A Policy-based Dependability Management Framework for Critical Services

Manuel Gil Pérez, Jorge Bernal Bernabé, Juan M. Marín Pérez, Daniel J. Martínez Manzano,
and Antonio F. Gómez Skarmeta
Departamento de Ingeniería de la Información y las Comunicaciones
University of Murcia, Spain
Email: {mgilperez,jorgebernal,juanmanuel,dmartinez,skarmeta}@um.es

***Abstract*—Many critical activities rely on the correct and uninterrupted operation of networked Computer Information Systems (CIS). Such systems are however exposed to many different kinds of risk, and thus many researches have been taking place for enabling them to perform self-monitoring and self-healing, and so maintaining their operation over time as specified by domain policies. These capabilities are the basis of what is commonly referred to as *dependability*. The DESEREC project has defined a tiered architecture as a policy based framework to increase the dependability of existing and new networked CIS, using technology-independent information which is translated at runtime to suit the managed components. This paper delves into how DESEREC builds and manages large critical systems through an agent-based distributed framework, and how it is able to respond to any adversity effectively, such as intrinsic failures, misbehavior or malicious internal use, and attacks from the outside. An illustrative example is used throughout this paper to demonstrate all the concepts and definitions presented.**

***Keywords*-Dependability; Policy Based Management; Self Healing; Configuration Constraints; Dynamic Reconfiguration**

I. INTRODUCTION

This article is an extended and revised version of the conference paper entitled "Towards a Policy-driven Framework for Managing Service Dependability" [1]. It contains a more comprehensive and detailed explanation of the proposed framework, and a complete running example with the aim of demonstrating the concepts and models herein introduced.

As networked Communication and Information Systems (CIS) become more pervasive, even more critical activities rely on their correct and uninterrupted operation. Such systems are however exposed to many different kinds of risk, including hardware failures, software bugs, connection and power outages, and even malicious use. In this context, much research has been taking place with the aim of providing networked CIS with a new capability, beyond the classic concept of fault tolerance and data redundancy [2]. This new feature will enable administrators to perform self-monitoring and self-healing to maintain the system operation over time, as specified by domain policies. This capability is often referred to as *dependability* in literature [3][4].

Dependability covers many properties relevant to the self-management of critical systems [5][6]. Among them, we can emphasize:

- Availability: probability that a service is available for use at any time; it allows for service failure, with the presumption that service rehabilitation is immediate.
- Reliability: measure of the continuous delivery of a service in the absence of failure.
- Safety: non-occurrence of catastrophic consequences or abuse to the environment or its users.
- Survivability: ability of a system to provide crucial services in front of attacks and failures, and restore those services in the least amount of time.
- Maintainability: capability of a system to support changes and evolutions, possibly under hostile conditions.
- Security: it includes some properties to maintain truthfulness and confidence in the managed data. These properties are mainly used for confidentiality, integrity and non-repudiability purposes.

Furthermore, it is important to note that security properties are generally limited to discrete values, e.g., a user is authenticated or not, the information is either available or it is not, etc., whereas, on the contrary, dependability properties are continuous or multi-valued, expressed in terms of probabilistic measures, e.g., a system is highly reliable beyond 95%.

Bearing in mind the above properties, a dependable framework with these goals should be pursued by following a multidisciplinary approach, in the search of a tiered architecture with the following responsibilities:

- Build abstract models of the managed system, the services to be provided by it, and any other needed management information (policies).
- Set up the technical components automatically in the system so that the intended services are provided.
- Perform intensive and extensive monitoring all over the managed system.
- Take appropriate actions when something wrong is detected.
- Quick containment as close as possible to the managed system.

At the sight of the above list, one can easily work out the value of a policy based management framework in such an architecture. Policy based management (PBM) [7][8]

is a management paradigm by which rule-like pieces of information are used to describe the desired operation and behavior of a system; these rules are typically stored in a repository from which they are selectively distributed across the concerned enforcement entities. These entities will then make use of their local knowledge in order to guarantee the correct operation of the areas under their management. Since these rules are described in an abstract fashion, enforcement entities may require to perform some technology-dependent translation on them so that they suit the managed components.

Having the above objectives as a goal, the DESEREC project [9] has defined a multi-tiered architecture as a framework to increase the dependability of existing and new networked CIS, by means of the following functional blocks:

- Modeling and simulation. DESEREC devises and develops innovative approaches and tools to design, model, simulate, and plan critical infrastructures to improve their resilience.
- Incident detection and quick containment. DESEREC integrates various detection mechanisms to ensure fast detection of severe incidents, and thereby avoiding any impact propagation.
- Fast reconfiguration with priority to critical activities. DESEREC provides a framework to respond in a quick and appropriate way to a large range of incidents to mitigate the threats to the dependability and thwarts the problem.

Next sections will give a deeper insight on some aspects of the overall solution, as follows: first, Section II introduces the main related work as background information for the reader; then, Section III explains the designed framework, describing its tiered architecture; Section IV presents a running example that will be used by the following sections to demonstrate the explained concepts; Section V describes the abstractions used by the DESEREC framework for binding configurations to services; Section VI delves into the implementation of the policy-based models and engine; the complete reconfiguration framework is explained in detail in Section VII; Section VIII summarizes our experience in the design of a dependable system following this approach; and lastly, some conclusions are drawn in Section IX.

II. Related Work

The modeling of large systems with the aim of reconfiguring automatically the services they offer has been the cornerstone for the last few years, precisely due to the great complexity that these systems convey.

There currently exist a great amount of tools that provide high availability of critical services in case of physical failures, whether due to hardware failures or even natural disasters.

Among them, the most commonly used are: *backup systems* which facilitate an automation of the backup process and a quick restoration of the data; and *clustering of servers* (a "battery" of servers that are monitored each other to detect temporary failures or system crashes). In both cases, these tools are able to restructure dynamically to continue offering the service. However, as main drawback, they are unable to detect attacks from malicious users (whether internal or external).

Many of the existing tools with a similar purpose are mainly focused on a single kind of services that the system must handle, like Web-based [10] or computer-based approaches [11]. New efforts are being made to provide a compact solution that includes the complete life cycle of any large system without human intervention; from the configuration, the monitoring until the reconfiguration. The great challenge behind this is to achieve truly 24x7 systems, or continuous availability.

In [12], the authors presented a framework based on the monitoring, performance evaluations and dynamic reconfiguration of the SIENA network. On the other hand, in [13] the authors extended this management to mobile environments following a similar approach to [12], by monitoring and estimating the redeployment architecture to maintain the availability of this kind of systems.

As a solution to these emerging issues, two initiatives have stood out. On one hand, the SERENITY EU-funded R&D project [4] aims at supplying security solutions and high availability in Ambient Intelligence (AmI) systems and services. These systems are mainly concerned in human and services interactions, especially in mobile distributed environments. In this context, in [14] the authors delve into the SERENITY approach, providing security and dependability (S&D) solutions for dynamic, highly distributed and heterogeneous systems.

On the other hand, the Willow approach [15] focuses on the design of an architecture to provide a great resilience to failures in large distributed information systems. This architecture offers mechanisms based on specifications for fault-tolerance techniques, but it sets aside many fundamental issues related to the dependability, like misconfiguration, misbehavior or malicious use. Such an approach has proven its value when it has been successfully applied to Grid management [16], enabling the dynamic reconfiguration of a grid without human intervention in response to environment changes. Additionally, this research group is currently focusing on the applicability of the Willow approach to the novel computing paradigm of Cloud Computing [17].

All these initiatives offer some initial results for the critical systems management with high availability, but they are still far from providing an integral system that allows managing large systems in a completely autonomous way, as standardized as possible, trying to save costs in managing the inactivity of a service, and so on.

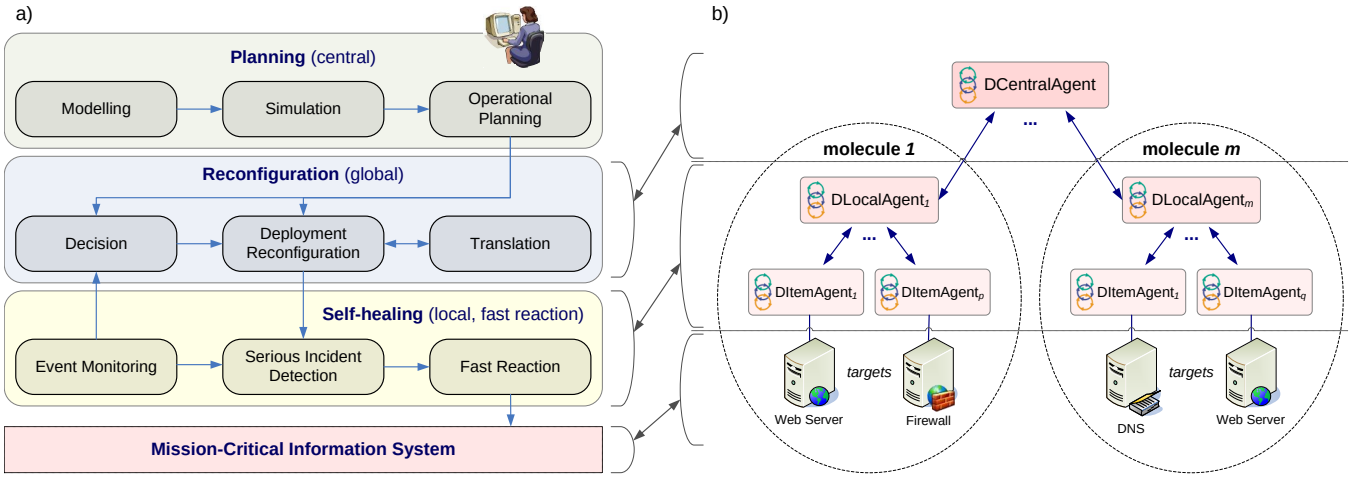

Figure 1. Model-based approach: a) high level functional blocks; and b) the DESEREC framework

III. DESEREC: A FRAMEWORK TO ENHANCE DEPENDABILITY

The DESEREC project [9] has defined a 3-tier architecture to increase dependability of any CIS by means of three different and connected approaches: modeling and simulation; detection; and response. Its main goal is to react appropriately upon incidents of any nature, e.g., errors, failures, malicious actions, to maintain critical services always available. The proposed approach supports a mid-term strategy with planning and simulation tools for modeling in a proactive way the performance and dependability of any CIS. Figure 1 depicts how DESEREC manages these critical systems using a model-based approach, which is organized around a 3-tier reaction loop.

As can be seen in Figure 1a), the three objectives proposed by DESEREC can be identified clearly: firstly, plan and model the operations of the system, as well as configuring it properly by the system manager at design time; as second objective, detection and prevention of incidents and potential faults proactively, from the events harvested on the target system; and finally, react to the detected incidents by either reallocating the services or executing a set of abstract commands to fix the problems caused. Both of them (services configuration and abstract commands) are defined in a generic way with the aim of abstracting system managers from technology-dependent details.

A. Three-Tier Agent-based Reaction Loop

The DESEREC runtime architecture has been developed following a multi-layered approach in order to manage large systems by means of splitting the underlying CIS in different areas of autonomic management [18]. As can be seen in Figure 1b), a *molecule* is the minimal sub-division of any CIS with the aim of grouping physical components (servers and network equipments) under the same management control at local level [19]. This division can host one or more technical services such as authentication, IP allocation, etc. Each molecule accommodates one local molecule agent, called *DLocalAgent*, which should:

1) Monitor low level events reported by the managed elements, or directly harvested from them.
2) React locally to serious incidents with enforcement capabilities (usually through secure channels).
3) Apply high-level reconfiguration orders, coming from a central agent, to enforce a new operational plan.

Each DLocalAgent in turn manages one or more item agents, called *DItemAgent*, which are in charge of handling the target infrastructure elements. Each DItemAgent is responsible for monitoring those underlying elements and to enforce available reactions in case of a system failure. Note that in this architecture only this last kind of agent has knowledge about the final technical service implemented (software and version, how should be started/stopped, etc.); that is, they manage vendor-specific information, whereas the rest of agents use technology-independent information.

Finally, a single central agent is placed to have a global view of the whole infrastructure. This agent is called *DCentralAgent* and will receive local events and alarms from DLocalAgents to detect incidents which could have passed as undetected by these latter; for example, by correlating events from different molecules to detect a distributed attack. The DCentralAgent is also able to take decisions based on the above information and thereby launching a global reconfiguration process, which could imply to more than one molecule.

As seen, apart from splitting the CIS in different areas, the main goal is to manage the underlying system as close as possible to the lower layers (target infrastructure). Thus, the detection, decision and reaction logic will be much faster and it will avoid overloading the higher level entities.

B. Planning Block

Initially, the administrator defines both a formal description of the networked system and the Business Services that the system should offer. On the one hand, a high level language has been defined to obtain a suitable description of the network environment, called *System Description Language* (SDL) [20][21]. SDL permits to specify the system design in great detail about its physical and logical infrastructure: network elements such as interfaces, gateways and links; technical services, e.g., software and version; etc. On the other hand, the services information is modeled using the W3C's *Web Services Choreography Description Language* (WS-CDL) [22]. With this language, the administrator is able to describe the services of the system and the relationships between them by way of choreography. This allows defining the sequence and conditions in the information exchange between the participants.

The next step for the administrator is to provide the requirements or constraints that define the behavior of the Business Services. From these constraints, defined at high level, our framework is able to generate automatically the configuration model in a software-independent way. These configurations, plus a set of rules, are what we denominate *Operational Plan* (OP) [18].

An OP is an allocation strategy that defines how the services should be mapped onto molecules or technical services, and how this allocation should change when undesired incidents happen.

It comprises the following items:

- One or more *Operational Configurations* (OC). Each of them describes a particular allocation of the offered services onto either molecules (global) or technical services (local), depending on the level where the OC will be applied, called *High-level Operation Configurations* (HOC) and *Low-level Operation Configurations* (LOC) respectively. For each of these allocations, an OC includes one or more high-level configuration policies for setting up the final service.
- *Detection Scenario*. It describes which foreseen incidents we are interested in; for example, the free disk space is reaching a critical point (above 95%) or a certain command is executed by an unauthorized user. This Detection Scenario is divided into two different abstraction levels, called *Global Detection Scenario* (GDS) and *Local Detection Scenario* (LDS).
- *Reaction Scenario*. It specifies how to react when each one of the above incidents happens. This reaction consists of either switching between Operational Configurations or executing a set of abstract commands in order to fix the problems caused by the incident detected. As before, this Reaction Scenario is divided into two different ones, called *Global Reaction Scenario* (GRS) and *Local Reaction Scenario* (LRS).

Thus, an OP can be seen as a graph of Operational Configurations in which nodes are individual OCs whereas links are allocation and configuration changes launched by the detection of known incidents. Due to the multi-layered strategy employed by this architecture, there exist two kinds of OPs, called *High-level Operational Plan* (HOP) and *Local-level Operational Plan* (LOP), which can be better seen as:

$$HOP = \{\{HOC_1, HOC_2, ..., HOC_l\}, GDS, GRS\}$$

$$HOC_i = \{\{LOP_{i1}, LOP_{i2}, ..., LOP_{im}\}\} \ \forall i \in [1..l]$$

$$LOP_{ij} = \{\{LOC_{ij1}, ..., LOC_{ijn}\}, LDS_{ij}, LRS_{ij}\} \ \forall j \in [1..m]$$

Where *m* is the number of molecules in the system, whereas *l* and *n* can vary according to the possible Operational Configurations defined for each given plan.

A HOP -there will normally be only one per managed system- contains a list of possible HOCs (high-level allocation of the services onto molecules) and two scenarios for detection and reaction purposes, GDS and GRS respectively, which specify how to switch between HOCs when a problem arises. Each of these HOCs carries a set of LOPs (one per molecule) which represent a low-level allocation and configuration plan. Each LOP contains in turn a list of possible LOCs and two refined scenarios for detection and reaction purposes at local level (LDS and LRS, respectively). These LOCs represent services that are mapped onto the system components, all belong to a particular molecule, whereas the scenarios express how to switch between LOCs after detecting a fault in the system. Note that each of these local allocations will only affect to the molecule where the problem arises without involving others.

C. Reconfiguration Framework

The DESEREC modeling framework is completed with a detection and reaction model, which describes when and how to reconfigure the system in response to an incident. The reaction stage is carried out once the DESEREC framework detects that a serious incident has occurred and, in consequence, the system should react by either switching from the current OC to another or executing a set of abstract commands. The former is the imposition of a new allocation for the system services, with (possibly) a new configuration for them, whereas the latter is a minor change in the target system but without changing the running operational plan; for instance, by executing a *ddns* command to add a new RR to the domain name server dynamically.

The detection engine constantly receives events from the lower down layers which are mapped against the current Detection Scenario. When a coincidence is detected an alarm is fired, thereby starting the reaction process. Since different reaction rules (defined in the Reaction Scenario) can be associated to the same Detection Scenario rule, the decision engine should determinate which of them is the

most suitable one; that is, which is the best OC or set of abstract commands to be enforced in the target system to fix the detected problem.

This reaction process is driven by a *Policy-Based Network Management* (PBNM) approach [23], which is responsible for deploying, installing and enforcing policies in target devices; in our case either an *Operational Configuration* (OC) or a set of abstract commands, depending on the kind of reaction chosen by the decision engine, and defined in the Reaction Scenario.

An in-depth explanation of the DESEREC framework and its modules is provided in Section VII.

IV. Illustrative Example

This section introduces a complete running example, which represents usual problematic situations with the aim of clearly demonstrating the concepts and definitions that will be explained in the following sections. These situations capture typical dependability and security problems which can be managed and solved through the DESEREC framework.

A. Services

The physical testbed simulates a railway signaling and control system, where the network and services infrastructure is depicted in Figure 2. It is composed of a private network which is connected to the corporative Intranet through a firewall component. This network makes services accessible for railway administrators.

Let us suppose the following services:

- Railway Web service: it provides a Web service interface for the railway signaling and control system.
- DNS service: it defines a domain name for the IP address of the previous Web service.
- Firewall service: it keeps packet filtering rules to services and network components.
- Timing service: it provides time synchronization for all the components that need it.

The railway Web service is configured to listen on two ports, depending on the requested services: on port 5486 a Web service interface is offered for interaction with the signaling and control system; and on port 443 administrators can gain access to a management Web page which provides statistics and monitoring services for the system. Both interfaces are offered through a secure HTTPS communication by using, for example, SSL or TLS with X.509 certificates.

The DNS service will be configured with the IP address where the Web service will be placed and, at the beginning, the firewall will allow connections to both service ports. In order for the service to be accessed through its domain name by administrators, the firewall will also allow connections on port 53.

B. Testbed Description

The scenario used for this example is comprised by two molecules. Figure 2 depicts the molecules and the software distribution in the testbed.

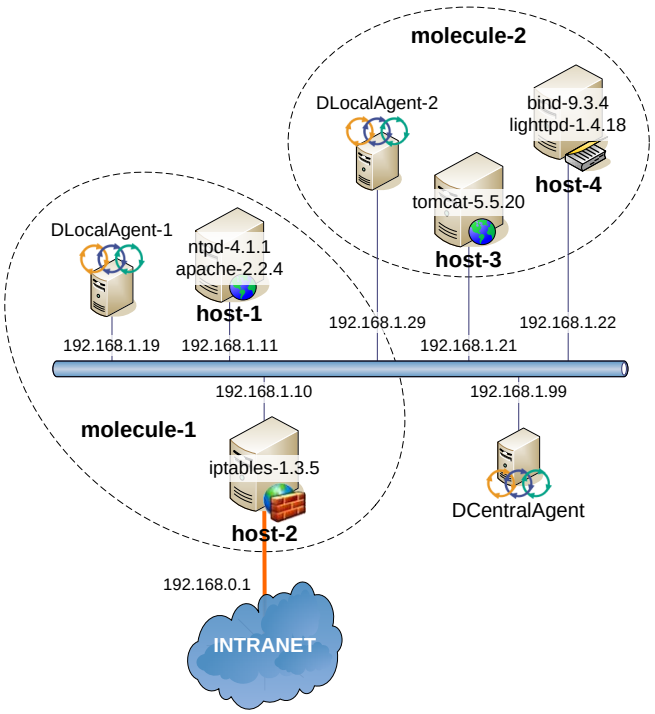

Figure 2. Molecules distribution and DESEREC components in the testbed

Table I summarizes the software requirements, along with their version numbers, needed to properly deploy this scenario.

Table I
Elements into the molecule breakdown

MOLECULE	COMPONENT	SPECIFIC SOFTWARE
molecule-1	Web server	apache-2.2.4
	Firewall	iptables-1.3.5
	Timing server	ntpd-4.1.1
molecule-2	Web server	lighttpd-1.4.18
	Web server	tomcat-5.5.20
	Name server	bind-9.3.4

It is worth noting that there are three Web servers available, installed in different elements. Two of them (Apache and Tomcat) support secure connections through SSL and TLS, whereas the other one (LigHTTPd) is not able to provide this feature. This last server has been included in the testbed, although it will be discarded during the allocation process since it does not provide the required features.

In this scenario, the corresponding DESEREC framework entities have also been included: one DLocalAgent per molecule, which will receive event notifications and will

launch the detection and reaction processes; and a DCentralAgent in charge of supervising those molecules.

Three different situations are considered to show how the DESEREC framework works to confront dependability and security problems. Either of these situations could cause dependability or security problems that the DESEREC framework should correct. They will make the framework react in three different ways and will serve to illustrate the following three kinds of reactions:

1) *Reallocation*: The Web service goes down due to a DoS attack from a compromised host in the Intranet and it becomes unavailable. In this case, the framework reaction is to move (reallocate) that service to another place which is able to get it running again.
2) *Reconfiguration*: An unauthorized user gains access to the private Web page. The reaction of the framework in this situation is to reconfigure the firewall by changing its current configuration to another more restrictive. Only connections to the critical system through HTTPS will be permitted on port 5486, blocking accesses to the Web page.
3) *Abstract command execution*: Some important Web service files are removed from the server due to a temporal hard disk failure and the Web service becomes inconsistent. The reaction of the framework is to execute a command, specified using an abstract syntax, that will copy the removed files from a backup folder to the appropriate directory in the Web server.

V. High-level Configuration Constraints

The deployment of Operational Configurations, used to enhance the system dependability, requires the definition of the desired prerequisites that Business Services should accomplish. These prerequisites are comprised by a set of requirements or constraints defined during the business process and specified by the administrator using a set of high level configuration policies for the different Business Services.

Each Business Service in a DESEREC-managed system is comprised by a number of individual *Business Service Components* (BSC). These BSCs carry out specific tasks of their Business Service, which can be mapped to one or more technical services of some kind. For example, a Business Service that provides a Web portal could be comprised by one BSC for delivering the Web content, another BSC for data storage, and another for performing access control operations.

Typically, any technical service needs to have a proper configuration in order to implement a BSC. The system administrator must supply some configuration policies or constraints (ideally, vendor-independent ones) about how the BSCs should operate. These policies will serve as the bases for generating the configuration of the technical services that will finally implement the BSCs. These guidelines are referred to as *service configuration constraints*.

The service configuration constraints are modeled in DESEREC via the *Service Constraints Language* (SCL) [20]. This language specifies which are the service configuration constraints for a given DESEREC-managed system, and how the constraints defined in it are related to the BSCs of the system.

Figure 3 gives a basic view of the Business Services and their BSCs defined for the illustrative example introduced in Section IV, and how each of them has a set of one or more configuration constraints.

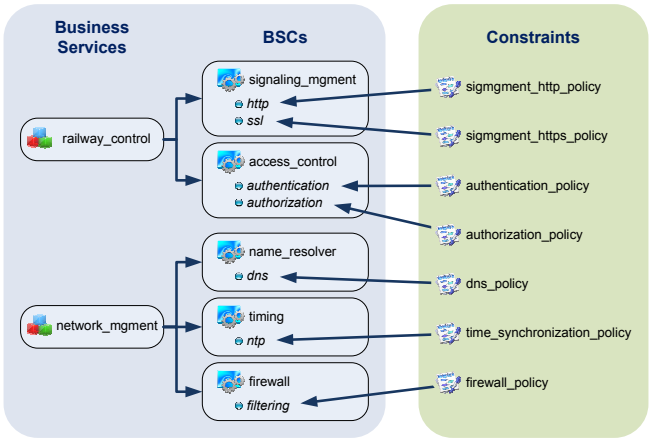

Figure 3. Constraints assigned to the BSCs

The relationship between BSCs and constraints lies on the concept of *capability*. A BSC is associated with a set of capabilities which define the type of service the BSC provides. For instance, the BSC providing access control has two capabilities: *authentication* and *authorization*.

The set of capabilities associated to one BSC determines the possible service configuration constraints that can be assigned to it. Every configuration policy defined in SCL belongs to a constraint type, being different constraint types able to configure specific capabilities. Thus, the BSC *access_control* has two constraints assigned to it: an *AuthenticationConstraint* specifying the authentication policy for the *authentication* capability; and an *AuthorizationConstraint* specifying the authorization policy for the *authorization* capability.

It is worth noting that configuration constraints may be reused. If more than one BSC requires exactly the same behavior for a given type of service, then they may share the same constraint. For example, there may be a new Business Service in the system containing another BSC for time synchronization which should have the same behavior and synchronize with the same time servers, as the one shown in Figure 3. That BSC may share the already defined *time_synchronization_policy* constraint, that is, the administrator can assign the same constraint to both BSCs.

This however does not mean that exactly the same technical configuration will be used on both, because they could be implemented by different software packages. Constraints are vendor-independent and they will have to be translated to final configurations only when the final software to which they will be applied is known.

Since configuration constraints are defined before the final software implementing the services is known, SCL is able to describe the configuration semantics in a vendor independent way. Moreover, dependability management implies service reallocation; e.g., a service that is running on a machine can be reallocated to another if the previous one fails. The first situation in the illustrative example of Section IV introduces this kind of reaction for the railway Web service, and it will be further described in Section VII-E (Situation 1). This requires the description of configuration constraints with a high level of abstraction, based on the BSC concept and avoiding the usage of allocation-dependent data.

The definition of domain name resolution constraints shows this problem about service reallocations. In the example, the managed system provides a website which should be accessed from the Intranet through a given URL, for instance `https://signaling.example.org`. The domain `example.org` is managed by the company and, therefore, the DNS service should be configured to map `signaling` to the company's railway signaling website. In this scenario, there are two different BSCs, one representing the website and another one representing the DNS service. The configuration constraint defining the behavior for the BSC *name_resolver* should specify a record with the IP address of the website as response for the `signaling` query by the DNS service. However, the IP address of the website is unknown since the service can be reallocated to a different machine in case of failure.

Because of this, SCL supports BSC references in policies. This allows administrators to specify BSC identifiers instead of IP addresses, or any other static information, which may vary depending on where the service is finally allocated by the dependability management system.

Other examples to illustrate these high level configurations may be the time synchronization policy specifying *Listen to the default NTP port* and *synchronize with another timing BSC named "corporative_timing"*. Or the filtering policy specifying *Allow traffic from the Intranet to the BSC "signaling_mgment"* and *Deny everything else*.

A further policy refinement process will resolve these references and will translate the high level configuration constraints defined in SCL into a lower level language. This language will contain all the specific and detailed data needed to generate the precise configuration for the current system.

To show the SCL structure and some of the aforementioned characteristics, Listing 1 presents the corresponding constraints for the illustrative example in Section IV.

```
<scl id="example_policies" plan="example_plan">
  <constraintAssignments>
    <bscConstraints bsc="example.access_control">
      <constraint ref="authentication_policy"
          requiredCapability="authentication"/>
      <constraint ref="authorization_policy"
          requiredCapability="authorization"/>
    </bscConstraints>
    ...
  </constraintAssignments>
  <constraintDefinitions>
    <authenticationConstraint id="authentication_policy">
      <authenticationRules>
        <authenticationRule name="user1_id">
          <identity>user1@example.org</identity>
          <credentials>
            <accountCredential>
              <accountId>user1</accountId>
              <accountContext>example.access_control</accountContext>
            </accountCredential>
          </credentials>
        </authenticationRule>
        ...
      </authenticationRules>
    </authenticationConstraint>
    <authorizationConstraint id="authorization_policy">
      <roles>
        <role name="signaling_admins">
          <identity>user1@example.org</identity>
          ...
        </role>
      </roles>
      <authorizationRules>
        <authorizationRule name="allow_signaling_mgment">
          <role>signaling_admins</role>
            <privileges>
              <privilege granted="true" name="signaling_mgment_access">
                <activities>
                  <activity>Read</activity>
                  <activity>Write</activity>
                </activities>
                <qualifiers>
                  <qualifier type="Packets"/>
                </qualifiers>
                <target>signaling.example.org</target>
              </privilege>
            </privileges>
        </authorizationRule>
      </authorizationRules>
    </authorizationConstraint>
    <httpConstraint id="sigmgment_http_policy"> ... </httpConstraint>
    <sslConstraint id="sigmgment_https_policy"> ... </sslConstraint>
    <dnsConstraint id="publicdns_policy">
      ...
      <zones>
        <zone domain="example.org" type="Master">
          ...
          <records>
            ...
            <record type="A">
              <query>signaling</query>
              <response type="BSC">example.signaling_mgment</response>
            </record>
          </records>
        </zone>
      </zones>
    </dnsConstraint>
    <filteringConstraint id="firewall_policy"> ... </filteringConstraint>
    <ntpConstraint id="time_synchronization_policy"> ... </ntpConstraint>
  </constraintDefinitions>
</scl>
```

Listing 1. Service Constraints Language

For clarity reasons, XML namespaces have been removed from the listing and some fragments have also been replaced by dots. It can be seen that this set of constraints and policies contains an identifier (*example_policies*) and it is

defined in the scope of a concrete Operational Plan, named *example_plan* in this case.

An SCL description is formed by two main parts: constraint assignments and constraint definitions. The former defines the policy which is associated to each capability of each BSC defined in the operational plan. The latter contains the actual definitions of the policies. This separation allows reusing policies defined by different BSCs; i.e., a policy defined in the definitions section can be assigned to two different BSCs in the assignments section.

Listing 1 shows just one constraint assignment corresponding to the BSC *access_control*. In the full description there is a constraint assignment for every BSC. Moreover, all policies appear in the description section, but their content has been replaced by dots and only the two assigned to the BSC *access_control* appear almost complete, just to illustrate the aspect of policies in SCL. Moreover, some fragments of the DNS policy of the BSC *name_resolver* are also shown to illustrate the previous example of BSC references in constraints. It can be seen how the record for `signaling` within the domain `example.org` assigns that domain name to the BSC *signaling_mgment* without still knowing its actual IP.

VI. Policy Modeling in DESEREC

As stated in Section V, within the DESEREC framework the administrator defines the behavior of the Business Services by means of a set of configuration constraints or policies defined in SCL. This language operates in a high level fashion, making use of platform-independent semantics as well as being transparent to the allocation process from Business Services to technical services.

In this context, it is worth reaching a trade-off between a high level formal language able to express the administrator's abstract requirements, in a human readable way, and a formal language able to be parseable by an automatic translation process. This is exactly what the SCL language deals with, defining models as clear as possible to be later used by an intelligent software process.

However, a standard model with the proper level of detail to allow generating configurations for the real system is also needed in order for the enforcement phase to be done properly in a vendor independent way. DESEREC uses the *Common Information Model* (CIM) [24] as this final model since it is a very complete information model. It covers almost all the different aspects required in a networking scenario, including systems, services, networks, applications, policies, security, etc. Moreover, CIM is independent of the language used to represent it, free, open source and extensible. Additionally, this information model has been used in a wide variety of research works, such as [25] or [26] among others.

Both DMTF and DESEREC define an XML representation of this model following two different approaches. The DMTF, in its standard CIM-XML of WBEM, uses a metaschema mapping which defines an XML schema to describe CIM, where both classes and instances are valid documents to the CIM metaschema. On the other hand, DESEREC uses an XML schema to describe CIM classes as XML complex types. Thus, CIM instances are described in valid XML documents for that schema. DESEREC reuses and extends the xCIM language, originally defined in the POSITIF EU-IST project (*Policy-based Security Tools and Framework*, IST-2002-002314), to provide this last kind of XML format representation of CIM. Furthermore, and thanks to the usage of the xCIM language, a wide range of possibilities become available. For example, there are related standards and technologies grouped under the WBEM specifications [27], which allow dynamically gathering the current state of the system by means of CIM.

Therefore, although the xCIM language is a useful implementation of CIM (including extended classes), it does not fit perfectly in the DESEREC requirements due to the great amount of classes that compound the model, and the lack of some DESEREC-specific requirements. To solve this issue, DESEREC defines a sublanguage, called *xCIM Service Constraints Language* (xCIM-SCL), that allows the representation of service constraints and policies.

So far, in DESEREC, different kinds of policies have been modeled for both SCL and xCIM. Among them, we have security policies, such as authentication, authorization, filtering or SSL, and also common constraints to specify service configurations, such as HTTP, NTP, NAT, DNS, streaming, DHCP or load balancing. Anyway, both languages, SCL and xCIM, can be easily extended to hold any new kind of policy.

A. SCL Console

In order to assist the administrator with the task of defining service configuration constraints in SCL, DESEREC has developed an intuitive console that permits to generate and manage SCL models using a graphical interface.

Through the SCL Console [28], a system administrator can create a service constraints model in SCL which defines how the system is expected to operate. This is done by working on the system model (physical/logical infrastructure) and the service model (Business Services description, decomposition and interaction). This console generates the aforementioned constraints model (SCL), which contains the configuration constraints defined by the administrator, as well as the assignments between them and the present BSCs in the service model.

Figure 4 is a snapshot of the SCL Console showing the Business Services, BSCs, capabilities and service constraint policies defined for the illustrative example introduced in Section IV. The console presents three well-defined areas in the frame: one for browsing through the Business Services, BSCs and capabilities (top-left area); a bigger one for viewing and editing the SCL constraints (right area); and a

last one for browsing through the constraints and assigning them to the BSCs (bottom-left area).

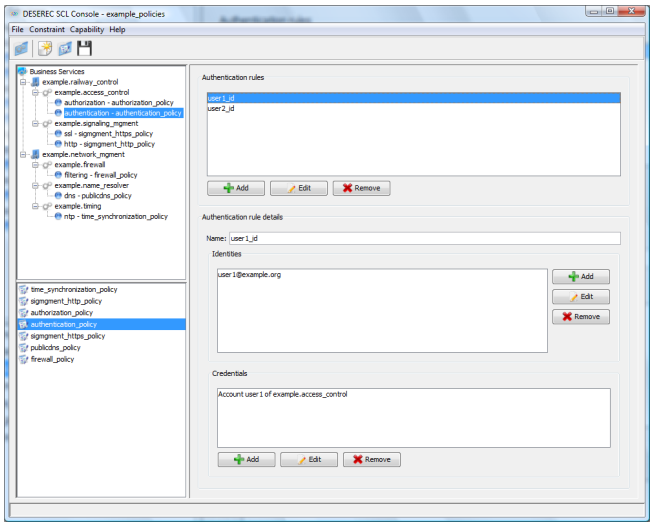

Figure 4. Snapshot of the SCL Console

The SCL Console supports creation, edition and management of the whole set of configuration policy types, defined by DESEREC and mentioned in this section. Nevertheless, the constraints processing in the SCL Console is implemented by following a plug-in based approach. That is, each supported constraint is implemented as a separate plug-in which is loaded on-the-fly. This means that the set of supported constraints can be extended as desired, just by developing additional plug-ins. Moreover, the console is able to manage SCL models with constraint types for which there is no plug-in to support its edition or creation. In such a case, the SCL Console will show the constraint as an XML text, thereby allowing the administrator to work with it.

B. Service allocation

Once the system administrator has defined the service configuration constraints, the Planning block must automatically allocate, in a first step, each BSC onto the molecules at global level; that is, it will provide the set of the HOCs that will compose the final global plan. To this end, the allocation process maps the required capabilities that each BSC needs against the available capabilities that each molecule provides (defined previously in SDL). The next step of this process is to calculate all possible local allocations for each HOC generated previously, thus providing each of the LOCs that will compose each HOC. As before, this local allocation process is carried out by means of mapping the required capabilities that each BSC needs against the available capabilities that each technical service provides (also defined in SDL).

Note that if one allocation does not fulfill the requirements defined by the administrator, the service cannot be allocated and the corresponding Operational Configuration (HOC or LOC) is discarded.

Table II shows the output of this process for the running example. As can be seen, two HOCs have been generated, in which the only difference is the allocation of the Business Service *railway_control* (for both BSCs, *signaling_mgment* and *access_control*). In the first case (`HOC1`), both BSCs are allocated onto *molecule-1* and, in the second one (`HOC2`), they are allocated onto *molecule-2*. In both cases the Business Service *railway_control* can be allocated onto the two defined molecules since they provide the required capabilities needed by it.

At local level, each HOC generates one LOC: a first LOC belonging to `HOC1` (`HOC1.LOP`$_{1\&2}$`.LOC1`) in which the complete Business Service *railway_control* is allocated onto the *apache-2.2.4* software; and another first LOC belonging to `HOC2` (`HOC2.LOP`$_{1\&2}$`.LOC1`) in which the same Business Service is now allocated onto the *tomcat-5.5.20* software. Note that, for example, the BSC *signaling_mgment* cannot be allocated onto the *lighttpd-1.4.18* software since it does not provide one of the required capabilities (specifically the *ssl* one to provide secure connections), thus being discarded during this mapping process.

The LOCs introduced in Table II will be later split and packaged in LOPs depending on the molecule to which they are addressed. In this case, all the allocations to *molecule-1* will be packaged as LOP1, and those addressed to *molecule-2* will be packaged as LOP2.

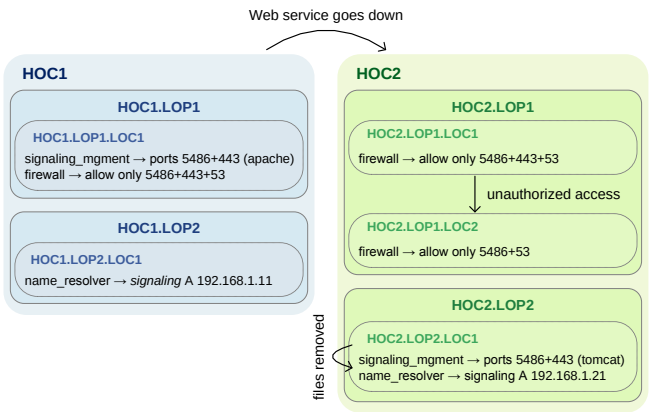

Figure 5. Complete allocation graph including detection/reaction logic

From these allocations, the Planning block generates the complete allocation graph, as the one shown in Figure 5 for the running example.

For clarity reasons, the BSCs *timing* and *access_control* have not been included since the former is always allocated in the same technical service (the only one that provides the required capability), and the latter is always allocated together with the BSC *signaling_mgment*. In this allocation graph, the allocation and configuration changes have also been included to follow how the DESEREC framework will work in runtime after detecting the aforementioned incidents.

Table II
ALL POSSIBLE ALLOCATIONS AT BOTH GLOBAL AND LOCAL LEVEL

GLOBAL LEVEL (mapping onto molecules)	LOCAL LEVEL (mapping onto technical services)
HOC1	**HOC1.LOP$_{1\&2}$.LOC1**
railway_control.signaling_mgment –> molecule-1	railway_control.signaling_mgment –> molecule-1.host-1.apache-2.2.4
railway_control.access_control –> molecule-1	railway_control.access_control –> molecule-1.host-1.apache-2.2.4
network_mgment.name_resolver –> molecule-2	network_mgment.name_resolver –> molecule-2.host-4.bind-9.3.4
network_mgment.timing –> molecule-1	network_mgment.timing –> molecule-1.host-1.ntpd-4.1.1
network_mgment.firewall –> molecule-1	network_mgment.firewall –> molecule-1.host-2.iptables-1.3.5
HOC2	**HOC2.LOP$_{1\&2}$.LOC1**
railway_control.signaling_mgment –> molecule-2	railway_control.signaling_mgment –> molecule-2.host-3.tomcat-5.5.20
railway_control.access_control –> molecule-2	railway_control.access_control –> molecule-2.host-3.tomcat-5.5.20
network_mgment.name_resolver –> molecule-2	network_mgment.name_resolver –> molecule-2.host-4.bind-9.3.4
network_mgment.timing –> molecule-1	network_mgment.timing –> molecule-1.host-1.ntpd-4.1.1
network_mgment.firewall –> molecule-1	network_mgment.firewall –> molecule-1.host-2.iptables-1.3.5

It is worth mentioning that in Figure 5 it has been added a second LOC in `HOC2.LOP1`, which represents the reaction thrown after detecting an unauthorized access (second problematic situation introduced in Section IV-B). This local reconfiguration is just a change in the firewall configuration but without implying changes in the service allocation; that is, the allocations are maintained exactly the same as the ones defined before the reaction.

C. Policy Refinement

Definition of high-level objectives is usually the way administrators work. To make these objectives a reality in terms of configuration, lots of information need to be provided to a refinement process, from high-level objectives to final configurations [29]. Afterwards, these configurations can then be deployed to the final devices and services, in order to maintain the system configured properly based on the administrator requirements. This avoids administrators to generate a wide range of different and specific configuration files for each device or service.

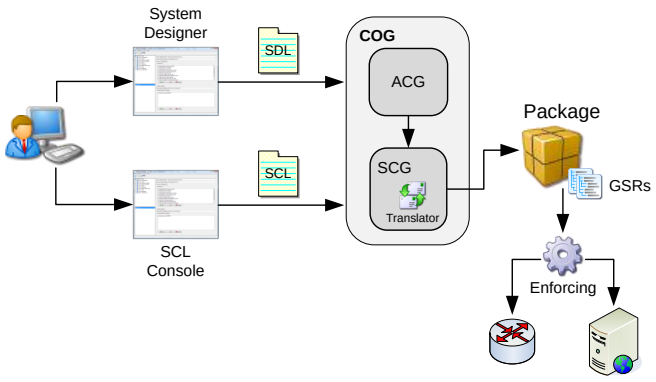

Figure 6. Refinement process workflow

In this context, a top-down engineering approach of this refinement process has been designed and implemented. Figure 6 depicts the refinement workflow followed in DESEREC. Firstly, the administrator defines both the Business Services and a complete system description using the *System Description Language* (SDL). The requirements, i.e., configuration constraints defining Business Services behavior, are also defined in SCL by means of the SCL Console as explained in Section VI-A. Then, all this information is used to generate the generic configuration model in xCIM format, which will be finally used by the reconfiguration framework to configure the system resources.

DESEREC relies on the *Configuration Generator* (COG) to perform the translation process (see Figure 6), shich is the central part of the Planning block in the DESEREC framework. It is in charge of taking the requirements from the administrator in terms of a system description, Business Services specification and the desired behavior of services. Then, it produces as output an *Operational Plan* (OP) containing the configuration models in xCIM needed to enforce it. The OP will contain a list of alternative Operational Configurations, and the needed detection/reaction logic that will implement the dependability features, as explained in Section III-B.

There are two submodules in the COG, the *Automatic Configuration Generator* (ACG) and the *Services Configuration Generator* (SCG). The former generates a first version of the Operational Plan containing Operational Configurations; that is, allocations of Business Services onto infrastructure elements, supporting the molecule abstraction. The latter analyzes the allocations present in the Operational Configurations, adds semantics to the constraint model, and produces configuration packages as a *Generic Service Ruleset* (GSR) [20]. This is a generic model that will enable the DESEREC runtime framework to actually set up the software elements to operate as desired.

The SCG takes the Operational Configurations one by one and launches the process autonomously for each one. As a result, a versatile package is generated, which contains the GSRs as well as the whole information about the system

where the GSR is assigned to. The SCG model has been developed as a Java application which allows loading and processing the Operational Plans in order to produce GSR packages in xCIM-SCL format. Indeed, the *Translator* is the SCG submodule in charge of translating from policies/constraints (defined in SCL) to xCIM-SCL.

In order to generate the GSRs properly, the *Translator* needs as input both the policy information to be refined and the information about the system that is being managed. For this second issue, DESEREC relies on the SDL language. Since the SDL model is also included as a part of the Operational Plan, this model is available to the translation process.

It is worth noting that the *Translator* module is composed of smaller and specific *translation submodules*, as many as different kinds of policies are supported by the DESEREC framework; i.e., HTTP, DNS, filtering, etc. Thus, each submodule is specialized on translating each kind of policy, taking into account its idiosyncrasy. This fact leads our approach to be easily extended with new kinds of policies. This refinement process is based on the one defined in [30].

```
<gsr:GSR xmlns:gsr="http://www.deserec.eu/xsd/gsr">
  <gsr:xDESEREC_DNSServerSettingData>
    <InstanceID>publicdns_policy</InstanceID>
    <Forwarders>198.41.0.4</Forwarders>
  </gsr:xDESEREC_DNSServerSettingData>
  <gsr:xDESEREC_DNSZoneSettingData>
    <InstanceID>publicdns_policy.zone1</InstanceID>
    <Domain>example.org</Domain>
    <Type>1</Type>
    <TimeToRefresh>7200</TimeToRefresh>
    ...
  </gsr:xDESEREC_DNSZoneSettingData>
  <gsr:xDESEREC_DNSRecordSettingData>
    <InstanceID>publicdns_policy.zone1.record3</InstanceID>
    <ElementName>publicdns_policy.zone1.record3</ElementName>
    <Query>signaling</Query>
    <Type>1</Type>
    <Response>192.168.1.11</Response>
  </gsr:xDESEREC_DNSRecordSettingData>
  <gsr:xCIM_ConcreteComponent>
    <GroupComponent>DESEREC_DNSZoneSettingData.InstanceID=
      'publicdns_policy.zone1'</GroupComponent>
    <PartComponent>DESEREC_DNSRecordSettingData.InstanceID=
      'publicdns_policy.zone1.record1'</PartComponent>
  </gsr:xCIM_ConcreteComponent>
  ...
  <gsr:xDESEREC_GSRHeader>
    <TransportationMethod>COPS−PR</TransportationMethod>
    <GSRTarget>system.netw.servers.M2.DNSServer.dnsd1</GSRTarget>
    <GSRTargetSoftware>
      system.netw.servers.M2.DNSServer.dnsd1.PublicDNSAGTsw
    </GSRTargetSoftware>
    <MoleculeID>system.netw.servers.M2</MoleculeID>
  </gsr:xDESEREC_GSRHeader>
</gsr:GSR>
```

Listing 2. xCIM representation of the DNS policy

Listing 2 shows how the translation process works, based on the running example defined in Section IV. It shows a fragment of a GSR that represents the aforementioned DNS policy, but now translated into xCIM format. The *Translator* generates this GSR document taking into account the SCL policy, the allocation information and the system description in SDL.

At first sight, it can be noticed that the xCIM language is more complex and, therefore, harder to understand than SCL. For instance, it codifies different kinds of SCL options by means of numbers. In order to understand this point, let us compare the Listing 2 with its equivalent specified in SCL, and shown in Listing 1. The DNS zone typed as *Master* in the SCL policy is now codified as $< Type > 1 < /Type >$ inside the xCIM class *xDESEREC_DNSZoneSettingData*.

Furthermore, it is important to note how the *Translator* resolves the BSC references in the DNS policy to IP addresses; e.g., the previous *A* record response reference called *example.signaling_mgment* has been replaced by its corresponding IP according to the allocations defined in `HOC1.LOP1.LOC1` of the running example. Note that it is possible since the allocation process is done before the translation process takes place.

Additionally, every GSR maintains some control information which is later used by the DESEREC framework to perform the enforcement and other operations. Thus, the xCIM class *xDESEREC_GSRHeader* contains some useful parameters, such as the reference to the target element where the GSR is going to be enforced or the transportation method used in such an enforcement.

Listing 3 shows another GSR, but now with respect to the access control policies that can be also found in SCL, and shown in Listing 1.

```
<gsr:GSR xmlns:gsr="http://www.deserec.eu/xsd/gsr">
  <gsr:xCIM_Role>
    <CreationClassName>CIM_Role</CreationClassName>
    <Name>authorization_policy.role.signaling_admins</Name>
    <CommonName>signaling_admins</CommonName>
    <ElementName>signaling_admins</ElementName>
  </gsr:xCIM_Role>
  <gsr:xCIM_Identity>
    <InstanceID>authorization_policy.role.signaling_admins.identity.
      user1@example.org</InstanceID>
    <ElementName>user1@deserec.org</ElementName>
  </gsr:xCIM_Identity>
  <gsr:xCIM_MemberOfCollection>
    <Collection>CIM_Role.CreationClassName='CIM_Role',
      Name='authorization_policy.role.signaling_admins'</Collection>
    <Member>CIM_Identity.InstanceID='authorization_policy.role.
      signaling_admins.identity.user1@example.org'</Member>
  </gsr:xCIM_MemberOfCollection>
  <gsr:xCIM_Privilege>
    <InstanceID>authorization_policy.allow_signaling_mgment.
      signaling_mgment_access</InstanceID>
    <PrivilegeGranted>true</PrivilegeGranted>
    <Activities>5</Activities>
    <Activities>6</Activities>
    <QualifierFormats>11</QualifierFormats>
  </gsr:xCIM_Privilege>
  <gsr:xCIM_AuthorizationRuleAppliesToPrivilege>
    <PolicySet>
      CIM_AuthorizationRule.SystemCreationClassName='CIM_AdminDomain',
      SystemName='system.netw.servers.M1',PolicyRuleName='
      authorization_policy.allow_signaling_mgment'</PolicySet>
    <ManagedElement>CIM_Privilege.InstanceID='authorization_policy.
      allow_signaling_mgment.signaling_mgment_access'</ManagedElement>
  </gsr:xCIM_AuthorizationRuleAppliesToPrivilege>
  <gsr:xCIM_AuthorizationRuleAppliesToTarget>
    <PolicySet>
      CIM_AuthorizationRule.SystemCreationClassName='CIM_AdminDomain',
      SystemName='system.netw.servers.M1',PolicyRuleName='
      authorization_policy.allow_signaling_mgment'</PolicySet>
    <ManagedElement>signaling.example.org</ManagedElement>
  </gsr:xCIM_AuthorizationRuleAppliesToTarget>
```

```
...
<gsr:xCIM_AuthorizationRule>
  <SystemCreationClassName>
    CIM_AdminDomain
  </SystemCreationClassName>
  <SystemName>system.netw.servers.M1</SystemName>
  <CreationClassName>CIM_AuthenticationRule</CreationClassName>
  <PolicyRuleName>
    authorization_policy.allow_signaling_mgment
  </PolicyRuleName>
</gsr:xCIM_AuthorizationRule>
<gsr:xCIM_AuthorizationRuleAppliesToRole>
  <PolicySet>CIM_AuthorizationRule.SystemCreationClassName=
    'CIM_AdminDomain',SystemName='system.netw.servers.M1',
    PolicyRuleName='authorization_policy.
    allow_signaling_mgment'</PolicySet>
  <ManagedElement>CIM_Role.CreationClassName='CIM_Role',
    Name='authorization_policy.role.
    signaling_admins'</ManagedElement>
</gsr:xCIM_AuthorizationRuleAppliesToRole>
<gsr:xCIM_PolicyRuleInSystem>
  <Antecedent>CIM_AdminDomain.CreationClassName='CIM_AdminDomain',
    Name='system.netw.servers.M1'</Antecedent>
  <Dependent>CIM_AuthorizationRule.SystemCreationClassName=
    'CIM_AdminDomain',SystemName='system.netw.servers.M1',
    PolicyRuleName='authorization_policy.
    allow_signaling_mgment'</Dependent>
</gsr:xCIM_PolicyRuleInSystem>

<!-- Authentication Policy -->

<gsr:xCIM_AuthenticationRule>
  <SystemCreationClassName>
    CIM_AdminDomain
  </SystemCreationClassName>
  <SystemName>system.netw.servers.M1</SystemName>
  <CreationClassName>CIM_AuthenticationRule</CreationClassName>
  <PolicyRuleName>authentication_policy.user1_id</PolicyRuleName>
</gsr:xCIM_AuthenticationRule>
...

<gsr:xDESEREC_GSRHeader>
  <TransportationMethod>COPS-PR</TransportationMethod>
  <GSRTarget>system.netw.servers.M1.Apache2</GSRTarget>
  <GSRTargetSoftware>
    system.netw.servers.M1.Apache2.AAAsw
  </GSRTargetSoftware>
  <MoleculeID>system.netw.servers.M1</MoleculeID>
</gsr:xDESEREC_GSRHeader>
</gsr:GSR>
```

Listing 3. xCIM representation of the access control policy

At the sight of the above fragment of XML document, both authentication and authorization policies share the same target element in this GSR and, therefore, the GSR header is unique for both of them. It means that this configuration will be enforced in the same software of the same machine for a given molecule. Please bear in mind that the *Translator* module will generate some other GSRs, which are able to configure other target elements according to the allocations established before, always taking the same SCL access control policies as input. Moreover, as can be seen, the xCIM language is still defined as a generic configuration and it is not linked to any implementation or particular software. It will be the enforcing mechanisms, described in section VII-D, the ones in charge of generating the actual configuration files depending on the concrete software.

As before, and for the sake of clarity, the authentication policy has been nearly omitted from the GSR and it is not shown in Listing 3.

VII. Reconfiguration Framework

This section introduces an in-depth explanation of the reconfiguration framework, its modules and components. We also present a complete illustrative example as demonstration of the proposed framework.

A. General Requirements

In this subsection we summarize the general requirements that have been identified by DESEREC, and that our framework should fulfill. Among them, we include scalability, language interoperability, security assurance, autonomy and, finally, issues related to service continuity and reliability of reconfiguration. These requirements are summarized as follows:

- Only well-characterized incidents shall be treated, and their analysis needs to be very fast and non ambiguous for detecting an incident in runtime.
- The reaction process shall be automatically carried out as soon as possible after the detection of an incident. Therefore, human interaction cannot take place in this process, although the system administrator could be alerted with high priority.
- Strong mechanisms must be provided and supported to avoid intrusion.
- It is necessary to use as much standard languages as possible to exchange consistent information between heterogeneous managed components (target system components and the DESEREC framework).
- A distributed solution should be designed since large systems will produce a great amount of events that must be processed.
- The detection and reaction processes should take a maximum time interval of a few minutes in order to maintain the service continuity.
- This framework must provide mechanisms to guarantee the integrity of the requested reconfiguration since it could provoke a breakdown of the service if it is corrupted.

B. Workflow of the Reconfiguration Framework

Once the configuration information is released, by using the planning tools described above, the reconfiguration framework works autonomously without human intervention. This information must be deployed through the DESEREC architecture and applied both in the corresponding technical services, for configuring them properly, and in the different modules of the architecture for detection and decision purposes.

Figure 7 depicts the complete workflow inside this framework, labeled with the information exchanged between the modules. This exchange is done using SOAP-based Web service interfaces. Please note that only the local reconfiguration framework is shown in Figure 7, although it could be extended to the global one since both levels (global and

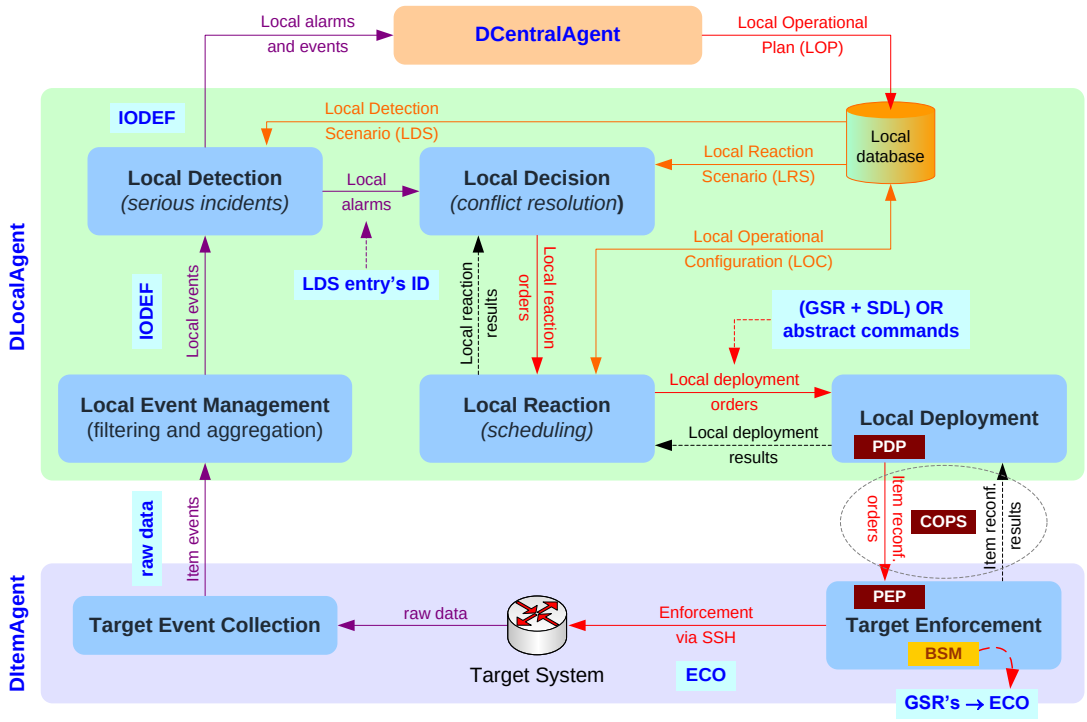

Figure 7. Local reconfiguration framework

local) work in a similar way to be composed of the same modules.

Through the *Target Event Collection*, which collects raw events from the target elements, the *Local Event Management* is continuously gathering item events, filtering and aggregating them to provide higher level events to the upper modules. This will avoid overloading these modules and will reduce the bandwidth occupation. Furthermore, the *Local Event Management* module also transforms the collected item events into a normalized format and sends them to the *Local Detection*. In our case, the chosen exchange format is IODEF [31] since it is a W3C standard format defined to represent and exchange operational and statistical information between components.

The Detection modules are constantly receiving events from the lower levels. DLocalAgents retrieve them from the target system through the DItemAgents, whereas the DCentralAgent retrieves them from the DLocalAgents. These events are matched against the possible problems defined in the *Detection Scenario* (GDS or LDS, depending on the level or agent) with the help of a set of signature-based rules included in that scenario. If there is a coincidence, it means that this module has detected a fault in the system and a response is required.

The Detection module notifies to the Decision one which alarm has occurred; that is, the problem that has been detected. In addition, if the problem has been detected at local level, the *Local Detection* module also forwards the appropriate alarm, i.e., the IODEF itself and some additional information about the problem, to the DCentralAgent.

The Decision module retrieves from the *Reaction Scenario* (GRS or LRS, depending on the level or agent) a list of possible reactions to solve the previously raised problem. Note that for each problem or alarm we have a list of $[1..n]$ reactions. This module will decide the most suitable reaction to carry out taking into account the statistical data harvested from the target system. The current system situation could also be taken into account for this decision-making process to choose the best reaction in a particular moment. When taking this decision it will always first try to apply a local reaction, which will be faster and less costly; otherwise, the DCentralAgent will be informed to take the corresponding global reconfiguration, if necessary.

At global level, the *Global Decision* module sends to the *Global Reaction* the HOC identifier with the new configuration to deploy in the system. On the other hand, at local level, the *Local Decision* module sends to the *Local Reaction* the XML-based reaction to apply, which includes either the LOC identifier with the new local configuration or a set of abstract commands to be executed to fix the problem.

The Reaction module retrieves the appropriate *Operational Configuration* (HOC or LOC) from its local database using the identifier (HOC ID or LOC ID) sent by the Decision module. Note that if the reaction is to apply a set of abstract commands, the Reaction module does not need to retrieve any information since these commands are already

included in the XML-based reaction instance itself. Later on, this last module queues the set of deployment orders and sends them progressively to the Deployment module; until an order is not correctly deployed and enforced, the next one is not sent. In the *Global Reaction* these orders will be the LOPs contained in the new HOC, whereas in the *Local Reaction* they will be the GSRs contained in the LOC (one GSR per service to be configured).

The Deployment module delivers the reconfiguration orders to the lower agents which will process them in a different way, depending on the agent level involved:

- At global level, each LOP sent by the *Global Reaction* module (one per molecule) is forwarded to the corresponding molecule associated with that LOP. This process is repeated for each molecule, in which each LOP is stored in its own repository. Then, the *Local Decision* module launches the deployment of the first configuration for the new LOP with the best possible LOC. During this phase, the *Local Detection* and the *Local Decision* modules are automatically reconfigured with the new scenarios specified in the new LOP as well.
- At local level, the GSR received by the *Local Deployment* module is sent to the appropriate *Target Enforcement* module which manages the underlying software. Each GSR will be translated to *End COnfigurations* (ECOs), thanks to the *Block Service Module* (BSM) submodule, see Section VII-D, just before enforcing it in the target system. It is worth noting that a local deployment order could also be a set of abstract commands, which also have to be translated to target-specific commands for being enforced in the system.

It is worth mentioning that during all this process, as can be seen in Figure 7, feedback information is sent to upper modules and/or layers for reporting the sending and proper execution of the requested orders. If a deployment error is reported, e.g., an Operational Configuration becomes unfeasible in that moment, the upper layers will then have to decide which of the rest of available Operational Configurations could be deployed as valid, taking into account the current system situation.

C. PBNM Approach for Deployment

The last action in the reconfiguration framework is to distribute and enforce the GSRs into the final technical services. By this, the deployment phase is based on a *Policy-Based Network Management* (PBNM) approach [23], which allows deploying, installing and enforcing policies in the target devices.

This architecture is basically composed of the following four elements:

1) *Policy Decision Point* (PDP). The PDP processes the policies of the system, along with other data such as network state information, and takes policy decisions regarding which policies should be enforced, and how and when this will happen. These policies are sent as configuration data to the appropriate PEPs.
2) *Policy Enforcement Point* (PEP). The PEP is communicated to the managed devices and it is responsible for installing and enforcing the policies sent by the corresponding PDP.
3) Policy repository. This is a policy database which the PDP uses for its decision-making process.
4) Target system. The final target device or element in which the above policies will be enforced.

Regarding the communication protocol between the PDP and its PEPs, the IETF has been focused on the definition of the *Common Open Policy Service* (COPS) protocol [32]. COPS is a simple query and response protocol based on a client-server model that can be used to exchange policy information between a policy server (PDP) and its clients (PEPs).

The inclusion of such a policy-based framework in the DESEREC architecture has been performed as follows: one PDP is included in each DLocalAgent, i.e., one PDP per molecule, whereas one PEP is placed in each DItemAgent. The policy repository is a database (local to the PDP) which receives policy information from the *Local Reaction* module and caches it for both performance and autonomy purposes.

When a new configuration needs to be deployed, the PEP can get the appropriate policy information from its PDP, adapt it (if needed) to the particular device which is being managed and, lastly, enforce it. This model also supports enforcement feedback, via the COPS reports which PEPs can send back to their corresponding PDP.

D. Block Service Module

The policy data provided by a PDP to one of its PEPs may need to be tuned for a specific managed device. This may include not only a change in the notation, but also other specific information; for example, updating the configuration of a service might require different steps to be taken, depending on the particular implementation of that service.

In the DESEREC architecture, just before enforcing the GSRs by the PEP, they should be translated according to the final software installed on the system, e.g., product, version, etc., since the GSRs represent software-independent configurations. This translation is performed by the *Block Service Module* (BSM) [20], using specific translation templates which generate the final device-specific configurations, called *End COnfigurations* (ECO). Finally, these configurations are enforced in the target device by the PEP that manages it through enforcement protocols like SSH/SCP or SNMP. The framework also allows the usage of proprietary protocols, depending on the managed software.

E. Illustrative Example Scenario

This section shows how the DESEREC framework, explained above, is used as a proof of concept to react at runtime when a problem in the system turns up. It describes a set of problematic situations that could cause different dependability and security problems which the DESEREC framework should fix, taking the testbed description in Section IV-B as the design and lab implementation.

Situation 0: everything is fine

This situation shows the whole system working properly in a nominal case. That is, interactions with the railway signaling and control system are made through the provided Web service and administrators can access the management website to request statistics and monitoring information.

Initially, during the first configuration of the DESEREC-managed system, the Global Detection and Reaction Scenarios (GDS and GRS) are automatically stored in the Global Detection and Decision modules, respectively, in order to provide detection and reaction logic at global level. As can be seen in Figure 5, the GDS contains at this level a signature-based rule of the type "*Web service goes down*". The associated reaction, defined in the GRS, will be switching between high-level configurations; in this case, from `HOC1` to `HOC2`.

Then, the most suitable global configuration (`HOC1` in this case) is released and distributed through all the DESEREC components, until configuring the technical services as the configuration policies dictate. At global level, the DCentralAgent extracts from `HOC1` each LOP that will configure each of the defined molecules; that is, `HOC1.LOP1` will be addressed to *molecule-1* and `HOC1.LOP2` to *molecule-2*. In both cases, the corresponding LOP is stored into the local database of each DLocalAgent. The local detection and reaction logic (LDS and LRS) included in each LOP, is automatically stored in the Local Detection and Decision modules. In this example, both local scenarios are empty without defining any reaction capacity.

The most suitable configurations (`HOC1.LOP1.LOC1` in *molecule-1* and `HOC1.LOP2.LOC1` in *molecule-2*) at local level are then deployed to the corresponding target software, with the aim of configuring them properly. In this example, and according to the service allocations defined above, the `HOC1.LOP1.LOC1` contains the GSR belonging to the BSC *signaling_mgment*, which is delivered to the DItemAgent that manages the *apache-2.2.4* software, and the GSR of the BSC *firewall*, which is delivered to the DItemAgent that manages the *iptables-1.3.5* software. Note that each DItemAgent will translate these GSRs to ECOs (final configurations) just before enforcing them in the underlying technical service.

On the other hand, the `HOC1.LOP2.LOC1` contains the GSR associated to the BSC *name_resolver*, which is delivered to the DItemAgent that manages the *bind-9.3.4* software. That DItemAgent will translate it to BIND format before finally enforcing it. This last GSR can be seen in Listing 2, which contains the entire required DNS configuration at high level: information about the zone for the `example.org` domain; and an "A" record to resolve the *signaling* name to a specific IP address for pointing out that the railway control Web service is running on `192.168.1.11`.

Note that although the BSCs *timing* and *access_control* have not been included in Figure 5, their GSRs are also deployed together with the previous ones. The GSR for the BSC *timing* will be always delivered to the DItemAgent that manages the *ntpd-4.1.1* software, independently of the operational plan and configuration, since this is the only technical service that can provide it. On the other hand, the GSR for the BSC *access_control* (see Listing 3) will be delivered to the DItemAgent that manages the *apache-2.2.4* software as defined by the service allocation.

As can be seen, two GSRs are addressed to the same *apache-2.2.4* software for configuring, in the same technical service, the BSCs *signaling_mgment* and *access_control*.

After all this process, the final configuration of the system is as the one depicted in Figure 8a), labeled with the running OCs and scenarios for each agent.

Situation 1 (reallocation): the Web service has become unavailable

This situation shows a global reconfiguration process, i.e., the DCentralAgent is involved on it, when the Web service becomes down and needs to be reallocated. This could be due to the fact that a DoS attack has been performed from a compromised host in the Intranet, an internal error of the Web server itself, etc. As a consequence, the Web server goes down.

Through the left-hand side of the framework, i.e., the monitoring part, shown in Figure 7, different local events harvested from the target system are going up on both molecules until their *Local Detection* modules.

Suddenly, one of these events in DLocalAgent-1 carries a possible problem of the type "*the Web service has become unavailable*". Since the *Local Detection* module in DLocalAgent-1 has no rule in its LDS, as can be seen in Figure 8a), this event is forwarded directly to the DCentralAgent for being managed at global level if necessary. Once the DCentralAgent receives the above event including the actual problem, sent in this case by the DLocalAgent-1 where the service was running, it is capable of detecting that a possible problem has occurred by mapping the event against its GDS. In this example, the GDS includes a global detection rule of the type "*Web service goes down*" and the framework needs to react for fixing it. The associated reaction is to

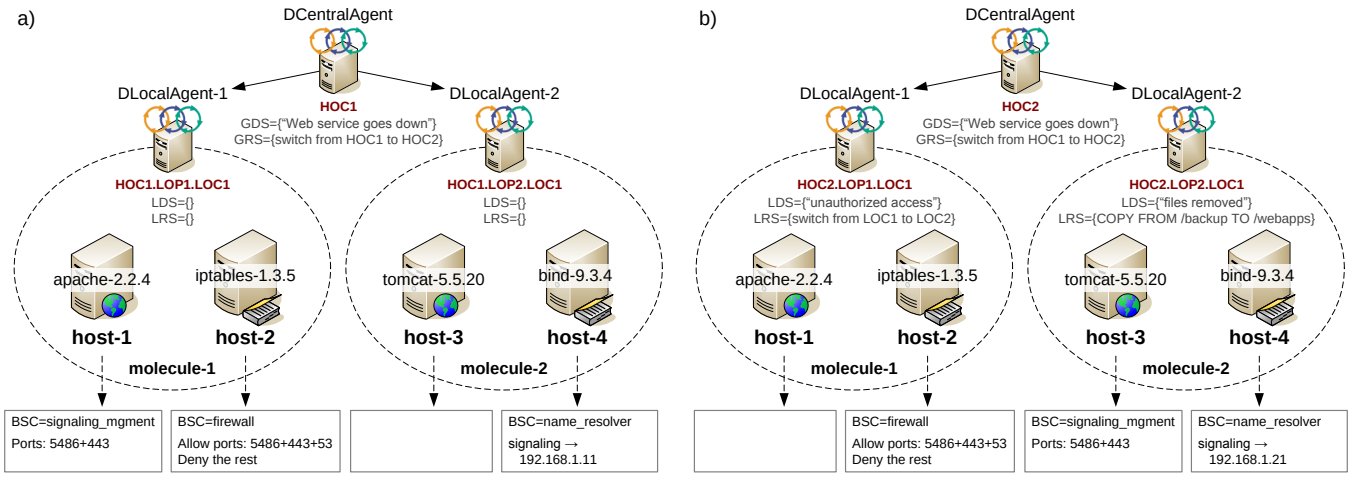

Figure 8. System status: a) initial configuration, situation 0; and b) after a reallocation, situation 1

move (reallocate) that service to another place where it is able to be provided again; in this case, by switching from the current global configuration (`HOC1`) to a new one (`HOC2`) as is defined in the GRS.

As in the previous situation, the new global configuration (`HOC2`) is distributed through all the DESEREC components. Because of this, two new LOPs (`HOC2.LOP1` for *molecule-1* and `HOC2.LOP2` for *molecule-2*) are sent to their corresponding DLocalAgents for deploying a new configuration. After being stored into their local databases, the local detection and reaction logic is updated as the new LOPs dictate. Looking again at Figure 5, the LDS of `HOC2.LOP1` contains a local detection rule of the type "*unauthorized access*" whose reaction (which is later used in Situation 2) is to switch from `HOC2.LOP1.LOC1` to `HOC2.LOP1.LOC2`. On the other hand, the LDS of `HOC2.LOP2` contains another local detection rule of the type "*files removed*" whose reaction (which is later used in Situation 3) is to execute an abstract command to *COPY* these files *FROM* a backup folder *TO* the appropriate place.

After this, the most suitable local configurations for both LOPs are then deployed (`HOC2.LOP1.LOC1` in *molecule-1* and `HOC2.LOP2.LOC1` in *molecule-2*) in a similar way as the one described in Situation 0, letting the system as shown in Figure 8b): the railway control Web service running again, but now in a different allocation (namely, in the *tomcat-5.5.20* software); the BSC *firewall* remains unchanged as before; and a new configuration has been applied in the *bind-9.3.4* software reflecting the reallocation change of the BSC *signaling_mgment* (this service is now running on `192.168.1.21`).

It is worth pointing out that, at global level, the GDS and GRS remain the same that before, although they will have no effect since the rules included in them are only applicable to `HOC1`, and the current running configuration after executing this situation is `HOC2`.

Situation 2 (reconfiguration): unauthorized access to a private resource

In this situation we show how the DESEREC framework is able to reconfigure a service. In this sense, we suppose that an unauthorized user gains access to the website. The response of the framework will be to put into quarantine the website by establishing a more restrictive configuration in the firewall component.

Only the DLocalAgent-1 will be into play in this case since, after locally receiving a possible problem of the type "*a user has made an unauthorized access to a resource*", the *Local Detection* module in DLocalAgent-1 has a location detection rule in its LDS, as can be seen in Figure 8b), that will throw a local alarm. The reaction associated to this incident, defined in the LRS, is to deploy a new local configuration (by switching between `HOC2.LOP1.LOC1` and `HOC2.LOP1.LOC2`) to reconfigure the firewall with a more restrictive filtering rules. In this new configuration, the firewall will only permit connections on ports 5486 and 53, blocking the access to the 443 (HTTPS) port.

Note that, despite reconfiguring a service, the allocations are maintained as before since the BSC *firewall* still continues to run on the same technical service, but now with a different configuration. As seen in this situation, the *molecule-2* remains the same since the reaction has been carried out locally in *molecule-1* without involving the rest.

Situation 3 (abstract command execution): some important files have been removed

In this last situation we show a third sort of reaction that the DESEREC framework is capable of managing by means of executing a set of abstract commands. Due to a temporal hard disk fail, important Web service files have been removed and the Web service becomes inconsistent. In

this case, when a request is made to the Web service, it will trigger an internal error.

Once the DESEREC framework catches that internal error by means of the events received from the target system, a possible problem of the type "*some files have been either removed or modified*" is detected. In this last situation, the associated reaction to this alarm (defined in the LRS) is sending of an abstract command which will copy the removed files from a backup folder to the appropriate place in the Web server. This abstract command is generically defined as `COPY FROM <source> TO <destination>`, which will have to be translated to the specific command depending on the system where it will be enforced. For example, in Linux systems this abstract command will be translated to `cp -r /backup /webapps`.

As before, this situation does not suppose any change in the allocation map, although in this case neither in the configurations. The reaction is only a very slight adjustment in the target system without changing anything in the running allocations and configurations.

VIII. Deployment and Validation

In this section we summarize how to design a dependable system following the proposed framework, as well as our experience in its deployment and how it has been validated and tested in existing mission critical systems.

A. Deployment of the DESEREC framework

The administrator(s) of the system, where the proposed framework has to be installed, must initially provide the molecule breakdown of the system. This will usually be done from scratch, although it could be also done in a non-intrusive mode by starting from an existing CIS. In both cases, the system administrator knows the Business Services that should be provided, for making them secure and thereby ensuring a given QoS. These Business Services have to be decomposed into Business Service Components for which the system administrator should define a list of constraints or policies using SCL. On the other hand, a system description should be provided by means of SDL, containing subsets of the whole CIS that are able to support technical services; that is, molecules to provide the required Business Services. In turn, these molecules have to be decomposed into software and network components, up to the level where they can be monitored, configured and deployed.

In this process, the administrator defines a synthetic and graphical view of the technical services, as well as the linked molecules. With them, and thanks to the Planning tools, the administrator can simulate errors, crashes, security attacks, etc., which means that few molecules could become unavailable. In each case, our framework is able to compute the best allocation of available molecules (according to the defined high-level policies), with the minimum number of reconfiguration steps involving molecule instances.

From the above defined molecule type description, the system administrator can generate and deploy the global and local configurations, together with their reaction plans. From this moment, it starts the runtime execution of the reconfiguration framework, in an autonomous way, as presented in Section VII.

B. Validation in real environments

The fulfillment of the DESEREC objectives have been evaluated by taking three typical cases of critical systems, which were provided by three end-user partners belong to the DESEREC consortium; namely:

- Hellenic Telecommunications Organization (OTE) [33]. OTE is the leading telecommunication operator in Greece and the Balkan area and, as such, operates most of the critical telecom infrastructures installed in that country. Therefore, securing its telecom infrastructure is a critical issue. The exploitation of the DESEREC project results has been very interesting in its lab testbed through a *TV over IP* (IPTV) scenario. It is worth mentioning that on this testbed the DESEREC consortium presented its results to the European Commission through a final demonstration at OTE premises.
- EADS Defence & Security Systems (DS) [34]. This partner provided to the consortium its *Security Command and Control System*, as main provider for the French Army. The main goal in this testbed was to minimize the risk exposure through protecting people and territories. In this case, EADS proposed a scenario where a border guard checks the passport of a person which is detected as blacklisted. Then, the border guard creates an alarm in the Web application in order to signal the problem, which is dispatched to the *Command and Control* application thanks to the *Enterprise Service Bus* (ESB). However, the border security employee plugs a USB key on the computer that hosts the ESB and it executes a malicious code which is present on his key. The code triggers an anomaly in the ESB service, which goes down and is no more able to forward client requests to the access control system. The DESEREC framework was successfully deployed to compute and execute the appropriate reaction, by making the service available again.
- RENFE-Operadora [35]. RENFE is the national railway operator in Spain, providing public service of train transportation for both passengers and trade goods. In this case, the DESEREC partners used the RENFE testbed to test a first approach of the proposed framework. These tests were focused on the railway signaling and control system, presented along with this paper, as well as the management of the *Ticket Selling* service.

IX. Conclusion and Future Work

In this paper we have presented a specific framework for managing service dependability in a policy based fashion. The concept of policy based management has been around in the research scene for several years now, with proven validity as an intuitive and scalable way for administrators to keep large information systems under control, ensuring the continuous enforcement of domain directives. Here we have checked how building a dependability management framework on a policy based core has indeed achieved to leverage the potential of this paradigm, applying it to a novel field. The proposed framework allows using the same abstract approach inherent to policy based solutions for managing also the automatic, on-demand configuration of system services.

Lastly, the tools developed for administrator interaction have allowed putting this proposal into practice, serving as further validation of the claimed achievements. By this, a complete example has been developed throughout this paper to stage how the proposed framework works in a fully autonomous way without human intervention.

As future work, some extensions remain to be taken into account which would improve this framework considerably. The current configuration policies that govern the system should be extended to include also setting up in a similar manner our own modules belonging to the framework; for example, the collection of sensors needed for a concrete operational plan, indicating the configuration for each of them.

Acknowledgment

This work has been funded by the DESEREC EU IST Project (IST-2004-026600), within the EC Sixth Framework Programme (FP6). Thanks also to the Funding Program for Research Groups of Excellence granted by the Seneca Foundation with code 04552/GERM/06.

References

[1] J. Bernal Bernabé, J.M. Marín Pérez, D.J. Martínez Manzano, M. Gil Pérez, and A.F. Gómez Skarmeta. "Towards a Policy-driven Framework for Managing Service Dependability". In *DEPEND '09: Proceedings of the 2nd International Conference on Dependability*, pages 66–72, 2009.

[2] E. Amidi. "System and Method for Providing a Backup-Restore Solution for Active-Standby Service Management Systems". U.S. Patent Application 20060078092, April 2006.

[3] A. Avizienis, J. Laprie, and B. Randell. "Fundamental Concepts of Dependability". Research Report 1145, LAAS-CNRS, April 2001.

[4] The SERENITY EU-IST Project (System Engineering for Security & Dependability). http://www.serenity-project.org [22 February 2010].

[5] R. Sterritt and D. Bustard. "Towards an Autonomic Computing Environment". In *DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, pages 699–703, 2003.

[6] R. Sterritt and D. Bustard. "Autonomic Computing-a Means of Achieving Dependability?". In *ECBS '03: Proceedings of IEEE International Conference on the Engineering of Computer Based Systems*, pages 247–251, 2003.

[7] D.C. Verma. "Simplifying Network Administration using Policy-based Management". *IEEE Network Magazine*, 16(2):20–26, 2002.

[8] J. Schönwälder, A. Pras, and J.P. Martin-Flatin. "On the Future of Internet Management Technologies". *IEEE Communications Magazine*, 41(10):90–97, 2003.

[9] The DESEREC EU-IST Project (Dependability and Security by Enhanced Reconfigurability). http://www.deserec.eu [22 February 2010].

[10] L. Ardissono, A. Felfernig, G. Friedrich, A. Goy, D. Jannach, G. Petrone, R. Schäfer, and M. Zanker. "A Framework for the Development of Personalized, Distributed Web-based Configuration Systems". *AI Magazine*, 24(3):93–108, 2003.

[11] A. Felfernig, G.E. Friedrich, and D. Jannach. "UML as Domain-Specific Language for the Construction of Knowledge-based Configuration Systems". *International Journal of Software Engineering and Knowledge Engineering*, 10:449–469, 2000.

[12] M. Caporuscio, A. Di Marco, and P. Inverardi. "Model-Based System Reconfiguration for Dynamic Performance Management". *Journal of Systems and Software*, 80(4):455–473, 2007.

[13] M. Mikic-rakic, S. Malek, and N. Medvidovic. "Improving Availability in Large, Distributed, Component-Based Systems via Redeployment". In *CD '05: Proceedings of the 3rd International Working Conference on Component Deployment*, pages 83–98, 2005.

[14] G. Spanoudakis, A. Maa Gomez, and S. Kokolakis. *"Security and Dependability for Ambient Intelligence"*. Springer Publishing Company, Incorporated, 2009.

[15] "Willow Survivability Architecture", Dependability Research Group, University of Virginia. http://dependability.cs.virginia.edu/research/willow [22 February 2010].

[16] Z. Hill, J. Rowanhill, A. Nguyen-Tuong, G. Wasson, J. Knight, J. Basney, and M. Humphrey. "Meeting Virtual Organization Performance Goals through Adaptive Grid Reconfiguration". In *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 177–184, 2007.

[17] Z. Hill and M. Humphrey. "Applicability of the Willow Architecture for Cloud Management". In *ACDC '09: Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, pages 31–36, 2009.

[18] D.J. Martínez Manzano, M. Gil Pérez, G. López Millán, and A.F. Gómez Skarmeta. "A Proposal for the Definition of Operational Plans to provide Dependability and Security". In *CRITIS '07: Proceedings of the 2nd International Workshop on Critical Information Infrastructures Security*, pages 223–234, 2007.

[19] A. Cotton, M. Israël, and J. Borgel. "Molecular Approach Paves the Way towards High Resilience for Large Mission-Critical Information Systems". In *SECURWARE '08: Proceedings of the 2nd International Conference on Emerging Security Information, Systems and Technologies*, pages 332–337, 2008.

[20] The DESEREC EU-IST Project. Deliverable D2.1, "Policy and System Models", March 2007.

[21] M.D. Aime, P.C. Pomi, and M. Vallini. "Policy-Driven System Configuration for Dependability". In *SECURWARE '08: Proceedings of the 2nd International Conference on Emerging Security Information, Systems and Technologies*, pages 420–425, 2008.

[22] "Web Services Choreography Description Language v1.0". W3C Candidate Recommendation 9, November 2005. http://www.w3.org/TR/ws-cdl-10 [22 February 2010].

[23] G. Martínez Pérez, A.F. Gómez Skarmeta, S. Zeber, J. Spagnolo, and T. Symchych. "Dynamic Policy-Based Network Management for a Secure Coalition Environment". *IEEE Communications Magazine*, 44(11):58–64, 2006.

[24] "Common Information Model (CIM) Standards", Distributed Management Task Force, Inc. http://www.dmtf.org/standards/cim [22 February 2010].

[25] M. Debusmann and A. Keller. "SLA-driven Management of Distributed Systems using the Common Information Model". In *IM '03: Proceeding of the 8th IFIP/IEEE International Symposium on Integrated Network Management*, pages 563–576, 2003.

[26] H. Mao, L. Huang, and M. Li. "Web Resource Monitoring Based on Common Information Model". In *APSCC '06: Proceedings of the 2006 IEEE Asia-Pacific Conference on Services Computing*, pages 520–525, 2006.

[27] C. Hobbs. *"A Practical Approach to WBEM/CIM Management"*. CRC Press, April 2004.

[28] "SCL Console and User Manual v1.0", University of Murcia. http://deserec.inf.um.es/console [22 February 2010].

[29] J. Rubio Loyola. *"A Methodological Approach to Policy Refinement in Policy-based Management Systems"*. PhD thesis, Technical University of Catalonia, Spain, April 2007.

[30] J.M. Marín Pérez, J. Bernal Bernabé, J.D. Jiménez Re, G. Martínez Pérez, and A.F. Gómez Skarmeta. "A Proposal for Translating from High-Level Security Objectives to Low-Level Configurations". In *SVM '07: Proceedings of the 1st International DMTF Academic Alliance Workshop on Systems and Virtualization Management: Standards and New Technologies*, 2007.

[31] R. Danyliw, J. Meijer, and Y. Demchenko. "The Incident Object Description Exchange Format". IETF RFC 5070, December 2007.

[32] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry. "The COPS (Common Open Policy Service) Protocol". IETF RFC 2748, January 2000.

[33] Hellenic Telecommunications Organization (OTE). http://www.ote.gr [22 February 2010].

[34] EADS Defence & Security Systems (DS). http://www.eads.com [22 February 2010].

[35] RENFE-Operadora. http://www.renfe.es [22 February 2010].