriptions in order to automate their processing and to extend their use.

Perhaps the first and most valuable contribution as pioneer in security patterns as we know them at present, is the work from Joseph Yoder and Jeffrey Barcalow that adapts the object-oriented solutions to recurring problems of information security [27]. A natural evolution of [27] is the work presented by Romanosky in [28], which takes into consideration new questions that arise when securing a networked application. Indeed, there is an increasing interest in proposing more formal and precise descriptions to enhance the special needs of secure-ware systems with high dependency on the environment in which these systems are deployed. Konrad et al. took an step in that direction in [26], studying the security patterns proposed by Gamma et al. in [29] and using UML to represent both the structural and behavioral information that had not been considered in general design patterns but appears as mandatory in the new security context.

In an ambitious paper [30], Eduardo B. Fernandez follows the track initiated in [31] and proposes a methodology for using security patterns at every stage of the software lifecycle, combining for the first time the idea of multiple architectural levels with the use of design patterns [32]. Using more precise representations based on UML, this methodology was a first attempt to bridge the gap between design patterns and their final implementation. Evolutions of that approach use logic formalisms [33] and formal methods [34] with an associated graphical notation to specify rich structural properties.

In an attempt to enhance the organization and applicability of the emerging patterns, Wassermann and Cheng presented in [35] a revision of most of the patterns from [31, 27] and categorized them in terms of their abstraction level. Unfortunately, none of these approaches include enough semantic information for automating their processing, making it impossible to face the possible change of requirements at runtime and the consequent need of adapting or changing the patterns in use. In order to find some advances in this direction, we must refer the use of Design by Contract (DbC) [36]. Contracts are meant to precisely specify a given class using class invariants and pre and post-conditions for characterizing the behaviors of individual methods of the class. In [33, 37] authors follow this approach to preserve the design integrity of a system so that it continues to be faithful to the patterns used in its initial design even as it evolves to meet changing requirements.

Some security patterns have also been proposed for Web Services systems. From the very beginning, the tendency has been to use the object oriented paradigm: in [38] an object oriented access control (OOAC) was firstly introduced as a result of consequently applying the object oriented paradigm for providing access controls in object and interoperable databases. Fernandez proposes in [39] some specific solutions oriented to web services: a pattern to provide authentication and authorization using Role-based access control (the so-called Security Assertion Coordination pattern) and a pattern for XML Firewalls. Other source for patterns on XML Firewalls is [40]. To end with, [41] is a good source to study the historical approaches that have been appearing in the scientific literature as pattern systems for security.

5.3 S&D Artefacts: Supporting Security and Dependability at Runtime

This section presents the expression of security knowledge following the SERENITY approach. Current processes for providing security and dependability (S&D) in computing systems require a detailed a priori knowledge about the target systems and their environments. However, in emergent computing scenarios like ubiquitous computing or ambient intelligence, it is not possible to foresee all possible situations that may arise at runtime, so the necessary knowledge is not available at development time. This section introduces the concept of S&D Artefacts: a set of S&D Classes, Patterns and Implementations; an approach that makes accessible security and dependability knowledge at runtime by extracting and capturing sensitive information of S&D Solutions in a machine-readable way. The structure of these artefacts, along with the relations established among them is described in detail here, including the definition of roles, preconditions, provided properties ...

5.3.1 The Conception of S&D Patterns

It is leisure time at “Las Acacias” College and Alice and Bob enjoy a Race Game using their wireless ACME game-consoles. Charlie asks them to join the race using his brand new BOXX630 SERENITY-enabled game-console. Alice and Bob are willing to accept their friend to join the game, but the ACME consoles require confidentiality for wireless connections to other devices. Basically, this is a preconfigured setting for preventing eavesdroppers from obtaining information about the parties that are interacting and the services they use.

Charlie’s console identifies the requirement (a confidential channel) and looks for the best solution. At design time, the developers of the game identified the need to securely connect players, but because at that stage they could not foresee the possible types of counterparts and the different circumstances under which the communication would take place, they decided not to restrict the range of possible solutions to use. One of the SERENITY artefacts called S&D Class represents security and dependability services and is especially designed to support system developers in these situations. 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. In particular S&D Classes allow developers to delay the decision about the most appropriate solution to runtime, when the information required to select a specific solution (the context, type and capabilities of the other party, etc.) is available. Thus, they selected and used the S&D Class named "TransmisionConfidentiality.etsi.org", which represents confidentiality services and includes a predefined high-level interface. In this way, the game developers were able to use the confidentiality services without knowing which solution will be used to provide them at runtime.

Going back to our scenario, Charlie's console must now select one specific solution to provide the confidentiality services to the game application. At this point the console uses the second of the SERENITY artefacts called S&D Pattern, used to represent abstract solutions. The main purpose of this artefact is to guarantee the interoperability of different solutions. A number of different S&D Patterns belong to the selected S&D Class. After analyzing them, only two are found to be adequate given the current context: Charlie's using an open wireless network susceptible to possible eavesdropping as well as passive and active attacks. The suitable patterns are: "SSL 3.0 Channel" and "TLS Channel". At this point Charlie's device negotiates with the other parties and eventually, the SSL option is selected as the most appropriate.

Once the abstract solution has been selected, which ensures the interoperability between the different systems, Charlie's console needs to find an implementation (i.e. an instance) of the "SSL 3.0 Channel" S&D Pattern. The third artefact provided by SERENITY comes into play. This artefact is the S&D Implementation, used to represent working solutions.

Given Charlie's console context (underlying O.S., running software, other active S&D Solutions, user preferences, etc.), only three S&D Implementations are available for that S&D Pattern and that context: `mod_ssl` module from Apache 2.0, Cisco's `OpenSSL`, and `Java SSL` using JSSE. Java implementation of SSL is selected and activated to provide confidentiality for the game-connection. Charlie is informed of the successful establishment of the confidential connection and he finally joins the game.

It goes without saying that our main characters are all but security experts. Consequently, it is important to remark that the provision of a solution for a concrete context should be as transparent as possible for the user. That is, S&D Patterns must be designed for automated processing, so that Charlie's awareness of technical details should be reduced to the minimum.

5.3.2 S&D Artefacts Definition

Components, frameworks, middleware, and patterns have been proposed as alternatives to simplify the design of complex systems and to capture the specialized expertise of security engineers, making it available for non-expert developers. However, all approaches already reviewed in section 5.2 have important drawbacks and limitations that hamper their practical use, especially when it comes to runtime scenarios. Our approach aims at integrating the best of these approaches in order to overcome the problems that have prevented them to succeed individually.

The main pillar to build sound solutions is the development of artefacts to capture security expertise allowing its automated processing. Note that for this purpose we do not need to describe the internal functioning of the solution but its semantics (i.e. properties provided, limitations, etc.). This is an essential difference between our S&D Patterns and the widespread concept of security pattern. These semantic descriptions allow solutions to be automatically selected, adapted, used and monitored at runtime. Nevertheless, the approach presented here not only covers runtime aspects but adopts an integral methodology covering the complete system life cycle. Thus, an additional goal for S&D artefacts is to support system developers in the development process. Furthermore, because secure interoperability is an essential requisite for the widespread adoption of our model, trust mechanisms are provided for them. With these two purposes in mind, we have developed the following artefacts to capture the different aspects of the S&D Solutions that are necessary at different stages of the system life cycle.

5.3.2.1 S&D Patterns

We define S&D Solutions as well-defined mechanisms (i.e. security protocols, encryption algorithms, etc.) that provide one or more S&D Properties (i.e. confidentiality, integrity, availability, etc.). Based on this definition, S&D Patterns are semantic descriptions of S&D Solutions. These semantic descriptions contain all the information necessary for the selection, instantiation and adaptation, and dynamic application of the solution represented. In other words, given a well-known security solution (e.g. IDEA encryption algorithm), S&D Patterns allow you to isolate and capture its fundamental characteristics (e.g. block encryption, 128-bit key...) so that you can: (i) consult these characteristics by automated means at runtime and (ii) apply the solution when such characteristics match the requirements of the actual environment.

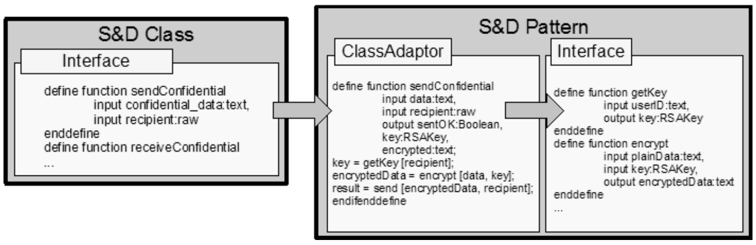
One important aspect of the solutions represented as S&D Patterns is that they can contain a description of the results of any static analysis performed on them. Such descriptions provide a precise foundation for the informed use of the solution and enhance the trust in the model. Despite of that, the limitations of the current static analysis tools introduce the need to support the dynamic validation of the behavior of the described solutions by means of monitoring mechanisms. S&D Patterns are meant for runtime scenarios, and more specifically, for evolving scenarios as those foreseen in AmI. In such scenarios, it is natural for the requirements of the environment to change from time to time so the soundness of the running solutions must be checked somehow. That is what the monitoring mechanisms stand for. When a running S&D Pattern is found to be inaccurate or inappropriate, the corresponding solution must be stopped, the system informed and then, if found convenient, the solution replaced by a new one.

S&D Patterns represent not only simple solutions, but also complex ones. In fact, a special type of S&D Patterns, called Integration Schemes, is used to represent solutions that are built by combining other S&D Patterns. 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 smaller S&D Solutions.

5.3.2.2 S&D Classes

S&D Classes represent abstractions of a set of S&D Patterns characterized for: (i) providing the same S&D Properties and (ii) complying with a common interface. According to this, S&D Patterns that belong to an S&D Class can have different interfaces but, in such circumstances, they must describe how these specific interfaces map into the S&D Class interface. We could describe this artefact as an extension of the interface concept, augmented with semantic information, in a similar way as proposed in [24].

Although S&D Classes are mainly used at development time, when system developers are defining the guidelines and general functionality of their models, the main purpose of introducing this artefact is to facilitate the dynamic substitution of the running S&D mechanisms at runtime. This is a basic pillar behind the whole idea of S&D Artefacts: first, select at development time a Class with an abstract definition of the intended functionality (i.e. abstract methods from Classes); second, find the Patterns complying to (i) this definition and (ii) the foreseen application environment; and third, at runtime, Patterns will be selectable and interchangeable because (though having different interfaces) they all comply with the same abstract one.



```
define function sendConfidential
          input confidential_data:text,
          input recipient:raw
enddefine
define function receiveConfidential
...
```

```
define function sendConfidential
          input data:text,
          input recipient:raw
          output sentOK:Boolean,
          key:RSAKey,
          encrypted:text;
key = getKey [recipient];
encryptedData = encrypt [data, key];
result = send [encryptedData, recipient];
endifenddefine
```

```
define function getKey
          input userID:text,
          output key:RSAKey
enddefine
define function encrypt
          input plainData:text,
          input key:RSAKey,
          output encryptedData:text
enddefine
...
```

Fig. 5.1 In the representation, the Class Adaptor specifies how to map from `sendConfidential()` call at Class level to the sequence `getKey()`, `encrypt()`, `send()` at Pattern level

Actually, interoperability is a key issue addressed via these concepts. Given that this artefact defines the high-level interface (i.e. the set of functions, calls, or methods that form the functionality offered by an artefact), with this approach it is possible to create an application bound to an S&D Class. Then, 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.

For this approach to be applicable, there must exist a mechanism to map from the original high level interface — described in the S&D Class, to the medium-level interface – used in the S&D Pattern. Figure 5.1 shows how this mapping is expressed. The pattern captures the correspondence from its own *Pattern operations* and the corresponding *Class calls*.

It is easy to see that the translation between one interface to the other is direct but not one-to-one, since it is possible for a single operation at S&D Class level, to be mapped into a sequence of operations at S&D Pattern level. Thus, the Class Adaptor must provide some statements for the flow of control that expresses the actual mapping. Moreover, as it is feasible for an S&D Pattern to belong to more than one S&D Class, it is possible to find several *Class Adaptors* for the same S&D Patterns Interface (each adaptor linked to the S&D Class that adapts).

5.3.2.3 S&D Implementations

S&D Implementations represent the components that realize the S&D Solutions and, given that S&D Patterns represent S&D Solutions, S&D Implementations realize S&D Patterns. Figure 5.2 represents a partial set of solutions, starting from a Class that provides confidentiality for simple transmissions, and ending in several Implementations of the solution (and their corresponding *Executable Components*).

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. A specific component providing encryption services or a web service providing time stamping services are susceptible to be described using an S&D Implementation. We must emphasize that S&D Implementations are not the actual components but their representation. The actual components are made accessible to applications thanks to the SERENITY Runtime Framework (presented in Chapter 11), who maps from the S&D Implementations to the actual Executable Components.

An S&D Implementation is not just an implementation of the S&D Solution, but 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 to meet before deployment, their performance, target platform, programming language or any other feature not fixed yet at Pattern level.

An S&D Implementation represents a working solution and therefore it contains a reference to the corresponding Executable Component that realizes it. While an S&D Implementation is only a description of an implementation, the Executable Component is the actual implementation as an executable code or entity. There exist

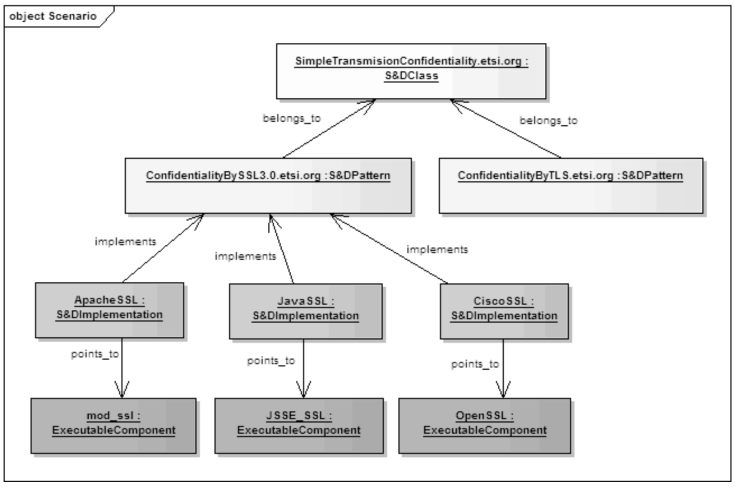

object Scenario
SimpleTransmisionConfidentiality.etsi.org : S&DClass
belongs_to
belongs_to
ConfidentialityBySSL3.0.etsi.org :S&DPattern
ConfidentialityByTLS.etsi.org :S&DPattern
implements
implements
implements
ApacheSSL : S&DImplementation
JavaSSL : S&DImplementation
CiscoSSL : S&DImplementation
points_to
points_to
points_to
mod_ssl : ExecutableComponent
JSSE_SSL : ExecutableComponent
OpenSSL : ExecutableComponent

Fig. 5.2 In the representation, `Apache SSL` implements `Confidentiality By SSL 3.0` and points to `mod_ssl` component, while two Patterns are shown that provide `Simple Transmission Confidentiality`, namely: `SSL 3.0` and `TLS`

a one to one relation between S&D Implementations (the descriptions of the working solutions) and Executable Components (the actual working solutions), so that no S&D Implementation is effective without an Executable Component associated.

5.3.3 Rationale for Classes, Patterns and Implementations

There are several reasons that justify the introduction of these artefacts and their separation at different levels of abstraction. First, from the point of view of the type of information they contain, only S&D Patterns can be verified using a handful of S&D engineering tools and techniques (for instance, static analysis using formal methods). This is possible for S&D Patterns because all information referring to the provided properties and the result of the verifications only concerns to the abstract solution, which is defined at pattern level. In contrast, S&D Classes can not be verified because they only provide the minimum amount of information required at development time (i.e. the properties provided and the interface used to access the services). The same applies to S&D Implementations, since software implementations are not feasible to verify for the most part. Therefore it is wise to separate their definitions since: (i) all information referring to the provided properties and the available proofs concern only the abstract solution (that is, the S&D Pattern) and not the interface or the specific implementation; and (ii) 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.

From the point of view of their role, S&D Classes are mainly interface definitions introduced in order to facilitate the development phase in the software life-cycle. On the other hand, S&D Patterns are used by S&D experts as a mechanism to capture their knowledge about one S&D solution in such a way that ensures the correct usage of the solution at runtime. S&D Implementations are used to capture information about the components that realize the S&D Patterns, so their role is analogous to that of the patterns. However, the knowledge for S&D Patterns is extracted by means of formal methods or any other analytic process, while S&D Implementations take advantage of certification processes or proof-carrying code schemes [46] to study each specific component.

Finally, the consideration of the producers of the different specifications makes it advisable to separate their definitions. For instance, S&D Classes will be defined by entities mainly interested in interoperability (e.g. industry associations, standardization bodies). Independent entities such as IT consulting companies and S&D experts will take these definitions to produce S&D Patterns, although it will be possible for standardization bodies to create S&D Patterns as well. Using this artefact, they will not only enhance security and dependability, but will also contribute to 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).

5.3.4 The Structure of S&D Artefacts

Each of the artefacts has a predefined structure, specified as an XML Schema, so there is a different XML Schema for S&D Classes, S&D Patterns and S&D Implementations. The definition of an S&D Artefact is in fact an XML document written according to the corresponding XML Schema. Understanding XML itself is rather simple, allowing data and meta-data sharing, and enabling interoperability; properties that are all inherited by S&D Artefact. However, given that S&D Artefact are conceived for their automatic use, the main reason behind the selection of XML as the standard for S&D Artefacts definition is its machine-readability.

There is one element that S&D Classes, S&D Patterns and S&D Implementation have in common: the distinction between *informational* and *operational* data. The former groups the information mainly devoted to development time, while the latter integrates the information used at runtime, when artefacts must be selected, the trust mechanisms verified and the functionality monitored. Next sections present the main elements and structure for each of the artefacts, and it provides several examples based on an authentication scenario.

The authentication scenario is composed by the following set of S&D Artefacts (see Figure 5.3). There are two S&D Classes, one of them representing authentication S&D Solutions (“UserAuthetication”), and the other devoted to represent “Observers”. “Observers” S&D Solutions do not provide a S&D mechanisms but checking particular context conditions. For instance, the S&D Pattern “smartCard-Connected” checks if there is an available smart card connected to the platform. There are two S&D Artefacts belonging to the “UserAuthentication”: the “Smart-CardAuthentication” S&D Pattern and the “SmartCardAuthenticationIS” Integration Scheme. The “SmartCardAuthentication” S&D Pattern represents an user authentication based on an smart card S&D Solution. the “SmartCardAuthenticationIS” Integration Scheme is a S&D Solution made by combining the aforementioned two patterns, consequently this S&D Solution includes the functionalities for both the authentication user and the check of the smart card connection. The rest of the S&D Artefacts in Figure 5.3 are S&D Implementations and Executable Components. Note, that the “SmartCardAuthenticationIS” Executable Component is the source of two *“use”* associations. This means that is this element (the Executable Component) the one implementing the composition of S&D Patterns. That is to say, once that this Executable Component has been instantiated it requires the functionalities from S&D Patterns under composition.

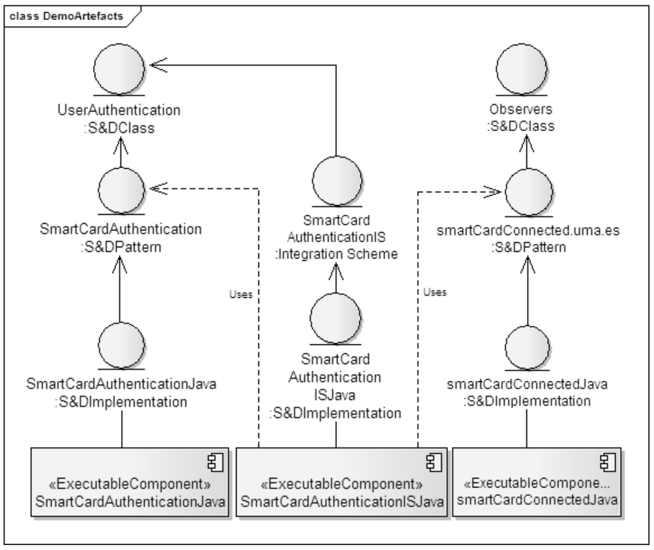

class DemoArtefacts
UserAuthentication :S&DClass
Observers :S&DClass
SmartCardAuthentication :S&DPattern
SmartCard AuthenticationIS :Integration Scheme
smartCardConnected.uma.es :S&DPattern
Uses
Uses
SmartCardAuthenticationJava :S&DImplementation
SmartCard Authentication ISJava :S&DImplementation
smartCardConnectedJava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

Fig. 5.3 S&D Artefacts involved in the proposed scenario.

5.3.4.1 Structure of S&D Classes

S&D Classes are introduced to relieve system developers from the need of specifying at development time the full interface for components that they include in their applications. Consequently, to make it manageable to search for classes, they must provide: (i) an exhaustive but simple description of their purpose, making it easy for developers to identify the desired class; and (ii) a reduced, high-level interface, making it easy for designers to incorporate in the programming code the calls to the operations defined in the class' interface.

Listing 5.1 Example of S&D Class definition in XML.

```
<?xml version="1.0" encoding="utf-8"?>
<SandDClass xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="file:///D:/serranito/trabajo/SERENITY/
     a5/version1/S&amp;DClass_v1.xsd" name="UserAuthentication" domain="uma
     .es" version="1.0">
  <informationalPart>
    <creator>
      <name>uma.es</name>
      <date>1214750275</date>
    </creator>
    <label>Autentication</label>
    <comments>This class represents a security solution providing user
        autentication.</comments>
    <providedProperties>
      <property>
        <name>authenticationProperty</name>
        <domain>>uma.es</domain>
        <version>1.1</version>
        <timestamp>1214750275</timestamp>
      </property>
    </providedProperties>
    <solutionFeatures>
      <feature>secure authentication</feature>
    </solutionFeatures>
    <roles>
      <role>
        <roleName>authenticator</roleName>
        <description>the application that performs the authetication</
             description>
        <interface>
          <calls>
        <call>
        <callName>authentication</callName>
        <description>ask for the authentication data to the user</
             description>
        <signature>bool authentication(void);</signature>
      </call>
       </calls>
          <sequence>
        <step>
        <order>1</order>
        <callName>authentication</callName>
      </step>
      </sequence>
        </interface>
      </role>
    </roles>
  </informationalPart>
  <operationalPart>
    <trustMechanisms> ––– </trustMechanisms>
    <validity>
      <validFrom>1214750275</validFrom>
      <validUntil>1449550800</validUntil>
    </validity>
  </operationalPart>
</SandDClass>
```

Thanks to the mapping mechanisms that automatically translate class' interfaces to pattern's and then implementation's interfaces, the selection of an S&D Class (instead of an S&D Pattern or Implementation) maximizes the flexibility and the number of possible S&D Solutions that can be later applied at runtime.

The description of S&D Classes is divided into two groups. Listing 5.1 introduce an example of an S&D Class following the presented structure. The S&D Class presented represents authentication S&D Solutions ("UserAuthetication"). Firstly, the informational part containing:

- `Creator`: this element is composed by the name of the creator and the date of creation of the artefact.
- `Label and Comments`: these two fields allow to included some information useful for developers using this artefact.
- `ProvidedProperties`: this element points to the descriptions of the S&D Properties fulfilled by S&D Patterns belonging to this S&D Class. Note that it is not the class but its patterns who really provide S&D Properties. One S&D Class can point to one or more properties (see Chapter 4 for further details on S&D Properties definition).
- `SolutionFeatures`: information about the specific characteristics of the solutions provided. They are meant to help developers discriminate among classes that point the same S&D Property.
- `Roles`: The use of the solution interface strongly depends on the role played by the solution user. In a secure transmission, they agree on the use of the same solution, but they use it in a different way. For instance, while the first *encrypts* and *sends* the data, the latter *receives* and *decrypts* it. The `Role` element covers this paradigm by specifying as many interface definitions as roles are. The `Role` elements contains the role name, its description and the interface offered.

Secondly, the operational part of S&D Classes contains:

- `validity`: this element express a period of time in which the artefact is applicable.
- `TrustMechanisms`: this element contains an enveloped XML Signature that, along with the trust infrastructure provided by SERENITY, allows the target system to check whether: (i) the document corresponds to the claimed artefact; (ii) it has been really produced by the creator; and (iii) it has not been modified or tampered with.

5.3.4.2 Structure of S&D Patterns

S&D Patterns are descriptions of reusable and validated S&D solutions that include a precise specification, along with applicability conditions. In the same fashion as S&D Classes, S&D Patterns are split into an informational part and an operational one. The informational part shares with S&D Classes a precise definition of the artefact, references to the S&D Properties provided, a list of features that helps to characterize the solution, and the description of the envisaged roles. In addition, it includes:

- `Static test performed`: every S&D Pattern can be proven, validated, best practice, recommendations by standards. Therefore, security engineers will be responsible for the static testing of the solution represented by the pattern and will use this element to specify the proofs that have been applied in order to claim that this solution is sound.
- `Models`: an S&D Pattern may have an associated model to conceptually describe the solution that the pattern represents. This element allows the inclusion of UML models, BPEL models, etc.
- `Roles`: the informational part of the roles, which is a declaration of the pattern roles. The declaration of each role includes the definition of its interface.

Since S&D Patterns are mainly devoted to runtime use, the operational side of S&D Patterns is richer than that of S&D Classes. In addition to the trust mechanisms, patterns include: an accurate behavioral description; a list of constraints on the context required for deployment; and information describing how to adapt and monitor the applied solution. More specifically:

- `monitors`: this element presents the declaration, by means of a list, of the monitors to be used by this pattern. These monitors are composed by an identifier, a location, the type of monitor and some initialization data.
- `Roles`: the operational part of the roles includes the following information:
  - `RolesName`: this element is specifies the name of the role.
  - `RequiredRoles`: this element is used to describe the complementary roles that need the application of this specific role.
  - `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 behavior of the solution. they can always be represented by a 2-tuple with a name and a value.
  - `Preconditions`: Every S&D Pattern represents a specific S&D Solution. For this reason, we assume that they are not universally applicable. `Preconditions` element collects the restrictions concerning the applicability context of the pattern, and it is task of the SERENITY Runtime Framework to check whether these preconditions hold before deploying it.
  - `Monitoring Information`: 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 smoothly. This element contains instructions for an external monitoring mechanism to perform this activity. Section 5.4 in this chapter presents an in-depth description of this element.
  - `ClassAdaptor`: already presented in section 5.3.2.2, this element describes how to map from the native interface of the S&D Pattern to the interface of an S&D Class. It is important to mention that, the `ClassAdaptor` element includes a number of references to the S&D Classes the S&D Pattern belongs to. Doing this, the mapping between S&D Classes and S&D Pattern is done through their roles. ClassAdaptors are expressing using Java syntax.

In order to facilitate the *ClassAdaptor* automated processing, the structure of the “adaptor” element is split into “name”, “imports”, “headerClass”, “globalVariables”, “classes” ... following the usual Java classes structure.

Listing 5.2 presents an example of an S&D Pattern following the proposed structure. It is an S&D Pattern representing an user authentication S&D Solution based on the usage of an smart card. This S&D Pattern belongs to the S&D Class presented in Listing 5.1.

Listing 5.2 Example of S&D Pattern definition in XML.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Created with Liquid XML Studio 1.0.8.0 (http://www.liquid-technologies
     .com) -->
<SandDPattern xmlns:tns="http://tempuri.org/ec/formula" xmlns:tnsa="http://
     www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="file:///D:/serranito/trabajo/SERENITY/
     a5/version1/S&amp;DPattern_v1.xsd" name="smartCardAuthentication"
     domain="uma.es" version="1.0">
  <informationalPart>
    <creator>
      <name>uma.es</name>
      <date>1214750275</date>
    </creator>
    <label>authentication smartcard</label>
    <comments>User Authentication based on Smart Card</comments>
    <providedProperties>
      <property>
        <name>authenticationProperty</name>
        <domain>uma.es</domain>
        <version>string</version>
        <timestamp>1214750275</timestamp>
      </property>
    </providedProperties>
    <staticTestsPerformed>
      <test name="name␣of␣the␣test">
        <description> -- </description>
        <attackModels> -- </attackModels>
        <result> -- </result>
        <comments> -- </comments>
        <date> -- </date>
      </test>
    </staticTestsPerformed>
    <features>
      <feature>secure authentication smartcard</feature>
    </features>
    <roles>
      <role>
        <roleName>authenticator</roleName>
        <description>the application performing the authentication</
             description>
        <interface>
          <calls>
            <call>
              <callName>authentication</callName>
              <description>this function is used in order to ask the
                   smartcard PIN to the user</description>
              <signature>bool authentication(void);</signature>
            </call>
          </calls>
          <sequence>
        <step>
        <order>1</order>
        <callName>authentication</callName>
      </step>
      </sequence>
        </interface>
      </role>
    </roles>
    <models>
      <model>
        <ModelType>UML</ModelType>
        <description>in XMI format</description>
        <modelData> —— </modelData>
      </model>
    </models>
  </informationalPart>
  <operationalPart>
    <trustMechanisms> —— </trustMechanisms>
    <validity>
      <validFrom>1214750275</validFrom>
      <validUntil>1214780653</validUntil>
    </validity>
    <monitors>
      <monitor>
        <id>1</id>
        <localization>localhost:3301</localization>
        <type>syncronous</type>
        <initialization>user:authenticatorPattern</initialization>
      </monitor>
    <roles>
      <role>
        <roleName>authenticator</roleName>
        <requiredRoles />
        <parameters />
        <preconditions />
        <systemConfiguration />
        <monitoring />
        <classAdaptors>
          <class>
            <classReference>UserAuthentication</classReference>
            <classRole>authenticator</classRole>
            <adaptor>
              <name>classToPattern</name>
              <imports></imports>
              <headerClass></headerClass>
              <globalVariables></globalVariables>
              <classes>
                <class>
                  <header>bool authentication(void) // class call</header>
                  <codeLines>return authentication(); // pattern call</codeLines>
                </class>
              </classes>
            </adaptor>
          </class>
        </classAdaptors>
      </role>
    </roles>
  </operationalPart>
</SandDPattern>
```

From S&D solution developer point of view, the development of an Integration Scheme follows the same process as the development of S&D Patterns. This so is since both artefacts use the same structure. 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. From the point of view of the Serenity runtime Framework, an executable component implementing an integration Scheme operates like 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.

5.3.4.3 Structure of S&D Implementations

S&D Implementations are the necessary link from the abstract solution, represented as an S&D Pattern, and the actual Executable Components that realize and implement that solution. In the same manner as two artefacts, implementations include: a precise definition; information about the creator of the artefact; a list of features; the compliance proofs applied to its Executable Component; and the essential trust mechanisms. Also devoted to runtime use, it includes a set of preconditions that are checked in the same way as S&D Patterns preconditions. Regarding the Executable Component pointed by the artefact, there is a element called `implementationReference` containing the following information:

- `URL`: this elements identifies where to find the executable component.
- `Type`: it describes the type of Executable Component (i.e. a web service, a software component, etc.).
- `Signature`: this is a security mechanisms to ensure that the executable component code is valid.

We must highlight that there is no specification of the S&D Implementation interface because all S&D Implementations share the same interface of the S&D Pattern they belong to.

5.4 S&D Artefacts for Monitoring

As aforementioned, S&D Patterns do not represent perfect solutions and therefore require the runtime monitoring of certain conditions to ensure that they are valid for the current executing context. Even perfect solutions depend on some conditions. These conditions are expressed as part of the description of an S&D Pattern and they refer to events which occur at the roles of the S&D Pattern. The Executable Components, which realise an implementation for the S&D Pattern, produce these events and emit them to the SRF. The SRF forwards the events to the monitor(s) that are responsible for the particular application and receives back the results of the evaluation of the monitoring conditions, which have been already been sent by the SRF to the monitor when the S&D Pattern had been instantiated.

The monitoring conditions are effectively a variant of Event Calculus [43] and the events appearing in them represent procedure/method calls or messages exchanged among the S&D Pattern roles. An example of such a condition is 5.1, shown in XML form in Listing 5.3, both taken from [42]. This condition is from an S&D Pattern for Optimistic Fair Exchange based on a Trusted Third Party (TTP). The condition is effectively derived from the requirement for the availability of the TTP. It states that when Bob sends a "solve" message to the TTP, then the TTP should respond with a "send_item" message withing time t_u.

$$\begin{aligned}\forall\quad & id_1, id_2, bob, TTP : \text{String}; t_1, t_2 : \text{Time}\\ & \text{Happens}(e(id_1, Bob, TTP, \text{REQ-B}, \text{solve}((Item\_A)Ka_1, Item\_B)),\\ & Bob), t_1, \mathbf{R}(t_1, t_1))\\ \Rightarrow & \\ & \text{Happens}(e(id_2, TTP, Bob, \text{RES-A}, \text{send\_item}(((Item\_A)Ka_1)Ka_2),\\ & TTP), t_2, \mathbf{R}(t_1, t_1 + t_u))\end{aligned} \tag{5.1}$$

Listing 5.3 XML representation of condition 5.1

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

```
                        <varName>t1</varName>
                        <varType>Time</varType>
                    </time>
                </fromTime>
                <toTime>
                    <time>
                        <varName>t1</varName>
                        <varType>Time</varType>
                    </time>
                </toTime>
            </happens>
        </predicate>
    </head>
</formulae>

```

Monitoring conditions have quantification elements for their variables, zero or one body elements (the RHS of the implication) and one head element (the LHS of the implication). Both the body and head elements are of type bodyHeadType, which effectively describe predicates. The XML schema for the monitoring conditions which is described in full details in section 3.2 of [42], has been extended in [44] to support further information for diagnosis and threat risk evaluation, since it is also used for reporting the monitoring results back to the SRF. Appendix C of [44] contains the full XML schema with the parts for diagnosis/threat risk evaluation highlighted - it is unfortunately too long to reproduce here. Figures 5.4 and 5.5 show the two entities which have been extended in order to support diagnosis and threat risk estimation. These are the resultType and the predicateType elements, which now include a minimum and maximum threat likelihood. Through these and the new confirmed attribute of predicateType, the monitor can inform the SRF of the threat level estimation for a rule, as well as for the diagnostic information that it has produced, as explained in detail in [44].

5.4.1 Reactions to Rule Violations

When a rule can be fully evaluated, the monitor reports the result of this evaluation to the SRF. The SRF then needs to decide how to react to the current situation. There are various reactions that the SRF could take - from simply logging a message, to deactivating the S&D Pattern itself. Of course, the choice of the reaction cannot be made by the SRF. Instead, the S&D Pattern developers need to associate a set of actions with each monitoring rule, so that the SRF knows how it should react in each case. The possible reactions that are available to a developer of an S&D Pattern are the following:

- `DeactivatePattern()` - The result of taking this action is to deactivate the pattern instance that the violation is related to. `DeactivatePattern()` takes no arguments since the SRF knows which pattern, pattern instance and rule the monitoring result which has triggered the execution of the action is related to.
- `RestartPattern()` - The result of taking this reaction is to start a new instance of the same pattern. The reaction does not need any arguments since the

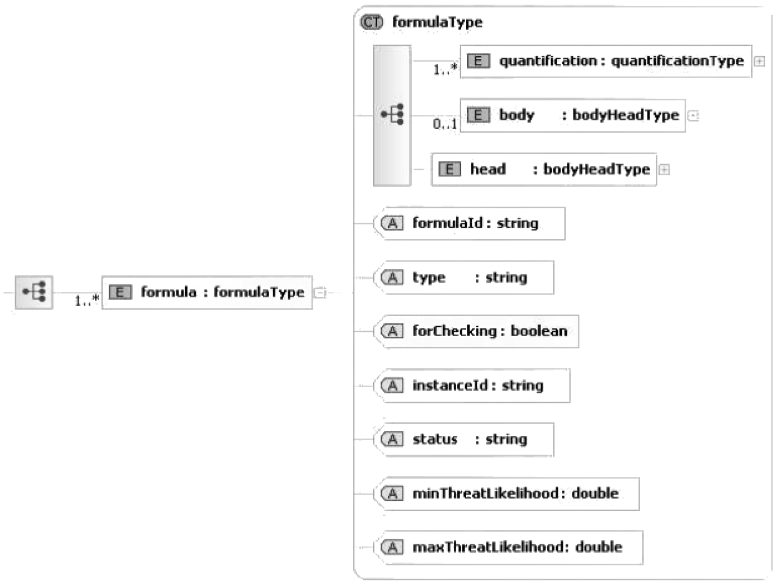


Fig. 5.4 Additions to resultsType in the new schema for monitoring results

SRF knows which pattern the monitoring result that has triggered the execution of the action is related to.

- `NotifySRF(String external_SRF_ID, String message)` - The result of taking this reaction is to notify an external SRF of the violation. The external SRF is identified by the reaction parameter `external_SRF_ID`. The information that will be sent to the external SRF is determined by the parameter message.
- `NotifyApplication(String message)` - The result of taking this reaction is to notify the application for which the implementation of the pattern is deployed of the violation. The notification to be sent is determined by the parameter message. The application to be notified does not need to be identified since the SRF knows it.
- `StopMonitoringRules(String` $\texttt{ruleID}_1$`, ...)` - The result of taking this reaction is to request the monitor to stop monitoring a given set of rules. These rules are identified by the parameters $\texttt{ruleID}_i$. All $\texttt{ruleID}_i$ are restricted to take as values IDs of the rules of the current S&D Pattern (i.e., the one incorporating the specific reaction), and when sent to the monitor the SRF should make sure that they are amended appropriately, so as to be unique (this may, for example, be ensured by adding the ID of the current S&D Pattern instance as a prefix to each $\texttt{ruleID}_i$.
- `StartMonitoringRules(String` $\texttt{ruleID}_1$`, ...)` - The result of taking this reaction is to request the monitor to start monitoring a given set of rules. These rules are identified by the parameters $\texttt{ruleID}_i$. Again, all $\texttt{ruleID}_i$ are

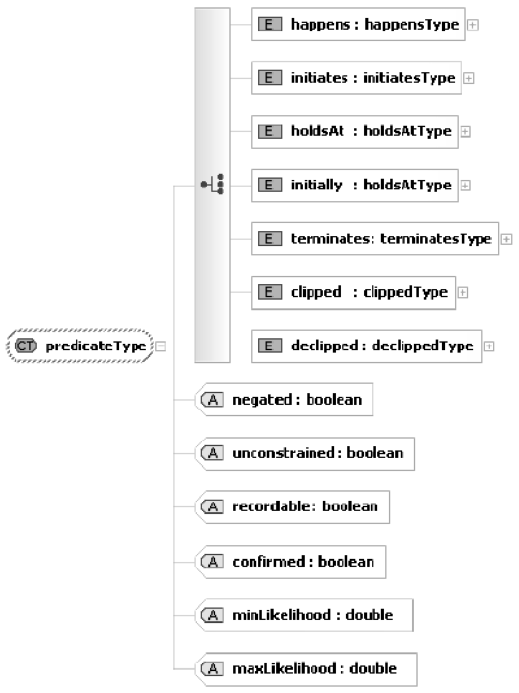


Fig. 5.5 Additions to predicateType in the new schema of monitoring results

restricted to take as values IDs of the rules of the S&D Pattern incorporating this reaction, and the SRF will render their IDs to globally unique ones. If one of the `ruleID`$_i$ corresponds to a rule which is already being monitored, the monitor will ignore the request for this particular rule and will not start monitoring a new instance of it (as this would be redundant and would decrease the performance of the monitor).

- `Log()` - The result of taking this reaction is to log the XML template which has been returned by the monitor to indicate the violation of a specific rule instance and the actions taken up to the point when `Log()` is executed for the particular violation. Thus, this reaction needs to be listed at the end of the action list for a particular rule and the SRF will need to keep a record of the actions that have been taken up to Log() in order to perform the required logging.

The aforementioned actions can be used freely by S&D Pattern developers - they can specify rules which have no actions associated to them or have more than one actions associated to them. In fact, the current S&D Pattern language allows developers to also express conditions under which an action should be performed for a specific rule result. So the specification of rules in the S&D Pattern will have the following form:

$$Rule[(action_1, condition_1), \cdots, (action_n, condition_n)]$$

Thus, when the value of a new rule instance has been established, the SRF will go through the rule's action list and execute all the actions whose conditions are true for that rule instance. These enabled actions will be executed in the order in which they have been specified in the S&D Pattern.

It should be noted here that the SRF does not react only when a rule instance is violated. As it has been briefly mentioned in the introduction, a rule instance may also have an associated threat risk level, which effectively estimates how probable it is that that rule instance will be violated. In addition to threat risk levels, rule instances can have associated diagnostic information (See Chapter 14). The conditions of the actions can refer to these extra threat & diagnostic information and, therefore, the SRF may very well take an action even when a rule instance has not been violated yet. For example, an S&D Pattern could be deactivated if the diagnostic information indicates that events received from its components are not genuine, even if there is no rule violation per se. In fact, the SRF may react, even when a rule instance is only partially evaluated, if for example its threat risk level is too high and an action has been conditioned on this.

5.5 Conclusions

This chapter has presented the languages of S&D Artefacts developed in the Project SERENITY. These artefacts are S&D Classes, S&D Patterns, S&D Implementations, and they have been developed in order to capture security expertise. The use of the languages of S&D Artefacts helps security experts to describe S&D Solutions in a standardized way.

It has started with a brief description of the artefacts and how they are related. This description reviews the most important elements conforming the structure of the S&D Artefacts, and it presents the rationale behind the chosen hierarchy. Next, it presents the detailed structure of each S&D Artefact. For every S&D Artefact both the language and an example is given. The examples are based on an AmI scenario. Finally, it extends the information about monitoring and reaction mechanisms.

The current line of this work focuses on the refinement of the structure, and on its extension in order to fulfil all issues resulting from the developed SERENITY prototypes.

References

1. Reiter, M. (1996) Distributing trust with the Rampart toolkit, Communications of the ACM, v.39 n.4, p.71-74.
2. BEA White Paper BEA WebLogic Security Framework Working with Your Security Eco-System. http://www.bea.com. Cited 6 July 2008.

3. Object Management Group. The Common Object Request Broker Architecture and Specification. http://www.omg.org. Cited 6 July 2008.
4. Llewellyn-Jones, D., Merabti, M., Shi, Q., B. Askwith (2004) An Extensible Framework for Practical Secure Component Composition in a Ubiquitous Computing Environment. In Proceedings of International Conference on Information Technology.
5. Fayad, M., Johnson, R., Schmidt, D.C. (1999) Building Application Frameworks Object-Oriented Foundations of Framework Design. Wiley & Sons.
6. Schumacher, M., Mouratidis, H., Giorgini, P. (2003) Security Patterns for Agent Systems. In Proc. of 8th European Conference on Pattern Languages of Programs.
7. Wooldridge, M., Jennings, N.R., Kinny., D. (2000) The Gaia methodology for agent-oriented analysis and design. Journal of Autonomous Agents and Multi-Agent Systems, **3(3)**, p.285.
8. Boudaoud, K., McCathieNevile, C. (2002) An Intelligent Agent-based Model for Security Management. In Proc. 7th International Symposium on Computers and Communications.
9. Nobukazu Y., Shinichi H., Anthony F. (2004) Security Patterns A Method for Constructing Secure and Efficient Inter-Company Coordination Systems. Enterprise Distributed Object Computing Conference.
10. Cigital Labs AOP An Aspect-Oriented Security Assurance Solution. http://www.cigital.com/labs/projects/1027/. Cited 6 July 2008
11. Shah, V., Hill, F. (2003) An Aspect-Oriented Security Framework. DARPA Information Survivability Conference and Exposition - Volume II, p. 143.
12. Llewellyn-Jones, D., Merabti, M., Shi, Q., Askwith, B. (2004) Utilizing Component Composition for Secure ubiquitous Computing. In Proceedings of 2nd UK-UbiNet Workshop.
13. Shi, Q., Zhang, N. (1998) An effective model for composition of secure systems. Journal of Systems and Software. **43(3)**, pp. 233-244.
14. Mantel, H. (2002) On the composition of secure systems. In Proc. of IEEE Symposium on Security and Privacy.
15. Canal, C., Fuentes, L., Pimentel, E.,Troya, J.M., Vallecillo, A. (2003) Adding Roles to CORBA Objects. IEEE Transactions on Software Engineering **29(3)**, pp. 242-260.
16. López, J., Maña, A., Ortega, J.J., Troya, J., Yague, M.I. (2003) Integrating PMI Services in CORBA Applications. In Computer Standards & Interfaces, **25, 4,** pp. 391-409. Elsevier.
17. Meling, R. (2000) Storing and Retrieving Software Components A Component Description Manager. In Proc. of the Australian Software Engineering Conference. IEEE.
18. Becker, S. (2006) Coordination and Adaptation Techniques Bridging the Gap between Design and Implementation. Report on the ECOOP'2006 Workshop on Coordination and Adaptation Techniques for Software Entities. Springer.
19. Khan, K., Han, J. (2002) Composing Security-aware Software. IEEE Software, **Vol. 19**, Issue 1, pp 34- 41. IEEE.
20. Brogi, A., Cmara, J., Canal, C., Cubo, J., Pimentel, E. (2006) Dynamic Contextual Adaptation. Workshop on the Foundations of Coordination Languages and Software Architectures. Electronic Notes in Theoretical Computer Science. Elsevier.
21. McDermid, J.A, Shi, Q. (1992) Secure composition of systems. In Proc. of Eighth Annual Computer Security Applications Conference, pp. 112-122.
22. Jaeger, T. (1998) Security Architecture for component-based Operating System. In ACM Special Interest Group in Operating Systems (SIGOPS) European Workshop.
23. Ghosh, A.K., McGraw, G. An Approach for Certifying Security in Software Components.
24. Kienzle, D.M., Elder, M.C. Final Technical Report Security Patterns for Web Application Development.
25. IBM's Security Strategy team (2004) Introduction to Business Security Patterns. An IBM White Paper. http //www-3.ibm.com/security/patterns/intro.pdf. Cited 6 July 2008.
26. Konrad, S., Cheng, B.H.C., Campbell, Laura, A., Wassermann, R. (2003) Using Security Patterns to Model and Analyze Security Requirements. In Proc. Requirements for High Assurance Systems Workshop.
27. Yoder, J., Barcalow, J. (2000) Architectural Patterns for Enabling Application Security. In Pattern Languages of Program Design, pp. 301-336. Addison Wesley.

28. Romanosky, S. (2001) Security Design Patterns, Part 1, 1.4.
29. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994) Design patterns Elements of Reusable Object-Oriented Software. Addison-Wesley.
30. Fernandez, E.B. (2006) Security patterns. In Procs. of the Eighth International Symposium on System and Information Security.
31. Fernandez, E.B., Rouyi, P. (2001) A pattern language for security models. PLoP'01.
32. Fernandez, E.B. (2000) Metadata and authorization patterns. Technical report, Florida Atlantic University.
33. Allenby, K., Kelly, T. (2001) Deriving Safety Requirements Using Scenarios. In Proc. of the 5th IEEE International Symposium on Requirements Engineering.
34. Mikkonen, T. (1998) Formalizing design patterns. In Proc. of 20th ICSE, pp. 115-124. IEEE Computer Society Press.
35. Wassermann, R., Cheng, B.H.C. (2003) Security Patterns. Technical Report **MSU-CSE-03-23**, Computer Science and Engineering.
36. Hallstrom, J. O., Soundarajan, N., Tyler, B. (2004) Monitoring Design Pattern Contracts. In Proc. of the FSE-12 Workshop on Specification and Verification of Component-Based Systems, pp. 87-94.
37. Hallstrom, J. O., Soundarajan, N. (2006) Pattern-Based System Evolution A Case-Study. In Proc of the 18th International Conference on Software Engineering and Knowledge Engineering.
38. Pernul, G., Essmayr, W., Tjoa, A.M. (1997) Access controls by object oriented concepts. In Proc. of 11th IFIP WG 11.3 Working Conference on Database Security.
39. Fernandez, E. B. (2004) Two patterns for web services security. In Proc. International Symposium on Web Services and Applications.
40. Delessy-Gassant, N., Fernandez. E.B., Rajput. S, Larrondo-Petrie, M.M. (2004) Patterns for Application Firewalls. PLoP'04 Conference.
41. Torsten, P, Fernandez, E.B., Mehlau, J.I., Pernul, G. (2004) A pattern system for access control. 18th IFIP WG 11.3 Conference on Data and Applications Security.
42. Androutsopoulos K, Ballas C, Kloukinas C, Mahbub K, Spanoudakis G (2007) Version 1 of the dynamic validation prototype. Deliverable A4.D3.1, SERENITY EU Research Project 027587, available from http //www.serenity-forum.org/Work-package-4-3.html
43. Shanahan MP (1999) The event calculus explained. In Wooldridge MJ, Veloso M (eds) Artificial Intelligence Today, vol 1600, pp 409-430.
44. Spanoudakis G, Tsigkritis T, Kloukinas C (2008) Second version of diagnosis prototype. Deliverable A4.D5.2, SERENITY EU Research Project 027587, available from http //www.serenity-forum.org/Work-package-4-5.html
45. Tsigkritis T, Spanoudakis G, Kloukinas C, Lorenzoli D (2009) Security and Dependability for Ambient Intelligence, Springer Verlag, chap Diagnosis and Threat Detection Capabilities of the SERENITY Monitoring Framework. Information Security Series.
46. Barthe G, Grgoire B, Pavlova M. (2008) Preservation of Proof Obligations from Java to the Java Virtual Machine. IJCAR 2008. 83-99.