SIXTH FRAMEWORK PROGRAMME
PRIORITY 2
"Information Society Technologies"

Project acronym: DESEREC

Project full title: Dependability and Security by Enhanced Reconfigurability

Proposal/Contract no.: IST-2004-026600-DESEREC

D2.1
Policy and System Models

Project Document Number: DESEREC/D2.1/PU[1]/v1.06 (official version)

Project Document Date: 30/05/2007

Workpackage Contributing to the Project Document: WP 2

Deliverable Type and Security: R[2]-PU

Author(s): POLITO, UMU, IEIIT, PWR, THALES

Abstract:

This document includes the initial description of the DESEREC modelling framework. It describes the different models needed to support the DESEREC architecture: both meta models design, language representations, and examples of their usage are presented.

Keywords: WP2, system models, policy models, meta models, requirements, services, infrastructure, operational plan

[1] Security Class: PU- Public, PP – Restricted to other programme participants (including the Commission), RE – Restricted to a group defined by the consortium (including the Commission), CO – Confidential, only for members of the consortium (including the Commission)

[2] Type: P - Prototype, R - Report, D - Demonstrator, O - Other

History

Version	Date	Description, Author(s), Reviser(s)
1.06	30/05/2007	PMT Approved version.

Executive Summary

This document is the D2.1 deliverable of the DESEREC project and it includes the initial description of the DESEREC modelling framework. It describes the different models needed to support the DESEREC architecture: both meta models design, language representations, and examples of their usage are presented.

The DESEREC project aims at increasing the dependability of existing and new networked mission-critical Communication and Information Systems (CIS) with a Business Services (BS) point of view.

The main levers are:

- Detect as proactively as possible incidents and contain them in a limited area of the CIS.
- When an incident causes a failure, the DESEREC Framework should handle it smoothly if not seamlessly.
- Reconfigure a subpart of the CIS based on the same resources (redistribution or substitution of existing resources).
- When no resources are available to resume performance of a high-priority service in accordance with the business rules, take the risk to stop low-priority services to release the needed resources.

To build the DESEREC framework a three-tiered approach is proposed

1. Planning of optimal configuration for anticipated operational modes on central level
2. Fast self-healing with priority to critical activities on global level
3. Incident detection and quick containment on local level

After having defined an Initial Architecture (*Initial System Architecture* deliverable) the end-user requirements (*Final User Scenarios and User Requirements* deliverable), this document lists the prototypes of the modelling tools at central level.

The *Policy and System Models* deliverable (D2.1) is very close to this D1.3 document as it defines the specifications for the modelling languages supported by the modelling tools.

The approach of this document is much related to the *Initial System Architecture* deliverable and it starts directly with this deliverable results and especially its functional decomposition.

Contents

Page

Figures

Tables

1 Introduction

1.1 Aim of Policy and System Models

The main interest of the proposed DESEREC approach is to improve the dependability of Communication and Information Systems (CIS) by a smart and powerful combination of three technological domains: Modelling & simulation, Incident detection, and Response. To succeed in this approach and to cover the whole problematic, DESEREC propose the following organisation of its activities, as illustrated by the Figure 1:

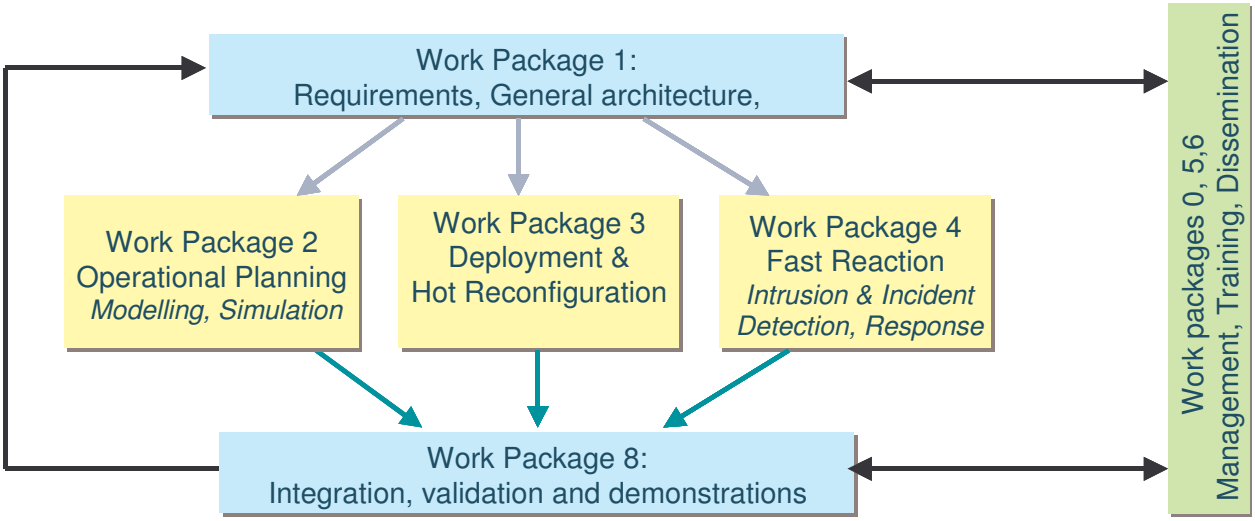

Figure 1: Work package breakdown of the project

- **WP1** (Horizontal Processes): addresses the user requirements, the risk analysis and the architecture design.
- **WP2 (operational planning)** deals with the creation of planned operational modes (or standard configurations). Its main activities are:
  - a thorough multi-level modelling of networked information systems, that keeps into account all aspects related to dependability
  - a policy system to formally express the desired dependability attributes
  - a validation and simulation environment, to assess the dependability properties and verify by simulation the expected behaviour under a certain configuration and expected faults
- **WP3 (deployment and hot reaction)** addresses the automated reconfiguration process (to take place in a time scale of minutes). Based on alarms generated by elements in WP4, the system is able to launch resume operations with under-nominal resources, targeting downgraded objectives but keeping mission-critical functions running.
- **WP4: (Fast self-healing):** The goal is to detect as close as possible from the origin (both in space and time) a fault, being that the result of an intrusion or an incident, to propagate the alert and launch local quick responses.
- WP2, WP3 and WP4 will produce prototypes that will be gathered in one global demonstrator **(WP8: Integration, Validation, Demonstration)** to show the improvement brought by DESEREC in the field of resilience.

The DESEREC modelling framework stands at the basis of the work package 2 "*Operational Planning*": it provides the basic conceptual and technical tools for formally describing the target information system, a policy system to formally express the desired dependability attributes, and the evaluated configurations of the system that are input to the Deployment and hot reaction, and Fast self-healing modules.

In order to design the DESEREC modelling framework, the main objectives of this document, DESEREC project must provide answers to the following questions:

- What models are actually needed to describe a large, service-oriented information system and its policies?
- What are the relationships among the different models?
- What is the information required to support configuration evaluation?
- What is the information required to support configuration deployment and management by DESEREC?
- How the information that populates the models is collected?

1.2 Scope and Structure of the Document

The goal of this document is to design part of the DESEREC modelling framework and in particular the policy and system models. In order to achieve this goal, it has been necessary to define precisely what models are needed and their relationships, and to specify a meta model and a suitable representation for each selected model.

This document consists of the following parts:

- Chapter 2 presents a list of requirements for the DESEREC modelling framework and a brief description of the usage of models within the whole DESEREC architecture.
- Chapter 3 presents the rationale and architecture of the DESEREC modelling framework.
- Chapters 4 and 5 present the models selected to describe the target information system and its dependability relevant aspects.
- Chapter 6 presents the models selected to describe the policy system to formally express the desired dependability attributes.
- Chapter 7 presents the models selected to describe the information system configurations.
- Chapter 8 summarises the achieved results and the future work.
- Chapter 9 lists the publication references.

2 Requirements on the DESEREC modelling framework

This section discusses the key requirements taken into consideration while developing the DESEREC modelling framework. Most of these requirements are documented in details in [1] and [2].

2.1 Multi-level models for system and policies

The DESEREC project aims at addressing in an integrated way the dependability and security issues arising in large networks providing high level services. In particular, the analysis, monitoring and dynamic reconfiguration of the system (with different time scales) have to be carried out, starting with the description of the whole system (i.e. nodes, links, software, network devices, high level services, and their configurations and policies). For this reason the DESEREC integrated approach has to cope with several domains, situations and details that have been dealt with separately till today, or even not addressed at all.

On the one hand existing well-assessed approaches (modelling, analysis, monitoring and reconfiguration techniques) must be preserved as much as possible, while, on the other hand, they need to be integrated into a single global framework.

In particular, such integration has to cope with techniques and formalisms already developed and available, and to merge them by adopting suitable extensions and enhancements in order to fill gaps coming from interoperability requirements and/or due to the integration process itself.

In addition, modelling should enable the description of the whole system in such a way it useful for all the processing activities that can be carried out on the model itself, taking into account the needs that are peculiar to each one of them (i.e. formal analysis, simulation, monitoring, reconfiguration and so on).

The meta-models this deliverable deals with are the following:

- System models: description of each single component of the system, ranging from physical objects (nodes, links) to business services, and related operational policies.
- Analysis models: description of the whole set of system behaviours whose analysis is considered useful (i.e. faults and vulnerabilities).
- Operational models: description of each (meaningful) system configuration, by means of operational configurations, detection scenarios and reaction scenarios.

Moreover, in order to make the DESEREC approach viable and manageable, a structured, hierarchical organisation of models is needed where the system can to be viewed as either a single entity (e.g. a global level where few details are taken into account) or consisting of interacting sub-systems (e.g. a local level where each sub-system is described by its peculiar attributes in their broadest sense).

The meaning of the global system and the sub-systems models, their structure and nature and their relationships heavily depend on the processing activities that have to be carried out on them: for example, let us consider a sub-system consisting of either a set of resources grouped on a geographical basis, when fault propagation analysis has to be carried out, or a set of resources geographically uncorrelated which is able to support a certain business service.

This approach is well understandable, but its feasibility requires that each (sub-) system view is kept coherent with respect to both the other ones and the global system model. This goal can be achieved by guaranteeing that each description related to the whole system has a coherent counterpart (or specialization) in each sub-system, and vice-versa.

The final result is a set of meta-models able, on the one hand, to describe all the system components and behaviours and, on the other hand, to scale easily and coherently from the high level (global) views to the local views and vice-versa.

2.2 Models in the design part of the DESEREC architecture

The design part of DESEREC Architecture consists of a toolbox split into three separate domains: Modelling, Planning and Analysis (for more details refer to section 3.1.2.2 of D1.3 document) as shown in Figure 2.

This subsystem provides the main input to the DESEREC tools dedicated for direct increasing dependability and security by enhanced reconfigurability (i.e. online part). Major functions of each domain (specified in details in D1.3) may be briefly presented as follows:

- Planning – Its main role is to provide validated configurations that can be enforced in the system. Moreover, Planning provides information that helps to decide when and how to change the current configuration.
- Analysis – Its main role is to evaluate a set of known configurations against dependability metrics, i.e. security and performability.
- Modelling – Provides meta-models to describe mainly the system and the offered services, the configurations and the constraints they are subjected to. Moreover, Modelling design functions to validate the consistency of the models built at design time based on its meta-models.

The very basic idea of design part's workflow is:

- DArchitect develops the way how to describe managed system and issues closely related to it (like requirements – e.g. Performance Level Agreements & Dependability Level Agreements, constraints – e.g. security policies, and all known threats that might affect the system), i.e. provides so-called meta-models (languages).
- DDesigner uses meta-models to build managed system and its environment representation; the results are Infrastructure, Inputs, Faults, Vulnerabilities, and Requirements models, and constraints referring to system configuration and scenarios.
- Modelling module check the validity of the whole System Model based on the DDesigner input.
- Planning produces Operational Configurations.
- Analysis tools are supplied with Operational Configurations and appropriate models to perform conformance, security and performance analysis.
- Analysis results are merged, and associated to the relevant Operational Configuration.
- Planning updates the Operational Plan, i.e. set of Operational Configurations available to employ on the real system alongside with related to each configuration: Detection and Reaction Scenarios.
- Design part delivers its products (i.e. model – System Model together with models contained in Operation Plan) to the online part.

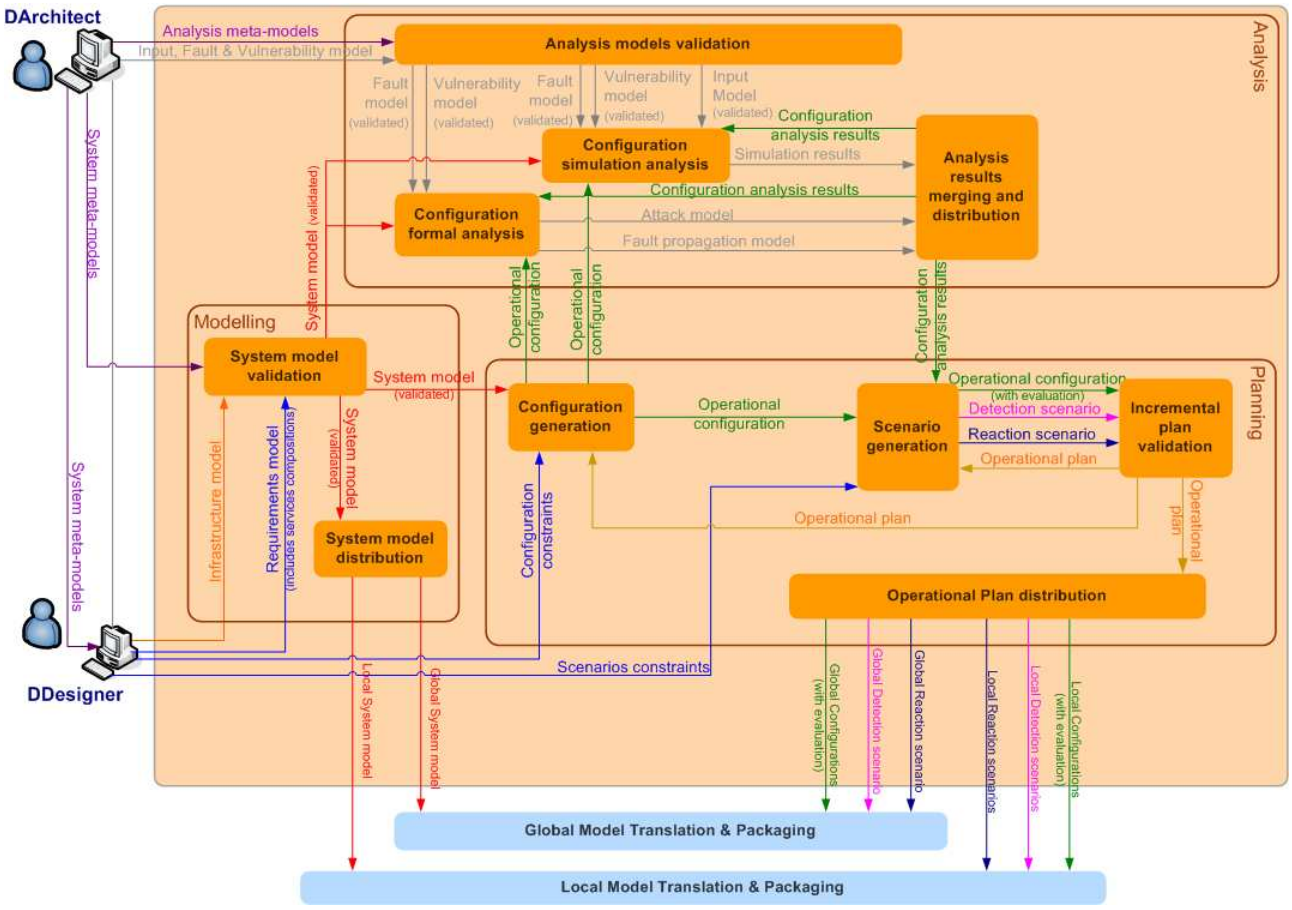

Figure 2 - DESEREC architecture (design part)

2.2.1 *System Models*

This complex model is in fact a group of diverse models. It lies the basis for DESEREC operability by representing the models that are used to describe the target system in terms of its structure, behaviour and characteristics. Typical for System Model is its capability to define diverse components at different level of abstraction. This allows finally obtaining and disseminating a local system and global system models. In particular System Models addresses

- Physical infrastructure – System Model must reflect in great detail components composed in the following groups: computer hardware (e.g. processing power, memory, NIC, etc.), computer software (e.g. OS, server applications), and network devices (e.g. links, traffic controllers – hubs/switches/gates/routes).
- Services – The model at least has to define implemented services, the way how they interact among them and how they are composed to realize higher level services (business services), i.e. orchestration. If this would be desired due to nature of service – also choreography description (e.g. communication between modelled IS's service1 and banking IS's service2) may be applied. Additionally for analysis purposes time orientation will be very helpful.
- Operational policies – This model precise system functionality by defining a set of rules which represent requirements and constraints with respect to the two areas: dependability (security and performability) and configuration. Hence this model contributes not only to the System Model, but also to a few other models (which are: Configuration Constraints and Scenarios Constraints models) directly used by various functional blocks within design part.

The meta-model for System Model is provided by DArchitect. Model itself is concatenation of Infrastructure Model and Requirements Model offered by DDesigner.

2.2.2 Analysis models

2.2.2.1 Inputs Model

The Inputs Model formulates a set of so-called use-cases which are needed by Configuration Simulation Analysis tool to exercise the System Model correlated with Operational Plan by exposing it to the specific scenarios (regular and corrupted) that it should be able to handle. Use-cases are also used to test the system dependability attributes. For this reason use-cases contain information on external events representing occurrence of incidents that might affect the system, i.e. inoperability (hardware and software failures described by the Faults Model) and known methods of attacking a system (as defined by Vulnerabilities and Faults models).

A single use-case describes an instance of interaction between an end-user and the system: sequence of events and awaited reactions. However, time oriented analysis requires not just the sequence of events, but must include information on time orientation and undesired (erroneous or malicious) events.

The Inputs Model is provided by the DDesigner.

2.2.2.2 Faults Model

Faults Model contain the set of faults the System Model can be subjected to. Furthermore this model has to cover the following issues: faults of different types, their preconditions, and their direct effects.

Particularly three types of faults causing service failure are in scope of DESEREC's interest, i.e.:

- physical (accidental);
- information (accidental) – being reason for software temporary inoperability (e.g. hang-up due to implementation mistake), usually maintained by restart action;
- deliberate (malicious) – attacks.

These faults require completely different describing capabilities, thus Faults Model has to be flexible enough to enclose all. For example physical fault of some hardware component typically triggers failure of all services running on that host. Moreover they are rather simple to model and forecast using Dependability Theory, as well as detect and maintain. But information faults are more complex and unpredictable by nature. Even though software falls are not easy to model and forecast but rather easy to detect and maintain, attacks require special treatment. In this case Faults Model must provide patters of known attacks together with additional information, like: target, typical result, etc.

The Faults Model is provided by the DDesigner.

2.2.2.3 Vulnerabilities Model

The Vulnerabilities Model contains the set of known vulnerabilities an attacker can try to exploit to conduct an intrusion against the target system. It represents part of the initial knowledge of the attacker. For each vulnerability, the model must indicate the most important characteristics of it, e.g. CVE id, pre-conditions that make the vulnerability exploitable, and effects of its exploitation on the victim system. This model should be created in form of repository based on public vulnerability databases, e.g. NIST-NVD, US-CERT-VND, OSVDB, etc.

The Vulnerabilities Model is provided by the DDesigner.

2.2.3 Operational models

2.2.3.1 Operational Plan

The Operational Plan is the main output of the DESEREC design part towards the online part. It is mainly a set of pre-validated Operational Configurations that have been considered eligible to be enforced in the target system, along with associated information that helps to detect problems affecting specific configurations (Detection Scenarios) and to select proper configuration changes as a reaction (Reaction Scenarios).

The Operational Plan is built incrementally as the joint effort of the Configuration Generation, Scenario Generation and Incremental Plan Validation functional blocks. The workflow is defined as follows:

- Configuration Generation generates configurations that may be included in the current Operational Plan;
- Scenario Generation selects configurations that are actually added to the plan and generates the associated detection and reaction information;
- Incremental Plan Validation maintains synchronisation of all the different parts of the current Operational Plan.

2.2.3.2 Operational Configurations

Operational Configurations create a set of instances of the System Model supplied by operational policies, which role is modelling specific configurations that can be enforced in the system. The configuration model mainly includes: the deployment of services onto available resources, and the configurations of these resources.

The configuration model is organised along multiple layers of abstraction, i.e. a Global Configuration Model dealing with the service deployment and configuration of a set of resource aggregation blocks (molecules), and a Local Configuration Model describing the internal services and configuration of each block.

DESEREC implements Operational Configurations at policy level descriptions.

Operational Configuration is produced by the Configuration Generation functional block based on the System Model, and Configuration Constraints.

2.2.3.3 Detection Scenarios

This is associated to a specific Operational Configuration "configuration of the detection process" at appropriate level of abstraction. By including a set of rules describing the way to identify possible problems regarding some events in the managed system, it helps to detect incidents that may affect a specific configuration. Events previously collected by DESEREC resources (refer to sections 3.2.1.1, 3.2.1.4/3.2.2.2 of D1.3) are matched against the events description in the current Detection Scenario, and thus they become a true triggers of alarms generated during detection process within Local/Global Detection functional blocks (refer to sections 3.2.1.5 and 3.2.2.3 of D1.3). Hence, Detection Scenario describes:

- Set of measures to be monitored/computed;
- List of pairs event-problem, where problem identifies the supposed trouble with managed system;

Whereas the event within Detection Scenario is connected with specific sets of measures acquiring values satisfying specific conditions, thresholds, ranges, and possibly significant correlations between them.

The Detection Scenario model reflects the multiple layers of abstraction used to model the system and its configuration – we have: Global Detection Scenarios helping to detect problems affecting the global configuration, and Local Detection Scenarios helping to detect problems affecting local configurations.

Detection Scenario is produced by the Scenario Generation functional block based on the associated Operational Configuration, Scenarios Constraints and configuration analysis results coming from the Analysis Results Merging and Distribution functional block.

2.2.3.4 Reaction Scenarios

Reaction Scenarios is a set of prescriptions that specify for each problem identified using Detection Scenario the available reactions to be triggered. Basically reaction is equivalent to switching action (from present to a new proposed validated Operational Configuration). For that reason Reaction Scenarios present to DAdmin available modifications from current Operational Configuration to the new ones, together with data associated to each of them, that helps to make a right choice assuming customized criterion. Such principle may be based on e.g. information about cost of alternation, predicted new values of system state metrics under comparable environmental conditions, etc.

Reaction Scenario model reflects the multiple layers of abstraction used to model the system and its configuration – we have: Global Reaction Scenarios helping to select changes in the global configuration, and Local Reaction Scenarios helping to select changes in the local configurations.

Reaction Scenarios is generated by the Scenario Generation functional block based on the associated Operational Configurations, Scenarios Constraints and configuration analysis results coming from the Analysis Results Merging and Distribution functional block.

2.3 Usage of models in the online part of the DESEREC architecture

As stated in the previous section, we enumerate two main products of design part that are utilized by the online part during management control of DESEREC Framework over underlying real Information System. These are System Model, and Operational Plan. Both of them are models being passed to the same online part's functional blocks, for further processing and dissemination. Since we distinguish in both cases two levels of abstraction, both for System Model and Operational Plan, according to the description granularity products are being passed to appropriate module – Local or Global Model Translation & Packaging functional block.

2.3.1 Global level

The Global Model Translation & Packaging module as the intermediary between design and online parts of the Framework receives all models, transforms them into an operational format, suitable and understandable by the rest of DESEREC modules, and pushes to appropriate recipients within online part.

Accordingly to this, first this Functional Block takes delivery of the Global System Model from the System Model Distribution module (referred to also as Systems Distribution), translates it into the appropriate operational format which is services orchestrations & global infrastructure description and sends it to the Global System Situation block.

Furthermore, global distributor has to deal with Global Operational Plan. As result Global Model Translation & Packaging produces global detection scenarios and global reaction scenarios information in order to be made them available for the Global Detection and the Global Decision modules, respectively. On the basis of received information these modules generate several communicates internal to the online part, e.g. global alarms, global reaction proposals, global reaction orders, etc. At the end evaluated Global Configurations that are also received by Global Model Translation & Packaging as part of Global Operational Plan are being translated into an operational format. Next they come across to two other blocks: the Global System Situation, and Global Reaction. In the first case directly obtained global operational configurations together with previously gathered services orchestrations & global infrastructure description allow to construct global state information, and in the latter case this is more "package to be deployed" delivery since these packages in essence are groups of configurations, which have been created based on specific criteria.

Product	**Functional Block (producer)**	**Global Model Translation & Packaging information (supplier)**	**Offline part model (source)**
global state information	Global System Situation	services orchestrations & global infrastructure description	*Global System Model*
		global operational configurations	Global Operational Plan (*Global Configurations* with evaluation)
global reaction proposals	Global Decision	global reaction scenarios	Global Operational Plan (*Global Reaction Scenario*)
		ad-hoc reactions info[3]	none
global reaction orders		global reaction scenarios	Global Operational Plan (*Global Reaction Scenario*)

[3] This information is not mentioned in the above wording because this is rather Reaction Scenarios model instance, created by administrator the fly, than design part product.

global alarms	Global Detection	global detection scenarios	Global Operational Plan (*Global Detection Scenario*)

Table 1 - Usage of Offline part models within global section of the Online part

2.3.2 Local level

The Local Model Translation & Packaging module as the intermediary between design and online parts of the Framework receives all models, transforms them into an operational format, suitable and understandable by the rest of DESEREC modules, and pushes to appropriate recipients within online part.

Accordingly to this, first this Functional Block takes delivery of the Local System Model from the System Model Distribution module (referred to also as Systems Distribution), translates it into the appropriate operational format which is local infrastructure & services description and sends it to the Local System Situation block.

Furthermore, local distributor has to deal with Local Operational Plan. As result Local Model Translation & Packaging produces local detection scenarios and local reaction scenarios information in order to be made them available for the Local Detection and the Local Decision modules, respectively. On the basis of received information these modules generate several communicates internal to the online part, e.g. local alarms, local reaction orders, etc. At the end evaluated Local Configurations that are also received by Local Model Translation & Packaging as part of Local Operational Plan are being translated into an operational format. Next they come across to two other blocks: the Local System Situation, and Local Reaction. In the first case directly obtained local operational configurations together with previously gathered local infrastructure & services description allow to construct local state information, and in the latter case this is more “package to be deployed” delivery since these packages in essence are groups of configurations, which have been created based on specific criteria.

Product	**Functional Block (producer)**	**Global Model Translation & Packaging information (supplier)**	**Offline part model (source)**
local state information	Local System Situation	local infrastructure & services description	*Local System Model*
		local operational configurations	Local Operational Plan (*Local Configurations* with evaluation)
local reaction orders	Local Decision	local reactions scenarios	Local Operational Plan (*Local Reaction Scenario*)
local alarms	Local Detection	local detection scenarios	Local Operational Plan (*Local Detection Scenario*)

Table 2 - Usage of Offline part models within local section of the Online part

3 The DESEREC modelling framework

For supporting the description of large heterogeneous systems, the DESEREC modelling framework articulates along three layers (see Figure 3):

- *The goal level* – describes the goals of DESEREC in the target system, that is deploying and managing a set of business services given an available network infrastructure and associated resources.
- *The policy level* – describes the set of policies DESEREC should comply with when fulfilling its goals, that is constraints to the way DESEREC (re)configure the target system. This includes constraints on which resources the business service components are deployed (allocation policies), and constraints on how to configure those resources to fulfil both functional and security properties.
- *The plan level* – describes the configurations selected by DESEREC to fulfil its goals and the rules to switch among them (the switching rules are not shown in Figure 3). Each configuration includes the description of a specific service deployment, the additional resources added to the infrastructure to support the deployment (structural configuration), and the configuration rules of every involved resources (behavioural configuration). Configurations are generated based on the information included in the previous levels.

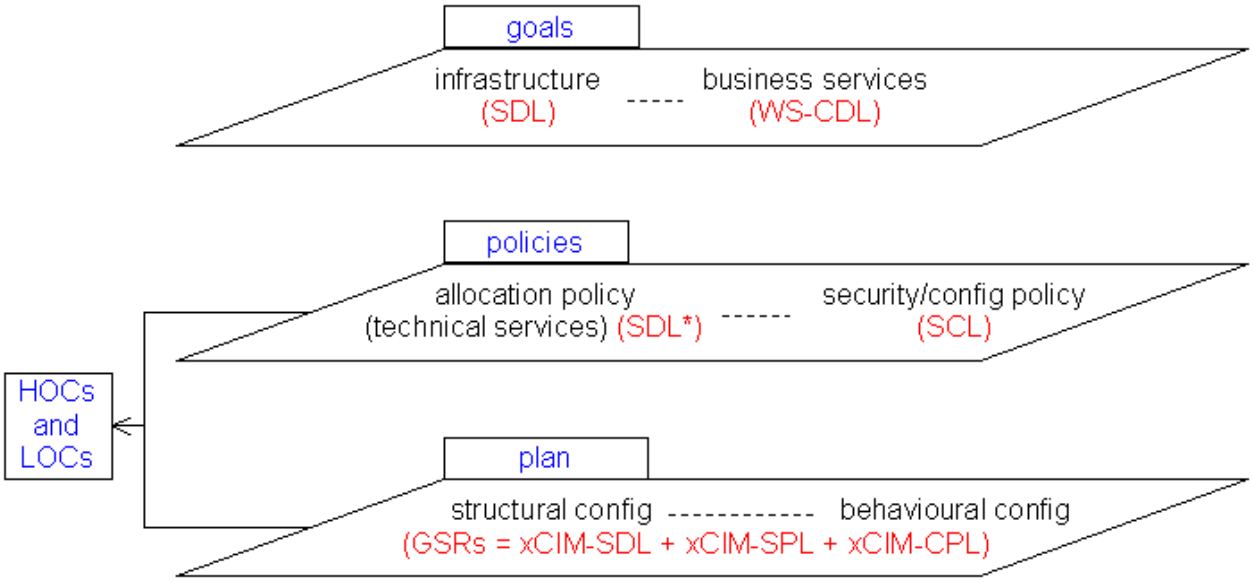

Figure 3 - Meta models and languages (configuration)

The separate modellisation of services, infrastructure, policies, and possible configurations has several advantages:

- Each model can be built and maintained by the most appropriate people, e.g. infrastructure models can be handled by system operators, service models by application designers, security policy models by security experts;
- Each model can be enhanced more independently to fulfil the requirements of specific systems, e.g. in terms of specific equipment, security and dependability mechanisms, levels of abstraction;
- Complex service interactions can be analysed independently of the exact security and dependability mechanisms put in place during their deployment (e.g. replication level, communication channels, service outsourcing);
- Policies can be chosen, evaluated and updated independently form the rest of the system to comply with mutable strategic decisions or normative constraints;

- Multiple configurations can be pre-built and analysed that have different security and dependability properties (e.g. better offered QoS at the cost of decreased resiliency to attacks), which can be switched a runtime upon reaction to specific events.

The following modelling languages were selected for the current implementation of the DESEREC modelling framework:

- The Services are modelled with the W3C's WS-CDL (*Web Services Choreography Description Language*) empowered with a set of extensions.
- The infrastructure and resource descriptions are modelled with the PSDL (*POSITIF System Description Language*) developed by the POSITIF project, empowered with a set of extensions.
- The allocation policies are modelled based on extensions to PSDL.
- The configuration and security policies are modelled with the SCL (*Services Constraint Language*), an ad-hoc XML language to express service configuration constraints and security policies.
- The configurations are modelled based on an XML representation of the DMTF's CIM: in particular the GSR (*Generic Service Rulesets*) format allows to describe the configuration of the system elements including element description (xCIM-SDL, CIM-based *System Description Language*), security configuration rules (xCIM-SPL, CIM-based *Security Policy Language*), and other service configuration rules (xCIM-CPL, CIM-based *Configuration Policy Language*).

Figure 4 depicts how the DESEREC modelling framework is completed with a detection and reaction model, which describes when and how to reconfigure the system in response to an incident:

- The detection scenarios describe how a set of events detected on the monitored system are mapped to a specific problem (this also helps to define which events must be monitored on the target system).
- The reaction scenarios describe how the enforced configuration should change in reaction to a set of detected problems.

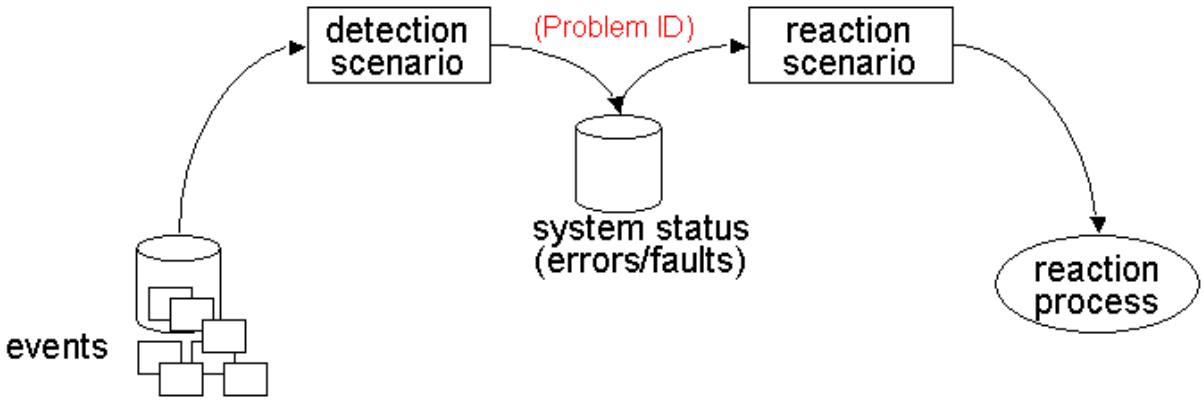

Figure 4 - The detection and reaction model

Based on the configuration, detection and reaction scenarios, we can roughly describe the model of the DESEREC detection and reaction decision process:

1. A set of collected and correlated events is processed against the detection scenario rules.
2. Each detection rule compute a set of changes in the system status model. Each change set is associated to the set of events that the detection rule has matched.
3. The current system status model (that is the change sets induced by the detection scenario) is processed against the reaction scenario rules.
4. Each reaction rule compute a set of changes to the current configuration (at limit the change of the whole configuration).

5. The suggested changes computed by the reaction scenario are made available to the re-configuration process.

Both detection and reaction scenarios are generated based on a set of policies modelled with the MPL (*Monitoring Policy Language*), an ad-hoc XML language to express monitoring policies.

The operational plan model associates the selected configurations and the detection and reaction scenarios to switch among them. Actually, the DESEREC architecture, as defined in [2], requires the operational plan model to be split into two levels as shown in Figure 5:

- *High (global) level operational configurations* (HOCs) – describe how business services components are deployed on the available molecules.
- *Low (local) level operational configurations* (LOCs) – describe how business service components are deployed onto specific resources available in a molecule, and how those resources are configured.
- *High (global) level detection and reaction scenarios (GDS, GRS)* – describe when a set of globally detected events suggests changing the current HOC, and which other HOCs are to be considered for reaction.
- *Low (local) level detection and reaction scenarios (LDS, LRS)* – describe when a set of locally detected events suggests changing the current LOC, and which other LOCs are to be considered for reaction.
- *The high (global) level operational plan* (HOP) – describes the possible HOCs selected for global system reconfiguration and how to switch among them in case of global incidents.
- *The low (local) level operational plan* (LOP) – describes the possible LOCs selected for reconfiguration within a specific molecule and how to switch among them in case of local incidents.

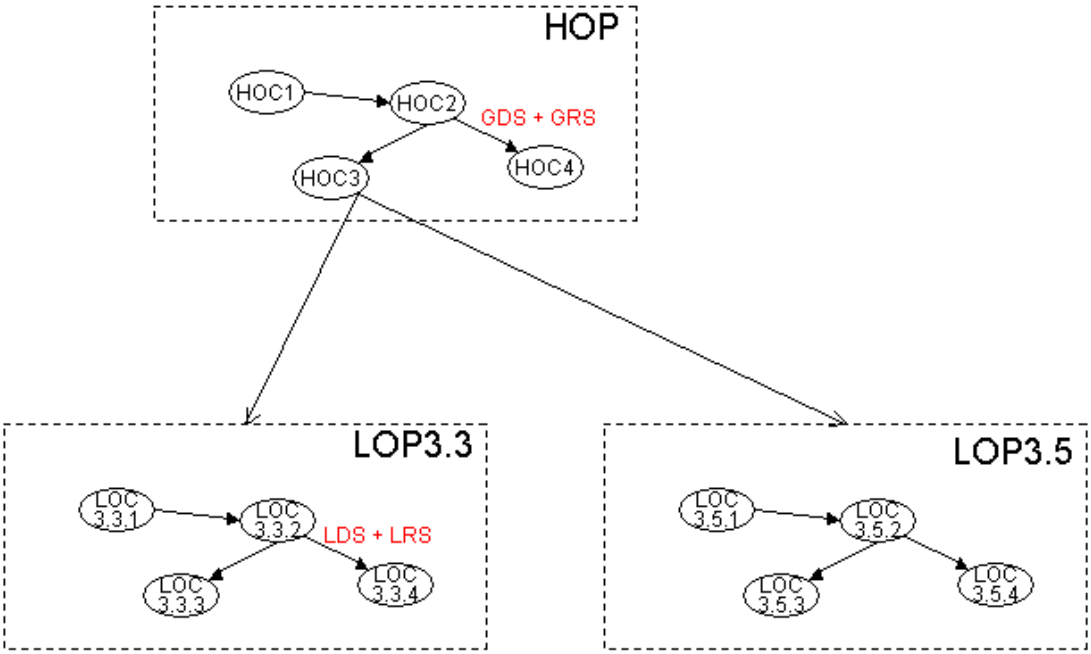

Figure 5 - The operational plan model

Figure 6 shows how the cited modelling languages are used within the design part of the DESEREC architecture. The following subsections discuss the models in additional details.

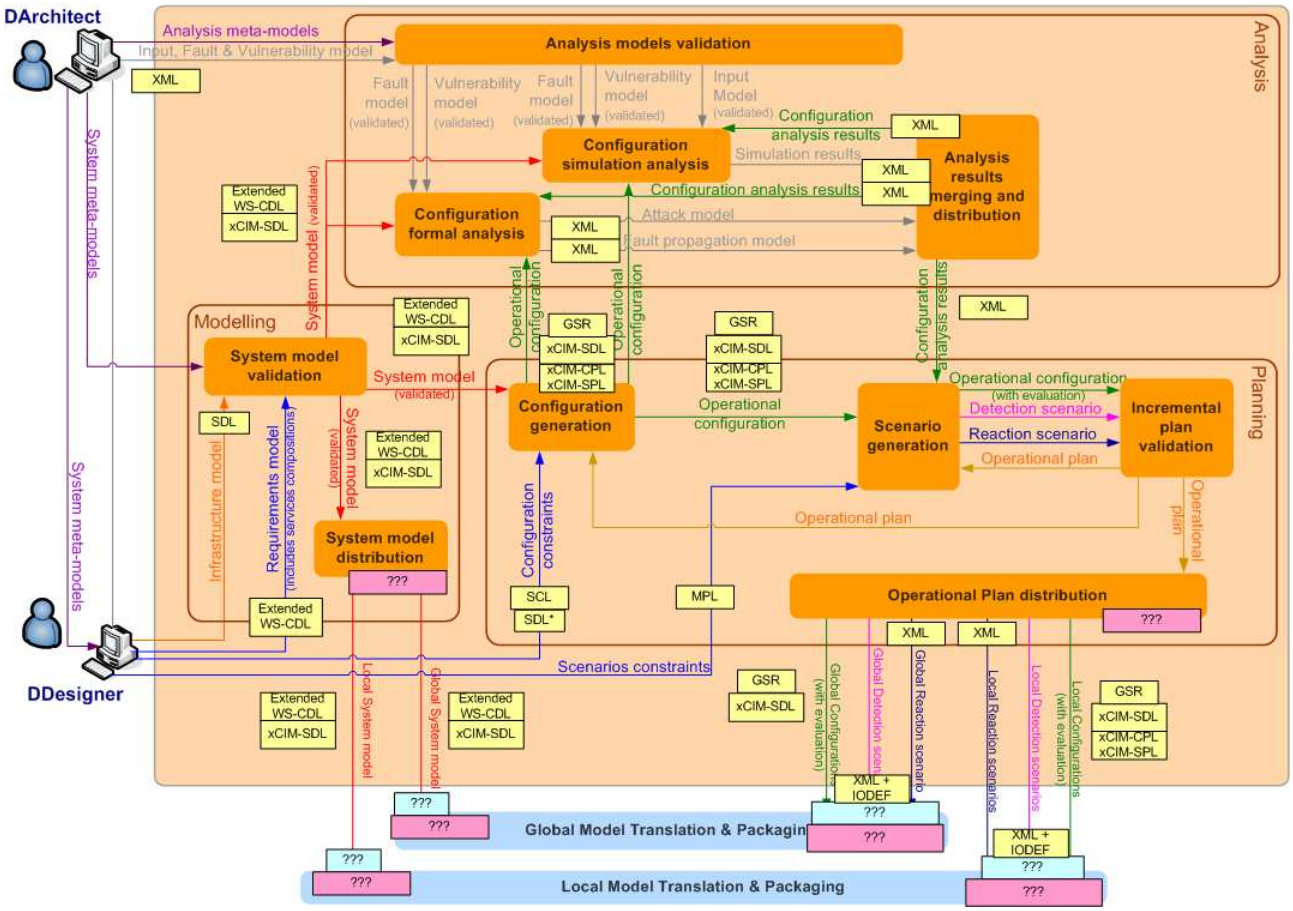

Figure 6 - Models and languages within the DESEREC architecture (design part)

3.1 Rationale of the service models

The next modelling concepts were selected to satisfy requirements on the service model:

- *interface modelling* – besides service components we explicitly model their interfaces. This helps thinking in terms of component diagrams rather than class diagrams, much like you compose an IC. In fact, service composition is wiring together the interfaces exposed by a set of service components.
- *behavioural modelling* – besides static composition, we model the interaction of service components. In accordance with the previous concept, we describe the behaviour of components by describing the protocols they expose through a set of interfaces. Thus, the model of the behaviour associated to an interface should include not only (1) the messages exchanged through the interface, but also (2) the activities carried out by the component in response to messages received through the interface, (3) the state internal to the protocol as it evolves in response to message exchanges, (4) internal events (such as timeouts) that may affect the protocol internal state.
- *multi-party protocols* – we address modern service architectures that go beyond the client-server paradigm and may involve interaction among more than two parties. According to the previous concept, the behaviour we model can encompass message interactions involving more than two parties, and the protocol state can be distributed among more than two parties.
- *process composition* – besides services composition we support process composition, that is the ability to reuse, merge, and possibly transform basic behaviours into more complex ones. The simplest example is a component exposing multiple independent interfaces. A typical example is the reuse of patterns such as an authentication protocol within a more complex interaction.
- *requirements modelling* – the service model must support the specification of the requirements the service behaviour should comply to. This includes at least performance, security and dependability requirements.

The service model selected for the DESEREC framework bases on three pillars:

- *Choreography description* – we chose a choreography view, where choreography "defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state" [6], because it natively supports multi-party protocol description and fit better interface modelling.
- *Choreography transformation* – in order to fully support process composition, we explicitly model the merging of multiple choreographies.
- *External Service Level Agreement (SLA) modelling language* – the model provides interface to external languages for service level agreement description.

Choosing a choreographic description as the base paradigm fits a top-down approach to service modelling. A top-down approach is the best when viewing the service model as a way to express requirements for the DESEREC operational planning. It makes easy to define high-level service descriptions, not dependent on the actual behaviour of specific COTS; without precluding a subsequent refinement of the model with additional, COTS-dependent details. On the other side, provided the choreography description complies with a set of constraints (e.g. assigning each activity to a specific actor in the choreography), we can extract per-component descriptions.

On the practical side, the choice is to extend the W3C's Web Services Choreography Description Language [9] with the following set of enhancements:

- *WS-CDL light* – a front-end to WS-CDL that makes descriptions lighter and easier to be provided directly by the DESEREC user. The translation between WS-CDL light and WS-CDL is without any information loss;
- *Choreography transformation extensions* – extended support for choreography composition.

The service meta model and its usage are discussed in section 4.

3.2 Rationale of the infrastructure and resource models

SDL is the language defined by the POSITIF project (www.positif.org) to provide a format for description of networked ICT systems, mainly in its physical (i.e. hardware) and logical (i.e. software) infrastructure. Although SDL attempts to be general and complete, its main purpose is to provide the data needed to perform various kinds of security analysis.

This means that obtaining a suitable description of a given network environment is a required goal: this description must be formal, precise, compliant with a simple high level meta-model, and detailed enough to support the automatic evaluation of security and dependability properties.

SDL is a user-friendly language, based on XML. SDL is meant to be a high level language: having its own XML Schema, it's easy to write a network description. In order to be saved on a repository, and used by automatic systems, the SDL description is translated into xCIM, an XML version of the Common Information Model (CIM) developed by DMTF. This translation is done using an ad-hoc XSL Stylesheet. Since CIM is very verbose, with many useless (for our purposes) elements, and not very human readable, the choice fell on the definition of a new language like SDL.

The resource meta model and its usage are discussed in section 5.

3.3 Rationale of the policy models

Different kinds of policies are needed in DESEREC to implement the concepts depicted in Figure 6 about how the system must behave normally. Bearing in mind those concepts, we must model:

- Service requirements: what features must implement the services.
- Configuration policies: the configuration itself of the different components of the system.
- Security policies: they are rules that dictate how to isolate attacks and to prevent risks of security.
- Monitoring policies: the dependability features that the system must fulfil in order to prevent and react properly when not desirable things could occur in the system.

The policy models above should define requirements, policies and capabilities in a simple and effective way, allowing the extension of the specification to suit any particular needs. The choice for this policy model should ideally comply with some properties such as readability or extensibility, among others. More specifically, we could point out the following features that we could desire in the chosen model:

- Simple and easily extensible in order to model any kind of policy; that is, it must be able to represent both security and dependability requirements. Thus, we should be able to future updates in order to include new features.
- Readability. Ideally, this model should be text-based for improved human understanding.
- Based on consolidated standards by the international community.
- Automatic processing and validation.
- Capability to express static information, as well as dynamic rules.
- Support for hierarchical grouping of data.
- Ease of interoperation.
- It must support the definition of event-triggered actions in order to model operational rules.
- It must be independent of any implementation or repository, allowing match-based retrieval.

The enumerated list of desired features makes XML-based notations an excellent choice for this policy model. Some properties such as readability, extensibility or vendor independence are inborn to XML. Others such as automatic processing and validation can be achieved by using XML processors and transformers. XML databases are widely available, with a fully standard XML-RPC interface which unifies the access to all of them, and with support for the XPath locator and pattern-matching language.

The policy meta model and its usage are discussed in section 6.

3.4 Rationale of the configuration models

Operational *configurations* play the main role in operational plan models. They draw a link between the service and resource models by describing how to deploy services onto resources available in the system. In fact, service and resource models describe two static views of the system, the intended behaviour of the system and its infrastructure, that are out of the control of the DESEREC re-configuration functions. Configuration models fill the gap by specifying the configurable part of the system that, under the control of DESEREC, is deployed on the infrastructure to match the intended behaviour.

On the practical side, specifying the deployment of a service component on the system infrastructure requires to describe:

- the collection of processes that provides the service;
- where each of these processes runs and which process is in charge to activate and maintain them;
- the collection of software packages (program and data) needed to run and activate these processes;
- how each of these software packages can be accessed either by the running service processes or by processes in charge of activating and maintaining the service processes.

Configuration models build upon a base model of the configuration actions they can be deployed. For configuration sake, resources can be classified as:

- *permanent versus temporary* - Permanent resources are pre-installed, may be only turned on/off (e.g. all hardware resources, pre-installed software), while temporary resources can be installed/removed (e.g. most software packages).
- *fixed versus flexible* - Fixed resources are pre-configured, beyond DESEREC authority (e.g. firewall managed by third party), while flexible resources can be dynamically configured by DESEREC.

From the previous classification, we can identify three main classes of configuration actions:

- CA1 - turn on/off (start/stop) permanent hardware (software) resources
- CA2 - install/remove temporary resources

- CA3 - install configuration rules on flexible resources (configuration rules corresponds to the behavioural part of the configuration)

We can now define operational *configurations* as composed of two set of information:

- *structural* - for each service component, the hardware and software resources that provide it, along with the deployment of any other resources they depend upon;
- *behavioural* - for each hardware and software resource at the previous point, how they are configured to satisfy the service requirements (including functional and non-functional ones) and the global policies enforced in the system.

Finally, we may want to describe configurations in two alternative ways:

- *absolute* - for each OCx, describe the changes needed to modify a common initial state OC0 into the desired target configuration OCx. The structural part of OC0 is the system infrastructure outside the DESEREC control, while the behavioural part is an empty configuration set.
- *differential* - allow to describe configuration OCx starting from any existent configuration OCy. This is equivalent to describe the changes needed to modify OCy into OCx.

For the moment, we model only absolute configuration and leave differential configuration as a next extension. Note that this choice allows us to describe only additions to the base infrastructure model (i.e. OC0) and postpone the description of deletions.

The structural part can be modelled as a mapping between a service component instance in the service model and an element in the resource model. Associated to every mapping is a list of additional resources the service component depends upon. More formally:

```
structural( OCx , Sy , PRz ) = { ( TRi , Rj ) }
```

where OCx is the configuration, Sy is a service component instance, PRz is a permanent element in the static resource model where Sy is deployed, TRi is a temporary resource required Sy depends upon, and Rj is resource (either a permanent or a temporary one) where TRi is deployed.

The behavioural part can be modelled essentially in terms of a set of generic rules that prescribe the runtime configuration of the elements, either permanent ones in the static resource model or temporary ones defined in the structural part. More formally:

```
behavioural( OCx , Ry ) = { { Ei } -> Aj }
```

where OCx is the configuration, Ry is a configurable resource addressed by the configuration, { Ei } is a condition expressed as a set of events, and Aj is some configurable action that can be enforced by Ry.

A goal of the configuration model is to decouple the function that designs target operational configurations, and the function that designs an optimised sequence of deployment commands. These two functions work at different levels: the former could be independent from the specific element configuration interfaces that are the main concern of the latter. Only at enforcement time, an operational configuration will be transformed into a sequence of element configuration actions (e.g., configuration data and sequences of commands for specific elements).

The configuration meta model and its usage are discussed in section 7.

3.5 Rationale of the detection and reaction models

A Detection Scenario (DS) is a list of pairs of type "*event → problem*", associated to a specific Operational Configuration; when such event (that may itself be a group of more events) is taking place, then any of the problems might be present and therefore we are detecting them. More formally:

```
DS(OCx) = {({Ei} → Pj)}
```

where

- `OCx ∈ {OC1,…,OCn}` is the current configuration, an element of the set of all possible Operational Configurations;
- `{Ei}` is a set of predicates over all observable events;

- `Pj ∈ {P1,…,Pk}` is the possible problem that could explain `{Ei}`.

Note that there could be more than one pair with the same {Ei} as different problems could generate the same events.

Events make the connection with the monitoring and correlation functions working in real time on the target system. The sets of events collected and inferred by these functions are matched against the events description in the current DS (that is the DS associated to the configuration currently enforced). The events model should use a representation that helps the interaction with monitoring and correlation functions.

In order to match this requirement, the events model is parametric to allow compact representation of a DS and handling events referring to thresholds and ranges. In fact, each Ei is either an atomic event or a predicate over a set of atomic events and their attributes. We intend an "atomic" event as an event that can be computed by some monitoring function at runtime, either by direct inspection of the target system or by correlating the results of multiple direct inspections.

Problems describe specific faulty or error conditions and represent the set of possible causes of the events being observed: see section 3.5.1 for further details.

On the other hand, and conceptually joined with the Detection Scenario, a Reaction Scenario (RS) is a list of pairs of type "*problem → <suggested new OC to solve the problem>*", associated to a specific Operational Configuration. More formally:

$$RS(OCx) = \{(\{Pj\} \rightarrow OCy)\}$$

where

- `{Pj} ∈ {P1,…,Pk}` is a subset of the set of all possible problems;
- `OCy ∈ {OC1,…,OCn}` is the suggested configuration, an element of the set of all possible Operational Configurations.

Note that there could be more that one OC suggested as a possible reaction to the same problem.

The reaction scenario makes the connection with the reaction functions working in real time on the target system. Reaction functions use RS to select a new configuration to switch to when specific faulty or error conditions have being inserted in the system context model. Reaction functions are responsible to choose among the set of suggested OCs: evaluation data associated to each OC may help with this choice.

Detection and reaction meta model and its usage are discussed in sections 6 and 7.

3.5.1 The system status

The definition of the detection and reaction scenarios in section 3.5 implies the modelling of *system status* (or system context), that is a set of conditions in the status of the system and its environment eventually alleging a reaction. The goal of system status modelling is in fact to link the events occurring in the managed system and the set of models upon which we are basing our management procedures.

The detection scenario specifies how to update the system status starting from events collected from the target system. The reaction scenario specifies how to elicit a potential reconfiguration starting from the current system status.

We distinguish the following main classes of information that may be part of the system status:

- Running Configuration – the currently enforced configuration.
- Resource model problems – problems in the system infrastructure (hw/sw failures, communication link overload, inaccessible/unavailable data).
- Policy model problems – problems in policy enforcement (some policies can't be enforced anymore).
- Security model changes – changes in the security model (new vulnerabilities affecting system resources, detected attacks, some credentials have been compromised).
- Dependability model changes – changes in the dependability model (increased load of requests to a service).

- High-level metrics – security and dependability metrics built on top of the previous information.

4 Meta models for service description

4.1 WS-CDL model overview

In the sequel we present the key modelling capabilities directly inherited from native WS-CDL. We point the attention on the most interesting part, trying to extract a first draft for a WS-CDL profile suitable for our scopes.

The WS-CDL model and language are extensively described in [8] and [9]. The content of this section is freely extracted from [8], which unfortunately is in some point out of date, and some small differences arise with respect to the final language specification [9].

4.1.1 Roles, Participants and Relationships

A *Role* identifies a set of related behaviours, for example the Buyer role is associated with purchasing of goods or services and the Seller role is associated with providing those goods or services for a fee.

A *Participant* identifies a set of related Roles, for example a Commercial Organization could take both a Buyer Role when purchasing goods and a Seller role when selling them.

A *Relationship* is the association of two Roles for a purpose. A relationship represents the possible ways in which two roles can interact. For example the Relationships between a Buyer and a Seller could include: (1) a "Purchasing" Relationship, for the initial procurement of goods or services, and (2) a "Customer Management" Relationship to allow the Supplier to provide service and support after the goods have been purchased or the service provided. Although Relationships are always between two Roles, Choreographies involving more than two Roles are possible.

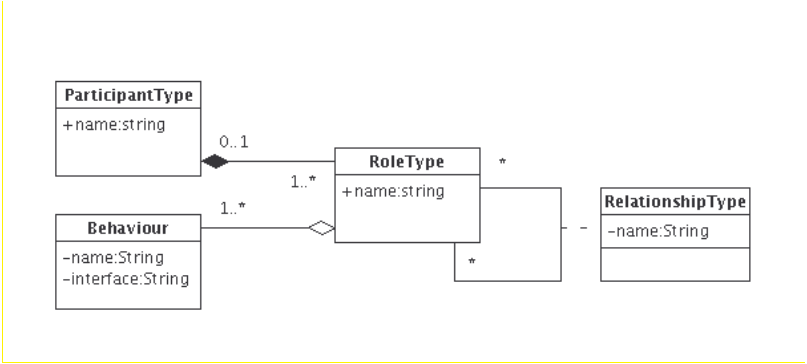

Figure 7 - Participant, Roles and Relationships

We use Roles to define services. Every Role has some Behaviours, which represent the service interfaces.

We use Participant to define processes, i.e. applications that implement a set of Roles; more precisely in WS-CDL we may define only ParticipantTypes, which correspond to abstract descriptions of applications. Participants are instances of ParticipantTypes. For instance, a ParticipantType may describe a web server, whereas the two instances of this web server are two Participants. According to the WS-CDL model, we describe only ParticipantTypes, using channels to identify the particular Participant if needed (we will expand this topic in section 4.1.4, after introducing channels).

Relationships are undirected, despite the fact that two fields named FromRole and ToRole are defined. So they cannot be used alone to define service dependencies. Refer to Example 4.3.2.1 for more information.

4.1.2 Choreography Structure

A Choreography Definition defines the information required by the choreography and sequence in which it is exchanged. It contains the following:

- Zero or more "sub" Choreography Definitions which define Choreographies that can be performed by the Choreography being defined.

- A Definition Block that contains set of Variable Definitions that defines information about objects used by the choreography.
- The actual Choreography that in turn contains: (1) a required Base Choreography part, that defines the normal sequence of information exchanges that should occur; (2) an optional Exception Block, that contains the sequence of information exchanges that are followed when some exceptional or unusual circumstance has occurred while the Choreography was being performed, and (3) an optional Transaction Block which, if present can make the Choreography "transactional" in that it contains information exchanges that are followed when the effects of the choreography need to be Compensated.

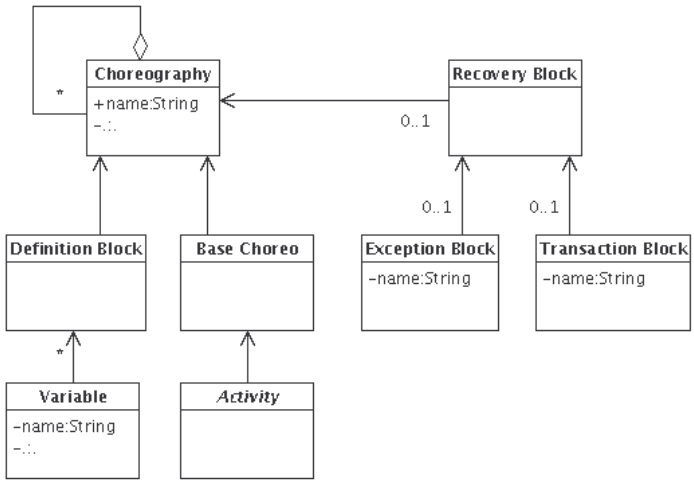

Figure 8 - Choreography structure

4.1.3 Choreography Composition

Choreography Composition is the creation of new Choreographies by reusing existing choreography definitions. Composition is realized through the Perform activity.

At the moment the composition supports Variables binding, but not Roles binding. In [8] also an "Import" statement is announced, but it is not defined in the specification language [9]. In order to actually import part of a description from different XML files, the standard Xinclude should be used. Refer to [9] for more information.

4.1.4 Types, Variables and Tokens

Variables contain information about objects in the choreography such as the messages exchanged or the state of the Roles involved. *Tokens* are aliases that can be used to reference parts of a Variable. Both Variables and Tokens have *Types* that define the structure of what the Variable or Token contains.

Particular kinds of variables are Channel Variables, that identify where and how to send information for a Role. As other variables, also channel variables can be "passed" within an interaction (this is usually referred as "Channel passing", derived from π-calculus).

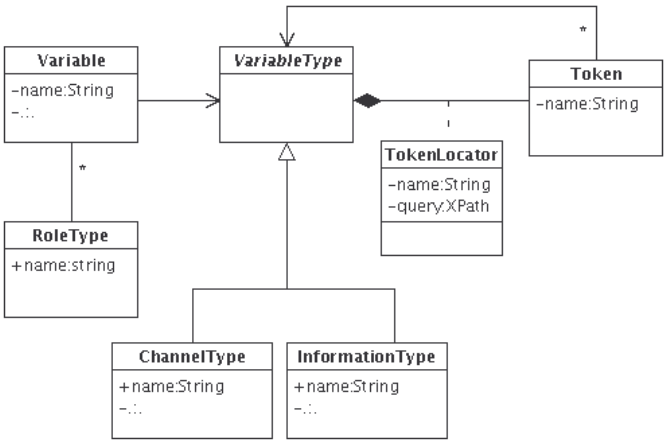

Figure 9 - Types, variables and tokens

In [8] four different kinds of variables are defined: Channel Variables, State Variables, Information Exchange Variables and Other Variables. In the final Description Language [9] only two Types exists: Information Type and Channel Type. The first one is used to type all the variables except, of course, channels.

Channel variables are used to dynamically discover the end point of a conversation. This may include to select a particular Participant (instance) among a collection described as a ParticipantType (refer back to section 4.1.1 for more details). Since WS-CDL in the current specification does not address this problem, we choose to solve by directly assigning the end point reference before the channel is used, i.e. we concretely describe the channel initialization as a variable assignment. It should be noticed that channels are non mutable variables, so the initialization can be done only once.

At the moment, WS-CDL does not support aggregation of a channel and an information type, i.e. a variable can be a channel or (exclusive) any other kind of variable (even composed).

4.1.5 Interactions

An *Interaction* always involves the exchange of information between two Roles in a Relationship that conform to a Message Exchange Pattern as defined by WSDL 1.2. This means an Interaction can be one of two types: (1) a One-Way Interaction that involves the sending of single messages, or (2) a Request-Response Interaction when two messages are exchanged.

An interaction calls an operation on the target role; the operation is part of the interface of the target role as specified by the roleType and behaviour fields within of the channelType description. The channelType used by an interaction is identified by the channelVariable field.

In order to describe an Interaction, WS-CDL requires two roles, a relationship, a channel (variable), an operation and optionally up to four variables (what is sent/received at the two end points). Parts of this information is redundant, for instance from relationship and channel one can extract the roles involved.

An interaction of type Request-Response can be defined with a single Interaction tag or with two tags, one for Request and the other for Response (not necessarily near, maybe something else is described in between).

Interactions are directed, and they can be used to describe services dependencies: if a service A makes a request to a service B, the A depends upon B. The direction of the interaction is given assigning the “from” and “to” role. See Example 4.3.2.1 for more information.

Within an interaction, optional message exchanges are used to describe the data flow and hence the actual source and target role for each message; a message exchange can be a “request” (source is the “from” role, target is the “to”) or a “response” (source is the “to” role, target is the “from”).

Interaction can be defined as "Aligned". Alignment (of interaction and variables) is a feature of WS-CDL that allow a monitor to find unexpected behaviour (two variables not aligned when they should), but it does not describe how to grant alignment.

4.1.6 Activities and Control Structures

Activities are the lowest level components of the Choreography which do the actual work, such as the Interactions described earlier or the performance of other Choreographies.

Control Structures combine these Activities with other Control Structures in a nested way to specify the sequence and flow of the exchange of information within the Choreography. However at the highest level, the Choreographies consist of Work Units, that each contains a single Activity that is performed whenever an optional enabling condition on the Work Unit, called a Guard, is true. Each Activity within a Work Unit is then either:

- A Basic Activity that does the actual work. These are: An ***Interaction***, i.e. the Work Unit consists of a single Interaction as described earlier; A ***Perform***, which means that a complete, separately defined choreography is performed; An ***Assign***, which assigns, within one Role, the value of one Variable to a Variable or Token, or makes a Token reference a Variable or another Token; ***No Action***, which means that the Choreography should take no particular action at that point.
- A Control Structure that groups Basic Activities and Control Structures together in a nested structure to express the logic and decision flow involved in the Choreography. The Control Structures are: ***Sequence***, which specifies a list of Activities that are performed in sequence; ***Choice***, which specifies that just one of two or more Activities are performed depending on the condition on a Guard; ***Parallel***, which means that all the Activities are performed at the same time.

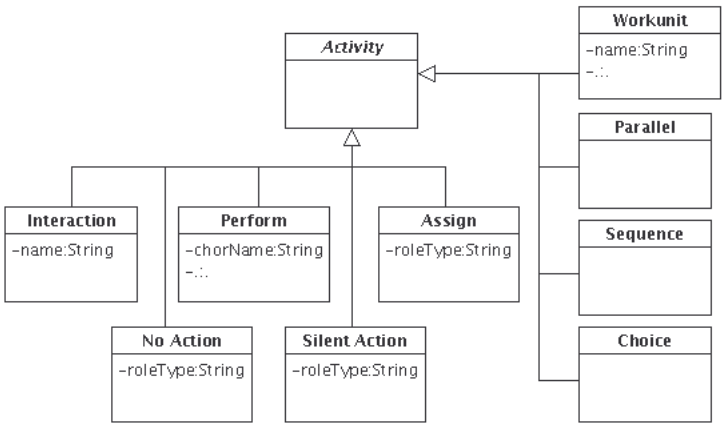

Figure 10 - Activities classification

Interaction and Assign can be used to express data dependency (see section 4.3.2).

In [9] the additional Silent Action activity is defined. It can be useful to semantically express that some Role, in some point, do something.

4.2 WS-CDL Extensions

4.2.1 Light WS-CDL

We believe that in most cases part of the information stated in a WS-CDL document can be implicitly defined. There is no need, for instance, to define variable type for every variable; relationships can often derived from interactions; behaviour is a useless information when a role has a single interface, and so on…

Furthermore, in many descriptions all these "extra" information are named in a standard way, for instance relationship between roles A and B is often named A2B, and this is an evidence of their scarce relevance.

Of course, these considerations are not always true, and one must be free to specify more details when needed.

This motivates the proposal of a light version of WS-CDL, a language with the same syntax of WS-CDL, where some parts are implicitly defined and others, if missed, can be automatically generated (e.g. by a tool) in order to build a complete WS-CDL description.

In details we propose to implicitly define:

- two roles: theClient and theServer.
- for each pair of roles interacting, say roles A and B:
    - a behaviour "A4B" within role A, and a behaviour "B4A" within the role B
    - a relationship, named "relA2B"
    - a channel type, named "ctA2B"
    - a channel variable (in the choreography), named "cvA2B" and of type "ctA2B"
- for each interaction:
    - relationship and channel, if not specified, are the ones implicitly defined
    - in the exchange details, the send and receive tags are generated if not specified
- optionally:
    - two participants Client and Server are created, the first containing only the Client role and the latter containing all the other roles (not related to other participants)
    - for each interaction, an operation with the same name is defined
    - a generic data type "Opaque", assigned by default to any variable (that does not have a type)
    - data types and/or variable for each interaction, for both the interacting roles

This means, in particular, that multiple interactions between two roles refer to the same behaviours, relationship and channel. This is the most common case, but if it would not be, one is allowed to explicitly define other behaviours, relationship and/or channels.

By design from this light description it is possible to retrieve a complete WS-CDL description.

4.2.2 Transformation extension

One of the main requirement for the service model is the *composition*. The choreography point of view, with respect to the orchestral one, allows for a simpler description of the interactions among services, which is an important part of the composition. But composition is not only limited to this.

WS-CDL specification [9] uses the term "composition" with different meanings:

- Composition as a contract among participants: "A choreography description is the multi-participant contract that describes this composition from a global perspective". In this sense, choreography *is* a composition (of participants).
- Choreography composition (and/for reuse): "Goals […] Composability. Existing choreographies can be combined to form new choreographies that may be reused in different contexts.
- Composition with other specifications: "Goals […] Specification Composability. This specification is intended to work alongside and/or complement other specifications such as […] Business Process Execution Language for WS…".

In our opinion composition should also address the possibility to have a layered description of the model. This would allow to choose the quantity of details to be extracted, for instance, at analysis time.

We then propose an additional form of composition that we call *transformation* (just to distinguish it from the other form of composition already supported by WS-CDL). In particular we put in evidence the following arguments, with grown complexity, that would fulfil our requirements. At the moment we propose

a preliminary solution for the first topic, but the others are still open problems, objects of further investigation.

- **Role mapping.** This is form of reuse of choreography descriptions achieved by "calling" a choreography, i.e. executing it, after having modified the roles involved. For instance we may have a description for a DNS Server involving 3 roles: a client, a server and the DNS Server that provides to the client the correct server's IP address. Every time a role A needs the IP address of role B, the DNS choreography may be called by mapping role A onto client and role B onto server.
- **Choreography alteration.** This is a more powerful form of reuse, which allows to (slightly) modify a choreography before executing it. For instance suppose to have a choreography defining a complex protocol between client and server. We may wish to reuse it in another context where the protocol is almost the same, except for a piece of information that does not come from client to server, but from a third party. This could be done by transforming the choreography modifying just the interaction from client to server that involves that piece of information.
- **Choreography refinement.** This would allow to modify a choreography by substituting one of its part (e.g. a single interaction) with another complete choreography. For instance a client-server interaction may be refined at a lower level showing that a third element is in the middle and forwards everything it receive from the client to the server and back. At the moment we are able to describe both the high and the low level scenarios (a number of example will be presented in section 4.3); the challenge is to find a way to describe how to transform the first in the second.

4.3 How to use the service meta model

The following subsections illustrate how to use the service meta model to describe different service architectures and properties. Though originally designed to model web service architectures, we use WS-CDL to describe the interaction among services both at business and network level.

4.3.1 Web service description

For an extended example of WS-CDL model at the business service level, refer to the one discussed in details in the WS-CDL Primer [7].

The primer is intended to give an overview of WS-CDL and can be read both by WS-CDL users and WS-CDL implementers. It describes the following scenario:

- Buyer does a bartering loop with Seller, to define the price of a product;
- Buyer submits an order request to Seller, which includes payment details (e.g. credit card details) and a channel towards Buyer (e.g. the customer address, where the product will be sent);
- Seller performs the payment using the CreditChecker;
- Seller sends instructions to Shipper for the product delivery, including the channel towards Buyer;
- Seller confirms the order acceptance to Buyer;
- Shipper sends the product to Buyer;

Figure 11 shows a diagram of the full description.

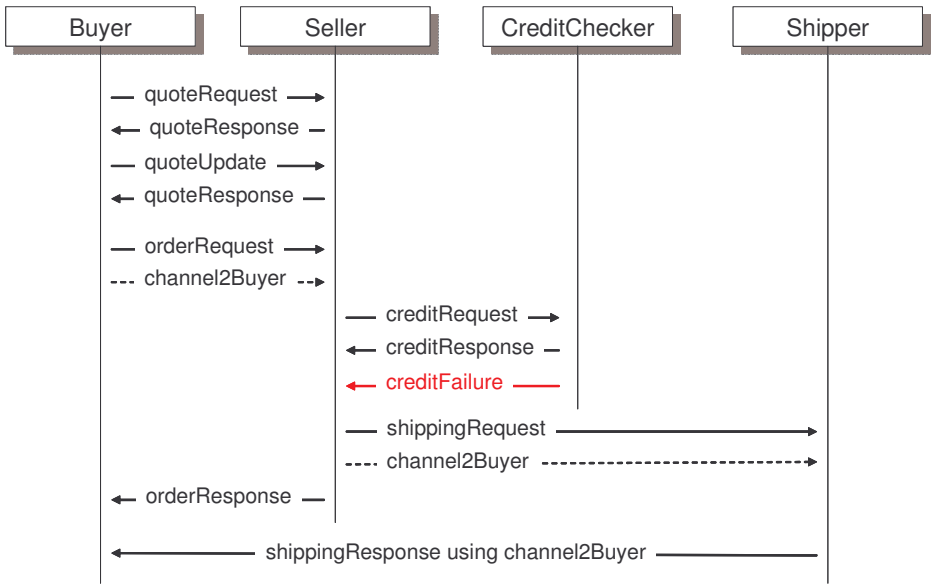

Figure 11 - WS-CDL Primer

A WS-CDL description, as the one shown before, may be quite complex and contains a lot of details on the whole service behaviour. It is also possible to give a description so detailed to be able to extract a running code (e.g. a Java code) for the involved participants.

However, in many cases, we can not expect to have all the details about a service or we do not want to express all of them.

A way to simplify a description, for instance, is to remove all the details about data exchanged. In such a description we still have all the interactions between roles and the control structures that define a precise work flow. Hence it would be possible to do some kind of analysis, for instance functional dependencies analysis, but other are precluded, in this case all regarding data (e.g. confidentiality, integrity, correctness…).

4.3.2 How to model Service Dependencies

In order to support analysis, the service models should provide information on dependencies between service components.

We propose to take into account both functional dependencies (analogous to the relation “depends upon”) and data dependencies, i.e. dependencies among data that constitute the information exchanged and modified during the service execution.

Moreover, we believe it is important to distinguish between static and dynamic dependencies. The firsts can be derived from the model, while the seconds require monitoring the service deployment. We concentrate on static dependencies in the sequel.

4.3.2.1 Functional dependencies

A *functional dependency* between two roles is defined by the interactions among them. In fact if a role A requests something to a role B, then it (functionally) depends upon B.

Functional dependencies are defined with respect to a property, for instance for a service we may define:

- Accessibility: the property of being accessible from authorized entity (i.e. the client is able to contact the service).
- Availability: the property of being accessible and usable upon demand by an authorised entity [ISO/IEC 13335:1-2004].
- Correctness: the property of being available, preserving the integrity of the information provided. (i.e. the service is able to provide a correct response to a correct and authorised request).

Notice that these properties can be checked between two generic roles, for instance we could check the availability of a DB from the point of view of the web server.

From the definitions it is evident that every time correctness is granted then availability is also granted and every time availability is granted then also accessibility is.

However, in practice, more complex relationships may arise as shown in the following example: suppose a Client needs a DNS service to access a Server. The server requires a DB to deploys its service. Moreover Client and Server rely on a Gateway in order to talk. If the DNS is not available, then the Client can not access the Server, i.e. the service's accessibility is compromised, while the availability is not: supposing the Server receives a request, it can both compute the response and send it back to the Client. If the DB is not available, then the Client can access the Server, but the Server can not deploy its service. So the accessibility is provided, while the availability is not. Finally, if the Gateway is not available, then both availability and accessibility are compromised, since Client and Server can not communicate.

Table resumes the above description.

	DNS unavailable	DB unavailable	Gateway unavailable
Service's accessibility	No	Yes	No
Service's availability*	Yes	No	No

Table 3 - Correlation among accessibility and availability

WS-CDL does not provide a specific command to define dependencies among services.

According to the definition, if we wish to add an explicit functional dependency between two services, we can model an interaction among them. In particular, we can do this with different granularity:

- Adding an abstract functional dependency – we can simply add an empty Interaction if we want to provide only an abstract description, the position of the interaction (before or after any original service interaction, in sequence or in parallel) does not matter.
- Adding a dependency on computing a response – we can substitute a new Interaction with another service component in place of an original local action (e.g. an Assign statement).
- Adding a dependency inside an interaction – we can split the original interaction into two interactions and insert in between the new interaction.
- Multiple dependencies – we may both use a sequence (using the Sequence statement) of interactions or multiple interactions put in parallel (using the Parallel statement).
- OR dependencies – we can use the Choice control structure, in a way very similar to the previous point (simply substitute Parallel with Choice).

In WS-CDL a piece of data is represented by a variable defined within a participant. The *value* of a variable may change during the choreography workflow and it may depends upon other variables (values). We refer to such kind of dependency with the term *data dependency*.

As well as functional dependencies, data dependencies are analysed with respect to one or more properties, for instance we may define:

- Confidentiality: the property that information is not made available or disclosed to unauthorized individuals, entities, or processes [ISO/IEC 13335-1:2004].
- Integrity: the property of safeguarding the accuracy and completeness of assets [ISO/IEC 13335-1:2004].

Different types of data dependencies can be modelled in the following ways:

- Data assignment – the assign activity change the value of a variable: this is the most simple example of data dependency between the *left part* (the target variable) and the *right part* (the source variable or expression) of an assignment.

- Interactions – this is similar to local assignment: exchanging a variable from a peer to another peer means that the value of the variable at the sending peer is copied into the variable at the receiving peer.
- Functions with more arguments – multiple data dependencies can be described by associating to an assign statement a function with more than one argument.
- OR dependencies – they can be described with the choice control structure; similarly to functional dependencies, a choice without condition represent an abstract way to express data dependencies in OR.

4.3.3 How to describe Network services

WS-CDL was designed to describe Web Services, even if it allows to describe general services.

In this section we are going to discuss how to model a number of non-WS examples, in particular network services.

A network level description may be either provided as it is, or as a refinement of a service level description; at the moment we do not model explicitly the refinement relationship.

For instance, a Client-Server interaction that is done at application-level (data is exchanged over an applicative channel, e.g. SOAP channel) may be refined to a network-level view, where channels are TCP/IP connections.

Figure 12 shows the network level refinement for Client-Server interaction. For each role, Client and Server, we added an extra role, respectively "Client (Net)" and "Server (Net)", that plays the part of the network stack, i.e. is able to establish network-level channels. Client and "Client (Net)" belong to the same participant, so the interactions between them are local interactions, i.e. function calls (the same happens on the Server side). When Client needs to send a request to Server, it no longer uses the application level channel, but calls its network stack sending a reference to Server and the request payload. The network stack is responsible to build a packet, in this case an HTTP or SOAP packet, and send it through a TCP/IP channel to the Server's network stack. "Server (Net)", once received the packet, extracts the request and forwards it to the Server.

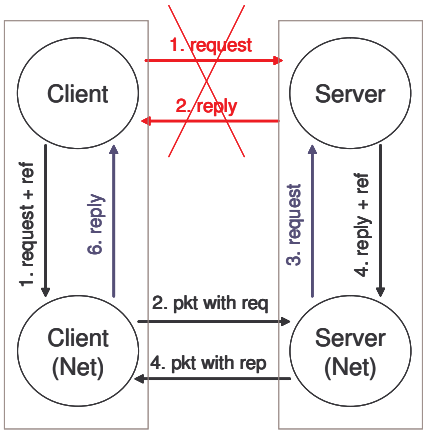

Figure 12 - Network-level refinement for a Client-Server interaction

This approach actually allows to describe services offered at network-level and by intermediary systems, including DNS servers, NFS servers, router/firewall/network-level gateways, DHCP servers, proxy/application-level gateways, load balancers.

4.3.4 Security mechanisms

Similarly to network services discussed in section 4.3.3, WS-CDL can be used to explicitly model some security services.

In particular, the use of secure channels (e.g. TLS) or firewalling services can be modelled through refinement mechanisms in the same way as described for network-level services in section 4.3.3.

The interaction with AAA (Authentication, Authorisation, and Accounting) services can instead be modelled as added functional and data service dependencies as discussed in section 4.3.2.

Finally, the event management infrastructure of an Intrusion Detection System (IDS) can be modelled as a client-server application where clients are sensors and server is the central engine.

5 Meta models for infrastructure and resources description

5.1 The System Description Language (SDL)

SDL is used to formally describe an ICT system. It can convey:

- the topology of the system, both physical (nodes and cables) and logical (such as that created by a VLAN);
- the network configuration and black-box functionality of each node (i.e. network and application services);
- additional information, like power supply, locations, ...

At the moment, SDL can be divided in two different parts:

- Core – main (mandatory) part. It describes the different network elements, their connections, and the software available in the network.
- Extensions – additional (optional) part. Each extension can be used to improve the core description, adding information not strictly related to the physical network.

5.1.1 Core

The main focus of the Core SDL language is to describe the nodes and the software in a generic network. The meta-model behind the SDL Core includes four main logical types:

- network nodes : in this class all the physical elements used to build a network (workstations, servers, switch, etc...) can be found. They are expressed as specific XML elements, like Computer, Router, Wireless Access Point, and so on . The Network itself is an element, and may contain other networks. In general, elements may have one or more application server running, one or more operating system, physical storages (i.e. hard disks), and network ports;
- interfaces : physically distinguishable network adapters. Each interface can be described with many attributes, like connector type, technology type (Ethernet, 802.11, ...), MAC address, IPv4/IPv6 address, and more. Each interface may also have one or more DNS names;
- links : describe a physical connection between two network elements, using their interfaces. The connections between different elements define the topology of the network under analysis. A connection may have attributes, representing the link type (Fiber-optic, Cat5, ...);
- logical elements : While network elements, interfaces and links describe the physical topology of the network, the logical elements focus on the representation of the different software running in a given network environment. Although many different elements fall in this category, the most interesting are:
    - services : these elements express the application servers (i.e. Web Server, SSH Daemon, ...) running on the node. Each service has at least a Service Access Point (SAP), representing the address (or addresses) and port where the service is bound. The information regarding the security of the communication with the server can also be expressed. A service may have a reference to one or more Software;
    - software elements : in this category many different elements can be found. For instance, Software (a commercial product or a program which can be installed on a system), Operating System, Kernel, and so on. Each of them may contain information about the product name, the version, a human readable description, the maintainer, and more. An analysis software can use these data, together with external tools like a library or a database, in order to retrieve more specific information and to perform security evaluation;
    - capabilities : these elements are properties assigned to a software element. They identify security and dependability mechanisms that can be subjected to policies, intended as a set of configuration rules. So they can express the ability of a given element to enforce a particular policy. For instance, the Packet Filtering capability tells that a software can be used as a firewall.

Each element necessarily falls inside one of these general categories.

5.1.2 Extensions

Sometimes, the details expressed by the SDL Core are not enough: in order to perform an in-depth analysis, an administrator may want to formally describe not only the network, but also additional information, like, for example, considerations about the physical environment.

So, aside from the Core part of the SDL language, the need to define some Extensions arose. These extensions are optional parts in an SDL description, providing information not strictly related to the network, and can be used to add modularity to the main SDL model.

At the moment, the Environment extension has been defined.

5.1.2.1 Environment

Environment is an SDL extension concerning the physical environment containing a network. It describes the locations where network elements are placed. Currently, the defined locations are Area, Building, Floor, Room, and Locker. Each location can contain other locations. The contained network elements are represented through references to the main SDL description.

The Environment extension may also give a basic description of the power supply system. Power lines for the different locations can be expressed, and UPS (Uninterruptible Power Supply) are also defined.

Moreover, for each location, a set of physical security features (alarm, fire sensor, EM shielding, etc...) can be defined, in order to give a better description of the overall security of a network environment. These security features are similar to software capabilities, but must be considered fixed, since no configuration change is possible for them.

5.1.3 SDL general design guidelines

Obtaining a suitable description of a given network environment, from the point of view of a network administrator, is a required goal in order to perform various security analysis: this description must be formal, precise, highly detailed and, above all, compliant with a general, high level model (for the sake of ease of use and expandability). These are the reasons why a meta-model has been defined. This meta-model follows a set of generic guidelines (which has been already used by the various languages originally developed by the POSITIF group), but hides most of the language-specific features.

The main idea behind these guidelines (which will be described with a greater degree of details later) is that, in a network environment, four main classes of elements can exist: network nodes (like PCs, routers, etc...), interfaces (i.e. level 3 network interfaces, VLANs...), links (connections between interfaces), and logical node features (like services running on a given node, security capabilities, etc...). Each network element necessarily falls inside one of these general categories. As previously said, the meta-model comes from the use of two already defined languages: xCIM and SDL. xCIM is used as an internal format, a common, standard language which can be understood by the different tools developed in different projects.

xCIM supports network description (and more, like policies...), but lacks of the possibility of setting specific logical limitations on network nodes (like, for example, the type or number of network interfaces on a switch). For this reason, it could be very difficult, for a user, to define the details of a given network using xCIM alone.

SDL, instead, is more user friendly (although verbose), and it's main goal is to 'guide' the user (typically, a system administrator) in the description of the network, setting strict limitations on the details of each and every node.

Since xCIM is a generic (and very verbose) language, obtaining the data required to perform any elaboration can be a difficult, time-consuming task.

For this reason, in order to simplify the use and standardize the access methods, a Java library has been developed. This optimized library permits to obtain, from the xCIM description of a network, any information available, in an abstract form strictly related to the general guidelines illustrated above.

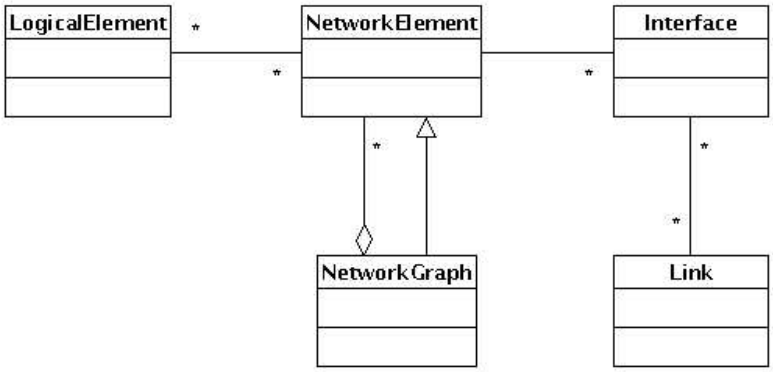

Figure 13 - UML model for general guidelines

A set of general guidelines has been introduced in order to make the model easier to understand, and to simplify the design. In figure 2 an UML model of these guidelines is proposed.

From a general point of view, we have four main types:

- *NetworkElement* : it's an abstraction for every network element (hub, printer, PC, ...), present in a given network. It's explicitly expressed in the diagram that a network (here defined as *NetworkGraph* in order to avoid possible ambiguities later) it's a node too, and aggregates a set of other nodes. A node has at least one communication point with some other entities;
- *Interface*: an interface it's a physical point of access and communication between network elements. Many network addresses can be associated to a single interface. It's the basic entry point to any connected element;
- *Link* : links represent the physical connection between two or more network nodes (through their own interfaces). Links may be wired or wireless;
- *LogicalElement* : features and abilities that a network node can have. These are logical elements.
    - *Services* : represent the ability to provide any given service, like a web server, an FTP server, and the like;
    - *Capability* : represent the ability to enforce a security policy, through a security protocol or an application gateway, for example.
    - Other logical elements which cannot be classified inside these two main classes, can exist, like, for example, an operating system, a software element or an application protocol.

Following these general guidelines, each and every component have been classified and related to the others.

6 Meta models for policies

6.1 Service Constraints Language (SCL)

The Service Constraints Language (SCL) represents the constraints associated with each service belonging to the Service View. These constraints are independent to where the service is implemented finally, and represent functional constraints of the service. They can produce the final configuration of the service, both more general constraints and specific ones about the security of underlying system.

For representing the final configuration for each service and its security requirements, two languages have been defined. The Configuration Policy Language (CPL) will be used to represent the configurations of each service, and the Security Policy Language (SPL) will be used to define what security requirements the above services must fulfil. Both languages are introduced in the next section. As it will be shown next, both languages will be used as extensions of xCIM (to be defined later); thus, we'll be referring to them as xCIM-CPL and xCIM-SPL.

In an initial approach, a set of basic constraints have been defined for the following services: Web, DNS and security (firewall). SCL will be able to be extended new constraints and services that could come up.

The complete XML schema for SCL is included as reference in section 6.1.1.

6.1.1 SCL meta model

The Service Constraints Language (SCL) defines a set of constraints specified by the DESEREC administrator which will be applied to each service in the Service View.

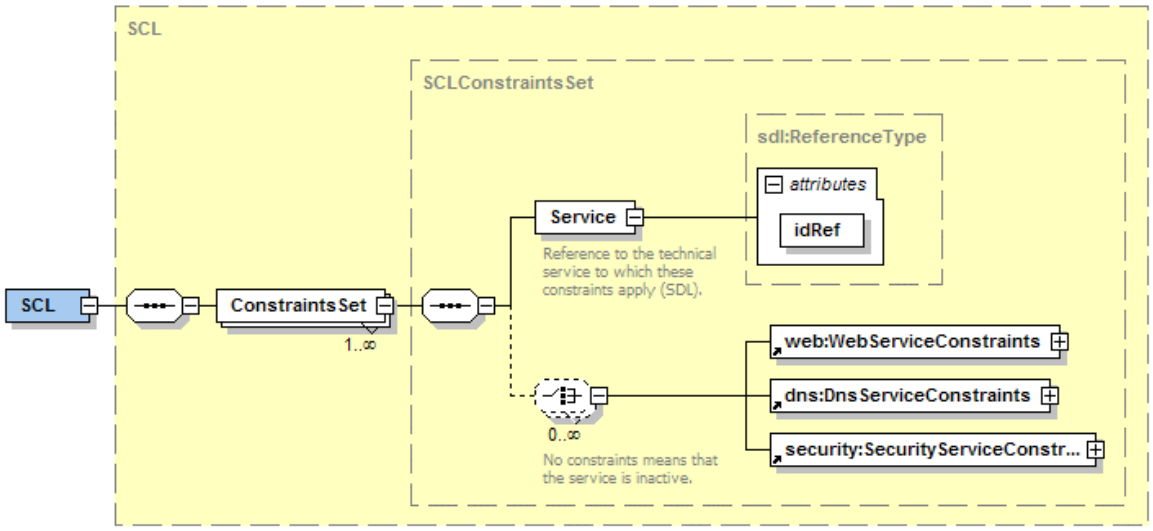

Figure 14 - XML schema for SCL

Figure 14 shows the XML schema for representing the constraints to a single service. Initially, three kinds of constraints have been defined for a Web server, for a DNS server and for a security server (firewall). When a *ConstraintsSet* element defines a target service but no constraints, it means that the indicated service should be inactive (i.e., disabled).

Each of the defined constraint types are defined in separate schemas, which are depicted in Figure 15, Figure 16, and Figure 17.

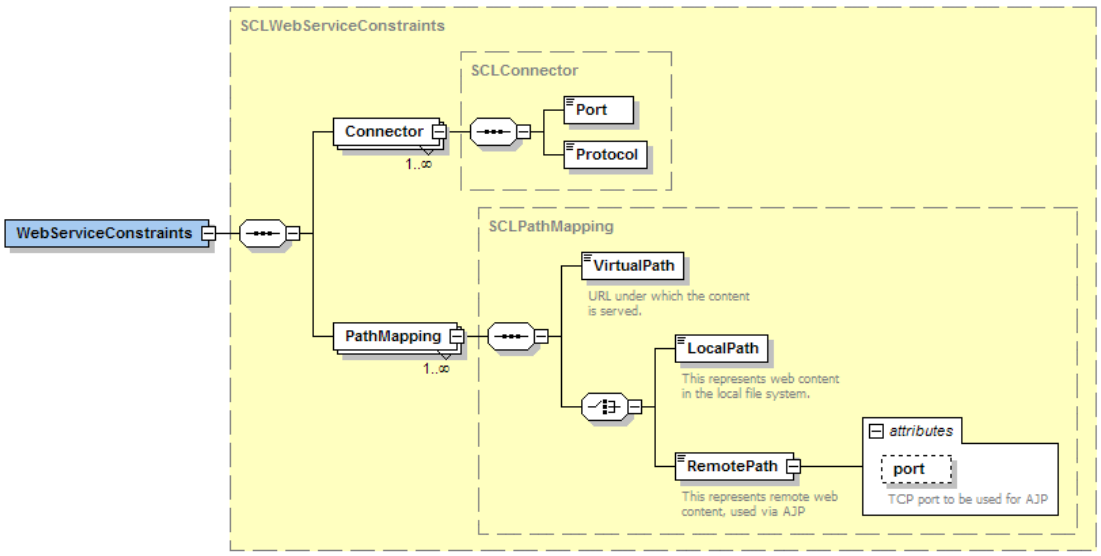

Figure 15 - XML schema for web constraints in SCL

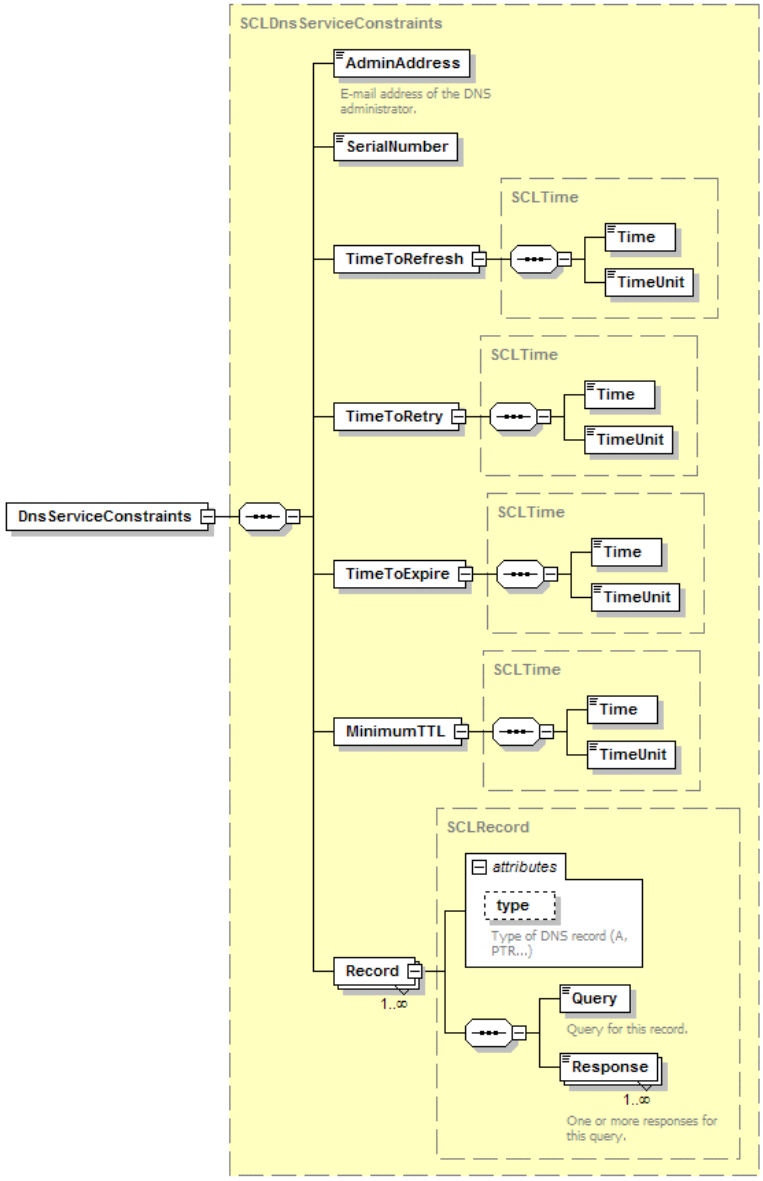

Figure 16 - XML schema for DNS constraints in SCL

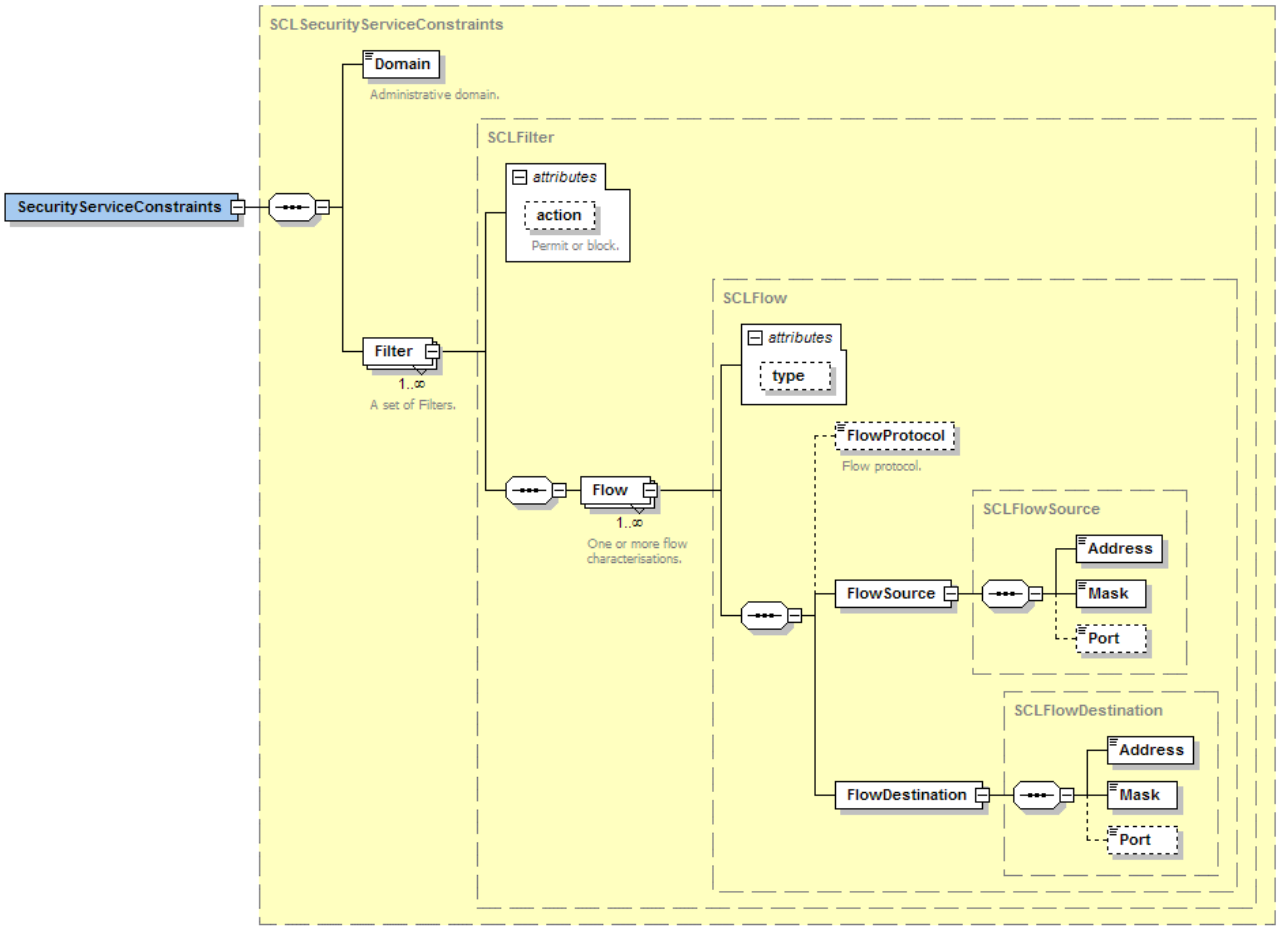

Figure 17 - XML schema for firewall constraints in SCL

As it can be seen in the XML schemas, this language is easily extensible to include more service constraints.

6.2 xCIM-CPL and xCIM-SPL

As cited before, the Configuration Policy Language (xCIM-CPL) represents the final configuration that each service (belonging to the Service View) will have, and the Security Policy Language (xCIM-SPL) describes the desired security policies used to guide the system. Both languages contain references to the elements of the system (belonging to the Resource View) which implement them. The meta models for system description are defined in chapter 5.

On the other hand, as it has been defined at the beginning of this chapter, CIM is the modelling framework chosen in DESEREC to describe dependability and security requirements. The Common Information Model (CIM) [10] provides a mechanism for modelling various types of information. The model is independent of any implementation or repository; although, for a model to be useful, it must be mapped into some implementation. In our case, and bearing in mind the different requirements expounded along the chapter 1, a XML-based implementation of CIM is proposed as follows in order to implement this system and policies modelling.

There are two main different models for mapping CIM into XML: meta-schema mapping and schema mapping. The DMTF [3] specifies a meta-schema mapping for the representation of CIM elements and messages in XML. This mapping defines a XML schema that is used to describe the CIM meta-schema, where both CIM classes and instances are valid XML documents for that schema. In other words the XML schema is used to describe in a generic fashion the notion of a CIM class or instance. In fact, in this approach CIM element names are mapped to XML attribute or element values, rather than XML element names.

The second approach, schema mapping, defines an XML schema to describe the CIM classes; in this approach CIM instances are mapped to valid XML documents for that schema. Essentially this means that

each CIM class generates its own XSD fragment whose XML element names are the same that the corresponding CIM element names.

The meta-schema mapping was mainly adopted by the DMTF, as it only requires one standardized DTD for the whole CIM regardless the version of this information model used in one particular implementation. However, several benefits have been detected regarding the use of the schema mapping rather than the meta-schema; the most important of which were: more validation power and a more intuitive representation.

As said above, the DMTF provides a XML encoding for the CIM model based on its meta-schema mapping. This encoding can be used to generate an XML schema by applying an XSL transformation (XSLT) [4] on it; this work has been developed using the popular XSLT processor *Saxon* [5] for performing the transformation. Figure 18 depicts the complete transformation process, from the original Managed Object Format (MOF) description by the DMTF to the final XSD schema.

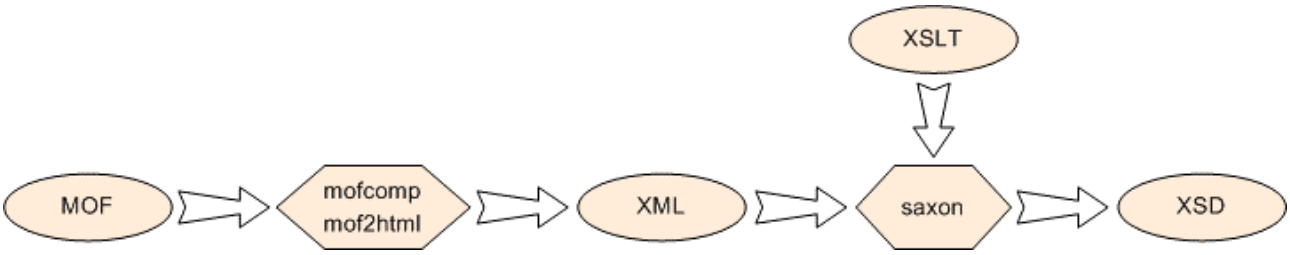

Figure 18 - Using XSLT for translating the CIM-XML encoding to an XML schema

The result of this transformation is an XSD schema which maps CIM to XML. We refer to this mapping as **xCIM**. The xCIM mapping is quite intuitive: a CIM class and its properties can be mapped directly to the corresponding components in XML.

The complete XML schema for xCIM-CPL and xCIM-SPL are included as reference in sections 6.2.1 and 6.2.2 respectively. Bear in mind that these notations are low-level, and not intended for manual use. Instead, the models based on them will always be auto-generated taking the higher level languages (SCL, SDL) as inputs.

6.2.1 xCIM-CPL meta model

Figure 19 shows a graphical representation of the xCIM-CPL schema which defines configuration policies for several services in Service View, such as DNS, Web and security (firewall).

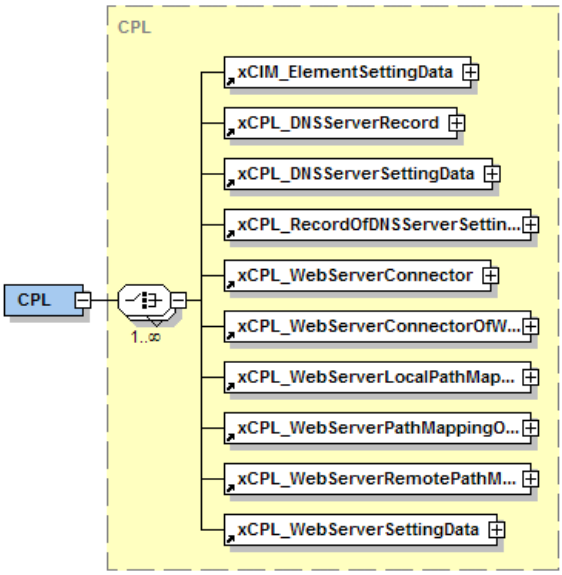

Figure 19 - Graphical representation of the xCIM-CPL schema

This schema aggregates the two kinds of configuration constraints currently defined (web and DNS) via a set of xCIM instances. Next figures show the actual meta-models implemented by these instances, for both kinds of constraints:

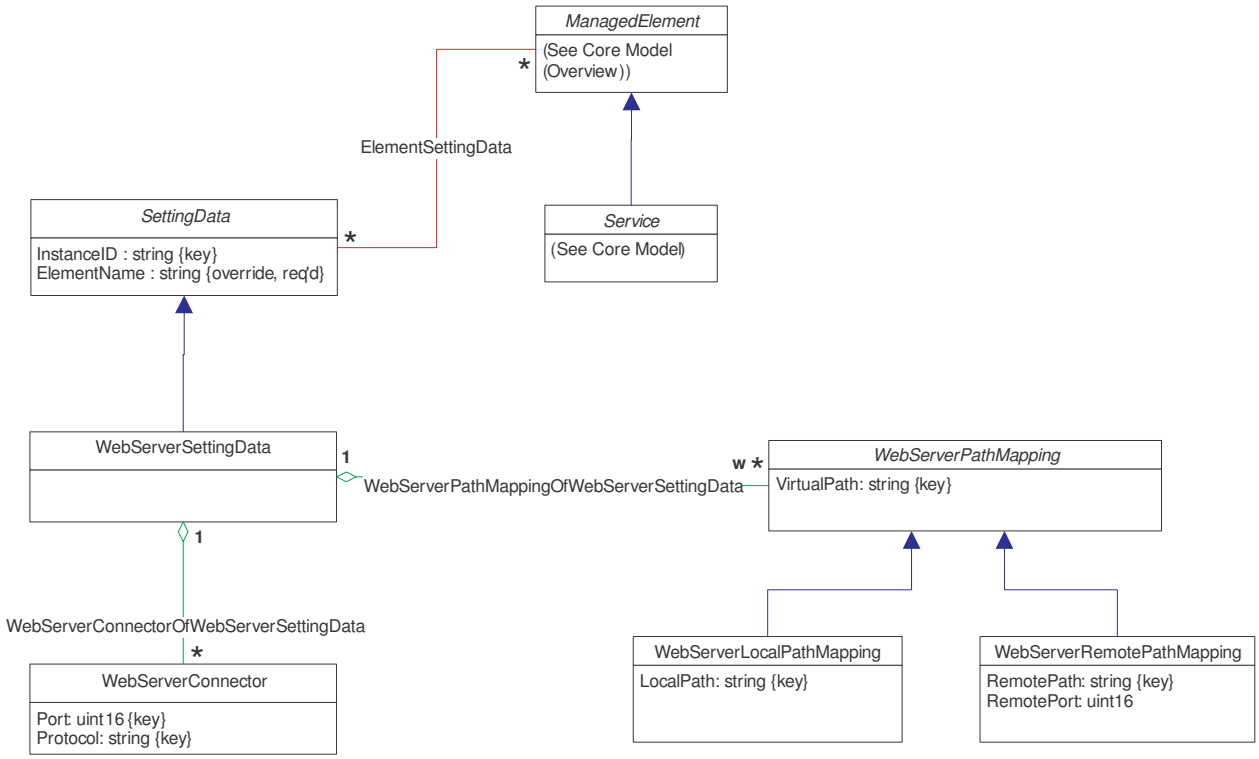

Figure 20 - UML model of web constraints in xCIM-CPL

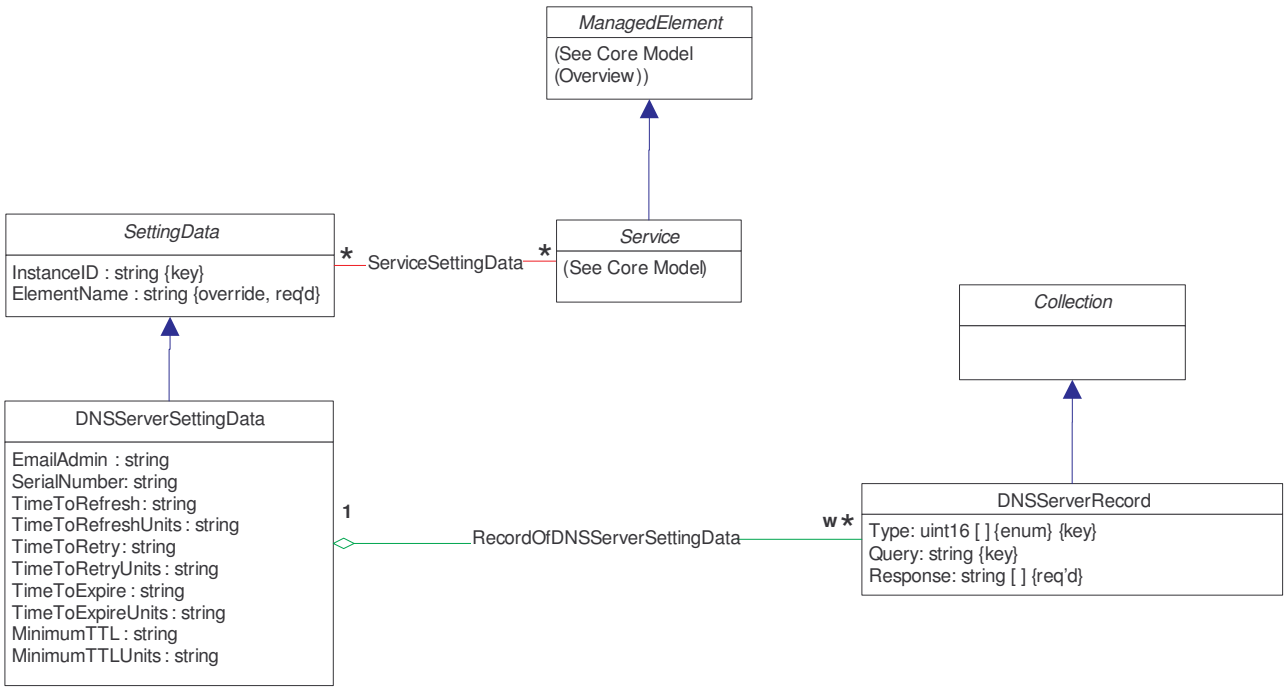

Figure 21 - UML model of DNS constraints in xCIM-CPL

6.2.2 xCIM-SPL meta model

Figure 22 shows a graphical representation of the xCIM-SPL schema which defines policies for security services in Service View, such as firewalls.

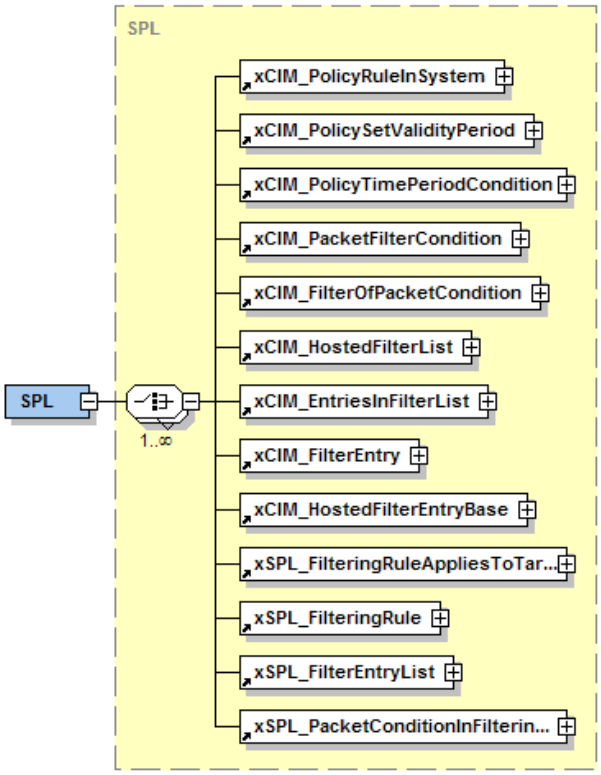

Figure 22 - Graphical representation of the xCIM-SPL schema

Like in the xCIM-CPL case, these instances define a meta-model which is depicted next:

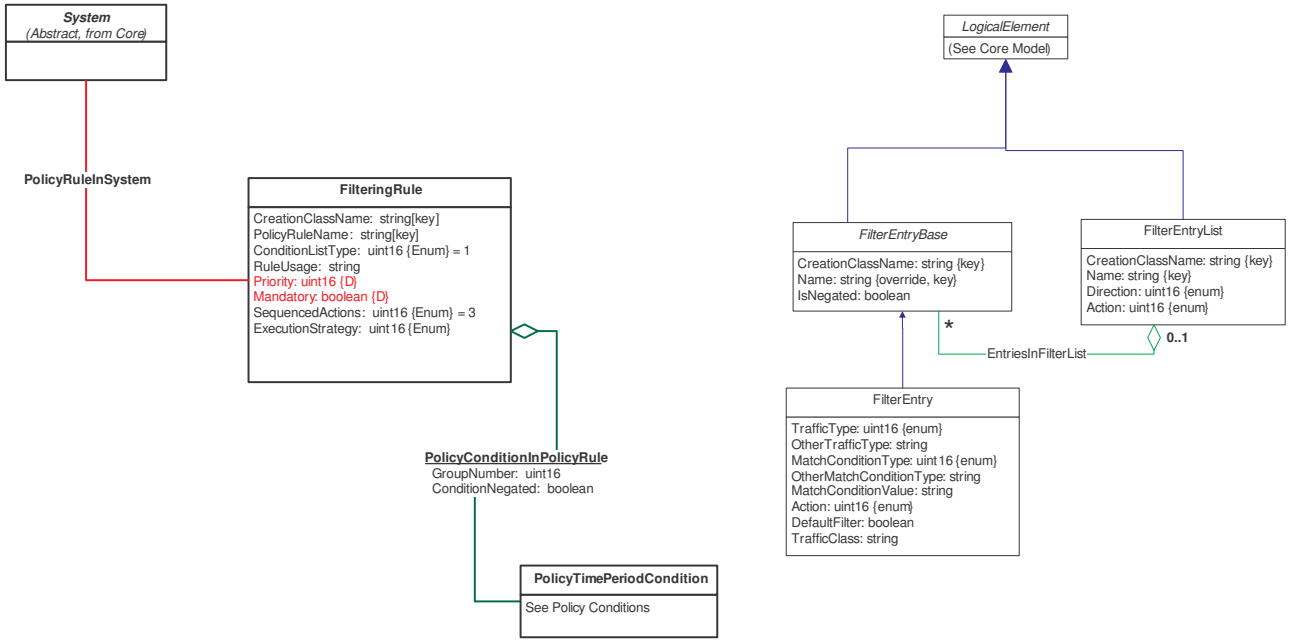

Figure 23 - UML model of security constraints in xCIM-SPL

6.3 Monitoring Policy Language (MPL)

One of the most important topics that DESEREC must address is how we can model the normal behaviour of a critical system, bearing in mind all the underlying components that comprise it; that is, what is the normal behaviour of the network, services and applications. Moreover, we must have special attention in what the system should perform when the system behaves abnormally, and how we can detect this variation of the normal behaviour.

For this, a set of monitoring rules could be modelled to manage the behaviour of a critical system, by establishing both detection and reaction associations. That is, what the DESEREC administrator wants to detect in his system and what reaction(s) should be taken when such incident happens.

We are modelled this set of monitoring rules to define the Monitoring Policy Language (MPL) basing us on the classical definition of policy; that is, a monitoring rule will be represent by a list of:

IF
certain condition is present
THEN
specific action(s) are taken

Thereby, a monitoring rule will be comprised by a set of pairs of type "condition → reaction(s)"; that is, for each condition one or more reactions can be taken to fix the problem detected.

The complete XML schema for MPL is included as reference in section 6.3.1.

6.3.1 MPL meta model

The Monitoring Policy Language (MPL) is comprised by one or more monitoring rules, based on a rule-based format of type "if <condition> then <reaction>".

Figure 24 shows the XML-based schema for the Monitoring Policy Language.

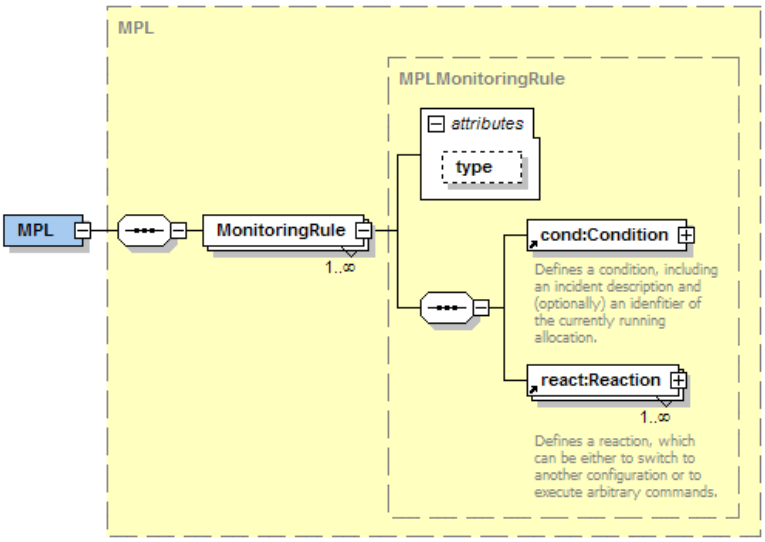

Figure 24 - XML schema for MPL

A monitoring rule is comprised by a set of pairs of type "condition → reaction(s)"; that is, for each condition one or more reactions can be taken to fix the problem detected (if more than one reaction is possible, the DESEREC framework will have to decide which one to apply). Both elements are a reference to the Condition and Reaction schemas which are shown hereunder.

On the other hand, the element *type* included in the MPL schema depicts at what level, global or local, this monitoring rule will be deployed.

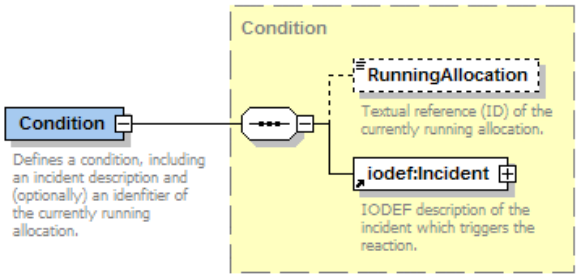

Figure 25 - XML schema for Condition

The Condition schema shown in above figure defines the left part of the rule "condition → reaction(s)". This schema includes the IODEF-based description of the incident that must be detected by the DESEREC architecture in order to trigger the reaction(s) associated to fix the problem. It is also included an optional element (*RunningAllocation*) to identify the currently running allocation; that is, a textual reference identifying in which operational plan this rule must be carried out. This is included to allow conditions to be triggered only when a specific allocation is applied, making the detection process more flexible.

Figure 26 shows the Reaction schema belonging to the right part of a monitoring rule.

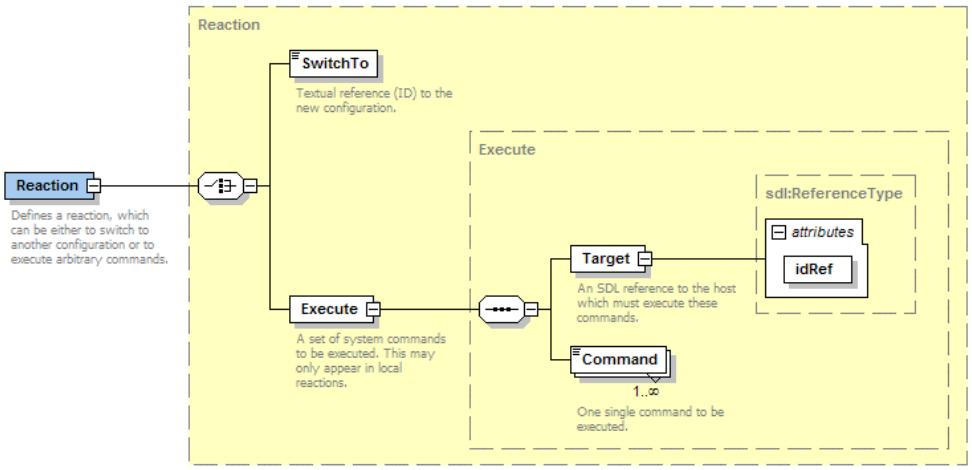

Figure 26 - XML schema for Reaction

As it can be seen in Figure 26, a reaction rule defines either the change of the current configuration to another (element *SwitchTo*) or the execution of a set of commands defined by the DESEREC administrator (element *Execute*). Note that this set of commands to be applied in the target system may only appear when local reactions are being modelled; the global ones only allow switching between configurations.

7 Meta models for operational plans

This chapter provides a meta-modelling approach for the operational plan described in sections 2.2.3 and 3, and introduces the different concepts on which the DESEREC framework relies, regarding mainly WP2 functionalities.

7.1 How to model configurations

Configurations are the merge point to describe how the logical entities, i.e. business services, are located upon the infrastructural resources; every component of the business service is identified by a *participant* in the service model, and every infrastructural resource is represented by an *SDL element* in the resource model.

The configuration is composed by two parts, the first one, named *structural*, shows that a business component is located upon one or more resources, the second one, named *behavioural*, describes in more details how this mapping is done focusing on the dependency between the *SDL elements*; furthermore some changes relating the infrastructure may be necessary (e.g.: installing or activating a software) adding the *temporary resources*.

7.1.1 Configuration meta model: structural part

Figure 27 shows the meta-model of the structural part of the configuration in the form of its XML schema representation; this includes:

- a root element “allocations” with two sub-elements:
  - “participant” - describing the mapping of each service component onto specific SDL elements;
  - “add” - listing the elements to be added to the infrastructure in order to support this specific allocation;
- the “participant” element refers to a service component described in the service model.
- the “participant” element also includes a list of sub-elements of two classes:
  - “host” – pointing to an SDL element onto which the service component is allocated (the element pointed by host is kept general to support different levels of allocations);
  - “data” – pointing to an SDL element that corresponds to some data needed by the service component.

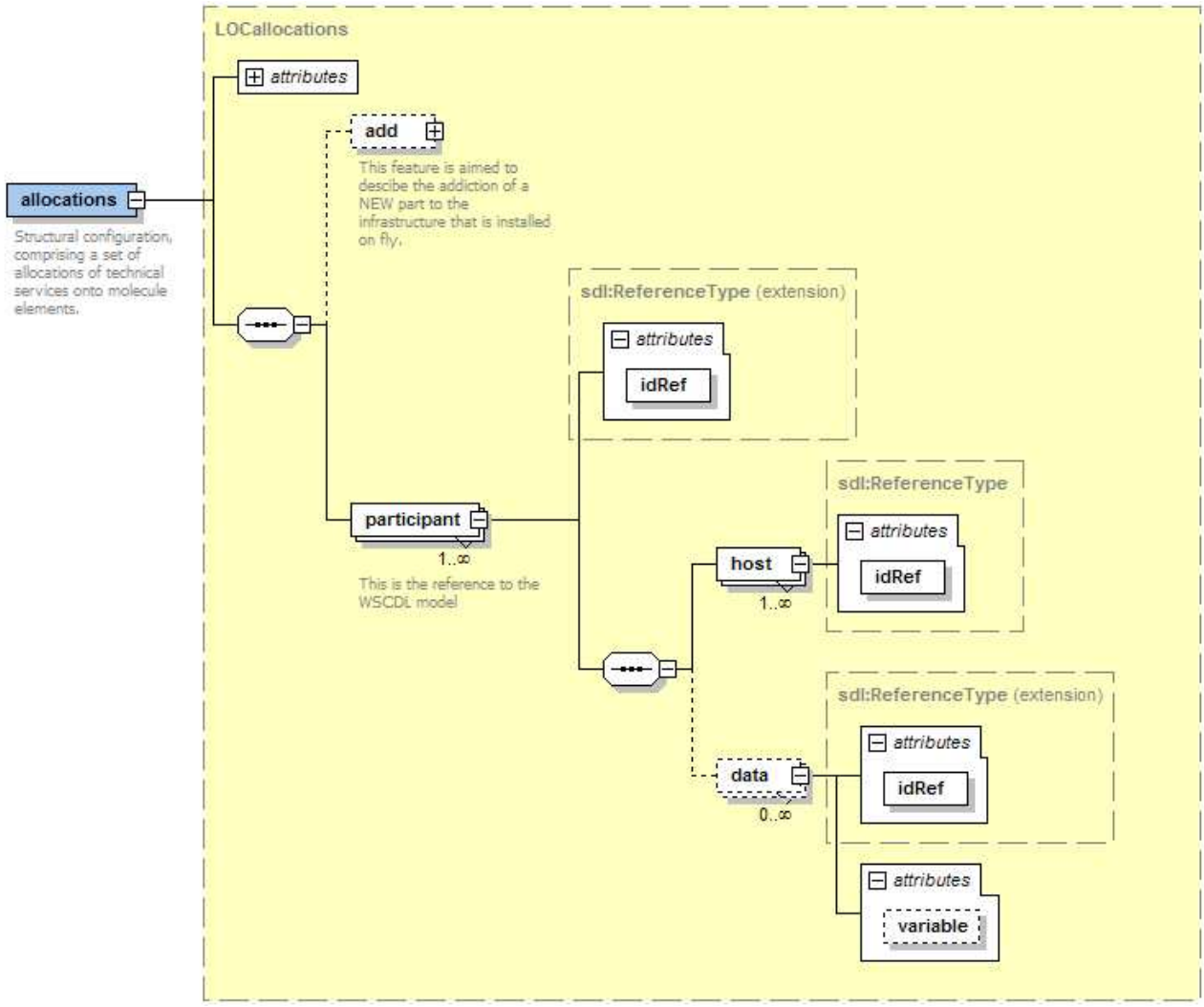

Figure 27 - Allocation meta model (configuration structural part)

Note that in general not every participant to the service choreography will be explicitly allocated in the allocation models. In fact, some of the potential participants are implicitly defined by some configuration dependencies: e.g. any host with networking capabilities can play the role of the client of any TCP/IP service, or can depend on a specific DNS service based on its DNS configuration.

The structural part of the configuration model is mapped into the CIM model based on the following guidelines:

- The "participant" element is translated into a CIM_ApplicationSystem element;
- For the "host" elements, the ApplicationSystem is associated to a set of CIM_Services through the CIM_SystemComponent association;
- For the "data" elements, the ApplicationSystem is associated to a set of CIM_SoftwareElements through the CIM_SystemComponent association;
- The "allocations" is mapped to a CIM_CollectionOfMSEs or a CIM_ConcreteCollection.

The structural part will be one of the information elements carried inside some of the operational plan meta-models, as will be described in next sections.

7.1.2 Generic Service Rulesets

Generic Service Rulesets (GSR's) are in charge of packing generic configuration data, represented in xCIM format. Each GSR describes the full desired configuration for a specific technical service, and is targeted to a specific element in a molecule. For example, if the module of a system contains two hosts (A and B), let's

suppose A runs one service and B runs two services. Then, in this case three GSRs will be generated: one per service.

One GSR is composed by three main components: first, the configuration data, that is, the configuration information (xCIM-CPL) and the security information (xCIM-SPL); second, the required parts of the system model (references to xCIM-SDL); finally, a reference to the target service, in this case one target per GSR. For these reasons, the XSD schema which describes a GSR (xCIM-GSR) consists basically of:

- A GSR header, including target information (target element and target element type)
- All the elements defined in the xCIM-CPL schema
- All the elements defined in the xCIM-SPL schema
- All the elements defined in the xCIM-SDL schema

GSR's are generated by the COnfiguration Generation (COG) module. This module receives as input the previously described information: xCIM-SDL, xCIM-CPL and xCIM-SPL, and generates the final GSR's, which will compose the main operational data unit.

GSR's are the main input for generating the ECO's (Element COnfiguration), that is, the final configuration for services. The translation process from GSR's to ECO's is performed by a Block Service Module (BSM) which will depend on the type of target service. It is important to note that whilst GSR's are generic (independent from the target type), BSM are target dependent. GSR's do not describe how to reach the desired configuration, only *what* we want to reach.

The xCIM-GSR schema is not included here since it adds no extra information to what has already been presented about xCIM models.

For the first prototype, the operational plan models will contain the final xCIM level configuration data. Next developments will decouple these kinds of information.

7.2 How to model the system status

Changes in the status of resource model elements can be modelled by associating a list of status values to every resources in the model. In xCIM-SDL this can exploit the *OperationalStatus* field of CIM_ManagedElement.

The incident *i* referenced in the GDS, GRS, LDS and LRS is in fact an undesired system status, an error. Part of this description may correspond to an xCIM-SDL description that describes an undesired operational status for some of the resources in the system.

The design of a full-featured system status meta model requires additional experimenting with the design and runtime parts of the DESEREC architecture and their interaction. Its definition is thus postponed to the next release of the DESEREC modelling framework.

7.3 High level Operational Plan (HOP)

A High-level Operational Plan (HOP) is a global allocation plan, which defines how the services should be mapped onto the molecules, and how this allocation should change when undesired incidents happen.

A HOP comprises the following items:

```
HOP = { HOC1, HOC2, …, HOCn } + GDS + GRS
```

Where:

- `HOC` is a High-level Operational Configuration.
- `GDS` is the Global Detection Scenario.
- `GRS` is the Global Reaction Scenario.

An overview of these items is as follows. Each HOC describes one particular allocation of the system services onto the molecules (this allocation will obviously have to be further specified inside each molecule). Then, any foreseen incidents which require an allocation change are modelled via the GDS and GRS: in

GDS we describe how these incidents will manifest themselves, and in GRS we define the set of possible global allocations (i.e., the new HOC's) which might be applied when each of these incidents happens.

This allows the HOP to define a global allocation graph, in which nodes are individual HOC's whereas links are allocation changes triggered by the detection of known incidents.

Next sections give additional details about each of the elements in which a HOP is divided.

The figure below depicts the XML-based schema that defines the HOP which includes one GDS, one GRS and a set of global allocations (HOC's). An *Id* attribute is also included. Each of these elements reference to its respective XML schema that define it: these schemas are introduced in the next subsections.

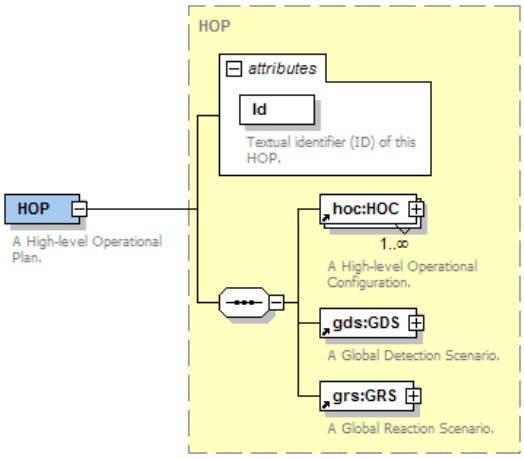

Figure 28 - XML schema for HOP

7.3.1 High level Operational Configuration (HOC).

High-level Operational Configurations (HOC's) are expected to allocate services on molecules, without worrying about which element inside each molecule will actually implement the service. The idea is to use a scalable approach, separating global from local allocation decisions, so that entities in the lower level can act autonomously when possible.

Thus, a HOC will contain a list of allocations in the form "$s \rightarrow m$", where s is a service in the Service View and m is one of the molecules defined in the system. This allocation list must be exhaustive, meaning that every service in the Service View must be allocated by such a rule.

The dependability requirements make it advisable to compute more than one HOC for the given system requirements. This way, the DESEREC framework will be able to switch from one HOC to another when needed. This allocation change can be the result of the decision process, or can be automatically triggered. The first case will be the usual behaviour when incidents are detected which were not foreseen, and thus decisions techniques (maybe AI-based) must come into play. The second case will occur when the detected incident is known in advance; hence, the adequate reaction (that is, the new allocation) can also be given in advance.

As said before, the HOC's are the nodes in the global allocation graph. The following items in the HOP (Global Detection Scenario and Global Reaction Scenario) will implement the graph links.

The XML schema depicted in Figure 29 defines a HOC, which is comprised by a unique identifier of the current HOC (*Id* attribute), and a set of mappings from a service in the Service View to a molecule in the Resource View. Both of these are xCIM-SDL references.

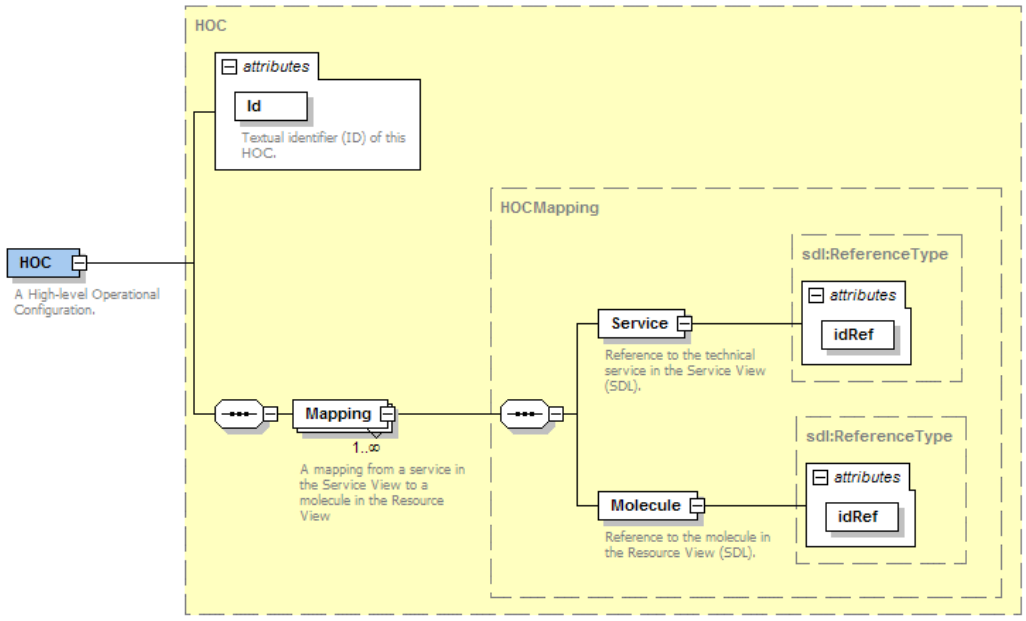

Figure 29 - XML schema for HOC

7.3.2 Global Detection Scenario (GDS)

The Global Detection Scenario (GDS) is basically a set or list of every foreseen system incident for which we desire an automated change in the global allocation. That is, every time the DESEREC framework detects an incident which is listed in the GDS, it will switch from the currently applied HOC to another.

Each item in the GDS is thus a description of an incident, which will need to be expressed by using a suitable notation. It could also include additional information, which may be needed (or simply useful) for triggering the reaction in an efficient way.

The next section describes the Global Reaction Scenario, which by definition is tightly coupled with the GDS.

The XML schema to represent a Global Detection Scenario (GDS) is shown in Figure 30. Based on it, a GDS comprises a set of entries, each of which contains is a unique identifier and the condition to be detected.

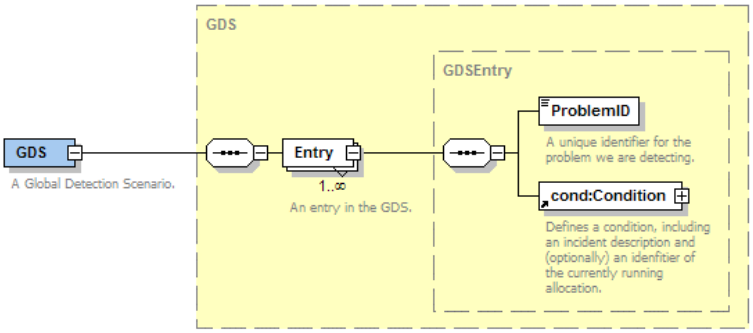

Figure 30 - XML schema for GDS

The identifier (element *ProblemID*) represents the problem we are detecting and the Condition is a reference to the XML schema stated in section 6.3. This *Condition* element will contain the IODEF object modelled by the DESEREC administrator in order to detect foreseen incidents beforehand globally.

7.3.3 Global Reaction Scenario (GRS)

The goal of the Global Reaction Scenario (GRS) is to define a target HOC for each incident described in the GDS. Hence, there must be some mechanism acting as a link between the items in the GDS and the items in the GRS.

The definition of GDS and GRS enable us to specify global reaction rules in the "if … then …" form, in which the condition part is taken from the GDS and the reaction part is taken from the GRS. More specifically, for each incident *i* in the GDS the system could contain one or more reaction rules like "$i \rightarrow h$", where *h* is the target HOC for *i*, according to the GRS.

The Figure 31 shows the XML schema for a Global Reaction Scenario (GRS) which defines what possible reactions may be carried out when an incident happens globally.

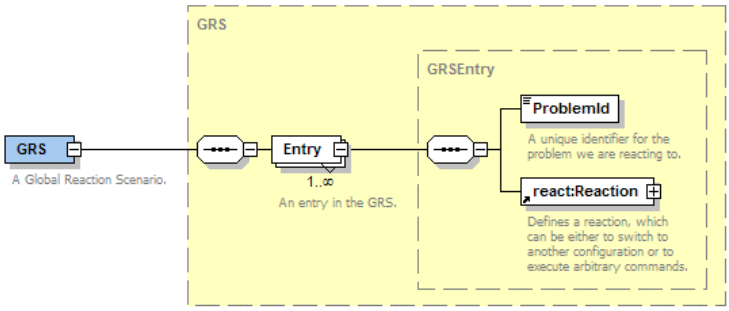

Figure 31 - XML schema for GRS

In this XML schema, a GRS is defined as a set of possible reactions, which are linked to the GDS (through the *ProblemID* element) for creating rules of type "condition → reaction(s)". That is, for each possible condition (defined in the GDS) a set of possible reactions can be taken into account in order to fix globally a foreseen incident. There may be multiple possible reactions for the same *ProblemID*; in this case, the DESEREC framework will have to decide which of them to apply.

The *Reaction* element is a reference to the XML schema stated in section 6.3.

7.3.4 Examples

This section introduces a simple example to envision how the rules described above can be used. Let's suppose a generic scenario where there is a set of services (defined in the Service View) which must be monitored for dependability purposes. Initially, the possible list of allocations of each service onto a particular molecule must be defined offline by the DESEREC administrator, using a simple GUI. A possible allocation of these services could be the following:

- $Service_1$ will be implemented on $molecule_1$.
- $Service_2$ will be implemented on $molecule_1$.
- $Service_3$ will be implemented on $molecule_2$.

Then, the administrator decides that when the $service_1$ suffers any dependability problem such an intrusion attack, network connectivity loss, internal hardware problems or simply the service goes down, the allocation configuration defined above should change to another, in order to fix this kind of dependability problem. In our example, let's suppose that the availability of $service_1$ is crucial, and when it goes down, the overall system must be reconfigured to deploy a new stable status of the system.

Therefore, the DESEREC administrator decides the two following allocations at high-level:

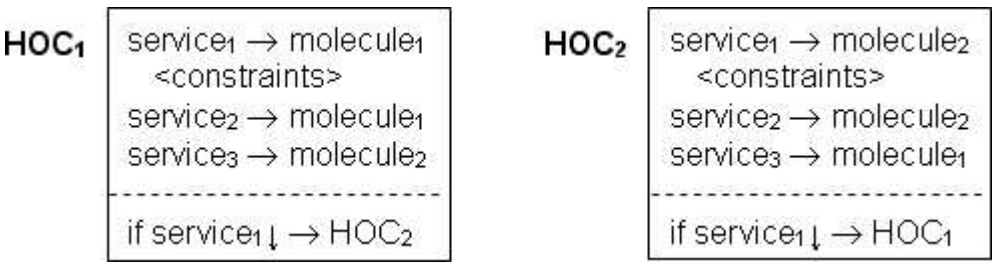

Figure 32 - Two possible high-level allocations (HOC's)

(Note: for clarity, throughout this example we'll be showing inside each HOC the reaction rules that are relevant to it. However, this does not mean that they are part of the HOC; they are really obtained from the GDS and GRS which are part of the HOP.)

The first high-level operational configuration (HOC_1) represents the above status of the system and the second one (HOC_2) represents the new stable status when $service_1$ goes down. In the same way, the DESEREC administrator decides another reaction for HOC_2 saying "when HOC_2 is active in the system and the $service_1$ goes down, the system must change to HOC_1". At this point, note that there might be a set of requirements and/or constraints associated to each possible defined service. For instance, one requirement/constraint for a web service could be that this one must be started with SSL/TLS support, or that this service need previous authentication by the user, and so on. This is illustrated above by the <*constraints*> line, which represents a set of SCL constraints for the current service.

As seen before, the complete set of these HOC's, together with the GDS and GRS which are implicit in the reaction rules, compose the High-level Operational Plan (HOP). In our example, this HOP will contain the following information:

$HOP = \{ [service_1 \rightarrow molecule_1, service_2 \rightarrow molecule_1, service_3 \rightarrow molecule_2], [service_1 \rightarrow molecule_2, service_2 \rightarrow molecule_2, service_3 \rightarrow molecule_1] \} + \{service_{1\downarrow} \text{ and } HOC_1, service_{1\downarrow} \text{ and } HOC_2\} + \{\text{change to } HOC_2, \text{change to } HOC_1 \}$

This HOP contains a list of the different possible allocations of services (defined in the Service View) onto molecules, as well as the GDS and GRS defined by the administrator to change between the allocations when undesired something happens. Note that, even though it is not shown, each detection rule in the GDS must be linked with its associated reaction in the GRS. For example, the incident "$service_{1\downarrow}$ and HOC_1" would be linked to the reaction "change to HOC_2", which means that when the service fails in HOC_1, the HOC_2 must be applied.

7.4 Low level Operational Plan (LOP)

A Low-level Operational Plan (LOP) is a local allocation plan, which defines how the services should be mapped onto the elements belonging to a particular molecule, and how this allocation should change when undesired incidents happen locally. Every local allocation plan will only affect to the molecule where the undesired incident happens, without involving the rest of the molecules. Thereby, for any given HOC, there is a LOP for each molecule defined in the system.

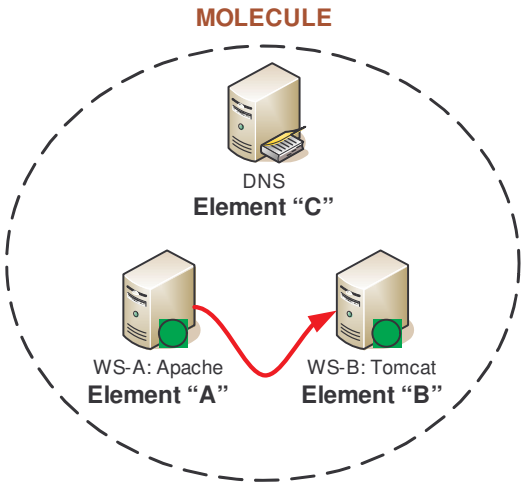

Figure 33 - Example of a local allocation plan

Figure 33 shows a simple example of a local allocation plan for two elements of the same molecule. At initial phase of the system, a web service is implemented into the element "A" and the DESEREC administrator has planned beforehand that when this element goes down, the same web service must be locally reallocated to another element inside the same molecule, which is capable of running it as well.

A LOP comprises the following items:

```
LOP = { LOC1, LOC2, …, LOCn } + LDS + LRS
```

Where:

- `LOC` is a Low-level Operational Configuration.
- `LDS` is the Local Detection Scenario.
- `LRS` is the Local Reaction Scenario.

An overview of these items is as follows. Each LOC describes one particular allocation of the system services onto the elements of a single molecule. Then, any foreseen incidents which require an allocation change are modelled via the LDS and LRS: in LDS we describe how these incidents will manifest themselves, and in LRS we define a set of possible local allocations (i.e., which new LOC's) which might be applied when each of these incidents happens. This allows the LOP to define a local allocation graph for each molecule, in which nodes are individual LOC's whereas links are allocation changes triggered by the detection of known incidents.

Figure 34 depicts a possible local allocation graph which represents the example defined above in Figure 33.

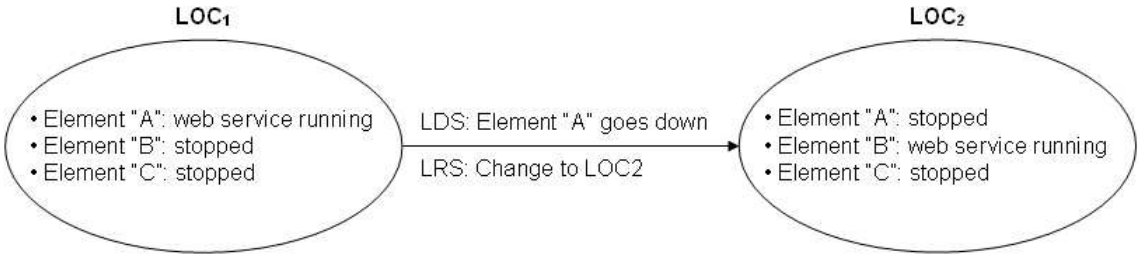

Figure 34 - Local allocation graph

In this example, two LOC's are defined to implement the target web service. Note that this web service can only be executed on the elements "A" and "B" since the element "C" supports DNS services only. Let's suppose the LOC_1 is currently applied on the molecule. When an incident such as "element A is down" occurs (identified by the LDS) the associated reaction is triggered. The pre-defined reaction for this event is to switch from the current LOC_1 to another (LOC_2) which is defined in the LRS. At this point, the molecule must be locally reconfigured to deploy the new configuration; that is, the web service must be started now in the element "B".

Next sections give additional details about each of the elements in which a LOP is divided.

Figure 35 depicts the XML-based schema that defines the LOP which is comprised by one LDS, one LRS and a set of local allocations (LOC's). Each of these elements references to its respective XML schemas that define them, which are introduced in the next subsections.

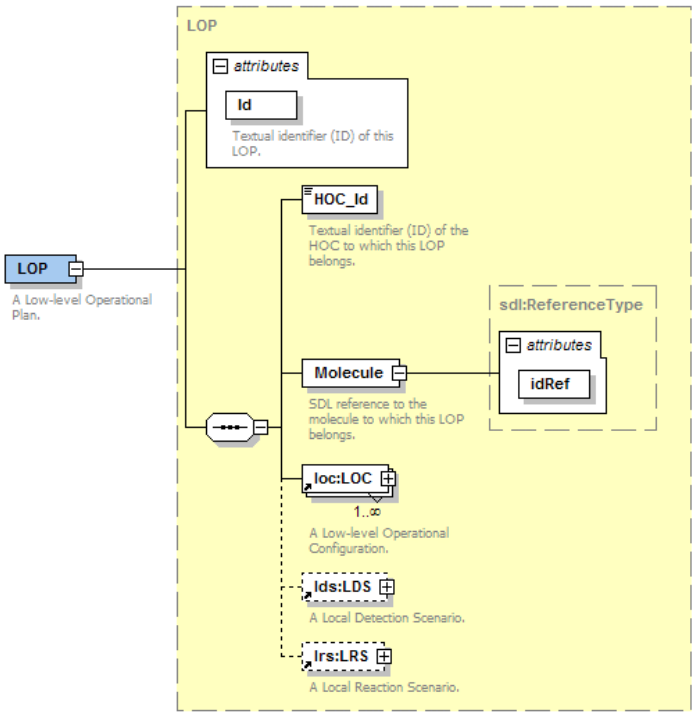

Figure 35 - XML schema for LOP

Note that this XML schema contains an identifier (element *HOC_Id*) for pointing to the HOC from which this LOP was generated. This identifier is used to link the different LOP's to a specific HOC. Thus, when a new HOC must be deployed, we can retrieve quickly all the LOP's that must be deployed to the corresponding molecules. An xCIM-SDL reference to the corresponding molecule is also included.

7.4.1 Low level Operational Configuration (LOC)

Low-level Operational Configurations (LOC's) are expected to locally allocate each service on one element of a particular molecule. Obviously, the selected element must be able to implement the service.

Thus, a LOC will contain a list of allocations in the form "$s \rightarrow e$", where s is a service in the Service View and e is one of the elements defined in the system. This element will only belong to a specific molecule of the system. The allocation list does not need to be exhaustive (as opposed to HOC's), meaning that not every service in the Service View needs to be implemented inside the same molecule (the one to which this LOC belongs).

As said before, LOC's are the nodes in the local allocation graph. The following items in the LOP (Local Detection Scenario and Local Reaction Scenario) will implement the graph links.

The XML schema depicted in Figure 36 defines a Low-Level Operational Configuration (LOC), which is comprised for the following fields:

- Unique identifier of the current LOC (*Id* attribute).
- The structural model for this configuration, using the *allocations* schema.
- The behavioural model for this configuration, using SCL.
- After the COG has processed the LOC, the generated xCIM-GSR instances will be appended as a set of *GSRBASE* elements.

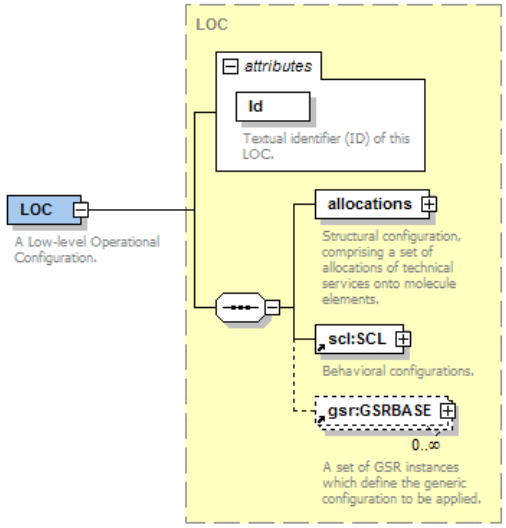

Figure 36 - XML schema for LOC

7.4.2 *Local Detection Scenario (LDS)*

The Local Detection Scenario (LDS) is basically a set or list of every foreseen system incident for which we desire an automated change in the local allocation. That is, every time the DESEREC framework detects an incident which is listed in the LDS, it will switch from the currently applied LOC to another.

Each item in the LDS is thus a description of an incident, which will need to be expressed by using a suitable notation. Similarly to what we have seen for the GDS's, it could also include additional information, which may be needed (or simply useful) for triggering the reaction in an efficient way.

The next section describes the Local Reaction Scenario, which by definition is tightly coupled with the LDS.

The XML schema to represent a Local Detection Scenario (LDS) shown in Figure 37 is comprised by a unique identifier and the condition to be detected.

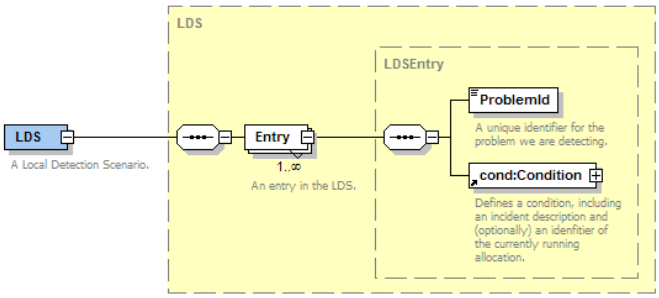

Figure 37 - XML schema for LDS

The identifier (element *ProblemID*) represents the problem we are detecting and the Condition is a reference to the XML schema stated in section 6.3. This *Condition* element will contain the IODEF object modelled by the DESEREC administrator in order to detect foreseen incidents beforehand for the current molecule.

7.4.3 *Local Reaction Scenario (LRS)*

The goal of the Local Reaction Scenario (LRS) is to define a target LOC for each incident described in the LDS. Hence, there must be some mechanism acting as a link between the items in the LDS and the items in the LRS.

The definition of LDS and LRS enable us to specify local reaction rules in the "if … then …" form, in which the condition part is taken from the LDS and the reaction part is taken from the LRS. More specifically, for each incident i in the LDS the system could contain one or more reaction rules like "$i \rightarrow l$", where l is the target LOC for i, according to the LRS.

shows the XML schema for a Local Reaction Scenario (LRS) which defines what possible reactions may be carried out when an incident happens inside the current molecule.

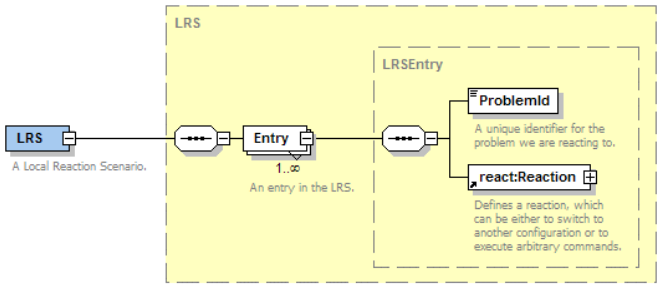

Figure 38 - XML schema for LRS

In this XML schema, a LRS is defined as a set of possible reactions, which are linked to the LDS (through the *ProblemID* element) for creating rules of type "condition → reaction(s)". That is, for each possible condition (defined in the LDS) a set of possible reactions can be taken into account in order to fix locally a foreseen incident.

The *Reaction* element is a reference to the XML schema stated in section 6.3.

7.4.4 Examples

Following the example stated in section 7.3.4, a set of LOP's must be calculated for each HOC defined above. As it has been seen before, there will have one LOP for each molecule defined in the system. For this reason, we need two LOP's in our example: one for $molecule_1$ and another for $molecule_2$. These LOP's will be either introduced manually by the administrator or automatically calculated by the analysis module.

Figure 39 shows an example of LOP's for the HOC_1 defined above. Let's focus on the LOP for $molecule_1$, that is, $LOPmolecule_1$. There we have the possible allocations of the services onto the elements belonging to $molecule_1$. In this case, we have two possible allocations for $service_1$, one located in $element_1$ and another in $element_2$. Although it is not shown in this figure, each LOC in this LOP would have to include an allocation for $service_2$ as well.

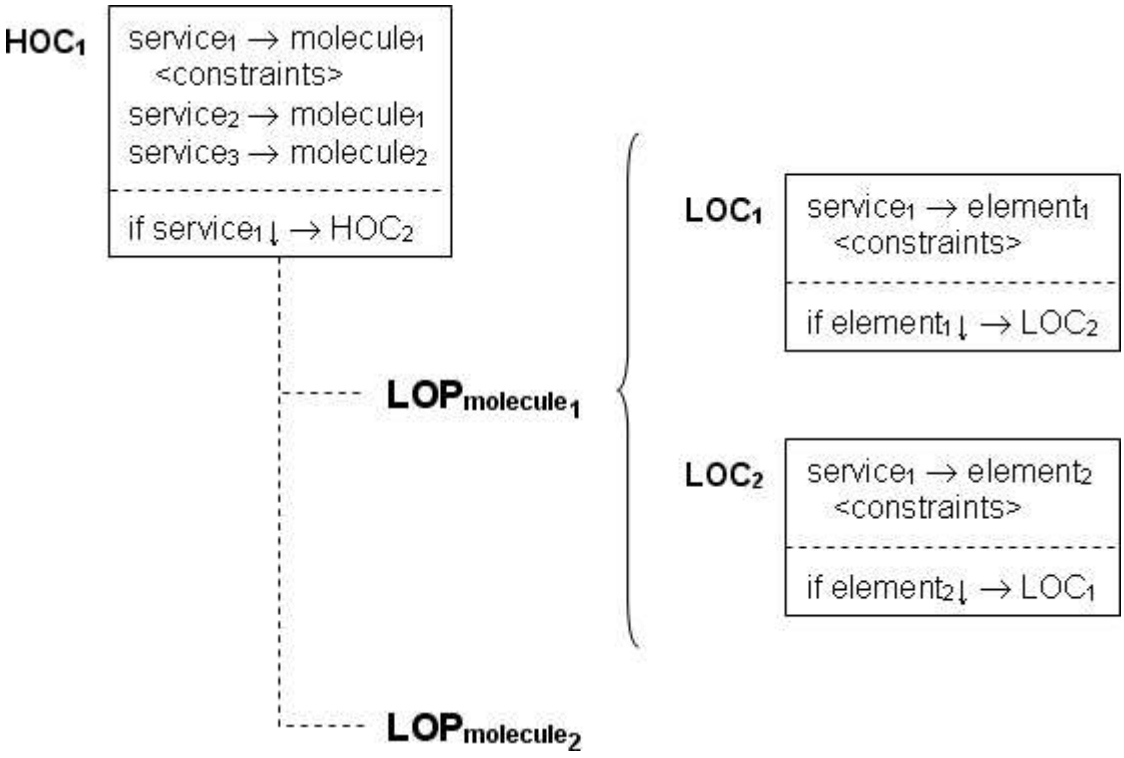

Figure 39 - Two possible low-level allocations (LOC's) inside a HOC

(Note: exactly like in the previous example, the relevant reaction rules are attached inside each HOC. However, this does not mean that they are part of the LOC; they are really obtained from the LDS and LRS which are part of the LOP.)

In this case, two local reactions have been defined to quickly respond when a known undesired event happens into molecule1. When LOC1 is applied and working in the molecule, and the monitoring system detects that element1 is down, a new local configuration (LOC2) must be deployed. In the same way, another local allocation is defined by LOC2 in case element2 goes down.

The complete set of these LOC's, together with the LDS and LRS which are implicit in the reaction rules, compose the Low-level Operational Plan (LOP). In our example, the LOP for molecule1 will be defined as follows:

$LOP = \{ [service_1 \rightarrow element_1], [service_1 \rightarrow element_2] \} + \{ element_1\downarrow, element_2\downarrow \} + \{ change\ to\ LOC_2, change\ to\ LOC_1 \}$

This LOP contains a list of the possible local allocations of services (defined in the service view) onto elements of the same molecule, just as the set of LDS and LRS defined to change between the allocations when undesired something happens. Note that each detection rule into LDS must be linked with its associated reaction into LRS.

8 Conclusions

The current document provides the general and detailed descriptions of the DESEREC modelling framework. Based on the initial specification, the first prototypes of the DESEREC modelling tools have been developed and released as the deliverable *D2.2 - Modelling Tools (first prototype)*. Thus, this document provides a strong basis for designing and prototyping the Operational Planning modules and the whole DESEREC architecture as initially defined in the deliverable *D1.3 - Initial System Architecture*.

The first design described in this document has outlined a set of issues and extensions that will be investigated for the final version of the DESEREC modelling framework: these aspects are well outlined along this document.

9 References

[1] DESEREC Project Consortium. “D1.1 - First requirements and scenarios”. May, 2006

[2] DESEREC Project Consortium. “D1.3 - Initial System Architecture”. December, 2006

[3] Distributed Management Task Force, Inc. (DMTF). “*Common Information Model (CIM) Standards*”. April, 2006. http://www.dmtf.org/standards/cim

[4] World Wide Web Consortium (W3C). “*XSL Transformations (XSLT)*”. W3C Recommendation. November, 1999. http://www.w3.org/TR/xslt

[5] Michael H. Kay. “SAXON - The XSLT and XQuery Processor”. April, 2006. http://saxon.sourceforge.net

[6] Hugo Haas, Allen Brown, “Web Services Glossary”, W3C Working Group Note, 11 Feb 2004, http://www.w3.org/TR/ws-gloss/

[7] Web Services Choreography Description Language: Primer, W3C Working Draft, 19 Jun 2006, http://www.w3.org/TR/2006/WD-ws-cdl-10-primer-20060619/

[8] Web Services Choreography Model Overview, W3C Working Draft, 24 Mar 2004, http://www.w3.org/TR/2004/WD-ws-chor-model-20040324/

[9] Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation, 9 Nov 2005, http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/

[10] Common Information Model (CIM) Standards. Distributed Management Task Force (DMTF), Inc. http://www.dmtf.org/standards/cim.

End of document.